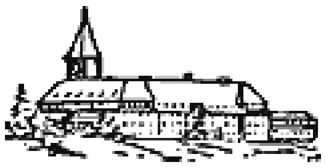
The Eighth IEEE Workshop on Hot Topics in Operating Systems

HotOS-VIII



May 20-23, 2001
Schoss Elmau, Germany

Sponsored by

IBM Research

HP Labs

Microsoft Research

This page is intentionally almost blank.

Table of Contents

The Eighth Workshop on Hot Topics in Operating Systems - HotOS-VIII

Beyond Address Spaces -
Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System

Michael Golm, Jürgen Kleinöder, Frank Bellosa
University of Erlangen-Nürnberg
Dept. of Computer Science 4 (Distributed Systems and Operating Systems)
Martensstr. 1, 91058 Erlangen, Germany
{golm,kleinoeder,bellosa}@informatik.uni-erlangen.de

Abstract

Early type-safe operating systems were hampered by poor performance. Contrary to these experiences we show that an operating system that is founded on an object-oriented, type-safe intermediate code can compete with MMU-based microkernels concerning performance while widening the realm of possibilities.

Moving from hardware-based protection to software-based protection offers new options for operating system quality, flexibility, and versatility that are superior to traditional process models based on MMU protection. However, using a type-safe language—such as Java—alone, is not sufficient to achieve an improvement. While other Java operating systems adopted a traditional process concept, JX implements fine-grained protection boundaries. The JX System architecture consists of a set of Java components executing on the JX core that is responsible for system initialization, CPU context switching and low-level domain management. The Java code is organized in components which are loaded into domains, verified, and translated to native code.

JX runs on commodity PC hardware, supports network communication, a frame grabber device, and contains an Ext2-compatible file system. Without extensive optimization this file system already reaches a throughput of 50% of Linux.

1 Introduction

For several years there has been an ongoing discussion in the OS community whether software-based protection is a promising approach [3]. We want to support the arguments for software-based protection with the experience we gained while building the JX operating system.

While MMU-based protection is commonly used in today's operating systems it has some deficiencies [10], [3]. From the point of functionality it neither meets the actual requirements of fine grained protection (page size is too coarse), nor offers it appropriate abstractions for access control (page tags are not capabilities).

These deficiencies justify the exploration of alternative protection mechanisms. Java popularized a protection mechanism that is based on a combination of type-safe intermediate code and load-time program verification.

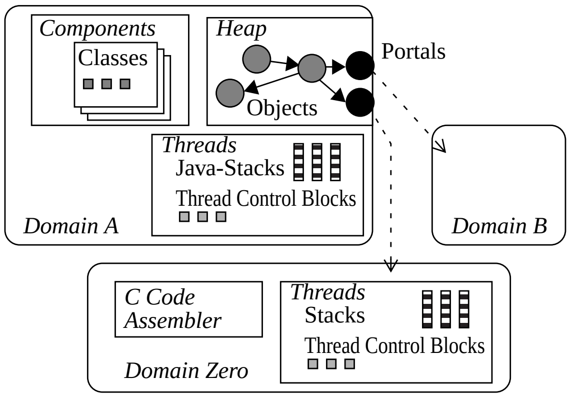
Several other research groups have been building Java-based operating systems: Sun's JavaOS [14], which was later replaced by "JavaOS for Business" [18], JN [16], J-Kernel [11], KaffeOS [2], and Joust [9]. But they are either limited by a monolithic structure or are built upon a full-featured OS and JVM. Furthermore, no performance figures for OS related functionality are published. KaffeOS and J-Kernel are two projects that try to overcome the monolithic structure by intruducing a process concept which is similar to the domain concept of JX. But their research is mainly concerned with introducing the traditional process concept and a red line [6] between user level and kernel into their Java operating system. While a red line between trusted and untrusted code is indeed important, we must free our mind from the MMU-enforced architecture of traditional operating systems. The aim of our research is a customizable and flexible [4] open OS architecture with fine-grained protection boundaries. Depending on functionality and deployment of a system there are different levels of trust and protection. An embedded real-time system needs a different red line than a single-user desktop system or a multi-user server system or an active network node OS [5]. In our architecture it is possible to draw red lines when and where they are needed.

While other Java operating systems require a microkernel, or even a full operating system including a JVM, JX runs on the bare hardware with only a minimal statically linked core (< 100kB). The remaining operating system functionality, including device drivers, is provided by Java components that are verified, compiled to native code, and optimized at load time.

The paper is structured as follows: In section 2 we describe the architecture of the JX system. The problems that appear when untrusted modules directly access hardware are discussed in section 3. Section 4 gives examples of the performance of IPC and file system access.

2 JX System Architecture

The JX system consists of a small core, written in C and assembler, which is less than 100 kilobytes in size. The majority of the system is written in Java and running in separate protection domains. The core runs without any protection and therefore must be trusted. It contains functionality that can not be provided at the Java level (system initialization after boot up, saving and restoring CPU state, low-level domain management, monitoring).



The Java code is organized in components (Sec. 2.2) which are loaded into domains (Sec. 2.1), verified (Sec. 2.4), and translated to native code (Sec. 2.5). A domain can communicate with another domain by using portals (Sec. 2.3).

The protection of the architecture is solely based upon the JX core, the code verifier, the code translator, and hardware-dependent components (Sec. 3). These elements are the *trusted computing base* [7] of our architecture.

2.1 Domains

A domain is the unit of protection, resource management, and typing.

Protection. Components in one domain trust each other. One of our aims is code reusability between different system configurations. A component should be able to run in a separate domain, but also together (co-located) with other components in one domain. This leads to several problems:

- The parameter passing semantics must be by-copy in inter-domain calls, but may be by-reference in the co-located case. This is an open problem.
- During a portal call a component must check the validity of the parameters because the caller could be in a different domain and is not trusted. When caller and callee are co-located (intra-domain call), the checks change their motivation—they are no longer done for security reasons but for robustness reasons. We currently parametrize the component whether a safety check should be performed or not.

Resource Management. JX domains have their own heap and own memory area for stacks, code, etc. If a domain needs memory, a domain-specific policy decides whether this request is allowed and how it may be satisfied, i.e., where the memory comes from. Objects are not shared between domains, but it is possible to share memory. Other Java systems use shared objects with the consequence of complicated and not interdependent garbage collection, problems during domain termination, and quality-of-service crosstalk [13] between garbage collectors.

Typing. A domain has its own type space, that initially contains exactly one type: java.lang.Object. Types (classes and interfaces) and code (classes) can then be loaded into the domain. Our type-space approach differs from the Java type spaces [12] as we do not use the class loader as type-space separator but tie type separation to resource management and protection. By this means a SecurityManager becomes redundant and protection boundaries are automatically enforced.

The C and assembler code of the JX core are encapsulated by a special domain, called *DomainZero*. All other domains contain only Java code. We do not allow native methods.

2.2 Components

Code is generally loaded as a component. JX does not support loading of single classes. A component is a collection of classes and interfaces. There are four kinds of components:

- *Library*: A simple collection of reusable classes and interfaces (example: the Java Development Kit).
- *Service*: A component that implements a specific service, e. g., a file system or a device driver. A service component is *started* after it has been loaded. To start a service means to execute a static method that is specified in a configuration file that is part of the component.
- *Interface*: Access to a service in another domain is always performed using an interface. An interface component contains all interfaces that are needed to access a service. An interface component also contains the classes of parameter objects. A special interface library *zero* contains all interfaces to access DomainZero.

• *Domain*: A domain is started by loading a domain component. An initial thread is created and a static method is executed.

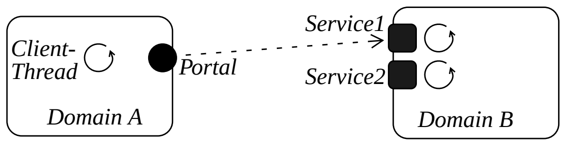
Components can be shared between domains. Sharing happens at two levels. At a logical level sharing establishes a window of type compatibility between two domains. At a lower level, sharing saves memory, because the (machine) code of the component has to be stored only once. While component sharing complicates resource accounting and domain termination, we believe that code sharing is an essential requirement for every real operating system. While code can be shared if the domains use the same type of execution environment (translator, memory layout), static variables are never shared. In JX this is implemented by splitting the internal class representation into a domain-local part, that contains the statics, and a shared part, that contains code and meta information.

2.3 IPC, Portals, and Services

Domains communicate solely by using portals. An object that may be accessed from another domain is called *service*. Each service is associated with a *service thread*.

A portal is a remote reference that represents a service, which is running in another domain. Portals are capabilities that can be passed between domains. Portals allow to establish the “principle of least privilege”. A domain gets only the portals it needs for doing its job.

A portal looks like a normal object reference. The portal type is an interface that is derived from the interface Portal. A portal invocation behaves like a normal synchronous interface method invocation: The calling thread is blocked, the service thread executes the method, returns the result and is then again available for new service requests via a portal. The caller thread is unblocked when the service method returns. While a service thread is processing a request, further requests for the same service are blocked.



An object reference can be passed as parameter of a portal invocation only if the object is a service. In this case a portal to the service is transferred and the reference counter of the service is incremented. Other parameters are passed by value. When a portal is no longer referenced in a domain, it is removed by the garbage collector and the reference counter of the associated service is decremented.

A portal/service connection between two domains requires that these domains have overlapping type spaces, i.e. the interface component must be logically shared. If the interface component depends on other components, they must be shared, too.

2.4 Component Verifier

When a component is loaded into a domain, its bytecode is verified before it is translated into machine code. As in the normal Java bytecode verifier, the conformance to the Java rules is checked. Basically this guarantees type safety. Furthermore the verifier performs additional JX-specific checks regarding interrupt handlers (Sec. 2.6), memory objects (Sec. 2.7), and schedulers (Sec. 2.9).

A type-safe operating system has the well-known advantages of robustness and ease of debugging. Furthermore, it is possible to base protection and optimization mechanisms on the type information. This is extensively employed in JX by using well-known interfaces (contained in a trusted library) and restricting the implementability of these interfaces (Sec. 2.6 and 2.7).

2.5 Component Translator

Components are translated from bytecode into machine code. The translator is a crucial element of JX to get a reasonable performance. The translator is domain-specific, so it can be customized for a domain to employ application-specific translation strategies. The same component may be translated differently in different domains. As the translator is a trusted component, this facility has to be used carefully because it affects the protection of the whole system.

Furthermore the translator is used to “short-circuit” several portal invocations. Special portals that are exported by DomainZero often do not need the domain context of DomainZero. Invocations of such portals can be inlined directly at the call site.

2.6 Interrupts

An interrupt is handled by invoking the handleInterrupt method of a previously installed interrupt handler object. The method is executed by a dedicated thread while interrupts on the interrupted CPU are disabled. This would be called the *first-level interrupt handler* in a traditional operating system. To guarantee that the handler can not block the system forever, the verifier checks all classes that implement the InterruptHandler interface whether the handleInterrupt method has certain time bounds. To avoid undecidable problems, only a simple code structure is allowed (linear code, loops with constant bound and no write access to the loop variable inside the loop). A handleInterrupt method usually acknowledges the interrupt at the device and unblocks a thread that handles the interrupt asynchronously.

2.7 Memory Management

Heap and Garbage Collection. The memory of the objects within a domain is managed by a heap manager with garbage collector. Currently, the heap manager is part of the JX core. It cooperates with the translator to obtain information about the object structure and stack structure. So far we are working with only one heap manager implementation and one translator implementation, but it is also possible to build domain-specific heap managers. They can even be written in Java and run in their own domain. The heap manager is a trusted part of the system.

Memory objects. To handle large amounts of data, Java uses arrays. Java arrays are useless for operating system components, because they do not provide access control and it is not possible to share only a part of an array. JX uses Memory objects instead. The memory that is represented by such a Memory object can be accessed via method invocations. These invocations are inlined by inserting the machine instructions for the memory access instead of the method invocation. This makes memory access as fast as array access. A Memory object can represent a part of the memory of another Memory object and Memory objects can be shared between domains like portals. Sharing memory objects between domains and the ability to create subranges are the fundamental mechanisms for a zero-copy implementation of system components, like the network stack, the file system, or an NFS server.

Avoiding range checks by object mapping. A memory range can be mapped to a (virtual) object that implements a marker interface (an interface without methods that is only used to mark a class as MappedLittleEndian or MappedBigEndian). The verifier ensures that a class that implements one of these interfaces is never instantiated by the new bytecode. Instead the objects are created by mapping and the translator generates code to directly access the memory range for access to instance variables. This makes the range check redundant.

2.8 Domain Termination

When a domain terminates, all resources must be released. Further interaction with the domain raises an exception.

All services are removed by stopping the service thread. A service contains a reference counter, that is incremented each time a portal to this service is passed to another domain. It is also incremented when a client domain passes the portal to another client domain. It is decremented, when the portal object in a client domain is garbage collected or when the client domain is terminated. As long as the reference counter is not zero, the service can not be completely removed when its domain terminates. Until all reference counters drop to zero, the domain remains in a *zombie* state.

Interrupt handlers are uninstalled. All threads are stopped and the memory (heap, stacks) is released.

2.9 Scheduling

CPU scheduling in JX is split into two scheduler levels. The *low-level scheduler* decides which domain should run on the CPU. Each CPU has its own low-level scheduler. The *high-level scheduler* is domain-specific - each domain has one high-level scheduler per available CPU. A domain may not be allowed to use all CPUs. To use a CPU, the domain must obtain a CPU portal for the specific CPU. The high-level schedulers are responsible for scheduling the threads of a domain.

The high-level scheduler may be part of the domain or may be located in a different domain.

To avoid that one domain monopolizes the CPU, the computation can be interrupted by a timer interrupt. The timer interrupt leads to the invocation of the low-level scheduler. The low-level scheduler first informs the high-level scheduler of the interrupted domain about the preemption. For this purpose it invokes a method of the high-level scheduler with interrupts disabled. An upper bound for the execution time of this method has been verified during the verification phase. When the method returns, the system switches back to the low-level scheduler. The low-level scheduler then decides, which domain to run next. After ensuring that it will be reactivated with the next (CPU-local) timer interrupt, the low-level scheduler activates the high-level scheduler of the selected domain. The high-level scheduler chooses the next runnable thread. It can switch to this thread by calling a method at the CPU portal. This method can only be called by a thread that runs on the corresponding CPU.

3 Device Drivers

Due to the enormous amount of new hardware that appeared in the last years, operating system code is dominated by device drivers. While it is rather straight forward to move most operating system parts, such as file systems or network protocols, out of the trusted kernel, it is very difficult for device drivers.

Developers of commodity hardware do not assume that their products are directly accessed by untrusted code. Although the Nemesis project has demonstrated that it is possible to build user-safe hardware [17], we do not expect such hardware to become commercially available in the near future.

Device drivers in JX are programmed in Java and are installed as service component in a domain. JX aims at only trusting the hardware manufacturer (and not the driver provider) in assuming that the device behaves exactly according to the device specification. When special functionality of the hardware allows bypassing the protection mechanisms of JX, the code for controlling this functionality must also be trusted. This code can not be part of the JX core, because it is device dependent. One example for such code is the busmaster DMA initialization, because a device can be programmed to transfer data to arbitrary main memory locations.

To reduce the amount of critical code, the driver is split into a (simple) trusted part and a (complex) untrusted part.

To understand the issues related to device drivers, we have developed drivers for the IDE controller, the 3C905B network card, and the Bt848 framegrabber chip. The IDE controller and network card basically use a list of physical memory addresses for busmaster DMA. The code that builds and installs these tables is trusted. The Bt848 chip can execute a program in a special instruction set (RISC code). This program writes captured scanlines into arbitrary memory regions. The memory addresses are part of the RISC program. We currently trust the RISC generator and thus limit extensibility. To allow an untrusted component to download RISC code, we would need a verifier for this instruction set.

All microkernel-based systems, where drivers are moved into untrusted address spaces run into the same problems, but they have much weaker means to cope with these problems. Using an MMU does not help, because busmaster DMA accesses physical RAM without consulting page tables. JX uses type-safety, special checks of the verifier, and splitted drivers to address these problems.

4 Performance

IPC. We measured the performance of a portal call. Table 1 compares the IPC round-trip performance of JX with fast microkernels and other Java operating systems.

System	IPC (cycles)
L4KA (PIII, sysenter, sysexit) [8]	800
Fiasco/L4 (PIII 450 MHz) [http://os.inf.tu-dresden.de/fiasco/status.html]	2610
J-Kernel (LRMI on MS-VM, PPro 200MHz) [11]	440
Alta/KaffeOS [1]	27270
JX/hosted (Linux 2.2.14, PIII 500MHz)	7100
JX/native (PIII 500MHz)	650

Table 1: IPC latency (round-trip)

Comparing IPC times for these systems is not easy because they were measured on different hardware (cache size, cache bandwidth, memory bandwidth, etc.), and, more importantly, they have different protection models. IPC is usually more expensive on a system with better protection. Currently the IPC path in JX is implemented in C and not optimized. It may be better compared with the Fiasco implementation of L4 than with L4KA. The emphasis of our work was on getting the architecture right and enabling performance, but not achieving it. The bad performance of Linux-hosted JX can be attributed to the use of sigprocmask to disable/restore signals.

The IPC cost of J-Kernel does *not* include thread switching costs, because the J-Kernel uses a “segmented” stack. IPC without switching threads complicates resource accounting, garbage collection, termination, and type separation.

File System. We have implemented the ext2fs in Java [19]. We reused the algorithms that are used in Linux-ext2fs.

We used the iozone benchmark to measure the Linux ext2fs re-read throughput (file size: 4 kB, record length: 4 kB — iozone -r 4 -s 4 -i 0 -i 1). To measure JX re-read throughput we wrote a Java benchmark, similar to iozone.

The system configuration that we measured works as follows: The virtual file system, the buffer cache, and the ext2 file system run in one domain (FSDomain). The IDE device driver runs in another domain. The client runs in a third domain. A service thread in the FSDomain accepts client requests. The client domain gets a portal to the virtual file system and calls lookup to get a FileInode portal. FSDomain uses one thread to asynchronously receive data from the block device driver. Only the service thread is active in this benchmark, because all data comes from the buffer cache.

System	Throughput (MByte/s)	Latency (µsec/4kB)
Linux (PIII 500 MHz)	400	10.0
JX (PIII 500MHz)	201	19.9
JX co-located (PIII 500MHz)	213	18.7

Table 2: File system re-read throughput and latency

We now try to estimate the necessary performance improvement to reach Linux throughput. The latency can be broken down as shown in table 3.

Operation	JX	JX goal
memory copy	5.2	5.2
IPC	1.3	1.3
file system logic	13.2	3.5

Table 3: Latency breakdown (in µsec)

Memory copy and IPC are relative constant costs in JX. The poor performance of the file system logic is not a problem of the JX architecture but of our non-optimizing compiler. With an improvement of factor 4 in Java performance, we would reach the Linux performance level. Although safety-related overhead cannot be avoided completely, recent research in JIT compiler technology has shown that an optimizing compiler can improve the performance of a Java program significantly. Performance differences of factor 10 are not unusual between non-optimizing and optimizing Java compilers.

5 Status and future work

The system runs either on standard PC hardware (i486, Pentium, and embedded PCs with limited memory) or as a guest system on Linux. The JX Java components also run on a standard JDK (with an emulation for Memory objects). When running on the bare hardware, the system can access IDE disks [19], 3COM 3C905 NICs [15], and Matrox G200 video cards. The network code contains IP, TCP, UDP, NFS2 client, and SUN RPC. JX also runs on a PIII SMP machine.

We have already implemented a heap manager that runs in its own domain and manages the heap of another domain. This heap manager is always called, when the managed domain tries to create a new object or array. Creating a new object with the build-in mechanism costs 250 cycles, calling another domain adds at least 650 cycles. This is not practical until we further improve IPC performance. There are also efforts to improve the quality of the machine code generated by the translator.

The JX architecture supports a broad spectrum of OS structures — from pure monolithic to a vertical structure similar to the Nemesis OS [13]. We are going to investigate the issues that are involved when reusing components between these diverse operating system configurations.

6 References

[1] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, J. Lepreau. Techniques for the Design of Java Operating Systems. In *Proc. of the 2000 USENIX Annual Technical Conference*, June 2000

[2] G. Back, W. C. Hsieh, J. Lepreau. Techniques for the Design of Java Operating Systems. In *Proc. of the 4th OSDI*, Oct. 2000

[3] B. Bershad. S.Savage, P. Pardyak. Protection is a Software Issue. In *Proc. of the Fifth Workshop on Hot Topics in Operating Systems*, pp 62-65, 1995

[4] V. Cahill. *Flexibility in Object-Oriented Operating Systems: A Review.* Technical Report TCD-CS-96-05, Dep. of Comp. Science Trinity College Dublin, 1996

[5] K. Calvert (ed.), *Architectural Framework for Active Networks*, Version 1.0, Active Networks Working Group, July 1999

[6] D. R. Cheriton. Low and High Risk Operating System Architectures. In *Proc. of OSDI*, pg. 197, Nov. 1994

[7] Department of Defense. *Trusted computer system evaluation criteria.* DOD Standard 5200.28, Dec. 1985

[8] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proc. of the 9th SIGOPS European Workshop*, Sep. 2000.

[9] J. Hartman, L. Peterson, A. Bavier, P. Bigot, P. Bridges, B. Montz, R. Piltz, T. Proebsting, and O. Spatscheckti. Experiences building a communication-oriented JavaOS, *Software--Practice and Experience*, 30 (10), Apr. 2000

[10] C. Hawblitzel, T. von Eicken. *A case for language-based protection.* Technical Report TR-98-1670, Dep. of Comp. Science, Cornell University, March 1998

[11] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, T. von Eicken. Implementing Multiple Protection Domains in Java. In *Proc. of the USENIX Annual Technical Conference*, New Orleans, LA, June 1998

[12] S. Liang, G. Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proc. of OOPSLA '98*, October 1998

[13] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7), pp. 1280-1297, Sept. 1996

[14] P. Madany, et. al. *JavaOS : A Standalone Java Environment.* White Paper, Sun Microsystems, May 1996

[15] M. Meyerhöfer. *Design und Implementierung eines Ethernet-Treibers und TCP/IP-Protokolls für dasJava-Betriebssystem JX*, Studienarbeit (supervised by M. Golm), University of Erlangen, IMMD4, Oct. 2000

[16] B. R. Montague. *JN: An Operating System for an Embedded Java Network Computer*, Technical Report UCSC-CRL-9629, University of California, Santa Cruz, 1996

[17] I. A. Pratt. *The User-Safe Device I/O Architecture*. Ph.D. thesis, King's College, University of Cambridge, 1997

[18] Sun Microsystems, *IBM. JavaOS for Business, Reference Manual*, Version 2.1, Oct. 1998

[19] A. Weissel. *Ein offenes Dateisystem mit Festplattensteuerung für metaXaOS.* Studienarbeit (supervised by M. Golm), Univ. of Erlangen, IMMD4, Feb. 2000

Design Issues in System Support for Programmable Routers

Prashant Pradhan, Kartik Gopalan, Tzi-cker Chiueh
Dept. of Computer Science, SUNY Stony Brook

Abstract

Placement of computation inside the network is a powerful computation model that can improve the overall performance of network applications. In this paper, we address the problem of providing sound and efficient system support for placing computation in a network router. We identify a set of requirements, related to protection, resource control, scheduling and efficiency, that are relevant to the design of this system support. We have developed a system that attempts to meet these requirements, and have used it to write a router application that performs aggregated congestion control.

1 Introduction

The success of the Internet can largely be attributed to the simplicity and robustness of its service model. Providing a simple, stateless forwarding service and keeping all complexity in end-systems has allowed the Internet to scale to enormous proportions. Good end-to-end algorithms, like TCP for congestion control, have further contributed to the adequacy of this model, by being able to control network stability using purely end-to-end mechanisms.

However, there is quantitative evidence to show that the ability to place computation inside the network can lead to significant performance optimizations for applications. Network-resident computation has topological advantage that allows routers to perform local recovery for reliable multicast [1] or media gateways to adapt to heterogenous receivers [2]. Also, by having access to multiple flows belonging to an application (or *global context*), network-resident computation can enable some global optimizations for all the flows of that application. For example, sharing of congestion state across a set of TCP flows that share a bottleneck link helps short TCP flows achieve better throughput [3], whereas global knowledge of session-to-server mappings in an SSL server cluster leads to improved connection throughput [4]. In many cases, the ability to place computation even in a restricted set of network nodes (e.g. edge routers) can provide a large subset of the possible benefits. We call such computation “router extensions” or “router applications”.

However, the success of this paradigm in real networks critically depends upon the existence of carefully designed system support for router programmability. Without appropriate resource control and protection mechanisms, dynamically added computation can effect the performance and integrity of a router in undesirable ways. Moreover, since routers are massively shared systems, the computation and I/O resources of a programmable router are scarce resources that must be effectively arbitrated. In addition, in order to make it practical for performance sensitive applications to use router extensions, router extensions must be efficiently executed. Our goal in this paper is to discuss some of the requirements for sound and efficient OS support for router programmability.

In the following sections, we describe the main requirements that we believe should be taken into account while designing a router OS.

2 Efficient Memory Protection

Memory protection is a basic requirement for maintaining system integrity in the presence of dynamically installed functions. A dynamically added function may not necessarily be malicious, but it may perform unintended operations that compromise the safety of the system. Since the execution of a function, in general, may be dependent upon the environment it executes in, it may not be possible to exhaustively test it for safe operation. Thus in general, a ”trusted function” may not be safe unless there are restrictions on what kind of computation the function may perform.

It is possible to write functions in a restricted programming language that guarantees safe execution. For certain kind of functions it may even be possible to statically determine safety, even though they are written in an unrestricted programming language. However, our interest is in an approach which does not restrict the expressiveness of the language in which these functions are written. Our experiences in writing two router applications that involved TCP congestion control and splice mechanisms [4] [3] show that router application code can be of significant complexity. We feel that writing these applications in a restricted language would have been substantially more complex, and perhaps suboptimal in performance. The key goal then becomes to provide safety efficiently for unrestricted router applications.

Efficient memory protection can be provided by utilizing the low-level hardware protection features of the router's processor architecture. Most general-purpose processors provide hardware primitives for protection, where all associated checks are embedded in the micro-architecture and thus do not incur any extra overhead. These primitives, when exploited at the lowest level, can provide efficiency as well as hard protection guarantees. We have been able to implement efficient protection domains in a router OS by utilizing the segmentation hardware of the X86 architecture [5]. The protection subsystem of our router OS exposes the segmentation hardware at a low enough level that router applications can use it easily, while keeping invocation overheads close to that of a protected function call in hardware. Similar approaches have been tried with other architectures as well [6]. In general, by tuning its protection subsystem implementation to the processor architecture, a router OS can provide efficient as well as strong memory protection without compromising expressiveness of router application code. In spirit, this design principle is similar to that of Exokernels [7], which would argue for exposing hardware protection features to the application.

3 Performance Protection

We distinguish between flows that are bound to some router application, called *application flows*, and flows that are processed by the router's standard forwarding code, called *generic flows*. We call generic flow processing and control plane processing as the router's *core tasks*. The goal of performance protection is to protect the performance seen by the router's core tasks in the presence of application flows. Performance protection limits the scope of the impact that dynamically added computation has on flows going through the router : application flows perceive the end-to-end effects of placing computation in the router (as desired), while the presence of this computation is transparent to generic flows.

Performance protection has two implications on a router OS. Firstly, core router tasks must be bound to an appropriate *core scheduling context*. This makes core task processing explicit in the scheduler, allowing it to deliver the appropriate performance guarantee. Secondly, the scheduling policy for the core tasks must be chosen, which may be prioritization or sharing. Recent studies using WAN traffic traces [8] and inter-domain routing message traces [9] show that traffic patterns and control plane processing load in Internet routers is difficult to characterize accurately, and might be bursty. Thus core tasks may need high short-term processing bandwidth, even though long-term requirements may be small. Thus, to provide true isolation to these tasks, prioritization is the appropriate scheduling primitive. Prioritization ensures that in a programmable router, the processing demands of core tasks will be handled with zero latency in the presence of router applications[1]. For generic flows and control processing, this essentially simulates a router in which there were no applications running. Moreover, if application flows are scheduled among themselves using a proportional share scheduler, they will adapt gracefully to short-term reduction in available resource bandwidth (system virtual time will not advance while the prioritized task is being run).

4 Event-Driven Control Flow

An important characteristic of many useful router applications is the use of functions that carry state across invocations. Protocol stacks are one example, where every ”layer” is a stateful function. Similarly, any router application that exploits global context across flows must use stateful functions. Typically, a single stateful function would be used by many flows. Similarly, a single flow would use several stateful functions that act like a ”processing pipeline” for the flow. This model localizes state in the functions, and carries a flow's invocations from function to function. This is in contrast with the ”thread” model, where it is expected that a thread executing on behalf of the flow has access to all the state. If different functions are in different protection domains (because some functions are privileged, or installed by mutually untrusted authorities), the thread approach must either resort to state sharing through an interface (since it cannot directly read/write it), or there should be a mechanism for a thread to ”cross” protection domains. The latter essentially takes the form of an explicit invocation, as proposed in [10] through descriptor passing.

Thus, we argue that the computation and composition model for router applications should be event-based, as opposed to thread-based. Besides providing a closer match to a computation model that uses stateful shared functions, a key advantage of an event-based model is that all invocations are *explicit* and *asynchronous*. Since invocations carry the identity of the resource principal making the invocation, the resource principal associated with a piece of work is always explicitly known throughout the system. This gives the scheduler complete knowledge of pending work in the system for each resource principal, and allows it to schedule work correctly. Further, by being asynchronous, every invocation acts as an instant when the scheduler gets control, leading to tighter resource control than that allowed in

[1]Modulo non-preemption. See section 4.

a constant time-slice based scheduler. Note that at every scheduling instant, the scheduler can look for invocations made in the core scheduling context (section 3), and prioritize them.

5 Integrated Resource Scheduling

An application flow requires CPU cycles as well as link bandwidth from the router to meet its performance requirement. However, the router application can only specify a flow's requirement in terms of the amount of work required from each resource, and a single, global deadline (or rate) requirement. For example, for each packet of a flow, it may specify the CPU cycles required, the packet size in bytes, and a single deadline for the packet to get serviced. The application does not specify how deadlines should be allocated in the CPU and the link. This task is best done by the router OS that should figure out how to best deliver the overall deadline by allocating a per-resource deadline. We call this router OS function *integrated resource scheduling*.

We generalize integrated resource scheduling in terms of deadlines, since rate requirements can be mapped to deadlines. Thus, we assume that a flow asks for a deadline (d) for each of its packets, and specifies the amount of work required from the CPU (W_C) and the link (W_L). The goal of integrated resource scheduling is to split d into d_C, a deadline for the CPU, and d_L, a deadline for the link, according to an optimization criterion. We briefly describe a deadline allocation algorithm here. At any time, there are a set of requests admitted into the system, corresponding to a set of reservations in each resource. If a resource has capacity C and has admitted a set of requests where request i needs work W_i and has been allocated a deadline d_i, then the residual capacity of the resource is $R = C - \sum_i W_i/d_i$. When a new request comes for this resource, asking for an amount of work W, its minimal service time in this resource is $d_{min} = W/R$. If the sum of the quantity d_{min} for every resource is less than or equal to the global deadline of the task, then the request is admissible. However, if the sum is less than the global deadline, these deadlines can be *relaxed* such that each resource has some spare capacity left (Note that allocating a deadline of d_{min} in a resource corresponds to using up *all* the residual capacity of that resource). It is in this relaxation step that the system-wide optimization criterion comes in. For instance, if the optimization goal is to keep all resources equally utilized, so that the system keeps spare capacity uniformly available across all resources, then the deadline allocated in the heavily utilized resource would be relaxed more. Such a mechanism should be an integral part of a router OS in order to achieve tight admission control for application flows.

6 Binding Resources to Flows

Typically router resources would be shared by a large number of application flows, which calls for appropriate resource arbitration. Moreover, many router applications would typically operate upon a *set* of flows belonging to a type of network application, as opposed to operating on single flows. In such cases, the router application would typically have an aggregate, as opposed to per-flow, performance requirement. This makes the task of accurately binding router resources to flows an important one. The *expressiveness* of the resource reservation interface determines how accurately router applications will be able to express their resource requirements. An inflexible interface may lead to coarse specifications, leading to underutilization of router resources. Likewise, an overly flexible interface may blow up the scheduling state in the system, while being a burden to a router application writer.

We believe that two key principles suffice to provide a simple and flexible interface.

1. Decouple execution contexts from scheduling contexts: This means that the interface should clearly distinguish between a thread of control associated with a flow, and the resource principal associated with it. Thus, an invocation made in the context of a flow should have two components : the identity of the flow, and the identity of the resource principal, which may be different in general.

2. Allow absolute as well as *symbolic* specification of resource reservations : A symbolic specification means a reference to another principal's resource reservation. Thus, a flow f_1 may specify that it requires an overall rate of 100 packets/sec with each packet having 64 bytes (independent link reservation of 6400 bytes/sec), but shares the CPU with flow f_2 (symbolic CPU reservation).

These principles have two important implications. First, binding a flow to a resource principal now becomes an explicit operation. Second, resources can be shared on a per-resource basis, as opposed to an all-or-none basis (where either both CPU and link resources are shared, or none is shared). An example application where this is needed is a multicast application that transcodes incoming data on a link and distributes it over three output links. Each output flow requires its own context in its output link, and may even have distinct link rate requirements due to receiver heterogeneity. However, the transcoding operation is done once on a single copy of every packet, and hence the CPU reservation should be shared. This application can be implemented using three flows that share their CPU resource. One of the flows can make the absolute CPU reservation, and the other two can refer to this reservation symbolically. A cursory look at the example might say that the same could be done by declaring one incoming flow that only reserves the CPU, and three outgoing flows reserving only the respective link rates. However, this would break the integrated CPU and link scheduling requirement described in section 5.

7 *Srishti* and Aggregate TCP

The ideas presented above have been incorporated in *Srishti*, a substrate for writing applications in a router that uses the X86 architecture for application flow processing. Using the above design principles, Srishti allows composition of router applications through stateful functions and flows. The functions are untrusted, preemptible functions that can be efficiently co-located with core router functions in a single address space. Flows are execution contexts, bound explicitly to resource principals using Srishti's API. All control transfer is explicit and asynchronous, and functions are called through references. These references are obtained by a *naming* service that acts like a dynamic symbol table of loaded functions.

We briefly share our experience in writing a router application over Srishti to perform aggregated TCP congestion control. TCP does not provide mechanisms to allow a new connection to reuse congestion estimates gathered by other connections that have used the same path. This forces a new TCP connection to always start from a conservative estimate of available bandwidth, causing short connections to never reach the correct value of the available bandwidth. Short HTTP connections can perform at significantly sub-optimal performance levels due to this, if there is a lot of opportunity for temporal and spatial congestion state reuse. ATCP is a router mechanism that allows congestion state reuse between TCP connections that are expected to share bottleneck links in the network[2], without changing end-system TCP implementations. The details on how ATCP implements its functionality and its evaluation on a real-world HTTP trace are available in [3].

While composing ATCP using Srishti's API, the most interesting choice is in how resources are to be allocated, and the *scope* of congestion state sharing. The scenario we envision is that ATCP is deployed in a router that serves a certain number of busy TCP servers, say from N different organizations. In this case, the scope of congestion state sharing is all TCP flows originating at these N servers, and the resource allocation goal is to be max-min fair to these N organizations. Thus the ATCP implementation uses N resource principals, to which incoming TCP packets are bound. One can choose to implement ATCP as a monolithic function that holds per-flow state ; or as K modules where there are K congestion sharing groups, each of which holds per-flow state only for the flows in that group. We have currently implemented ATCP as a monolithic function. Only one execution context is used, since all session state is centralized in one function. Since there are no blocking calls in the application, there is no need for multiple execution contexts to hide blocking latency.

[2]ATCP approximates this by grouping together flows destined to the same subnet.

8 Evaluation

We have implemented Srishti on a 400 MHz Pentium and tested it as a router with Intel eepro100 network interfaces. While the implementation uses a Linux skeleton, it depends more directly on the X86 architecture rather than on Linux[3]. In this section, we provide some microbenchmarks on the system that give some insight into the design decisions laid out earlier.

We begin with some microbenchmarks related to protection. A null router application function, co-located with the router kernel in a lesser privileged segment, incurs an overhead of 325 cycles for a call and return. When the function also makes a protected function call to a core router function before returning, the overhead becomes 814 cycles. This is more than twice of the single call, due to additional overheads of saving all general-purpose registers. To see the advantages of co-location, we ran a ping-pong test between two null functions in different address spaces, incurring on overhead of 1360 cycles per call. This overhead would be higher in general for a non-null function, due to the cost of re-populating flushed TLB entries with every address space switch.

The next measurement shows the role of event-driven control flow in providing fine-grained prioritization to the router's generic flows. We modified the eepro100 driver to use polling instead of interrupts. Thus, interrupt context is not used to process generic flows. We tried three ways in which the scheduler could get control in order to serve generic flows. In the first case, the scheduler only gets control at system timer interrupts (10 msec). This simulates a time-slice based scheduler, and a system that uses synchronous function invocations. The second case gives control to the scheduler only when application functions return, simulating asynchronous control flow, but with no timers. In the last case, the scheduler gets control every time a function returns or the timer fires, representing the finest scheduling granularity. The system continuously runs invocations whose running time is uniformly distributed from 3 msec to 21 msec in increments of 3, centered roughly around 10 msec. The router is fed with a uniform stream of packets with varying inter-packet gap. The metric

[3]Code for Srishti and ATCP is available via anonymous FTP from sequoia.ecsl.cs.sunysb.edu/pub/srishti.

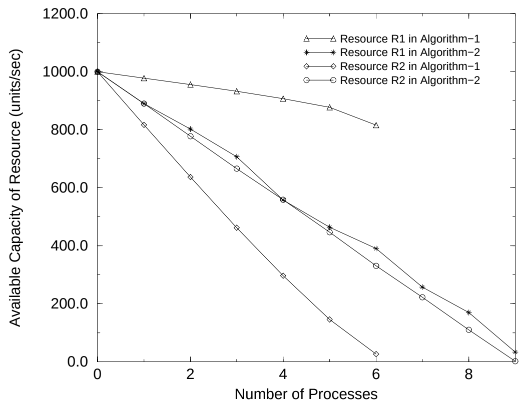


Figure 1. *Utilization of two resources v/s admitted flows for fixed allocation (algorithm 1) and load-dependent allocation (algorithm 2).*

that reflects the "disturbance" introduced in the forwarding path of generic flows is the standard deviation of the inter-arrival time at the receiver. As shown in table 1, the event-driven approach leads to lesser perturbation than a constant time-slice based scheduler, and the scheduler that combines events with time slices performs the best.

Sender Inter Pkt Gap	Timer Interrupt	Function Return	Timer OR Fn. Return
1.0	1.746	1.821	1.573
4.0	4.383	3.783	3.069
7.0	4.312	3.832	3.082
12.0	3.881	3.658	2.844

Table 1. *Standard Deviation (in msec) of received inter-packet gap for three ways in which the scheduler can get control from router applications.*

The final measurement shows the impact of integrated resource scheduling in providing tighter admission control. We assume two resources, each with a capacity of 1000 units/sec. Flows request these resources by asking for a rate of 1 packet per 100 msec, requesting 1 unit of work from one resource and 10 units from the other. Figure 1 shows how resource usage for each resource changes as flows are added, for the case when deadline slack is allocated in a fixed manner (algorithm 1), and when it is allocated in a load-dependent manner (algorithm 2). Algorithm 1 allcoates half the global deadline to each resource, leading to skewed utilization and stops admission control sooner than algorithm 2. Algorithm 2 tries to keep resource utilization balanced by allocating more slack to the more loaded resource.

9 Related Work

Recent interest in providing system support for router programmability has led to the specification of the NodeOS interface [11] which attempts to lay down implementation-independent primitives that a programmable router should provide. NodeOS *implementations* internally implement these primitives using substrates like language runtimes or specialized OSes [12], and expose the NodeOS interface to router applications. Placing our work in this context, the requirements we identify pertain to NodeOS implementations. In other words, we expose some of the design decisions which are hidden beneath the NodeOS interface, but are important in making router programming a practical paradigm. Some of the requirements that we propose are generic in the sense that they can be incorporated in existing implementations. For example, the requirements pertaining to resource scheduling can be incorporated into any framework that already supports scheduling, whereas efficient memory protection primitives can be utilized to sandbox non-protected approaches such as router plugins [13]. Stateful computation may not be directly applicable in some systems, for example those based upon functional languages. However fine-grained scheduling afforded by event-driven control flow can be supported in language runtimes by giving control to the language runtime upon a method invocation.

References

[1] Papadopoulos C., *et al*, An Error-Control Scheme for Large-Scale Reliable Multicast Applications, Proc. IEEE Infocom 1998.

[2] Amir E., *et al*, An Applical Level Video Gateway, Proc. ACM Multimedia 1995.

[3] Pradhan P., *et al*, Aggregate TCP Congestion Control Using Multiple Network Probing, Proc. ICDCS 2000.

[4] Apostolopoulos G., *et al*, Design, Implementation and Performance of a Content-Based Switch. Proc. IEEE Infocom 2000.

[5] Chiueh T., *et al*, Integrating Segmentation and Paging Protection for Safe, Transparent and Efficient Software Extensions, Proc. ACM SOSP 1999.

[6] Takahashi M., *et al*, Efficient Kernel Support for Fine-Grained Protection Domains for Mobile Code, Proc. ICDCS 1998.

[7] Kaashoek F., *et al*, Application Performance and Flexibility on Exokernel Systems, Proc. ACM SOSP 1997.

[8] Feldmann A., *et al*, Data Networks as Cascades : Explaining the Multifractal Nature of Internet WAN Traffic Proc. ACM SIGCOMM 1998.

[9] Malan R., *at al*, Internet Routing Instability, Proc. ACM SIGCOMM 1997.

[10] Banga G., *et al*, Resource Containers : A New Facility for Resource Management in Server Clusters, Proc. USENIX 1999.

[11] Peterson L., *et al*, NodeOS Interface Specification.

[12] Peterson L., *et al*, A NodeOS Interface for Active Networks, In IEEE JSAC 2001.

[13] Decasper D.*et al*, Router Plugins : A Software Architecture for Next-Generation Routers, Proc. ACM SIGCOMM 1998.

Lazy Process Switching

Jochen Liedtke Horst Wenske

University of Karlsruhe, Germany

liedtke@ira.uka.de

1 Motivation

Although IPC has become really fast it is still too slow on certain processors. Two examples motivating even faster IPC, critical sections in real-time applications and multi-threaded servers, are briefly discussed below.

Critical sections in real-time applications suffer from the well-known priority-inversion problem [7]. Multiple solutions have been proposed, e.g. priority inheritance (which is generally not sufficient), priority ceiling [7], and stack-based priority ceiling [2]. All methods need to modify a thread's priority while the thread executes the critical section. In the stack-based priority-ceiling protocol, for example, a thread has to execute the critical section always with the maximum priority of all threads that might eventually execute the critical section, regardless of its original priority.

A very natural solution for stack-based priority ceiling in a thread/IPC-based system is to have a dedicated thread per critical section. This thread's priority is set to the (static) ceiling priority. Any "client" thread calls the critical section through RPC (two IPCs). Priorities are automatically updated through the undelying thread switch. The synchronous IPC mechanism also serializes threads automatically that compete for the critical section. Provided that simultaneously pending request IPCs are delivered in prioritized order, we have a simple and elegant implementation of stack-based priority ceiling.

However, this method of implementing critical section requires very lightweight threads. In particular, IPC should be very fast. 180 cycles which is the current L4 time on a Pentium III is too expensive. Such costs are acceptable when real synchronization actions are necessary such as entering the invoker into a wait queue if the critical-section thread is blocked on a page fault. However, typically a critical-section thread can be called directly. 180 cycles are inacceptable in this case. *Therefore, we need much faster IPC!*

For achieving highest performance, *multi-threaded servers* often need *customized* policies how to distribute incoming requests to worker threads. For instance, a server might want to handle up to 3 requests in parallel but queue further requests. The natural solution is one distributor thread which also implements a request queue and 3 worker threads that communicate through IPC with the distributor. Again, 180 cycles are inacceptable. *Therefore, we need much faster IPC!*

In general, we see that the availability of fast IPC lets people think about fine-grain system componentization. Once they are on this path they ask for mechanisms that enable even more fine-grain componentization, in particular infinitely fast IPC. *Therefore, we need much faster IPC!*

2 Is User-Level IPC Possible?

Nicely, we seem to need superfast IPC particularly for intra-task communication which does not include an address-space switch. User-level threads which are no kernel objects [1, 6, 5] might achieve the required speed. Since a tasks's user-level threads are unknown objects for the kernel and execute all in the context of a single kernel thread user-level-thread switchs are invisible to the kernel and can entirely execute in user mode. However, the overhead required to make user-level threads kernel schedulable [1] more than compensates the above speed gain in a system that offers sufficiently lightweight threads and fast IPC. From our previous experience, we are convinced that the total costs of user-level threads in terms of time and total system complexity are much higher than their gain. Furthermore, having two concepts, kernel threads and user-level threads, is conceptually inelegant and contradicts the idea of conceptual minimality.

Therefore, our goal is to find an implementation of kernel threads that offers all speed advantages of user-level threads for intra-task communication.

Let us revisit how an intra-address-space thread switch happens. We assume an atomic *SendAndWaitForReply* IPC which is typically used for RPC. Client and server variant of this call differ only marginally. The client thread sends a request to a server thread and waits for a reply from that server. Correspondingly, the server thread replies to the client thread and waits for the next request which may arrive from any client. We show the client variant:

```
$A \to B$ :
  call IPC function, i.e. push A's instruction pointer ;
  **if** B is a valid thread id AND thread B waits for thread A
      **then** save A's stack pointer ;
           set A's status to "wait for B" ;
           set B's status to "run"
           load B's stack pointer ;
           current thread := B ;
           return, i.e. pop B's instruction pointer
      **else**
           more complicated IPC handling
  **endif** .
```

There are two reasons why to be execute this code in kernel mode:

1. *Atomicity.* Checking B's state and the following thread switch have to be executed atomically to avoid inconsistencies.

2. *Kernel Data.* Stack pointer, thread status, and "current thread" are protected data that can only be accessed by the kernel to prevent user-level code from compromising the system.

On processors with relatively expensive kernel/user-mode-switch operations such as x86, the above two reasons increase IPC costs from 20– cycles to 180 cycles (Pentium III, using systenter/sysexit instructions). Therefore, we should find a way to invalidate both reasons, i.e. to execute the above IPC operation entirely in user mode.

2.1 Atomicity

Ensuring atomicity in user mode is relatively simple as long as the kernel knows the executed code. The method goes back to an idea that Brian Ford [3] proposed in 1995: Let some unmodifiable "kernel code" execute in user space so that the kernel can act specifically to this code if an interruption within this "kernel code" occurs.

In our example, the kernel would simply reset the thread's instruction pointer to the beginning of the IPC routine if an interruption occurs before a real status modification has become effective. After the system state has been partially modified, the kernel would have to either undo those modifications or complete the IPC operation before handling the interruption. Such a method cannot ensure atomicity in general; e.g., it fails if the questionable code experiences a page fault. However, we can easily implement the IPC code such that the described forward-completion method works:

```
$A \to B$ :
    call IPC function, i.e. push A's instruction pointer ;
    save A's stack pointer ;
    — restart point —
    if B is a valid thread id AND thread B waits for thread A
        then  — forward point —
              set A's status to "wait for B" ;
              set B's status to "run"
              load B's stack pointer ;
              current thread := B ;
              — completion point —
              return, i.e. pop B's instruction pointer
        else  . . .
```

Interruptions including page faults between restart point and forward point occur before the system's state has really changed. Provided that no required registers have been overwritten, resetting to the restart point heals the interruption:

```
interruption between restart point and forward point:
    set interrupted instruction pointer to restart point .
```

The algorithm is robust against page faults[1] upon accessing thread-control blocks (TCBs): If a page fault occurs when TCB B is accessed to check B's status the IPC operation simply restarts after page-fault handling. We assume that *after* the forward point, no legal page faults can occur since both TCBs have been accessed in the check phase. However, illegal page faults might occur, e.g. if a user program jumps directly to the middle of the code or even to the middle of an instruction. Consequently, any page fault in this region is illegal and permits to kill the thread.

```
interruption between restart point and complete point:
    if is page fault
        then  kill thread A
        else  A's status := "wait for B" ;
              B's status := "run" ;
              load B's stack pointer ;
              current thread := B ;
              set interrupted instruction pointer to completion point
    endif .
```

On a uniprocessor, we have thus guaranteed atomicity without using privileged instructions. For multiprocessors, the method can be extended to work for threads residing on the same processor. (Cross-processor communication is anyhow an order of magnitude more expensive than intra-processor communication. Restricting user-level IPC to intra-processor is thus acceptable.)

[1]Some systems might hold TCBs in virtual memory.

2.2 Kernel Data

The kernel data involved are A-TCB and B-TCB variables *stack pointer, status* and the system variable *current thread*. We have to analyze whether these variables must be really protected from unautorized user access.

For the time being, assume that the above mentioned IPC code runs in user mode. Then, the TCB variable *stack pointer* holds a thread's *user* stack pointer. Remember that A and B both run in the same address space so that they can arbitrarily modify each other stacks and perhaps even code. Protection would therefore not be significantly better if A's stack pointer would be protected against access from B. *Consequently, the TCB variable* stack pointer *can be user accessible.*

The *status* case is a little more complicated. Assume that a thread's status can only be "run" or "wait for X". We have to analyze three cases when thread A maliciously switches thread B's status: from "run"[2] to "wait for X", from "wait for X" to "wait for Y", and from "wait for X" to "run".

Whenever A modifies B's status illegaly we see user-level effects and system effects. User-level effects within A's address space can be ignored (see above). Effects in different address space that indirectly result from user-level effects within A's address space are also irrelevant since A has full access to their data even without modifying the thread states. As long as only thread states within the same task are accessible, user-level effects are thus uncritical.

System effects are more serious. Whenever the system state depends on a thread's *status* variable we need provisions ensuring system integrity. Unauthorized modification of a *status* variable must in no case lead to system inconsistencies. For instance, the kernel can no longer assume that a thread of *status* "run" is always in the run queue. Similarly, a thread might be in the run queue although its *status* says "wait for X". This run-queue problem can, e.g., be solved by the lazy-scheduling technique [4] where the run queue is updated lazily.

A more generally applicable technique is based on the idea to have a *kernel twin* for each unprotected user-accessible kernel variable. Before the unprotected variable is used by the kernel, the kernel always checks consistency. If unprotected variable and kernel twin do not match the kernel takes appropriate actions to reestablish consistency. The fundamental problem is to determine whether the recognized inconsistency is legal or not. If it is legal the unprotected variable is used to update the protected kernel state. If it is illegal the unprotected variable can be reconstructed based on its kernel twin or the current thread can be killed.

For example, we could have an unprotected $status_u$ variable in user space and a protected kernel twin, $status_k$ in kernel space per thread. Whenever the kernel detects $status_u \neq status_k$ it will reestablish consistency by:

[2]On this level of abstraction, "run" is used to denote a ready-to-run thread as well as a thread that currently executes on a processor.

```
status inconsistency:
    if status$_u$ = "run" AND status$_k$ is wait for
        then  insert thread into run queue ;
              status$_k$ := status$_u$
    elif status$_u$ is wait for AND status$_k$ = "run"
        then  delete thread from run queue
              status$_k$ := status$_u$
        else  kill thread
    endif .
```

The algorithm can be straightforwardly extended to handle more thread states than only "run" and "wait for X". Ignoring performance questions and potential complications due to dependencies between multiple kernel objects, we can conclude that, in principle, *some kernel data can be made user-mode accessible.*

3 Lazy Switching

The fundamental insight is that twin inconsistencies need only to be checked on kernel entry. This sounds trivial. However, its immediate consequence is that an IPC executing completely in user level *does not need to synchronize with the kernel.*

In particular, this type of IPC can switch threads without directly telling the kernel. The kernel will synchronize, i.e. execute the thread switch in retrospect upon the next kernel entry, e.g. timer tick, device interrupt, cross-address-space IPC, or page fault.

In general, lazily-evaluated operations pay if more of them occur than have to be evaluated effectively. Correspondingly, lazy switching can pay if only a samll fraction of lazy-switching operations lead finally to real kernel-level process switches. Such behavior can be expected whenever a second IPC, for example the reply or a forwarding IPC, happens before an interrupt occurs. Our motivating examples "critical region" and "request distribution" fall into this category provided their real work phase is short.

3.1 UTCBs and KTCBs

Now let us try to apply the insights of the previous section to the concrete problem:

1. The IPC system-call code is mapped to a fixed address in user address space and can be executed in user mode; atomicity is guaranteed as described in Section 2.1.

2. We separate each thread's TCB into a UTCB and a KTCB. The UTCB is unprotected and user accessible. The KTCB can only be accessed by the kernel. A thread's UTCB holds its *user stack pointer* and its *status*$_u$. *Status*$_k$ is in the KTCB. Furthermore, the UTCB holds the KTCB address which is of course not trustworthy. However, the KTCB holds a backpointer to its corresponding UTCB so that the UTCB's KTCB pointer can be validated (see algorithm below).

3. An unprotected kernel variable *CurrentUTCB*$_u$ can be accessed from user mode. It is intended to point to the current thread's UTCB. Its protected twin *CurrentUTCB*$_k$ lives in kernel space.

The only variable that triggers synchronization is *CurrentUTCB.* Inconsistencies that include only *status* are ignored because they are always illegal. Due to lazy scheduling [4], *status* inconsistencies can be tolerated.

```
CurrentUTCB inconsistency:
    if CurrentUTCB$_u$ is in valid utcb region
        then  NewKTCB := CurrentUTCB$_u$–>ktcb ;
              if NewKTCB is in valid ktcb region and aligned
                  AND NewKTCB–>utcb = CurrentUTCB$_u$
                  then  switch from CurrentKTCB to NewKTCB ;
                        CurrentKTCB := NewKTCB ;
                        CurrentUTCB$_k$ := CurrentUTCB$_u$ ;
                        return
              endif
    endif ;
    kill thread (CurrentKTCB) .
```

3.2 Coprocessor Synchronization

Most modern processors permit to handle floating-point registers and those of other coprocessors lazily. Those resources can be locked by the kernel. If another thread tries to access them an exception is raised that permits the kernel to save the coprocessor register in that TCB which has used the coprocessor so far and reload the registers from the current TCB. Typically, coprocessors can only be locked by kernel-mode software.

Therefore, we have to extend the above CurrentUTCB-synchronization algorithm to make it coprocessor safe.

We introduce a pair of flags *CoprocessorUsed*$_{u/k}$. Both flags are set by the kernel whenever it allocates the coprocessor to the current thread. If *CoprocessorUsed*$_k$ is set the kernel locks the coprocessor when switching from this thread to another one and resets both flag twins. The user-level IPC code now checks whether *CoprocessorUsed*$_u$ is not set. If it is set user-level IPC is not possible and a full kernel IPC is invoked.

Of course, *Coprocessor*$_u$ is not trustworthy. Therefore, we might see an invalid coprocessor flag when switching through user-level code from A to B. A potential coprocessor confusion between A, B, and other threads of the same task can be ignored. However, we must ensure that the information "one of the current task's threads has currently allocated the coprocessor" never gets lost. Otherwise, the coprocessor confusion could infect threads of other tasks. Fortunately, this can be done lazily, i.e. needs only to be checked when a an UTCB inconsistency is handled:

```
Switch from CurrentKTCB to NewKTCB:
    . . .
    NewKTCB–>CoprocessorUsed := CurrentKTCB–>CoprocessorUsed ;
    . . .
```

Remember that user-level IPC never legally switches away from a thread that currently uses the coprocessor. As long as all lazy switches have been legal, the above statement copies therefor always a 0-flag. However, as soon as we have a coprocessor confusion through an illegally reset *CoprocessorUsed*$_u$, it copies a 1-flag and propagates the coprocessor confusion to the new thread. If later a kernel IPC or other kernel-level thread switch switches to another task the coprocessor is deallocated so that the coprocessor confusion can not infect the other task.

4 Prototype Performance

The current prototype takes 12 cycles for the fast IPC path on a Pentium III. Slight increases have to be expected when integrating it into a fully-functional L4 version 4 microkernel.

5 Conceptual Summary

Lazy switching enables very fast blocking intra-task IPC between kernel-implemented threads. This type of IPC can typically be entirely executed in user mode although it operates on kernel objects. We hope that lazy switching adds the advantages of user-level threads to kernel-level threads.

The work on lazy switching is ongoing research in its early stage. Whether all its promising properties can make it to reality is still open. Further open questions:

1. Can we include the structural modifications required for lazy switching into an existing microkernel at almost no cost?

2. Processors with low kernel/user-switch costs such as Alpha obviously do not require lazy switching. Can we find an API that permits lazy switching on x86 without impose additional costs on an Alpha implementation?

3. Can we extend lazy switching to certain cross-address-space process switches?

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *13th ACM Symposium on Operating System Principles (SOSP)*, Pacific Grove, CA, October 1991.

[2] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *11th Real-Time Systems Symposium (RTSS)*. IEEE, December 1990.

[3] B. A. Ford. private communication, December 1995.

[4] J. Liedtke. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, December 1993.

[5] F. Mueller. A library implementation of POSIX threads under UNIX. In *Winter USENIX Technical Conference*, page 29, January 1993.

[6] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multithreaded architecture. In *Winter USENIX Technical Conference*, page 65, El Cerrito, CA, January 1991.

[7] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

Robustness in Complex Systems

Steven D. Gribble
Department of Computer Science and Engineering
The University of Washington
gribble@cs.washington.edu

Abstract

This paper argues that a common design paradigm for systems is fundamentally flawed, resulting in unstable, unpredictable behavior as the complexity of the system grows. In this flawed paradigm, designers carefully attempt to predict the operating environment and failure modes of the system in order to design its basic operational mechanisms. However, as a system grows in complexity, the diffuse coupling between the components in the system inevitably leads to the butterfly effect, in which small perturbations can result in large changes in behavior. We explore this in the context of distributed data structures, a scalable, cluster-based storage server. We then consider a number of design techniques that help a system to be robust in the face of the unexpected, including overprovisioning, admission control, introspection, adaptivity through closed control loops. Ultimately, however, all complex systems eventually must contend with the unpredictable. Because of this, we believe systems should be designed to cope with failure gracefully.

1. Introduction

As the world grows more dependent on complex computing systems (such as scalable web sites, or even the Internet itself), it has become increasingly evident that these systems can exhibit unpredictable behavior when faced with unexpected perturbations to their operating environment. Such perturbations can be small and innocuous, but due to latent flaws in the design of the system combined with widespread coupling between its components, the effects of small perturbations may be large and destructive, possibly rendering the system inoperative.

For example, in [5], Floyd and Jacobson demonstrated that periodic signals (such as router broadcasts) in the Internet tend to become abruptly synchronized, leading to patterns of loss and delays. As another example, in [4], Druschel and Banga demonstrate that with web servers running on traditional interrupt-driven operating systems, a slight increase in load beyond the capacity of the server can drive the server into a persistent state of livelock, drastically reducing its effective throughput. As a third example, in [1], Arpaci-Dusseau et al. demonstrate that with conventional software architectures, the difference in performance resulting from placing data on the inner tracks vs. outer tracks of a single disk can affect the global throughput of an eight node cluster of workstations by up to 50%. A final example is that of BGP "route flap storms" [11, 12]: under conditions of heavy routing instability, the failure of a single router can instigate a storm of pathological routing oscillations. According to [11], there have been cases of flap storms that have caused extended Internet outages for millions of network customers.

By their very nature, large systems operate through the complex interaction of many components. This interaction leads to a pervasive coupling of the elements of the system; this coupling may be strong (e.g., packets sent between adjacent routers in a network) or subtle (e.g., synchronization of routing advertisements across a wide area network). A well-known implication of coupling in complex systems is the butterfly effect [14]: a small perturbation to the system can result in global change.

Avoiding Fragility

A common goal that designers of complex systems strive for is robustness. Robustness is the ability of a system to continue to operate correctly across a wide range of operational conditions, and to fail gracefully outside of that range. In this paper, we argue against a seemingly common design paradigm that attempts to achieve robustness by predicting the conditions in which a system will operate, and then carefully architecting the system to operate well in those (and only those) conditions. We claim that this design technique is akin to *precognition*: attempting to gain knowledge of something in advance of its actual occurrence.

As argued above, it is exceedingly difficulty to completely understand all of the interactions in a complex system *a priori*. It is also effectively impossible to predict all of the perturbations that a system will experience as a result of changes in environmental conditions, such as hardware failures, load bursts, or the introduction of misbehaving software. Given this, **we believe that any system that attempts to gain robustness solely through precognition is prone to fragility**.

In the rest of this paper, we expore this hypothesis by presenting our experiences from building a large, complex cluster-based storage system. We show that although the system behaved correctly when operating within its design assumptions, small perturbations sometimes led to the violation of these assumptions, which in turn lead to system-wide failure. We then describe several design techniques that can help systems to avoid this fragility. All of these techniques have existed in some form in previous systems, but our goal in this paper is to consolidate these techniques as a first step towards the design of more robust systems.

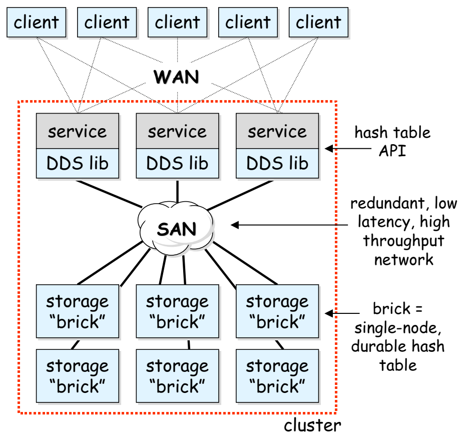


Figure 1. DDS architecture: each box in the diagram represents a software process. In the simplest case, each process runs on its own physical machine in a cluster, however there is nothing preventing processes from sharing physical machines.

2. DDS: A Case Study

In [7], we presented the design and implementation of a scalable, cluster-based storage system called a *distributed data structure (DDS)*. A DDS, shown in Figure 1, is a high-capacity, high-throughput virtual hash table that is partitioned and replicated across many individual storage nodes called bricks. DDS clients (typically Internet services such as web servers) invoke operations on it through a library that acts as a two-phase commit coordinator across replicas affected by the operation. These two-phase commits are used to achieve atomicity of all operations and one-copy equivalence across the entire cluster.

The design philosophy we used while building the DDS was to choose a carefully selected set of reasonable operational assumptions, and then to build a suite of mechanisms and an architecture that would perform robustly, scalably, and efficiently given our assumptions. Our design strategy was essentially predictive: based on extensive experience with such systems, we attempted to reason about the behavior of the software components, algorithms, protocols, and hardware elements of the system, as well as the workloads it would receive. In other words, we largely relied on precognition while designing mechanisms and selecting operating assumptions to gain robustness in our system.

Within the scope of our assumptions, the DDS design proved to be very successful. We were able to scale the number of nodes in the system across two orders of magnitude, and we observed a corresponding linear scaling in performance. We also demonstrated fault-tolerance by deliberately inducing faults in the system and showing that the storage remained available and consistent. However, as we operated the system for a period of more than a year, we observed several very unexpected performance and behavioral anomalies. In all cases, the anomalies arose because of an unforeseen perturbation to the system that resulted in the violation of one of our operating assumptions; the consequences of these violations were usually severe.

In this section of the paper, we describe several of the more interested and unexpected behavioral anomalies that we encountered over the one or two years' worth of experience we had with this system. Some may choose to consider these anomalies simply as bugs in the design of the system, arising from lack of foresight or naïvety on the part of its designers. We argue, however, that these "bugs" all shared similar properties: they were extremely hard to predict, they arose from subtle interactions between many components or layers in the system, and they bugs led to severe implications in our system (specifically, the violation of several operating assumptions which in turn led to system unavailability or data loss).

2.1. Garbage Collection Thrashing and Bounded Synchrony

Various pieces in the DDS relied on timeouts to detect the failure of remote components. For example, the two-phase commit coordinators used timeouts to identify the deaths of subordinates. Because of the low-latency (10-100 μs), redundant network in the cluster, we chose to set our timeout values to several seconds, which is four orders of magnitude higher than the common case round trip time of messages in the system. We then assumed that components that didn't respond within this timeout had failed: we assumed *bounded synchrony*.

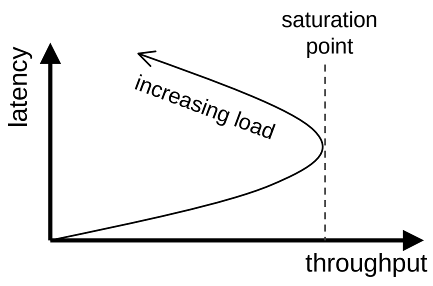


Figure 2. **Performance with GC thrashing:** this graph depicts the (parametric) curve of latency and throughput as a function of load. As load increases, so does throughput and latency, until the system reaches a saturation point. Beyond this, additional load results in GC thrashing, and a decrease in throughput with a continued latency increase. After saturating, the system falls into a hole out of which it must "climb".

The DDS was implemented in Java, and therefore made use of garbage collection. The garbage collector in our JVM was a mark-and-sweep collector; as a result, as more active objects were resident in the JVM heap, the duration that the garbage collector would run in order to reclaim a fixed amount of memory would increase. If the DDS were operating near saturation, slight (random) fluctuations in the load received by bricks in the system would increase the pressure on their garbage collector, causing the effective throughput of these bricks to drop. Because the offered load to the system is independent of this effect, this would cause the degraded bricks to "fall behind" relative to its peers, leading to more active objects in its heap and a further degradation in performance. This catastrophe leads to a performance response of the system as shown in Figure 2.

Once the system was pushed past saturation, the catastrophe would cause the affected node to slow down until its latency exceeded the timeouts in the system. Thus, the presence of garbage collection would cause the system to violate the assumption of bounded synchrony as it approached and then exceeded saturation.

2.2. Slow Leaks and Correlated Failure

We used replication in the DDS to gain fault-tolerance: by replicating data in more than one location, we gained the ability to survive the faults of individual components. We further assumed that *failures would be independent*, and therefore the probability that multiple replicas would simultaneously fail is vanishingly small.

For the most part, this assumption was valid. We only encountered two instances of correlated failure in our DDS. The first was due to blatant, naive bugs that would cause bricks to crash; these were quickly fixed. However, the second was much more subtle. Our bricks had a latent race condition in their two-phase commit handling code that didn't affect correctness, but which had the side-effect of a causing a memory leak. Under full load, the rareness of this race condition caused memory to leak at the rate of about 10KB/minute. We configured each brick's JVM to limit its heap size to 50MB. Given this leak rate, the bricks' heaps would fill after approximately 3 days.

Whenever we launched our system, we would tend to launch all bricks at the same time. Given roughly balanced load across the system, all bricks therefore would run out of heap space at nearly the same time, several days after they were launched. We also speculated that our automatic failover mechanisms exacerbated this situation by increasing the load on a replica after a peer had failed, increase the rate at which the replica leaked memory.

We did in fact observe this correlated failure in practice: until we isolated and repaired the race condition, our bricks would fail predictably within 10-20 minutes of each other. The uniformity of the workload presented to the bricks was itself the source of coupling between them; this coupling, when combined with a slow memory leak, lead to the violation of our assumption of independent failures, which in turn caused our system to experience unavailability and partial data loss.

2.3. Unchecked Code Dependencies and Fail-Stop

As mentioned above, we used timers in order to detect failures in our system. If a timer expired, we assumed that the corresponding entity in the system had crashed; therefore, in addition to assuming bounded synchrony, we also assumed nodes would behave in a *fail-stop* manner (i.e., a node that failed to respond to one message would never again respond to any message).

To gain high performance from our system given the highly concurrent workload, we implemented our bricks using an event-driven architecture: the code was structured as a single thread executing in an event loop. To ensure the liveness of the system, we strove to ensure that all long-latency operations (such as disk I/O) were performed asynchronously. Unfortunately, we failed to notice that portions of our code that implemented a network session layer made use of blocking (synchronous) socket `connect()` routines in the Java class library. This session layer was built to attempt to automatically reinstantiate a network connection if it was broken. The main event-handling thread therefore could be surreptitiously borrowed by the session layer to forge transport connections.

On several occasions, we noticed that some of our bricks would seize inexplicably for a multiple of 15 minutes (i.e., 15 minutes, 30 minutes, 45 minutes, ...), and then resume execution, egregiously violating our fail-stop assumption. After much investigation, we traced this problem down to a coworker that was attempting to connect a machine that was behind a firewall to the cluster. The firewall was silently dropping incoming TCP syn packets, causing session layers to block inside the `connect()` routine for 15 minutes for each connection attempt made to that machine.

While this error was due to our own failure to verify the behavior of code we were using, it serves to demonstrate that the low-level interaction between independently built components can have profound implications on the overall behavior of the system. A very subtle change in behavior (a single node dropping incoming SYN packets) resulted in the violation of our fail-stop assumption across the entire cluster, which eventually lead to the corruption of data in our system.

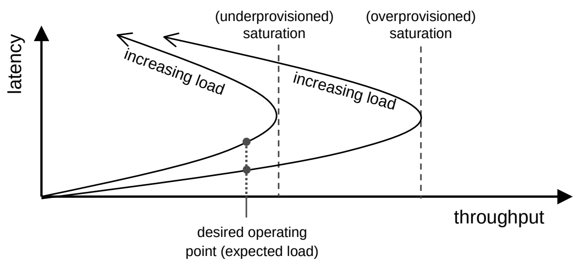


Figure 3. **An overprovisioned system:** by overprovisioning relative to the expected load, the system has slack: it can withstand unexpected bursts of load without falling into the "hole" associated with operating beyond saturation.

3. Towards Robust Complex Systems

The examples in the previous section served to illustrate a common theme: small changes to a complex, coupled system can result in large, unexpected changes in behavior, possibly taking the system outside of its designers' expected operating regime. In this section, we outline a number of design strategies that help to make systems more robust in the face of the unexpected. None of these strategies are a panacea, and in fact, some of them may add significant complexity to a system, possibly introducing more unexpected behavior. Instead, we present them with the hope of stimulating thought in the systems community for dealing with this increasingly common problem: we believe that an important focus for future systems research is building systems that can adapt to unpredictably changing environments, and that these strategies are a useful starting point for such investigation.

Systematic overprovisioning: as exemplified in Section 2.1, systems tend to become less stable when operating near or beyond the threshold of load saturation. As a system approaches this critical threshold, there is less "slack" in the system to make up for unexpected behavior: as a result, the system becomes far less forgiving (i.e., fragile). A simple technique to avoid this is to deliberately and systematically overprovision the system; by doing so, the system is ensured to operate in a more forgiving regime (Figure 3).

Overprovisioning doesn't come for free; an overprovisioned system is underutilizing its resources, and it is tempting to exploit this underutilization instead of adding more resources as the load on the system grows. In fact, it is only when the system nears saturation that many well-studied problems (such as load balancing) become interesting. However, we believe it is usually better to have a well-behaved, overprovisioned system than a poorly behaved, fully utilized one, especially given that computing resources are typically inexpensive relative to the cost of human designers.

However, overprovisioning contains the implicit assumption that the designers can accurately predict the expected operating regime of the system. As we've argued in Section 1, this assumption is often false, and it can lead to unexpected fragility.

Use admission control: given that systems tend to become unstable as they saturate, a useful technique is to use admission control to reject load as the system approaches the saturation point. Of course, to do this requires that the saturation point is identifiable; for large systems, the number of variables that contribute to the saturation point may be large, and thus statically identifying the saturation point may be difficult. Admission control often can be added to a system as an "orthogonal" or independent component. For example, high throughput web farms typically use layer 5 switches for both load balancing and admission control.

To reject load still requires resources from the system; each incoming task or request must be turned back, and the act of turning it back consumes resources. Thus, we view systems that perform admission control as having two classes of service: normal service, in which tasks are processed, and an extremely lightweight service, in which tasks are rejected. It is important to realize that the lightweight service has a response curve similar to that shown in Figure 2: a service, even if it is performing admission control, can saturate and then collapse. This effect is called livelock, and it is described in [4]. Admission control simply gives a system the ability to switch between two response curves, one for each class of service.

Build introspection into the system: an introspective system is one in which the ability to monitor the system is designed in from the beginning. As argued in [2], by building measurement infrastructure into a system, designers are much more readily able to monitor, diagnose, and adapt to aberrant behavior than in a black-box system. While this may seem obvious, consider the fact that the Internet and many of its protocols and mechanisms do not include the ability to introspect. As a result, researchers have often found it necessary to subvert features of existing protocols [9, 15], or to devise cunning mechanisms to deduce properties of network [13]. We believe that introspection is a necessary property of a system for it to be both managable and for its designers and operators to be able to help it adapt to a changing environment.

Introduce adaptivity by closing the control loop: the usual way for systems to evolve over time is for their designers and operators to measure its current behavior, and then to correspondingly adapt its design. This is essentially a control loop, in which the human designers and operators form the control logic. This loop operates on a very long timescale; it can take days, weeks, or longer for humans to adapt a complex system.

However, an interesting class of systems are those which include internal control loops. These systems incorporate the results of introspection, and attempt to adapt control variables dynamically to keep the system operating in a stable or well-performing regime. This notion of adaptation is important even if a system employs admission control or overprovisioning, because *internal* as well as external perturbations can affect the system. For example, modern disks occasionally perform thermal recalibration, vastly affecting their performance; if a system doesn't adapt to this, transient periods of poor performance or even instability may result.

Closed control loops for adaptation have been exploited in many systems, including TCP congestion control, online adaptation of query plans in databases [8, 10], or adaptive operating systems that tuning their policies or run-time parameters to improve performance [16]. All such systems have the property that the component performing the adaptation is able to hypothesize somewhat precisely about the effects of the adaptation; without this ability, the system would be "operating in the dark", and likely would become unpredictable. A new, interesting approach to hypothesizing about the effects of adaptation is to use statistical machine learning; given this, a system can experiment with changes in order to build up a model of their effects.

Plan for failure: even if a system employs all of the above strategies, as it grows sufficiently complex, unexpected perturbations and failure modes inevitably will emerge. *Complex systems must expect failure and plan for it accordingly*.

Planning for failure might imply many things: systems may attempt to minimize the damage caused by the failure by using robust abstractions such as transactions [6], or the system may be constructed so that losses are acceptable to its users (as is the case with the web). Systems may attempt to minimize the amount of time in which they are in a failure state, for example by checkpointing the system in known good states to allow for rapid recovery. In addition, systems may be organized as several weakly coupled compartments, in the hope that failures will be contained within a single compartment. Alternatively, systems may stave off failure by proactively "scrubbing" their internal state to prevent it from accumulating inconsistencies [3].

4. Summary

In this paper, we have argued that a common design paradigm for complex systems (careful design based on a prediction of the operating environment, load, and failures that the system will experience) is fundamentally fragile. This fragility arises because the diffuse coupling of components within a complex systems makes them prone to completely unpredictable behavior in the face of small perturbations. Instead, we argue that a different design paradigm needs to emerge if we want to prevent the ever-increasing complexity of our systems from causing them to become more and more unstable. This different design paradigm is one in which systems are given the best possible chance of stable behavior (through techniques such as overprovisioning, admission control, and introspection), as well as the ability to adapt to unexpected situations (by treating introspection as feedback to a closed control loop). Ultimately, systems must be designed to handle failures gracefully, as complexity seems to lead to an inevitable unpredictability.

In the future, we hope to explore the rich design space associated with robust, complex systems. Our plans include evaluating and extending the techniques identified in this paper in the context of adaptive, wide-area information delivery systems, such as caching hierarchies, content distribution networks, and peer-to-peer content sharing systems.

References

[1] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*, May 1999.

[2] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D. A. Patterson. ISTORE: Introspective storage for data intensive network services. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, March 1999.

[3] G. Candea and A. Fox. Reboot-based High Availability. In *Presented in the WIP Session of the 4th Symposium for Operating System Design and Implementation (OSDI 2000)*, San Diego, CA, October 2000.

[4] P. Druschel and G. Banga. Lazy Receiver Processing: A Network Subsystem Architecture for Server Systems. In *Proceedings of the USENIX 2nd Symposium on Operating System Design and Implementation (OSDI '96)*, Seattle, WA, USA, October 1996.

[5] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *ACM Transactions on Networking*, 2(2):122–136, April 1994.

[6] J. Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of VLDB*, Cannes, France, September 1981.

[7] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI 2000)*, San Diego, California, USA, October 2000.

[8] Z. G. Ives, M. Friedman, D. Florescu, A. Y. Levy, and D. Weld. An Adaptive Query Execution System for Data Integration. In *Proceedings of SIGMOD 1999*, June 1999.

[9] V. Jacobsen. Traceroute. `ftp://ftp.ee.lbl.gov/traceroute.tar.Z`, 1989.

[10] N. Kabra and D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proceedings of SIGMOD 1998*, Seattle, WA, June 1998.

[11] C. Labovitz, G. R. Malan, and F. Jahanian. Internet routing instability. In *Proceedings of ACM SIGCOMM ’97*, Cannes, France, September 1997.

[12] C. Labovitz, G. R. Malan, and F. Jahanian. Origins of internet routing instability. In *Proceedings of IEEE INFOCOMM ’99 Conference*, New York, New York, March 1999.

[13] K. Lai and M. Baker. Measuring Link Bandwidths Using a Deterministic Model of Packet Delay. In *Proceedings of the 2000 ACM SIGCOMM Conference*, Stockholm, Sweden, August 2000.

[14] E. N. Lorentz. *The Essence of Chaos*. University of Washington Press, Seattle, WA, 1993.

[15] S. Savage. Sting: a TCP-based Network Measurement Tool. In *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems (USITS ’99)*, October 1999.

[16] M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, May 1997.

Using Abstraction To Improve Fault Tolerance

Miguel Castro
Microsoft Research Ltd.
1 Guildhall St., Cambridge CB2 3NH, UK
mcastro@microsoft.com

Rodrigo Rodrigues and Barbara Liskov
MIT Laboratory for Computer Science
545 Technology Sq., Cambridge, MA 02139, USA
{rodrigo,liskov}@lcs.mit.edu

Abstract

Software errors are a major cause of outages and they are increasingly exploited in malicious attacks. Byzantine fault tolerance allows replicated systems to mask some software errors but it is expensive to deploy. This paper describes a replication technique, BFTA, which uses abstraction to reduce the cost of Byzantine fault tolerance and to improve its ability to mask software errors. BFTA reduces cost because it enables reuse of off-the-shelf service implementations. It improves availability because each replica can be repaired periodically using an abstract view of the state stored by correct replicas, and because each replica can run distinct or non-deterministic service implementations, which reduces the probability of common mode failures. We built an NFS service that allows each replica to run a different operating system. This example suggests that BFTA can be used in practice — the replicated file system required only a modest amount of new code, and preliminary performance results indicate that it performs comparably to the off-the-shelf implementations that it wraps.

1. Introduction

There is a growing demand for highly-available systems that provide correct service without interruptions. These systems must tolerate software errors because these are a major cause of outages [7]. Furthermore, there is an increasing number of malicious attacks that exploit software errors to gain control or deny access to systems that provide important services.

This paper proposes a replication technique, BFTA, that combines Byzantine fault tolerance [12] with work on data abstraction [11]. Byzantine fault tolerance allows a replicated service to tolerate arbitrary behavior from faulty replicas, e.g., the behavior caused by a software bug, or the behavior of a replica that is controlled by an attacker. Abstraction hides implementation details to enable the reuse of off-the-shelf implementations of important services (e.g., file systems, databases, or HTTP daemons) and to improve the ability to mask software errors.

This research was partially supported by DARPA under contract F30602-98-1-0237 monitored by the Air Force Research Laboratory.

We extended the BFT library [1, 2] to implement BFTA. The original BFT library provides Byzantine fault tolerance with good performance and strong correctness guarantees if no more than $1/3$ of the replicas fail within a small window of vulnerability. However, it requires all replicas to run the same service implementation and to update their state in a deterministic way. Therefore, it cannot tolerate deterministic software errors that cause all replicas to fail concurrently and it complicates reuse of existing service implementations because it requires extensive modifications to ensure identical values for the state of each replica.

The BFTA library and methodology described in this paper correct these problems — they enable replicas to run different or non-deterministic implementations. The methodology is based on the concepts of *abstract specification* and *abstraction function* from work on data abstraction [11]. We start by defining a *common abstract specification* for the service, which specifies an *abstract state* and describes how each operation manipulates the state. Then we implement a *conformance wrapper* for each distinct implementation to make it behave according to the common specification. The last step is to implement an abstraction function (and one of its inverses) to map from the concrete state of each implementation to the common abstract state (and vice versa).

Our methodology offers several important advantages.

Reuse of existing code. BFTA implements a form of state machine replication [14, 10], which allows replication of services that perform arbitrary computations, but requires determinism: all replicas must produce the same sequence of results when they process the same sequence of operations. Most off-the-shelf implementations of services fail to satisfy this condition. For example, many implementations produce timestamps by reading local clocks, which can cause the states of replicas to diverge. The conformance wrapper and the abstract state conversions enable the reuse of existing implementations without modifications. Furthermore, these implementations can be non-deterministic, which reduces the probability of common mode failures.

Software rejuvenation. It has been observed [9] that there is a correlation between the length of time software runs and the probability that it fails. BFTA combines proactive recovery [2] with abstraction to counter this problem. Replicas are recovered periodically even if there is no reason to suspect they are faulty. Recoveries are staggered such that the service remains available during rejuvenation to enable frequent recoveries. When a replica is recovered, it is rebooted and restarted from a clean state. Then it is brought up to date using a correct copy of the abstract state that is obtained from the group of replicas. Abstraction may improve availability by hiding corrupt concrete states, and it enables proactive recovery when replicas do not run the same code or run code that is non-deterministic.

Opportunistic N-version programming. Replication is not useful when there is a strong positive correlation between the failure probabilities of the different replicas, e.g., deterministic software bugs cause all replicas to fail at the same time when they run the same code. BFTA enables an opportunistic form of N-version programming [3] — replicas can run distinct, off-the-shelf implementations of the service. This is a viable option for many common services, e.g., relational databases, HTTP daemons, file systems, and operating systems. In all these cases, competition has led to four or more distinct implementations that were developed and are maintained separately but have similar (although not identical) functionality. Furthermore, the technique is made easier by the existence of standards that provide identical interfaces to different implementations, e.g., ODBC [6] and NFS [5]. We can also leverage the large effort towards standardizing data representations using XML.

It is widely believed that the benefits of N-version programming [3] do not justify its high cost [7]. It is better to invest the same amount of money on better development, verification, and testing of a single implementation. But opportunistic N-version programming achieves low cost due to economies of scale without compromising the quality of individual implementations. Since each off-the-shelf implementation is sold to a large number of customers, the vendors can amortize the cost of producing a high quality implementation. Additionally, taking advantage of interoperability standards keeps the cost of writing the conformance wrappers and state conversion functions low.

The paper explains the methodology by walking through an example, the implementation of a replicated file service where replicas run different operating systems and file systems. For this methodology to be successful, the conformance wrapper and the state conversion functions must be simple to reduce the likelihood of introducing more errors and introduce a low overhead. Experimental results indicate that this is true in our example.

The remainder of the paper is organized as follows. Section 2 provides an overview of the BFTA methodology and library. Section 3 explains how we applied the methodology to build the replicated file system. Section 4 presents our conclusions and some preliminary results.

2. The BFTA Technique

This section provides an overview of our replication technique. It starts by describing the methodology that we use to build a replicated system from existing service implementations. It ends with a description of the BFTA library.

2.1. Methodology

The goal is to build a replicated system by reusing a set of off-the-shelf implementations, $I_1, ..., I_n$, of some service. Ideally, we would like n to equal the number of replicas so that each replica can run a different implementation to reduce the probability of simultaneous failures. But the technique is useful even with a single implementation.

Although off-the-shelf implementations of the same service offer roughly the same functionality, they behave differently: they implement different specifications, $S_1, ..., S_n$ using different representations of the service state. Even the behavior of different replicas that run the same implementation may be different when the specification they implement is not strong enough to ensure deterministic behavior. For instance, the specification of the NFS protocol [5] allows implementations to choose arbitrary values for file handles.

BFTA, like any form of state machine replication, requires determinism: replicas must produce the same sequence of results when they execute the same sequence of operations. We achieve determinism by defining a *common abstract specification*, S, for the service that is strong enough to ensure deterministic behavior. This specification defines the abstract state, an initial state value, and the behavior of each service operation.

The specification is defined without knowledge of the internals of each implementation unlike what happens in the technique sketched in [13]. It is sufficient to treat them as black boxes, which is important to enable the use of existing implementations. Additionally, the abstract state captures only what is visible to the client rather than mimicking what is common in the concrete states of the different implementations. This simplifies the abstract state and improves the effectiveness of our software rejuvenation technique.

The next step, is to implement *conformance wrappers*, $C_1, ..., C_n$, for each of $I_1, ..., I_n$. The conformance wrappers implement the common specification S. The implementation of each wrapper C_i is a veneer that invokes the operations offered by I_i to implement the operations in S; in implementing these operations it makes use of a *conformance rep* that stores whatever additional information is needed to allow the translation from the concrete behavior of the implementation to the abstract behavior.

The final step is to implement the *abstraction function* and one of its inverses. These functions allow state transfer among the replicas. State transfer is used to repair faulty replicas, and also to bring slow replicas up-to-date when messages they are missing have been garbage collected. For state transfer to work replicas must agree on the value of the state of the service after executing a sequence of operations; they will not agree on the value of the concrete state but our methodology ensures that they will agree on the value of the abstract state. The abstraction function is used to convert the concrete state stored by a replica into the abstract state, which is transferred to another replica. The receiving replica uses the inverse function to convert the abstract state into its own concrete state representation.

To enable efficient state transfer between replicas, the abstract state is defined as an array of variable-sized objects. We explain how this representation enables efficient state transfer in Section 2.2.

There is an important trend that simplifies the methodology. Market forces push vendors towards extending their products to offer interfaces that implement standard specifications for interoperability, e.g., ODBC [6]. Usually, a standard specification S' cannot be used as the common specification S because it is too weak to ensure deterministic behavior. But it can be used as a basis for S and, because S and S' are similar, it is relatively easy to implement conformance wrappers and state conversion functions, these implementations can be mostly reused across implementations, and most client code can use the replicated system without modification.

2.2. Library

The BFTA library extends BFT with the features necessary to provide the methodology. Figure 1 presents a summary of the library's interface.

```
Client call:
int invoke(Byz_req *req, Byz_rep *rep,
                              bool read_only);

Execution upcall:
int execute(Byz_req*req, Byz_rep*rep,
            int client, Byz_buffer *non-det);

Checkpointing:
void modify(int nobjs, int* objs);

State conversion upcalls:
int get_obj(int i, char** obj);

void put_objs(int nobjs, char **objs,
                           int *is, int *szs);
```

Figure 1. BFTA Interface and Upcalls

The `invoke` procedure is called by the client to invoke an operation on the replicated service. This procedure carries out the client side of the replication protocol and returns the result when enough replicas have responded. When the library needs to execute an operation at a replica, it makes an upcall to an `execute` procedure that is implemented by the conformance wrapper for the service implementation run by the replica.

To perform state transfer in the presence of Byzantine faults, it is necessary to be able to prove that the state being transferred is correct. Otherwise, faulty replicas could corrupt the state of out-of-date but correct replicas. (A detailed discussion of this point can be found in [2].) Consequently, replicas cannot discard a copy of the state produced after executing a request until they know that the state produced by executing later requests can be proven correct. Replicas could keep a copy of the state after executing each request but this would be too expensive. Instead replicas keep just the current version of the concrete state plus copies of the abstract state produced every k-th request (e.g., k=128). These copies are called checkpoints.

As mentioned earlier, to implement checkpointing and state transfer efficiently, we require that the abstract state be encoded as an array of objects. Creating checkpoints by making full copies of the abstract state would be too expensive. Instead, the library uses copy-on-write such that checkpoints only contain the objects whose value is different in the current abstract state. Similarly, transferring a complete checkpoint to bring a recovering or out-of-date replica up to date would be too expensive. The library employs a hierarchical state partition scheme to transfer state efficiently. When a replica is fetching state, it recurses down a hierarchy of meta-data to determine which partitions are out of date. When it reaches the leaves of the hierarchy (which are the abstract objects), it fetches only the objects that are corrupt or out of date.

To implement state transfer, each replica must provide the library with two upcalls, which implement the *abstraction function* and one of its inverses. `get_obj` receives an object index i, allocates a buffer, obtains the value of the abstract object with index i, and places that value in the buffer. It returns the size for that object and a pointer to the buffer. `put_objs` receives a vector of objects with the corresponding indices and sizes. It causes the application to update its concrete state using the new values for the abstract objects passed as arguments. The library guarantees that the `put_objs` upcall is invoked with an argument that brings the abstract state of the replica to a consistent value (i.e., the value of a valid checkpoint). This is important to allow encodings of the abstract state with dependencies between objects, e.g., it allows objects to describe the meaning of other objects.

Each time the `execute` upcall is about to modify an object in the abstract state it is required to invoke a `modify` procedure, which is supplied by the library, passing the object index as argument. This is used to implement copy-on-write to create checkpoints incrementally: the library invokes `get_obj` with the appropriate index and keeps the copy of the object until the corresponding checkpoint can be discarded.

BFTA implements a form of state machine replication that requires replicas to behave deterministically. The methodology uses abstraction to hide most of the non-determinism in the implementations it reuses. However, many services involve forms of non-determinism that cannot be hidden by abstraction. For instance, in the case of the NFS service, the time-last-modified for each file is set by reading the server's local clock. If this were done independently at each replica, the states of the replicas would diverge. The library provides a mechanism [1] for replicas to agree on these non-deterministic values, which are then passed as arguments to the `execute` procedure.

Proactive recovery periodically restarts each replica from a correct, up-to-date checkpoint of the abstract state that is obtained from the other replicas. Recoveries are staggered so that less than $1/3$ of the replicas recover at the same time. This allows the other replicas to continue processing client requests during the recovery. Additionally, it should reduce the likelihood of simultaneous failures due to aging problems because at any instant less than $1/3$ of the replicas have been running for the same period of time.

Recoveries are triggered by a watchdog timer. When a replica is recovered, it reboots after saving the replication protocol state and the concrete service state to disk. The protocol state includes the abstract objects that were copied by the incremental checkpointing mechanism. Then the replica is restarted, and the conformance rep is reconstructed using the information that was saved to disk. Next, the library uses the hierarchical state transfer mechanism to compare the value of the abstract state it currently stores with the abstract state values stored by the other replicas. This is efficient: the replica uses cryptographic hashes stored in the state partition tree to determine which abstract objects are out-of-date or corrupt and it only fetches the value of these objects.

The object values fetched by the replica could be supplied to `put_objs` to update the concrete state, but the concrete state might still be corrupt. For example, an implementation may have a memory leak and simply calling `put_objs` will not free unreferenced memory. In fact, implementations will not typically offer an interface that can be used to fix all corrupt data structures in their concrete state. Therefore, it is better to restart the implementation from a clean initial concrete state and use the abstract state to bring it up-to-date.

3. An example: File System

This section illustrates the methodology using a replicated file system as an example. The file system is based on the NFS protocol [5]. Its replicas can run different operating systems and file system implementations.

3.1. Abstract Specification

The common abstract specification is based on the specification of the NFS protocol [5]. The abstract file service state consists of a fixed-size array of pairs with an object and a generation number. Each object has a unique identifier, *oid*, which is obtained by concatenating its index in the array and its generation number. The generation number is incremented every time the entry is assigned to a new object. There are four types of objects: files, whose data is a byte array; directories, whose data is a sequence of <name, oid> pairs ordered lexicographically; symbolic links, whose data is a small character string; and special *null* objects, which indicate an entry is free. All non-null objects have meta-data, which includes the attributes in the NFS `fattr` structure. Each entry in the array is encoded using XDR [4]. The object with index 0 is a directory object that corresponds to the root of the file system tree that was mounted.

The operations in the common specification are those defined by the NFS protocol. There are operations to read and write each type of non-null object. The file handles used by the clients are the *oids* of the corresponding objects. To ensure deterministic behavior, we define a deterministic procedure to assign *oid*s, and require that directory entries returned to a client be ordered lexicographically.

The abstraction hides many details; the allocation of file blocks, the representation of large files and directories, and the persistent storage medium and how it is accessed. This is desirable for simplicity, performance, and to improve resilience to software faults due to aging.

3.2. Conformance Wrapper

The conformance wrapper for the file service processes NFS protocol operations and interacts with an off-the-shelf file system implementation also using the NFS protocol as illustrated in Figure 2. A file system exported by the replicated file service is mounted on the client machine like any regular NFS file system. Application processes run unmodified and interact with the mounted file system through the NFS client in the kernel. We rely on user level relay processes to mediate communication between the standard NFS client and the replicas. A relay receives NFS protocol requests, calls the `invoke` procedure of our replication library, and sends the result back to the NFS client. The replication library invokes the `execute` procedure implemented by the conformance wrapper to run each NFS request.

The conformance rep consists of an array that corresponds to the one in the abstract state but it does not store copies of the objects; instead each array entry contains the generation number, the file handle assigned to the object by the underlying NFS server, and the value of the timestamps in the object's abstract meta-data. Empty entries store a null file handle. The rep also contains a map from file handles to *oids* to aid in processing replies efficiently.

The wrapper processes each NFS request received from a client as follows. It translates the file handles in the request, which encode *oid*s, into the corresponding NFS server file

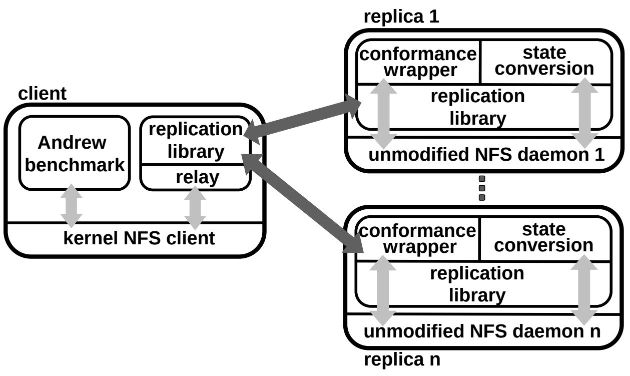


Figure 2. Software Architecture

handles. Then it sends the modified request to the underlying NFS server. The server processes the request and returns a reply.

The wrapper parses the reply and updates the conformance rep. If the operation created a new object, the wrapper allocates a new entry in the array in the conformance rep, increments the generation number, and updates the entry to contain the file handle assigned to the object by the NFS server. If any object is deleted, the wrapper marks its entry in the array free. In both cases, the reverse map from file handles to *oids* is updated. The wrapper must also update the abstract timestamps in the array entries corresponding to objects that were accessed. We use the library to agree on the timestamp value that is assigned to each operation [1]. This value is one of the arguments to the `execute` procedure implemented by the wrapper.

Finally, the wrapper returns a modified reply to the client, using the map to translate file handles to *oids* and replacing the concrete timestamp values by the abstract ones. When handling *readdir* calls the wrapper reads the entire directory and sorts it lexicographically to ensure the client receives identical replies from all replicas.

3.3. State Conversions

The abstraction function in the file service is implemented as follows. For each file system object, it uses the file handle stored in the conformance rep to invoke the NFS server to obtain the data and meta-data for the object. Then it replaces the concrete timestamp values by the abstract ones, converts the file handles in directory entries to *oids*, and sorts the directories lexicographically.

The inverse abstraction function in the file service works as follows. For each file system object o it receives, there are three possible cases depending on the state of the entry e that corresponds to o in the conformance rep: (1) e contains o's generation number, (2) e is not free and does not contain o's generation number, (3) e is free.

In the first case, objects that changed can be updated using the file handle in e to make calls to the NFS server. This is done differently for different types of objects. For files, it is sufficient to issue a `setattr` and a `write` to update the file's meta-data and data, and for symbolic links, it is sufficient to update their meta-data. Updating directories is slightly trickier. The inverse abstraction function reads the entire directory from the NFS server, computes its current abstract value, and compares this value with o. Nothing is done for entries that did not change. Entries that are not present in o or point to a different object are removed by issuing the appropriate calls to the NFS server. Then entries that are new or different in o are created but if the object they refer to does not exist in the current abstract state, it is first created using the value for the object that is supplied to `put_objs`.

In the second case, the NFS server is invoked to remove the object and then the function proceeds as in case 3.

In the third case, the NFS server is invoked to create the object (initially in a separate *unlinked* directory) and the object's data and meta-data is updated as in case 1. It is guaranteed that the directories that point to the object will be processed; the object is then linked to those directories and removed from the unlinked directory. When new objects are created, their file handles are recorded in the conformance wrapper's data structures.

3.4. Proactive Recovery

NFS file handles are volatile: the same file system object may have a different file handle after the NFS server restarts. For proactive recovery to work efficiently, we need a persistent identifier for objects in the concrete file system state that can be used to compute the abstraction function during recovery.

The NFS specification states that each object is uniquely identified by a pair of meta-data attributes: <fsid,fileid>. We solve the problem above by maintaining an additional map from <fsid,fileid> pairs to the corresponding *oids*. This map is saved to disk asynchronously when a checkpoint is created and synchronously before a proactive recovery. After rebooting, the replica that is recovering reads the map from disk. Then it traverses the file system's directory tree depth first from the root. It reads each object, uses the map to obtain its *oid*, and uses the cryptographic hashes from the state transfer protocol to check if the object is up-to-date. If the object is out-of-date or corrupt, it is fetched from another replica.

Instead of simply calling `put_objs` with the new object values, we intend to start an NFS server on a second empty disk and bring it up-to-date incrementally as we obtain the value of the abstract objects. This has the advantage of improving fault-tolerance as discussed in Section 2.2. Additionally, it can improve disk locality by clustering blocks from the same file and files that are in the same directory. This is not done in the current prototype.

4. Conclusion

Software errors are a major cause of outages and they are increasingly exploited in malicious attacks to gain control or deny access to important services. Byzantine fault tolerance allows replicated systems to mask some software errors but it has been expensive to deploy. We have described a replication technique, BFTA, which uses abstraction to reduce the cost of deploying Byzantine fault tolerance and to improve its ability to mask software errors.

BFTA reduces cost because it enables reuse of off-the-shelf service implementations without modifications, and it improves resilience to software errors by enabling opportunistic N-version programming, and software rejuvenation through proactive recovery.

Opportunistic N-version programming runs distinct, off-the-shelf implementations at each replica to reduce the probability of common mode failures. To apply this technique, it is necessary to define a common abstract behavioral specification for the service and to implement appropriate conversion functions for the state, requests, and replies of each implementation in order to make it behave according to the common specification. These tasks are greatly simplified by basing the common specification on standards for the interoperability of software from different vendors; these standards appear to be common, e.g., ODBC [6], and NFS [5]. Opportunistic N-version programming improves on previous N-version programming techniques by avoiding the high development, testing, and maintenance costs without compromising the quality of individual versions.

Additionally, we provide a mechanism to repair faulty replicas. Proactive recovery allows the system to remain available provided no more than $1/3$ of the replicas become faulty and corrupt the abstract state (in a correlated way) within a window of vulnerability. Abstraction may enable more than $1/3$ of the replicas to be faulty because it can hide corrupt items in concrete states of faulty replicas.

The paper described a replicated NFS file system implemented using our technique. The conformance wrapper and the state conversion functions in our prototype are simple — they have 1105 semi-colons, which is two orders of magnitude less than the size of the Linux 2.2 kernel. This suggests that they are unlikely to introduce new bugs.

We ran a scaled-up version of the Andrew benchmark [8, 2] (which generates 1 GB of data) to compare the performance of our replicated file system and the off-the-shelf implementation of NFS in Linux 2.2 that it wraps. Our performance results indicate that the overhead introduced by our technique is low; it is approximately 30% for this benchmark with a window of vulnerability of 17 minutes.

These preliminary results suggest that BFTA can be used in practice. As future work, it would be important to run experiments that apply BFTA to more challenging services, e.g., a relational database. It would also be important to run fault injection experiments to evaluate the availability improvements afforded by our technique.

References

[1] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

[2] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[3] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Fault Tolerant Computing, FTCS-8*, pages 3–9, 1978.

[4] Network Working Group Request for Comments: 1014. XDR: External Data Representation Standard, June 1987.

[5] Network Working Group Request for Comments: 1094. NFS: Network File System Protocol Specification, March 1989.

[6] Kyle Geiger. *Inside ODBC*. Microsoft Press, 1995.

[7] J. Gray and D. Siewiorek. High-Availability Computer Systems. *IEEE Computer*, 24(9):39–48, September 1991.

[8] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[9] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Fault-Tolerant Computing, FTCS-25*, pages 381–390, Pasadena, CA, June 1995.

[10] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[11] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[12] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[13] A. Romanovsky. Abstract Object State and Version Recovery in N-Version Programming. In *TOOLS Europe'99*, Nancy, France, June 1999.

[14] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

Fail-Stutter Fault Tolerance

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin, Madison

Abstract

Traditional fault models present system designers with two extremes: the Byzantine *fault model, which is general and therefore difficult to apply, and the* fail-stop *fault model, which is easier to employ but does not accurately capture modern device behavior. To address this gap, we introduce the concept of* fail-stutter *fault tolerance, a realistic and yet tractable fault model that accounts for both absolute failure and a new range of performance failures common in modern components. Systems built under the fail-stutter model will likely perform well, be highly reliable and available, and be easier to manage when deployed.*

1 Introduction

Dealing with failure in large-scale systems remains a challenging problem. In designing the systems that form the backbone of Internet services, databases, and storage systems, one must account for the possibility or even likelihood that one or more components will cease to operate correctly; just how one handles such failures determines overall system performance, availability, and manageability.

Traditionally, systems have been built with one of two fault models. At one extreme, there is the *Byzantine failure* model. As described by Lamport: "The component can exhibit arbitrary and malicious behavior, perhaps involving collusion with other faulty components" [25]. While these assumptions are appropriate in certain contexts (*e.g.*, security), they make it difficult to reason about system behavior.

At the other extreme, a more tractable and pragmatic approach exists. Known as the *fail-stop* model, this more limited approach is defined by Schneider as follows: "In response to a failure, the component changes to a state that permits other components to detect a failure has occurred and then stops" [33]. Thus, each component is either working or not, and when a component fails, all other components can immediately be made aware of it.

The problem with the Byzantine model is that it is general, and therefore difficult to apply. The problem with the fail-stop model is that it is simple, and therefore does not account for modern device behavior. Thus, we believe there is a need for a new model – one that is realistic and yet still tractable. The fail-stop model is a good starting point for a new model, but it needs to be enhanced in order to account for the complex behaviors of modern components.

The main reason an enhancement is in order is the increasing complexity of modern systems. For example, the latest Pentium has 42 million transistors [21], and future hardware promises even more complexity with the advent of "intelligent" devices [1, 27]. In software, as code bases mature, code size increases, and along with it complexity – the Linux kernel source alone has increased by a factor of ten since 1994.

Increasing complexity directly affects component behavior, as complex components often do not behave in simple, predictable ways. For example, two identical disks, made by the same manufacturer and receiving the same input stream will not necessarily deliver the same performance. Disks are not the only purveyor of erratic performance; as we will discuss within this document, similar behavior has been observed in many hardware and software components.

Systems built under the "fail-stop illusion" are prone to poor performance when deployed, performing well when everything is working perfectly, but failing to deliver good performance when just a single component does not behave as expected. Particularly vulnerable are systems that make static uses of parallelism, usually assuming that all components perform identically. For example, striping and other RAID techniques [28] perform well if every disk in the system delivers identical performance; however, if performance of a single disk is consistently lower than the rest, the performance of the entire storage system tracks that of the single, slow disk [6]. Such parallel-performance assumptions are common in parallel databases [16], search engines [18], and parallel applications [12].

To account for modern device behavior, we believe there is a need for a new model of fault behavior. The model should take into account that components sometimes fail, and that they also sometimes perform erratically. We term the unexpected and low performance of a component a *performance fault*, and introduce the *fail-stutter* fault model, an extension of the fail-stop model that takes performance faults into account.

Though the focus of the fail-stutter model is component performance, the fail-stutter model will also help in building systems that are more manageable, reliable, and available. By allowing for plug-and-play operation, incremental growth, worry-free replacement, and workload modification, fail-stutter fault tolerant systems decrease the need for human intervention and increase manageability. Diversity in system design is enabled, and thus reliability is improved. Finally, fail-stutter fault tolerant systems deliver consistent performance, which likely improves availability.

In this paper, we first build the case for fail-stutter fault tolerance via an examination of the literature. We then discuss the fail-stutter model and its benefits, review related work, and conclude.

2 The Erratic Behavior of Systems

In this section, we examine the literature to document the many places where performance faults occur; note that this list is illustrative and in no means exhaustive. In our survey, we find that device behavior is becoming increasingly difficult to understand or predict. In many cases, even when erratic performance is detected and investigated, no cause is discovered, hinting at the high complexity of modern systems. Interestingly, many performance variations come from research papers in well-controlled laboratory settings, often running just a single application on homogeneous hardware; we speculate that component behavior in less-controlled real-world environments would likely be worse.

2.1 Hardware

We begin our investigation of performance faults with those that are caused by hardware. We focus on three important hardware components: processors and their caches, disks, and network switches. In each case, the increasing complexity of the component over time has led to a richer set of performance characteristics.

2.1.1 Processors and Caches

Fault Masking: In processors, fault masking is used to increase yield, allowing a slightly flawed chip to be used; the result is that chips with different characteristics are sold as identical. For example, the Viking series of processors from Sun are examined in [2], where the authors measure the cache size of each of a set of Viking processors via micro-benchmark. “The Single SS-51 is our base case. The graphs reveal that the [effective size of the] first level cache is only 4K and is direct-mapped.” The specifications suggest a level-one data cache of size 16 KB, with 4-way set associativity. However, some chips produced by TI had portions of their caches turned off, whereas others, produced at different times, did not. The study measured application performance across the different Vikings, finding performance differences of up to 40% [2].

The PA-RISC from HP [35] also uses fault-masking in its cache. Schneider reports that the HP cache mechanism maps out certain “bad” lines to improve yield [34].

Fault-masking is not only present in modern processors. For example, the Vax-11/780 had a 2-way set associative cache, and would turn off one of the sets when a failure was detected within it. Similarly, the Vax-11/750 had a direct-mapped cache, and would shut off the whole cache under a fault. Finally, the Univac 1100/60 also had the ability to shut off portions of its cache under faults [37].

Prediction and Fetch Logic: Processor prediction and instruction fetch logic is often one of the most complex parts of a processor. The performance characteristics of the Sun UltraSPARC-I were studied by Kushman [24], and he finds that the implementation of the next-field predictors, fetching logic, grouping logic, and branch-prediction logic all can lead to the unexpected run-time behavior of programs. Simple code snippets are shown to exhibit non-deterministic performance – a program, executed twice on the same processor under identical conditions, has run times that vary by up to a factor of three. Kushman discovered four such anomalies, though the cause of two of the anomalies remains unknown.

Replacement Policy: Hardware cache replacement policies also can lead to unexpected performance. In their work on replicated fault-tolerance, Bressoud and Schneider find that: “The TLB replacement policy on our HP 9000/720 processors was non-deterministic. An identical series of location-references and TLB-insert operations at the processors running the primary and backup virtual machines could lead to different TLB contents” [10], p. 6, ¶ 2. The reason for the non-determinism is not given, nor does it appear to be known, as it surprised numerous HP engineers.

2.1.2 Disks

Fault Masking: Disks also perform some degree of fault masking. As documented in [6], a simple bandwidth experiment shows differing performance across 5400-RPM Seagate Hawk drives. Although most of the disks deliver 5.5 MB/s on sequential reads, one such disk delivered only 5.0 MB/s. Because the lesser-performing disk had three times the block faults than other devices, the author hypothesizes that SCSI bad-block remappings, transparent to both users and file systems, were the culprit.

Bad-block remapping is also an old technique. Early operating systems for the Univac 1100 series would record which tracks of a disk were faulty, and then avoid using them for subsequent writes to the disk [37].

Timeouts: Disks tend to exhibit sporadic failures. A study of a 400-disk farm over a 6-month period reveals that: “The largest source of errors in our system are SCSI timeouts and parity problems. SCSI timeouts and parity errors make up 49% of all errors; when network errors are removed, this figure rises to 87% of all error instances” [38], p. 7, ¶ 3. In examining their data further, one can ascertain that a timeout or parity error occurs roughly two times per day on average. These errors often lead to SCSI bus resets, affecting the performance of all disks on the degraded SCSI chain.

Similarly, intermittent disk failures were encountered by Bolosky *et al.* [9]. They noticed that disks in their video file server would go off-line at random intervals for short periods of time, apparently due to thermal recalibrations.

Geometry: Though the previous discussions focus on performance fluctuations *across* devices, there is also a performance differential present *within* a single disk. As documented in [26], disks have multiple zones, with performance across zones differing by up to a factor of two. Although this seems more “static” than other examples, unless disks are treated identically, different disks will have different layouts and thus different performance characteristics.

Unknown Cause: Sometimes even careful research does not uncover the cause of I/O performance problems. In their work on external sorting, Rivera and Chien encounter disk performance irregularities: “Each of the 64 machines in the cluster was tested; this revealed that four of them had about 30% slower I/O performance. Therefore, we excluded them from our subsequent experiments” [30], p. 7, last ¶.

A study of the IBM Vesta parallel file system reveals: “The results shown are the best measurements we obtained, typically on an unloaded system. [...] In many cases there was only a small (less than 10%) variance among the different measurements, but in some cases the variance was significant. In these cases there was typically a cluster of measurements that gave near-peak results, while the other measurements were spread relatively widely down to as low as 15-20% of peak performance” [15], p. 250, ¶ 2.

2.1.3 Network Switches

Deadlock: Switches have complex internal mechanisms that sometimes cause problematic performance behavior. In [6], the author describes a recurring network deadlock in a Myrinet switch. The deadlock results from the structure of the communication software; by waiting too long between packets that form a logical “message”, the deadlock-detection hardware triggers and begins the deadlock recovery process, halting all switch traffic for two seconds.

Unfairness: Switches often behave unfairly under high load. As also seen in [6], if enough load is placed on a Myrinet switch, certain routes receive preference; the result is that the nodes behind disfavored links appear “slower” to a sender, even though they are fully capable of receiving data at link rate. In that work, the unfairness resulted in a 50% slowdown to a global adaptive data transfer.

Flow Control: Networks also often have internal flow-control mechanisms, which can lead to unexpected performance problems. Brewer and Kuszmaul show the effects of a few slow receivers on the performance of all-to-all transposes in the CM-5 data network [12]. In their study, once a receiver falls behind the others, messages accumulate in the network and cause excessive network contention, reducing transpose performance by almost a factor of three.

2.2 Software

Sometimes unexpected performance arises not due to hardware peculiars, but because of the behavior of an important software agent. One common culprit is the operating system, whose management decisions in supporting various complex abstractions may lead to unexpected performance surprises. Another manner in which a component will seem to exhibit poor performance occurs when another application uses it at the same time. This problem is particularly acute for memory, which swaps data to disk when over-extended.

2.2.1 Operating Systems and Virtual Machines

Page Mapping: Chen and Bershad have shown that virtual-memory mapping decisions can reduce application performance by up to 50% [14]. Virtually all machines today use physical addresses in the cache tag. Unless the cache is small enough so that the page offset is not used in the cache tag, the allocation of pages in memory will affect the cache-miss rate.

File Layout: In [6], a simple experiment demonstrates how file system layout can lead to non-identical performance across otherwise identical disks and file systems. Sequential file read performance across aged file systems varies by up to a factor of two, even when the file systems are otherwise empty. However, when the file systems are recreated afresh, sequential file read performance is identical across all drives in the cluster.

Background Operations: In their work on a fault-tolerant, distributed hash table, Gribble *et al.* find that untimely garbage collection causes one node to fall behind its mirror in a replicated update. The result is that one machine over-saturates and thus is the bottleneck [20]. Background operations are common in many systems, including cleaners in log-structured file systems [31], and salvagers that heuristically repair inconsistencies in databases [19].

2.2.2 Interference From Other Applications

Memory Bank Conflicts: In their work on scalar-vector memory interference, the authors show that perturbations to a vector reference stream can reduce memory system efficiency by up to a factor of two [29].

Memory Hogs: In their recent paper, Brown and Mowry show the effect of an out-of-core application on interactive jobs [13]. Therein, the response time of the interactive job is shown to be up to 40 times worse when competing with a memory-intensive process for memory resources.

CPU Hogs: Similarly, interference to CPU resources leads to unexpected slowdowns. From a different sorting study: “The performance of NOW-Sort is quite sensitive to various disturbances and requires a dedicated system to achieve ’peak’ results” [5], p. 8, ¶ 1. A node with excess CPU load reduces global sorting performance by a factor of two.

2.3 Summary

We have documented many cases where components exhibit unexpected performance. As both hardware and software components increase in complexity, they are more likely to perform internal error correction and fault masking, have different performance characteristics depending on load and usage, and even perhaps behave non-deterministically. Note that short-term performance fluctuations that occur randomly across all components can likely be ignored; particularly harmful are slowdowns that are long-lived and likely to occur on a subset of components. Those types of faults cannot be handled with traditional methods, and thus must be incorporated into a model of component behavior.

3 Fail-Stutter Fault Tolerance

In this section, we discuss the topics that we believe are central to the fail-stutter model. Though we have not yet fully formalized the model, we outline a number of issues that must be resolved in order to do so. We then cover an example, and discuss the potential benefits of utilizing the fail-stutter model.

3.1 Towards a Fail-Stutter Model

We now discuss issues that are central in developing the fail-stutter model. We focus on three main differences from the fail-stop model: the separation of performance faults from correctness faults, the notification of other components of the presence of a performance fault within the system, and performance specifications for each component.

Separation of performance faults from correctness faults. We believe that the fail-stutter model must distinguish two classes of faults: absolute (or correctness) faults, and performance faults. In most scenarios, we believe the appropriate manner in which to deal with correctness faults such as total disk or processor failure is to utilize the fail-stop model. Schneider considers a component faulty "once its behavior is no longer consistent with its specification" [33]. In response to such a correctness failure, the component changes to a state that permits other components to detect the failure, and then the component stops operating. In addition, we believe that the fail-stutter model should incorporate the notion of a *performance failure,* which, combined with the above, completes the fail-stutter model. A component should be considered performance-faulty if it has not absolutely failed as defined above and when its performance is less than that of its performance specification.

We believe this separation of performance and correctness faults is crucial to the model, as there is much to be gained by utilizing performance-faulty components. In many cases, devices may often perform at a large fraction of their expected rate; if many components behave this way, treating them as absolutely failed components leads to a large waste of system resources.

One difficulty that must be addressed occurs when a component responds arbitrarily slowly to a request; in that case, a performance fault can become blurred with a correctness fault. To distinguish the two cases, the model may include a performance threshold within the definition of a correctness fault, *i.e.*, if the disk request takes longer than T seconds to service, consider it absolutely failed. Performance faults fill in the rest of the regime when the device is working.

Notification of other components. One major departure from the fail-stop model is that we do not believe that other components need be informed of all performance failures when they occur, for the following reasons. First, erratic performance may occur quite frequently, and thus distributing that information may be overly expensive. Further, a performance failure from the perspective of one component may not manifest itself to others (*e.g.*, the failure is caused by a bad network link). However, if a component is persistently performance-faulty, it may be useful for a system to export information about component "performance state", allowing agents within the system to readily learn of and react to these performance-faulty constituents.

Performance specifications. Another difficulty that arises in defining the fail-stutter model is arriving at a performance specification for components of the system. Ideally, we believe the fail-stutter model should present the system designer with a trade-off. At one extreme, a model of component performance could be as simple as possible: "this disk delivers bandwidth at 10 MB/s." However, the simpler the model, the more likely performance faults occur, *i.e.*, the more likely performance deviates from its expected level. Thus, because different assumptions can be made, the system designer could be allowed some flexibility, while still drawing attention to the fact that devices may not perform as expected. The designer must also have a good model of *how often* various performance faults occur, and *how long* they last; both of these are environment and component specific, and will strongly influence how a system should be built to react to such failures.

3.2 An Example

We now sketch how the fail-stutter model could be employed for a simple example given different assumptions about performance faults. Specifically, we consider three scenarios in order of increasingly realistic performance assumptions. Although we omit many details necessary for complete designs, we hope to illustrate how the fail-stutter model may be utilized to enable more robust system construction. We assume that our workload consists of writing D data blocks in parallel to a set of $2 \cdot N$ disks and that data is encoded across the disks in a RAID-10 fashion (*i.e.*, each pair of disks is treated as a RAID-1 mirrored pair and data blocks are striped across these mirrors a la RAID-0).

In the first scenario, we use only the fail-stop model, assuming (perhaps naively) that performance faults do not occur. Thus, absolute failures are accounted for and handled accordingly – if an absolute failure occurs on a single disk, it is detected and operation continues, perhaps with a reconstruction initiated to a hot spare; if two disks in a mirror-pair fail, operation is halted. Since performance faults are not considered in the design, each pair (and thus each disk) is given the same number of blocks to write: $\frac{D}{N}$. Therefore, if a performance fault occurs on any of the pairs, the time to write to storage is determined by the slow pair. Assuming $N - 1$ of the disk-pairs can write at B MB/s but one disk-pair can write at only b MB/s, with $b < B$, perceived throughput is reduced to $N \cdot b$ MB/s.

In the second scenario, in addition to absolute faults, we consider performance faults that are static in nature; that is, we assume the performance of a mirror-pair is relatively stable over time, but may not be uniform across disks. Thus, within our design, we compensate for this difference. One option is to gauge the performance of each disk once at installation, and then use the ratios to stripe data proportionally across the mirror-pairs; we may also try to pair disks that perform similarly, since the rate of each mirror is determined by the rate of its slowest disk. Given a single slow disk, if the system correctly gauges performance, write throughput increases to $(N - 1) \cdot B + b$ MB/s. However, if any disk does not perform as expected over time, performance again tracks the slow disk.

Finally, in the third scenario, we consider more general performance faults to include those in which disks perform at arbitrary rates over time. One design option is to continually gauge performance and to write blocks across mirror-pairs in proportion to their current rates. We note that this approach increases the amount of bookkeeping: because these proportions may change over time, the controller must record where each block is written. However, by increasing complexity, we create a system that is more robust in that it can deliver the full available bandwidth under a wide range of performance faults.

3.3 Benefits of Fail-Stutter

Perhaps the most important consideration in introducing a new model of component behavior is the effect it would have if systems utilized such a model. We believe such systems are likely to be more available, reliable, and manageable than systems built only to tolerate fail-stop failures.

Manageability: Manageability of a fail-stutter fault tolerant system is likely to be better than a fail-stop system, for the following reasons. First, fail-stutter fault tolerance enables true "plug-and-play"; when the system administrator adds a new component, the system uses whatever performance it provides, without any additional involvement from the operator – a true "no futz" system [32]. Second, such a system can be incrementally grown [11], allowing newer, faster components to be added; adding these faster components to incrementally scale the system is handled naturally, because the older components simply appear to be performance-faulty versions of the new ones. Third, administrators no longer need to stockpile components in anticipation of their discontinuation. Finally, new workloads (and the imbalances they may bring) can be introduced into the system without fear, as those imbalances are handled by the performance-fault tolerance mechanisms. In all cases, the need for human intervention is reduced, increasing overall manageability. As Van Jacobson said, "Experience shows that anything that needs to be configured will be misconfigured" [23], p. 6; by removing the need for intricate tuning, the problems caused by misconfiguration are eradicated.

Availability: Gray and Reuter define availability as follows: "The fraction of the offered load that is processed with acceptable response times" [19]. A system that only utilizes the fail-stop model is likely to deliver poor performance under even a single performance failure; if performance does not meet the threshold, availability decreases. In contrast, a system that takes performance failures into account is likely to deliver consistent, high performance, thus increasing availability.

Reliability: The fail-stutter model is also likely to improve overall system reliability in at least two ways. First, "design diversity" is a desirable property for large-scale systems; by including components of different makes and manufacturers, problems that occur when a collection of identical components suffer from an identical design flaw are avoided. As Gray and Reuter state, design diversity is akin to having "a belt and suspenders, not two belts or two suspenders" [19]. A system that handles performance faults naturally works well with heterogeneously-performing parts. Second, reliability may also be enhanced through the detection of performance anomalies, as erratic performance may be an early indicator of impending failure.

4 Related Work

Our own experience with I/O-intensive application programming in clusters convinced us that erratic performance is the norm in large-scale systems, and that system support for building robust programs is needed [5]. Thus, we began work on River, a programming environment that provides mechanisms to enable consistent and high performance in spite of erratic performance in underlying components, focusing mainly on disks [7]. However, River itself does not handle absolute correctness faults in an integrated fashion, relying either upon retry-after-failure or a checkpoint-restart package. River also requires applications to be completely rewritten to enable performance robustness, which may not be appropriate in many situations.

Some other researchers have realized the need for a model of fault behavior that goes beyond simple fail-stop. The earliest that we are aware of is Shasha and Turek's work on "slow-down" failures [36]. The authors design an algorithm that runs transactions correctly in the presence of such failures, by simply issuing new processes to do the work elsewhere, and reconciling properly so as to avoid work replication. However, the authors assume that such behavior is likely only to occur due to network congestion or processes slowed by workload interference; indeed, they assume that a fail-stop model for disks is quite appropriate.

DeWitt and Gray label periodic performance fluctuations in hardware *interference* [17]. They do not characterize the nature of these problems, though they realize its potential impact on parallel operations.

Birman's recent work on Bimodal Multicast also addresses the issue of nodes that "stutter" in the context of multicast-based applications [8]. Birman's solution is to change the semantics of multicast from absolute delivery requirements to probabilistic ones, and thus gracefully degrade when nodes begin to perform poorly.

The networking literature is replete with examples of adaptation and design for variable performance, with the prime example of TCP [22]. We believe that similar techniques will need to be employed in the development of adaptive, fail-stutter fault-tolerant algorithms.

5 Conclusions

Too many systems are built assuming that all components are identical, that component behavior is static and unchanging in nature, and that each component either works or does not. Such assumptions are dangerous, as the increasing complexity of computer systems hints at a future where even the "same" components behave differently, the way they behave is dynamic and oft-changing, and there is a large range of normal operation that falls between the binary extremes of working and not working. By utilizing the fail-stutter model, systems are more likely to be manageable, available, and reliable, and work well when deployed in the real world.

Many challenges remain. The fail-stutter model must be formalized, and new models of component behavior must be developed, requiring both measurement of existing systems as well as analytical development. New adaptive algorithms, which can cope with this more difficult class of failures, must be designed, analyzed, implemented, and tested. The true costs of building such a system must be discerned, and different approaches need to be evaluated.

As a first step in this direction, we are exploring the construction of fail-stutter-tolerant storage in the Wisconsin Network Disks (WiND) project [3, 4]. Therein, we are investigating the adaptive software techniques that we believe are central to building robust and manageable storage systems. We encourage others to consider the fail-stutter model in their endeavors as well.

6 Acknowledgements

We thank the following people for their comments on this or earlier versions of this paper: David Patterson, Jim Gray, David Culler, Joseph Hellerstein, Eric Anderson, Noah Treuhaft, John Bent, Tim Denehy, Brian Forney, Florentina Popovici, and Muthian Sivathanu. Also, we would like to thank the anonymous reviewers for their many thoughtful suggestions. This work is sponsored by NSF CCR-0092840 and NSF CCR-0098274.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks. In *ASPLOS VIII*, San Jose, CA, Oct. 1998.

[2] R. H. Arpaci, A. C. Dusseau, and A. M. Vahdat. Towards Process Management on a Network of Workstations. http://www.cs.berkeley.edu/ remzi/258-final, May 1995.

[3] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. The Wisconsin Network Disks Project. `http://www.cs.wisc.edu/wind`, 2000.

[4] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, J. Bent, B. Forney, S. Muthukrishnan, F. Popovici, and O. Zaki. Manageable Storage via Adaptation in WiND. In *IEEE Int'l Symposium on Cluster Computing and the Grid (CCGrid'2001)*, May 2001.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. Searching for the Sorting Record: Experiences in Tuning NOW-Sort. In *SPDT '98*, Aug. 1998.

[6] R. H. Arpaci-Dusseau. *Performance Availability for Networks of Workstations*. PhD thesis, University of California, Berkeley, 1999.

[7] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *IOPADS '99*, May 1999.

[8] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Bidiu, and Y. Minsky. Bimodal multicast. *TOCS*, 17(2):41–88, May 1999.

[9] W. J. Bolosky, J. S. B. III, R. P. Draves, R. P. Fitzgerald, G. A. Gibson, M. B. Jones, S. P. Levi, N. P. Myhrvold, and R. F. Rashid. The Tiger Video Fileserver. Technical Report 96-09, Microsoft Research, 1996.

[10] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. In *SOSP 15*, Dec. 1995.

[11] E. A. Brewer. The Inktomi Web Search Engine. Invited Talk: 1997 SIGMOD, May 1997.

[12] E. A. Brewer and B. C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *Proceedings of the 1994 International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[13] A. D. Brown and T. C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *OSDI 4*, San Diego, CA, October 2000.

[14] J. B. Chen and B. N. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 120–133, December 1993.

[15] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[16] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsaio, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[17] D. J. DeWitt and J. Gray. Parallel database systems: The future of high-performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[18] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *SOSP 16*, pages 78–91, Saint-Malo, France, Oct. 1997.

[19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[20] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, , and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *OSDI 4*, San Diego, CA, October 2000.

[21] Intel. Intel Pentium 4 Architecture Product Briefing Home Page. http://developer.intel.com/design/Pentium4/prodbref, January 2001.

[22] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM '88*, pages 314–329, August 1988.

[23] V. Jacobson. How to Kill the Internet. ftp://ftp.ee.lbl.gov/talks/vj-webflame.ps.Z, 1995.

[24] N. A. Kushman. Performance Nonmonotonocities: A Case Study of the UltraSPARC Processor. Master's thesis, Massachussets Institute of Technology, Boston, MA, 1998.

[25] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[26] R. V. Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, Jan. 1997.

[27] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. Intelligent RAM (IRAM): Chips That Remember And Compute. In *1997 IEEE International Solid-State Circuits Conference*, San Francisco, CA, February 1997.

[28] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, pages 109–116, Chicago, IL, June 1988. ACM Press.

[29] R. Raghavan and J. Hayes. Scalar-Vector Memory Interference in Vector Computers. In *The 1991 International Conference on Parallel Processing*, pages 180–187, St. Charles, IL, August 1991.

[30] L. Rivera and A. Chien. A High Speed Disk-to-Disk Sort on a Windows NT Cluster Running HPVM. Submitted for pulication, 1999.

[31] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[32] M. Satyanarayanan. Digest of HotOS VII. http://www.cs.rice.edu/Conferences/HotOS/digest, March 1999.

[33] F. B. Schneider. Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[34] F. B. Schneider. Personal Communication, February 1999.

[35] A. P. Scott, K. P. Burkhart, A. Kumar, R. M. Blumberg, and G. L. Ranson. Four-way Superscalar PA-RISC Processors. *Hewlett-Packard Journal*, 48(4):8–15, August 1997.

[36] D. Shasha and J. Turek. Beyond Fail-Stop: Wait-Free Serializability and Resiliency in the Presence of Slow-Down Failures. Technical Report 514, Computer Science Department, NYU, September 1990.

[37] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation*. A K Peters, 3rd edition, 1998.

[38] N. Talagala and D. Patterson. An Analysis of Error Behaviour in a Large Storage System. In *IPPS Workshop on Fault Tolerance in Parallel and Distributed Systems*, 1999.

Reconsidering Internet Mobility

Alex C. Snoeren, Hari Balakrishnan, and M. Frans Kaashoek
MIT Laboratory for Computer Science
Cambridge, MA 02139
{*snoeren, hari, kaashoek*}*@lcs.mit.edu*

Abstract

Despite the popularity of mobile computing platforms, appropriate system support for mobile operation is lacking in the Internet. This paper argues this is not for lack of deployment incentives, but because a comprehensive system architecture that efficiently addresses the needs of mobile applications does not exist. We identify five fundamental issues raised by mobility—location, preservation of communication, disconnection handling, hibernation, and reconnection—and suggest design guidelines for a system that attempts to support Internet mobility.

In particular, we argue that a good system architecture should (i) eliminate the dependence of higher protocol layers upon lower-layer identifiers; (ii) work with any application-selected naming scheme; (iii) handle (unexpected) network disconnections in a graceful way, exposing its occurrence to applications; and (iv) provide mobility services at the mobile nodes themselves, rather than via proxies. Motivated by these principles, we propose a session-oriented, end-to-end architecture called Migrate, *and briefly examine the set of services it should provide.*

1 Introduction

The proliferation of laptops, handheld computers, cellular phones, and other mobile computing platforms connected to the Internet has triggered much research into system support for mobile networking over the past few years. Yet, when viewed as a large-scale, heterogeneous, distributed system, the Internet is notoriously lacking in any form of general support for mobile operation.

We argue that previous work has failed to comprehensively address several important issues. This paper discusses some of these issues and describes a session-oriented architecture we are developing to preserve end-to-end application-layer connectivity under various mobile conditions.

Mobility raises five fundamental problems:

This research was funded by DARPA (Grant No. MDA972-99-1-0014), NTT Corporation, Intel, and IBM. Alex C. Snoeren is supported by a National Defense Science and Engineering Graduate (NDSEG) Fellowship.

1. **Locating the mobile host or service:** Before any communication can be initiated, the desired end-point must be located and mapped to an addressable destination.
2. **Preserving communication:** Once a session has been established between end points (typically applications), communication should be robust across changes in the network location of the end points.
3. **Disconnecting gracefully:** Communicating applications should be able to rapidly discern when a disconnection at either end, or a network partition, causes communication to be disrupted.
4. **Hibernating efficiently:** If a communicating host is unavailable for a significant period of time, the system should suspend communications, and appropriately reallocate resources.
5. **Reconnecting quickly:** Communicating peers should detect the resumption of network connectivity in a timely manner. The system should support the resumption of all previously established communication sessions without much extra effort on the part of the applications.

Most current approaches provide varying degrees of support for the first two problems. The last three—disconnection, hibernation, and reconnection—have received little attention outside of the file system context [17]. We argue that a complete—and useful—solution must address all these issues.

One need look no further than interactive terminal applications like `ssh` or `telnet`, one of the Internet's oldest applications, for a practical example of the continuing lack of support for these important components. A user with an open session might pick up her laptop and disconnect from the network. After traveling for some period of time, she reconnects at some other network location and expects that her session continue where it left off. Unfortunately, if there was any activity on the session during the period of disconnectivity, she will find the connection aborted upon reconnection to the network. The particular details of the example are irrelevant, but demonstrate just how lacking current support is, even for this simple scenario.

Based on our own experience developing various mobile protocols and services [1, 3, 12, 24] and documented reports of several other researchers over several years [7, 11, 13, 16, 26], we identify four important guidelines that we believe should be followed as hints in designing an appropriate network architecture for supporting mobile Internet services and applications:

1. *Eliminate lower-layer dependence from higher layers.* A large number of problems arise because many higher layers of the Internet architecture use identifiers from lower layers, assuming they will never change during a connection.
2. *Do not restrict the choice of naming techniques.* Dynamic naming and location-tracking systems play an important role in addressing mobility. In general, whenever an end point moves, it should update a naming system with its new location—but forcing all applications to use a particular naming scheme is both unrealistic and inappropriate.
3. *Handle unexpected disconnections gracefully.* We advocate treating disconnections as a common occurrence, and exposing them to applications as they occur.
4. *Provide support at the end hosts.* Proxies are attractive due to their perceived ease of deployment. However, it becomes markedly more difficult to ensure they are appropriately located when hosts are mobile.

We elaborate upon these guidelines in Section 2. They have served as a guide in our development of an end-to-end, session-oriented system architecture, called *Migrate*, over which mobile networking applications and services can be elegantly layered. We describe our proposed architecture in Section 3, discussing how it addresses four of the five problems mentioned above: preserving communication, and handling disconnection, hibernation, and resumption. We do not provide or enforce a particular location or naming scheme, instead leveraging domain-specific naming services (e.g., DNS, service discovery schemes [1, 10], etc.) for end-point location.

An attractive feature of our architecture is that it accomplishes these tasks without sacrificing common-case performance. Migrate provides generic mechanisms for managing disconnections and reconnections in each application session, and for handling application state and context. We briefly discuss related work in Section 4 before concluding in Section 5.

2 Design guidelines

In this section, we elaborate on our four design guidelines for supporting applications on mobile hosts.

2.1 Eliminate lower-layer dependence

The first step in enabling higher-layer mobility handling is to remove inter-layer dependences. In a 1983 retrospective paper on the DoD Internet Architecture, Cerf wrote [6]: "TCP's [dependence] upon the network and host addresses for part of its connection identifiers" makes "dynamic reconnection" difficult, "a problem . . . which has plagued network designers since the inception of the ARPANET project in 1968." The result is that when the underlying network-layer (IP) address of one of the communicating peers changes, the end-to-end transport-layer (TCP) connection is unable to continue because it has bound to the network-layer identifier, tacitly (but wrongly) assuming its permanence for the duration of the connection.

A host of other problems crop up because of similar linkages. For example, the increasing proliferation of network address translators (NATs) in the middle of the network has caused problems for applications (like FTP) that use network- and transport-layer identifiers as part of their internal state. These problems can be avoided by removing any assumption of stability of lower-layer identifiers. If a higher layer finds it necessary to use a lower-layer identifier as part of its internal state, then the higher layer should allow for it to change, and continue to function across such changes.

Furthermore, each layer should expose relevant changes to higher layers. In today's Internet architecture, applications have almost no control over their network communication because lower layers (for the most part) do not concern themselves with higher-layer requirements. When important changes happen at a lower layer, for example to the network-layer address, they are usually hidden from higher layers. The unfortunate consequence of this is that it makes it hard for any form of adaptation to occur.

For example, a TCP sender attempts to estimate the properties of the network path for the connection. A significant change in the network-layer attachment point often implies that previously discovered path properties are invalid, and need to be rediscovered. This consequence is not limited to classical TCP congestion management—for example, if mobile applications are notified of changes in their environment and given the power to effect appropriate changes, significant improvements in both performance and usability can be realized [17, 19]. Similar results have also been shown in the network layer [7, 11, 28], and in the area of transport optimization over wireless links [3, 5, 24].

2.2 Beware the Siren song of naming

Many researchers have observed that the first problem raised by mobility, namely locating the mobile host or service, can be addressed through a sophisticated naming system, hence most proposals for managing Internet mobility attempt to provide naming and location services as a fundamental part of the mobility system.[1] Unfortunately, the tight binding between naming schemes and mobility support often causes the resulting system to be inefficient or unsuitable for various classes of applications. For example, Mobile IP assumes that the destination of each packet needs to be *independently* located, thereby necessitating a home agent to intercept and forward messages to a mobile host. The utility of alternative proposals to use agile naming [1] or IP multicast [18] for mobility support hinges on widespread deployment of their location systems.

We believe that inexorably binding mobility handling with naming unnecessarily complicates the mobility services, and restricts the ability to integrate advances in naming services. On the face of it, it appears attractive that a "good" naming scheme can provide the level of indirection by which to handle mobility. In practice, however, it is important to recognize and separate two distinct operations. The first is a "location" operation: The process of finding an end point of interest based on an application-specific name. The second is a "tracking" operation: Preserving the peer-to-peer communication in some way. There are two problems with using a new idealized naming scheme: First, there are a large number of ways in which applications describe what they are looking for, which forces this ideal naming scheme to perform the difficult task of accommodating them all. Experience shows that each application is likely to end up using a naming scheme that best suits it (e.g. INS, DNS, JINI, UPnP), rather than suffer the inadequacies of a universal one. Second, if this tracking is done through the same name resolution mechanism, every packet would invoke the resolution process, adding significant overhead and degrading performance.

We therefore suggest that an application use whichever naming scheme is sufficiently adept at providing the appropriate name-to-location binding in a timely fashion. This service is used at the beginning of a session between peers, or in the (unlikely) event that all peers change their network locations "simultaneously." At all other times, the onus of preserving communication across moves rests with the peers themselves. In the common case when only a subset of the peers moves at a time, the task of reconnection is efficiently handled by the peers themselves. We have previously described the details of such a scheme in the context of TCP connection migration [24].

2.3 Handle unexpected disconnections

The area of Internet mobility that has received the least attention is support for efficient disconnection and reconnection. While significant work has been done in the area of disconnected file systems [13, 17], less attention has been paid to preserving application communication when a disconnection occurs, enabling it to quickly resume upon reconnection. The key observation about disconnections is that they are usually unexpected. Furthermore, they last for rather unpredictable periods of time, ranging from a few seconds to several hours (or more). Today's network stacks terminate a connection as soon as a network disconnection is detected, with unfortunate consequences—the application (and often the user) has to explicitly reinitiate connectivity and application state is usually lost.

Like all other aspects of network communication, we believe the system should therefore provide standard support for unexpected disconnection, enabling applications to gracefully manage session state, releasing system resources and reallocating them when communication is restored. Even if the duration of the disconnection period is short enough to avoid significantly impacting communication or draining system resources, the disconnection and ensuing reconnection events are often hidden by current network stacks, leaving the higher network layers and application to eventually discover (often with unfortunate results) that network conditions have changed dramatically.

2.4 Provide services at the end points

A great deal of previous work in mobility management has relied on a proxy-based architecture, providing enhanced services to mobile hosts by routing communications through a (typically fixed) waypoint that is not collocated with the host [3, 8, 9, 15, 20, 26]. It is often easier to deploy new services through a proxy, as the proxy can provide enhanced services in a transparent fashion, interoperating with legacy systems. Unfortunately, in order to provide adequate performance, it is not only necessary to highly engineer the proxy [15], but locate the proxy appropriately as well.

Several researchers have proposed techniques to migrate proxy services to the appropriate location, avoiding the need to preconfigure locations [8, 25]. Unfortunately, all candidate proxy locations must be appropriately preconfigured to participate. Further, in the face of general mobility, proxies (or at least their internal state) must be able to move with the mobile host in order to remain along the path from the host to its correspondent peers. This is a complex problem [26]; we observe that it can be completely avoided if the support is collocated with the mobile host itself.

3 Migrate approach

We now describe the Migrate approach to mobility, which leverages application naming services and informed transport protocols to provide robust, low-overhead communica-

[1]Indeed, the authors of this paper are guilty of having taken this position in the past.

tion between application end points. We describe a session-layer protocol that handles both changes in network attachment point and disconnection in a seamless fashion, but is flexible enough to allow a wide variety of applications to maintain sufficient control for their needs.

3.1 Service model

The number of communication paradigms in use on the Internet remains small, but the type and amount of mobility support needed varies dramatically across modalities [7]. In particular, the notion of a session is application-dependent and varies widely, from a set of related connections (e.g. FTP's data and control channels) to an individual datagram exchange such as those often found in RPC-based applications (e.g. a cached DNS response). As session lengths grow longer and sessions become more complex in terms of the system resources they consume, applications can benefit from system support for robust communication between application end points. However, due to the disparate performance and reliability requirements of different session-based applications, it is important that a mobility service enables the application to dictate its requirements through explicit choice of transport protocols and policy defaults.

Hence we propose an optional session layer. This layer presents a simple, unified abstraction to the application to handle mobility: a session. Sessions exist between application-level end points, and can survive changes in the transport, network, and even other session layer protocol states. It also includes basic check-pointing and resumption facilities for periods of disconnection, enabling comprehensive, session-based state management for mobile-aware applications. Unlike previous network-layer approaches, our session layer exports the specifics of the lower layers to the application, and provides an API to control them, if the application is inclined to do so.

3.2 Session layer

Applications specify their notion of a session by explicitly joining together related transport-layer connections (or destinations in connectionless protocols). Once established, a session is identified by a locally-unique token, or Session ID, and serves as the system entity for integrated accounting and management. The session layer exports a unified session abstraction to the application, managing the connections as a group, adapting to changes in network attachment point as needed. The selection of network end point and transport protocol, however, remains completely under the application's control.

To assist in the timely detection of connectivity changes, the session layer accepts notification from lower layers (e.g., loss of carrier, power loss, change of address, etc.), the application itself, or appropriately authorized external entities that may be concurrently monitoring connection state [2]. Since a session may span multiple protocols, connections, destinations, and application processes, there may be several sources of connectivity information. Regardless of the source, the session manager handles notification of disconnection and reconnection in a consistent fashion.

3.2.1 Disconnection. If a host can no longer communicate with a session end point due to mobility, as signaled by changes in the network layer state, transport layer failure, or other mechanisms, it informs the application. If the application is not prepared to handle intermittent connectivity itself, the session layer provides appropriate management services, depending on the transport layers in use, including data buffering for reliable byte streams. Specifically, it may block or buffer stream sockets, selectively drop unreliable datagrams, etc. Additional application and transport-specific services can be provided, such as disabling TCP keep-alives.

Depending on the system configuration, the session layer may need to actively attempt to reestablish communication, or it may be notified by network or transport layers when it becomes available again. System policy may dictate trying multiple network interfaces or transport protocols. In either case, if the period of disconnection becomes appropriately long (as determined by system and application configuration), it will attempt to conserve resources by reducing the state required in the network, transport, and session layers (with possibly negative performance implications upon reconnection), and notify the application, enabling additional, domain-specific resource reallocation.

3.2.2 Reconnection. Upon reattachment, a mobile host contacts each of its correspondent hosts directly, informing them of its new location. Some transport layers may be unable to adequately or appropriately handle the change in network contexts. In that case, the session layer can restart them, using the session ID to re-sync state between the end points. In either case, the session layer informs the application of reattachment, and resynchronizes the state of the corresponding session layers.

The complexity of synchronization varies with the transport protocols in use; a well-designed transport layer can handle many things by itself. By using a transport-layer token, and *not* a network layer binding, the persistent connection model can provide limited support for changes in attachment point, often with better performance than higher-layer approaches [21, 24]. Similarly, the performance of even traditional transport protocols can be enhanced when the network layer exposes the appropriate state [3, 5]. Similarly, grouping multiple transport instances between the same end points into sessions can provide additional performance improvement [2, 22].

Legacy transport protocols may be completely unable to handle changes in network addresses. In that case, the session layer may initiate an entirely new connection, and resynchronize them transparently at the session layer. In the worst case, the application itself may be unable to handle unexpected address changes, and provide no means of system notification. Such applications are still supported via IP encapsulation. The correspondent session layers establish an IP tunnel to the new end point, and continue to send application data using the old address.

If a correspondent end point is no longer reachable (possibly because the other end point also moved), the application is instructed to perform another naming/location resolution operation in attempt to locate the previous correspondent, returning a network end point (host, protocol, port) to use for communication. The particular semantics of suitable alternative end points and look-up failure are application specific. It may be a simple matter of another application-layer name resolution (perhaps a fresh DNS query), or the application may which wish to perform its own recovery in addition to or in place of reissuing the location query.

While the amount overhead varies with the capabilities of the available lower layer technologies, overhead is incurred almost exclusively during periods of disconnectivity and reconnection. This provides high performance for the common case of communication between static peers.

3.3 State management

In a spirit similar to Coda, our architecture considers disconnection to be a natural, transient occurrence that should be handled gracefully by end hosts. For extended periods of disconnection, resource allocation becomes an additional concern. While managing application state is outside the scope of our architecture, enabling efficient strategies is decidedly not. In particular, since disconnection often occurs without prior notice, applications may require system support to reclaim resources outside of their control.

There has been a great deal of study on application specific-methods of dealing with disconnected or intermittent operation. Most of it has focused on providing continued service at the disconnected client, and has not addressed the scalability of servers. If our approach becomes popular, and disconnected sessions begin to constitute a non-negligible fraction of the connections being served, servers will need to free resources dedicated to those stalled connections, and be able to easily reallocate them later. We are considering a variety of state management services the session layer should implement, and briefly hypothesize about two: migrating session state between the system and application, and providing contextual validation of session state.

3.3.1 State migration. We believe the session abstraction may be a useful way to compartmentalize small amounts of connection state, reducing the amount of state applications need to store themselves, and simplifying its management. Furthermore this state could be tagged as being associated with a particular communication session, and managed in an efficient fashion together with system state [4]. System support may allow intelligent paging or swapping of associated state out of core if the period of disconnection becomes too long.

3.3.2 Context management. There is a significant amount of context associated with a communication session, and it may be the case that some (or all) of it will be invalidated by disconnection and/or reconnection. In particular, previous work has shown that context changes in the transport layer can be leveraged to adapt application protocol state [23]. Hence any state the session layer manages needs to be revalidated, possibly internally, possibly through application-specific up-calls. Changes in context may dictate that buffers be cleared, data be reformatted, alternate transport protocols be selected, etc. This requires a coherent contextual interface between the application and the session layer.

4 Related work

The focus of the Migrate architecture is on preserving end-to-end application communication across network location changes and disconnections. Much work has been done in the area of system support for mobility over the past few years; this section outlines the work most directly related to ours.

At the network-layer, several schemes have been proposed to handle mobile routing including Mobile IP [20] and multicast-based mobility [18]. Mobile IP uses a home agent as to intercept and forward packets, with a route optimization option to avoid triangle routing. The home-agent-based approach has also been applied at the transport layer, as in MSOCKS [15], where connection redirection was achieved using a split-connection proxy, providing so-called transport-layer mobility. Name resolution and message routing were integrated to implement a "late binding" option that tracks highly mobile services and nodes in the Intentional Naming System [1].

Most TCP-specific solutions for preserving communication across network-layer changes [21, 24] do not handle the problems associated with connections resuming after substantial periods of disconnectivity. A "persistent connection" scheme where the communication end-points are location independent was proposed for TCP sockets and DCE RPC [27], but the mapping between global endpoint names and current physical endpoints is done through a global clearinghouse, which notifies everyone of binding updates. Session layer mobility [14] explored moving entire sessions by utilizing a global naming service to provide endpoint bindings; address changes are affected through a TCP-specific protocol extension.

5 Conclusion

In this paper, we have defined five salient issues concerning host mobility in the Internet. We presented a set of design guidelines for building a system to address these issues, distilled from a decade of research in mobile applications and system support for mobility on the Internet. Following these principles, we outlined *Migrate*, a basic session-based architecture to preserve end-to-end application-layer communication in the face of mobility of the end points. We believe the general abstractions for disconnection, hibernation, and reconnection provided by the session layer define an appropriate set of interfaces to enable more advanced system support for mobility.

References

[1] Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., and Lilley, J. The design and implementation of an intentional naming system. In *Proc. ACM SOSP* (Dec. 1999), pp. 186–201.

[2] Balakrishnan, H., Rahul, H., and Seshan, S. An integrated congestion management architecture for Internet hosts. In *Proc. ACM SIGCOMM* (Aug. 1999), pp. 175–187.

[3] Balakrishnan, H., Seshan, S., and Katz, R. H. Improving reliable transport and handoff performance in cellular wireless networks. *ACM Wireless Networks 1*, 4 (Dec. 1995), 469–481.

[4] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: A new facility for resource management in server systems. In *Proc. USENIX OSDI* (Feb. 1999), pp. 45–58.

[5] Caceres, R., and Iftode, L. Improving the performance of reliable transport protocols in mobile computing environments. *IEEE JSAC 13*, 5 (June 1995), 850–857.

[6] Cerf, V. G., and Cain, E. The DoD Internet architecture model. *Computer Networks 7* (Oct. 1983), 307–318.

[7] Cheshire, S., and Baker, M. Internet mobility 4x4. In *Proc. ACM SIGCOMM* (Aug. 1996), pp. 318–329.

[8] Dahlin, M., Chandra, B., Gao, L., Khoja, A.-A., Nayate, A., Razzaq, A., and Sewani, A. Using mobile extensions to support disconnected services. Tech. rep., UT Austin, Apr. 2000.

[9] Gritter, M., and Cheriton, D. An architecture for content routing support in the internet. In *Proc. 3rd USITS* (Mar. 2001), pp. 37–48.

[10] Guttman, E., Perkins, C., Veizades, J., and Day, M. Service Location Protocol, Ver. 2. RFC 2608, June 1999.

[11] Inouye, J., Binkley, J., and Walpole, J. Dynamic network reconfiguration support for mobile computers. In *Proc. ACM/IEEE Mobicom* (Sept. 1997), pp. 13–22.

[12] Joseph, A. D., Tauber, J. A., and Kaashoek, M. F. Mobile computing with the rover toolkit. *IEEE Trans. on Computers 46*, 3 (Mar. 1997), 337–352.

[13] Kistler, J., and Satyanarayanan, M. Disconnected operation in the Coda filesystem. In *Proc. ACM SOSP* (Oct. 1991), pp. 213–225.

[14] Landfeldt, B., Larsson, T., Ismailov, Y., and Seneviratne, A. SLM, a framework for session layer mobility management. In *Proc. IEEE ICCCN* (Oct. 1999), pp. 452–456.

[15] Maltz, D., and Bhagwat, P. MSOCKS: An architecture for transport layer mobility. In *Proc. IEEE Infocom* (Mar. 1998), pp. 1037–1045.

[16] Milojicic, D., Douglis, F., and Wheeler, R. *Mobility: Processes, Computers, and Agents*. Addison Wesley, Reading, Massachusetts, 1999.

[17] Mummert, L., Ebling, M., and Satyanarayanan, M. Exploiting weak connectivity for mobile file access. In *Proc. ACM SOSP* (Dec. 1995), pp. 143–155.

[18] Mysore, J., and Bharghavan, V. A new multicasting-based architecture for internet host mobility. In *Proc. ACM/IEEE Mobicom* (Sept. 1997), pp. 161–172.

[19] Noble, B., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile application-aware adaptation for mobility. In *Proc. ACM SOSP* (Oct. 1997), pp. 276–287.

[20] Perkins, C. E. IP mobility support. RFC 2002, Oct. 1996.

[21] Qu, X., Yu, J. X., and Brent, R. P. A mobile TCP socket. In *Proc. IASTED Intl. Conf. on Software Eng.* (Nov. 1997).

[22] Savage, S., Cardwell, N., and Anderson, T. The Case for Informed Transport Protocols. In *Proc. HotOS VII* (March 1999), pp. 58–63.

[23] Snoeren, A. C., Andersen, D. G., and Balakrishnan, H. Fine-grained failover using connection migration. In *Proc. 3rd USITS* (Mar. 2001), pp. 221–232.

[24] Snoeren, A. C., and Balakrishnan, H. An end-to-end approach to host mobility. In *Proc. ACM/IEEE Mobicom* (Aug. 2000), pp. 155–166.

[25] Vahdat, A., and Dahlin, M. Active names: Flexible location and transport of wide-area resources. In *Proc. 2nd USITS* (Oct. 1999).

[26] Zenel, B., and Duchamp, D. A general purpose proxy filtering mechansism applied to the mobile environment. In *Proc. ACM/IEEE Mobicom* (Sept. 1997), pp. 248–259.

[27] Zhang, Y., and Dao, S. A "persistent connection" model for mobile and distributed systems. In *Proc. IEEE ICCCN* (Sept. 1995), pp. 300–307.

[28] Zhao, X., Castelluccia, C., and Baker, M. Flexible network support for mobile hosts. *ACM MONET 6*, 1 (2001).

Protium, an Infrastructure for Partitioned Applications

Cliff Young, Lakshman Y.N., Tom Szymanski, John Reppy, David Presotto,
Rob Pike, Girija Narlikar, Sape Mullender, and Eric Grosse
Bell Laboratories, Lucent Technologies
{cyoung,ynl,tgs,jhr,presotto,rob,girija,sape,ehg}@research.bell-labs.com

Abstract

Remote access feels different from local access. The major issues are consistency (machines vary in GUIs, applications, and devices) and responsiveness (the user must wait for network and server delays). Protium attacks these by partitioning programs into local viewers that connect to remote services using application-specific protocols. Partitioning allows viewers to be customized to adapt to local features and limitations. Services are responsible for maintaining long-term state. Viewers manage the user interface and use state to reduce communication between viewer and service, reducing latency whenever possible.

System infrastructure sits between the viewer and service, supporting replication, consistency, session management, and multiple simultaneous viewers. The prototype system includes an editor, a draw program, a PDF viewer, a map database, a music jukebox, and windowing system support. It runs on servers, workstations, PCs, and PDAs under Plan 9, Linux, and Windows; services and viewers have been written in C, Java, and Concurrent ML.

1 Introduction

In the 1970's, one could walk up to any telephone in the world and use it as easily as one's home telephone. The computer revolution might have followed suit, but the opposite holds. It is nearly impossible to use the neighbor's computer: data files are unavailable or not available in a consistent place, the wrong applications are installed, and the preferences for those applications are personalized.

Why can't one use any computer on the planet? The reasons are historical and economic. First, the last twenty years of the computing industry have been about *personal* computing. Mainframe and minicomputer users shared a consistent (if sometimes unresponsive) environment in their single, shared system. In contrast, every PC is customized, binding large amounts of state to each PC. The move to PCs gave users responsiveness at the cost of consistency. Secondly, networked computing environments work "well enough" within a single security domain such as a corporate intranet or university-wide network. Such environments do allow users to log into multiple terminals, and this covers a large percentage of shared-use cases. Third, remote access tools are "good enough": users are willing to dial into or tunnel to their corporate intranet to get access, despite added latency or inconsistency. Lastly, the Internet grew up only in the last decade; before that one couldn't think of being connected to every computer in the world. A new generation of computing devices is upon us; each person will have many devices and each person will expect the multiple and remote devices to work consistently. We will have to rewrite all of our applications for the new devices anyway; why not rewrite them so they work better?

Our goal is to be able to use any Internet-connected device as if it were the machine in our office. Further, we want this access consistently and responsively. Consistency is similarity of experience across devices. The user's session must migrate from device to device. Each application will adapt to each end device so that it exploits the device's unique capabilities and works around the device's limitations. Responsiveness implies that remote access is as comfortable as a local application. Many remote access systems have addressed one or the other of these two goals; few address both.

Protium splits applications into two pieces. One runs near the user; the other runs in a service provider that is highly available, has persistent storage, and has abundant computation cycles. We call these pieces *viewers* and *services*, respectively, to emphasize that state is maintained by the service. Viewers and services communicate via an *application-specific* protocol; the application designer must partition the application to maximize consistency and responsiveness. Applications are built as if only a single viewer-service pair existed, with certain additional constraints. These constraints allow the Protium infrastructure to support connection, reconnection, multiple simultaneous viewers, state consistency and replication, and session management.

The Protium prototype includes a text editor, a MacDraw-style drawing program, a PDF viewer, a map viewer with map database, a digital music jukebox, and windowing system support. Viewers and services have been written in C, Java, and Concurrent ML and run under the Plan 9 Operating System, inside Java applets, and under Linux. All of these viewers and services interoperate.

2 A Better World

We'd like to be able to work all day at the office, using a research operating system such as Plan 9 or Linux. At the end of the day we'd like to walk away from the office computer, possibly with state uncommitted to stable storage. On the train home, we'd like to be able to pull out a wireless PDA or cellular phone, and have the portable device replicate the office session. PDA-specific viewers for each of our applications will be launched that show the exact same state, including uncommitted changes, as the work session back in the office. The PDA is limited, but one could imagine reading drafts of a document or fixing typos even on its small screen and using its limited input capabilities. When we get home, the home computer runs only Windows. But using a web browser and Java applets we again replicate the session, getting Java versions of each of our applications with the same session state as we had at the end of our train ride. In each remote case (PDA and Java), the applications respond immediately to user input; updates spool back to the office server.

Our two consistency-related goals are *session mobility* and *platform independence*. The example shows session mobility where the user accessed the same state of applications using three different systems. We will not supply a precise definition of “session” in this paper; however, an intuitive definition would be the state of all applications currently open on one's workstation screen. Platform independence involves accessing the same session on a variety of devices and operating systems: a workstation running a workstation OS, a PDA with its proprietary OS, and a Windows home PC with a standard browser. And to make things difficult, we will not sacrifice responsiveness for these consistency goals.

This example sounds like typical ubiquitous computing propaganda, but our system concretely provides them: we have a prototype system running. We next describe the assumptions about future technologies that underlie our engineering choices, then go on to describe our approach and prototype system. Before concluding, we explain why prior approaches fail some of our requirements.

3 Assumptions

We make some assumptions about the future. On the technical front, Moore's law continues, exponentially improving processing power, memory sizes, device sizes, heat dissipation, and cost. As a corollary, the world will move to multiple devices per person. Bandwidth will increase in the backbone network and will increase (albeit less quickly) to portable and home devices. Wired or wireless remote coverage will improve over the next decade; we thus choose not to focus on disconnected operation. However, we also assume that communications latency will *not* improve much in the next decade.

While the speed of light places a fundamental lower bound on communications, it takes light only about 130 milliseconds to go around the world. Our latency assumption instead rests on the current realities in data networking, where differentiated services have not yet been deployed and switching delays are significant. Even with a speedy core Internet, however, it seems believable that last hop communications services would still experience notable delays for the next decade (today, we regularly experience *10 second* round-trip times on CDPD modems and WAP cellular phones; systems must respond within 100 milliseconds to feel instantaneous and within 1.0 second not to disrupt the user’s flow of thought [3]). Furthermore, 130 milliseconds is very long for a computer, so services that rely on other services still face latency issues.

On the social front, we have two assumptions. First, we believe that there will be a new round of operating system wars for the PDA/cellphone market. The market will determine which (if any) of the current contenders (PalmOS, Windows CE, Psion, to name a few) will win. In the meantime, we should deploy systems that work well regardless of programming language or operating system.

Our second social assumption is that distributed programming is hard. If we can find ways to sweep many of the traditional distributed programming problems under the rug of our infrastructure, the average programmer might be able to write a robust distributed application. A secondary goal is that writing an application in our system will not be much harder than writing a standalone GUI-based application is today.

4 Partitioned Applications

Our approach draws its inspiration from two applications that work when a low-performance channel connects the user and his data: the Sam text editor [4] and the IMAP mail protocol [1]. Sam comes in two pieces. Sam’s service runs near the file system where editing takes place; Sam’s viewer runs on whatever device the user has at hand. Viewer and service communicate using a protocol that keeps track of the state of both halves. IMAP works similarly but for mail instead of editing. Both applications divide the task into two parts: a service that is highly available and has large compute and storage resources, and a viewer that needs a connection to the service and some kind of user interface but need not download the entire program state. Perhaps the central question of our project is: can we generalize from Sam and IMAP to all applications? And can we build infrastructure that makes this easy? We call this approach, “partitioned applications,” because the network breaks the application into parts.

Another way of looking at Protium is that we are “putting the network into the application.” Most previous approaches divide remote from local at an existing abstraction layer, for example the file system, the GUI API (The X Window System [6]), or the frame buffer. Partitioning incorporates these prior approaches; it just adds a new dimension of flexibility.

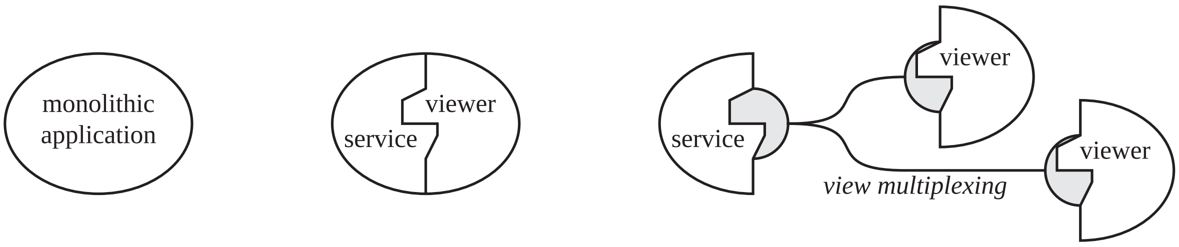


Figure 1: A monolithic application, the same application after partitioning, and multiple viewers connected to a single service through a view multiplexer.

Partitioning induces rich systems issues. For example, what manages a session? How does one connect or reconnect to a service? What maintains consistency, replicates state, or provides multicasting across simultaneously active viewers? What happens when a viewer crashes or the viewer device is lost? Our prototype system suggests preliminary answers to each of these questions.

5 Prototype System

Our prototype system currently supports five applications in addition to the session service/view manager. These are a simple text editor, a MacDraw-style drawing program, a PDF image viewer, a map program (with both photographic images and polygonal graphics), and a music jukebox. These represent a variety of interesting desktop applications, so we are encouraged that we have been able to build them with our current infrastructure. However, for us to really claim that we are building a general system, we need to build more applications. We are investigating video and hope to add PDA applications (calendar, email, address books) to our suite.

One of the most interesting research issues involves adapting viewers to the platform on which they run. There are at least four different kinds of platforms: big bitmaps (desktops and laptops), small bitmaps (PDAs and cellphones), text, and voice. We present some preliminary results about device-specific adaptation in the section on session management, but this topic remains largely untouched.

Building applications around a protocol gives us a high degree of language and operating system independence. Our prototype applications run under Plan 9 (all viewers and the edit and draw services), Java (all but the PDF viewer; map and juke services), and Linux (draw and PDF services). The Plan 9 programs were written in C; the Linux programs were written in Concurrent ML. Porting remains a significant task, but this wide variety of languages and systems supports a claim to language and OS independence.

Just rewriting applications into two pieces doesn't make a systems project. For Protium, the interesting issues are in the infrastructure, and we describe two pieces of the infrastructure here, followed by an application example. The first piece of infrastructure, the view multiplexer, supports multiple viewers on a single service, while simulating a connection to a single counterpart to each viewer or service. The second piece of infrastructure, the session service, bundles together multiple services into a session; it has a corresponding piece, the view manager, which runs on the viewing device. After describing the pieces of infrastructure, we will go on to an application example, our map program.

5.1 Multiplexed Viewers

Each viewer or service is designed as if it spoke to a single counterpart service or viewer, respectively. But we want to be able to support multiple viewers simultaneously connected to a single service. The view multiplexer simulates a single viewer to a service. New viewers that wish to connect to a running service do so through the view multiplexer, so the service need not be aware that a new viewer has connected. Figure 1 shows a view multiplexer interposed between a service and multiple viewers.

To do its job, the view multiplexer snoops the messages between service and viewer. Each message in the system has a tag to help the multiplexer. Most communication is synchronous from viewer to service, in viewer-initiated request-response pairs. Viewers can generate read, lightweight write, and heavyweight write messages. Services respond with either acknowledgements (ACKs) or negative acknowledgements (NACKs); the infrastructure is allowed to NACK a message without allowing the message to reach the service. Viewers must also be able to handle asynchronous update messages, which are generated by the infrastructure when one viewer receives an ACK; an update tells a viewer that some other viewer succeeded in updating the state. Since all viewers see all ACKs, they can keep their views of the state up-to-date. Lastly, services can asynchronously broadcast to all viewers; broadcast messages support streaming media. This consistency model is similar to publisher-subscriber consistency models.

In addition to multicasting ACKs (as updates) to all viewers, the view multiplexer helps build responsive viewers. Using a simple token-passing scheme, the view multiplexer allows one viewer to become privileged. Lightweight writes from the privileged viewer are immediately ACKed; this allows the privileged viewer to deliver local response time. These writes must then be propagated to the service and the other viewers; formal

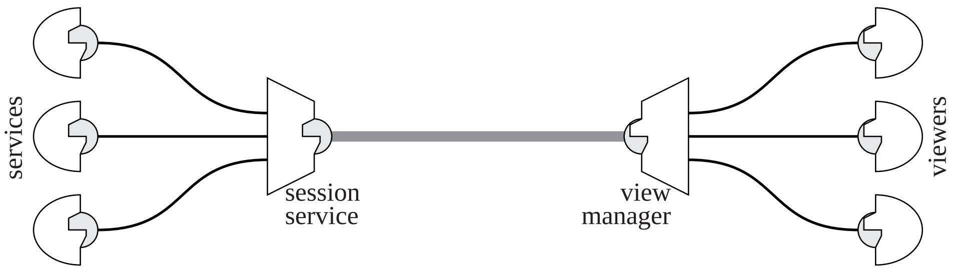


Figure 2: Three applications (with service and viewer pieces) connected through the session service/view manager multiplexor/demultiplexor pair.

requirements of the protocol and the service implementation guarantee that the service will acknowledge the write. Lightweight writes should be common actions that do not require support from the service, e.g., responding to keystrokes or mouse events. Global search-and-replace or commit to stable storage should be heavyweight writes. Token management matches our intended uses, where a single user expects immediate response from the device he uses but can tolerate delayed updates in other devices.

5.2 Session Management

Intuitively, a session is the state of one's desktop. The Protium *session service* runs on the service side and bundles together multiple services into a single session. A device-specific *view manager* connects to the session service and runs the viewers that correspond to the services in the session. The session service and view manager form a multiplexer/demultiplexer pair, linking multiple viewers to multiple corresponding services. In our introductory example, the view manager launched the viewers on the PDA and under the Java-enabled browser.

The session service and view manager behave as another application pair, so the pipe between them can be managed by the view multiplexer just like for any other application. The session and view managers can also hierarchically encapsulate and route messages for the underlying service/viewer pairs; however, services and viewers are free to communicate out-of-band if the designer so chooses. Figure 2 depicts a set of applications managed by a session service and view manager. We can compose view multiplexers and session/view manager pairs in arbitrary nesting; the two sets of multiplexers recurse.

In addition to managing the set of applications, the view managers adapt window system events to their devices. Moving or resizing a window on a big screen causes a corresponding move or resize on another big screen. Moves and resizes have no small screen analog (because applications typically use the whole screen); however, focus changes and iconification work similarly on big and small screens. We have not yet implemented the text-only or voice-based viewers, so we have no experience in this area; a text-only view manager should work like a shell. Another open problem is adapting to differently sized big screens.

5.3 A Protium Application: Map

Some of us (Szymanski and Lakshman) are interested in geographic data such as terrestrial maps, aerial images, elevation data, weather information, aviation maps, and gazetteer information; one goal is synthesizing these views coordinated by positional information. Gigabytes of data come from scattered sources and multiple servers. We had built a Java geodata viewer that could navigate and edit the data. Different types of geodata show as selectable display layers. Different layers or different parts of the same layer may be served by different machines. However, all servers and the viewer have the same abstract view of the data and hold some piece of the data locally. This uniform view results in a simple protocol between the viewer and the servers.

We used the Protium infrastructure to share the geodata viewer, the basic idea being that multiple viewers can share a session, moderated by a session server, that tracks what is being viewed and tells viewers what data to get and from where. Actual data travels out-of-band; only control messages route through the Protium infrastructure. The session server also supports textual messaging so that a shared viewer can be used to give driving directions to someone (with appropriate maps and messaged instructions) at a remote location.

The map application is designed to hide latency from the user. For example, the viewer displays street names and addresses in a tool tip; a remote query would make this feature too slow and too variable in latency. All geographical data is kept in a tiled format, compressed using a method appropriate to its type, and transmitted to the viewer upon demand or (sometimes) before. The map viewer stores this data in a two-level cache in which the lower level (which counteracts network latency) contains compressed data, and the upper level (which counteracts decompression latency) contains fully expanded data structures needed to support user interactions. Requests to map services are executed in batches that are satisfied in an out-of-order fashion. This overlaps server processing with both network transmission time and client decompression time. This architecture provides a degree of responsiveness that could not be approached with a conventional browser/server structure.

As part of this exercise, we implemented a new viewer in C under Plan 9. The Java and Plan 9 viewers differ greatly: the Plan 9 viewer targets the small but colorful iPAQ display and uses pen input; the Java viewer runs on big screens and uses the real estate for a complex GUI.

The session server (excluding marshalling code) is about 330 lines of Java. An additional 286 lines allow the existing viewer to interoperate with the session server. The Plan 9 viewer required 3527 lines of C of which about 300 lines deal with communication and the rest deal with graphics and event handling. Thus, with a small amount of effort, we were able to convert an existing single-user application (which was already split into service and viewer parts) into a shared application.

6 Remote Access

The body of related work is far too vast to survey in a position paper; remote access systems span many disciplines including operating systems, networking, distributed systems, and databases. This section instead highlights major approaches to remote access and explains why they do not meet our goals.

Most remote access systems fail one or both of our consistency and responsiveness requirements. Rlogin and its more secure modern descendant, ssh [7], are platform independent but do not provide session mobility. Distributed file systems (examples abound; AFS, Coda, and Locus/Ficus to name a few) allow one to access stable storage wherever the network reaches but say nothing about how to provide applications. Remote Procedural Call packages similarly do not show how to provide applications. Distributed object frameworks such as CORBA and DCOM address the same problem as we do, but suffer performance problems because their abstraction of remote and local objects hides latency from designers [8]. Recent work in thin-client computing and its predecessor, client-server computing, give some forms of consistency but force clients to wait during both network and server latencies.

A number of systems apply the brute-force approach of sending screen differences and raw user input information across the network. Examples include Virtual Network Computing (VNC) from AT&T Research [5], the SunRay product from Sun Microsystems, Citrix System's Windows-based product, and Microsoft's NetMeeting. All of these systems provide bit-for-bit consistency but suffer when network latency increases. They also do not adapt to the constraints of local devices: viewing large virtual screens on small physical devices is difficult, and the system architecture prevents further device-specific adaptation.

Philosophically, the Berkeley Ninja project is closest to our approach [2]. We follow Ninja's approach of keeping stable storage in a service provider (Ninja calls this a "base") and allowing "soft" state in the viewers to be lost. Ninja focuses on scalable services; Protium focuses on the applications we use daily on the desktop. Protium's infrastructure works primarily between service and viewer.

7 Experiences

To partition an application, one must focus on the application-specific protocol. We would like to present a how-to guide on partitioning, but the lessons so far sound like platitudes, including "match messages to user input", "separate control and data", and "beware round trips." In an early version of the draw protocol, each object deleted required a separate message. If the user selected a number of objects and issued a delete command, some of the deletes might fail, leaving the application in a confusing state. The juke, map, and PDF applications have large data streams (music, graphics/image, and image); waiting for a large object to be transmitted can keep a small control message from taking effect. Designing protocols without a delay simulator is dangerous: what works acceptably on a LAN may be unusable with a 1-second round-trip time. We hope to be able to summarize and illustrate more such principles in the future.

Protocol designers must decide which application state is viewer-specific and which is service-mediated. For example, the edit application keeps scroll bar position and text selection local to the viewer. The draw application tries to do the same, but some operations (grouping, ungrouping, and deleting objects) reset the selection to be consistent across all viewers. More ambitiously, it might be useful to be able to preview the next PDF page on one's remote control device while continuing to show the current page on the video projector device, but this is not yet supported by the PDF protocol. Some applications have added state expressly for collaborative or remote-control purposes: the map and PDF programs both support telestrator-style overlays, and the map program also includes a chat room.

Writing viewers is harder than we would like. Viewers include all of the state and complexity of a traditional stand-alone application, augmented by the complexity of managing a single outstanding request while being able to accept asynchronous updates and broadcasts. Backing out attempted changes when a NACK arrives further complicates design. All of our viewer programs are multi-threaded; this seems a higher standard than we would care to impose on the average programmer. We are exploring programming idioms, APIs, and library support that might simplify viewer development.

8 Discussion and Conclusion

This system is not about the next killer application. If anything, we are rebuilding all of our old applications to live in a new world. This follows our biases as system builders: we know how to build infrastructure. If this project or one like it succeeds, we will have universal data service, like universal telephone service. The new devices require rewriting all of our old applications anyway. We might as well get some benefit out of it.

The Protium approach makes additional demands on application programmers. The initial designer of an application creates an application-specific protocol, while designers of new viewers or services must adhere to that protocol (if our project succeeds, then perhaps standards for application protocols will emerge). Porting an application to a new platform involves at least porting the viewer. Building a viewer combines both traditional GUI issues and communicating back to the service. Services may also need to be ported.

What is the best way for Protium to support existing applications? It depends on the application. Programs with clean separation between display and state integrate easily with Protium; most programs, however, are large, complex, and have tangled state- and display-management code. We observe that the move to new devices such as PDAs and phones will force such programs to be rewritten anyway; integrating the program with Protium as part of the rewrite will be a modest extra requirement and will benefit the application by making it use the network more effectively.

Considering applications in a partitioned context provides new opportunities to use old tricks. Persistence is a service-only problem; the service need not worry about geographic distribution, so known persistence techniques apply. Viewers that are lost or lose state are easily replaced or restored because the service is the repository of record. The connection between service and viewer can be a network socket; known techniques for authentication and encryption therefore apply. Security, logging, caching, and prefetching seem like obvious features to add. This paper concentrates on the single user; Protium also gives limited support for collaboration and remote control. We think of these as bonuses rather than our primary research goal; it seems a high enough goal to be able to use any computer in the world.

Protium is the most common isotope of hydrogen, the most common element in the universe. A protium atom has two pieces that are closely coupled and essential to the nature of hydrogen, but the two pieces are different from each other. And while the two pieces are themselves basic, the exploration of their interaction has occupied scientists for more than a century.

9 References

[1] M. Crispin. Internet Message Access Protocol – Version 4rev1. RFC2060 (December 1996).

[2] S. D. Gribble, et al. The Ninja Architecture for Internet-Scale Systems and Services. To appear in a *Special Issue of Computer Networks on Pervasive Computing*.

[3] R. B. Miller. Response time in man-computer conversational transactions. *Proc. AFIPS Fall Joint Computer Conference*, 33:267–277, 1968.

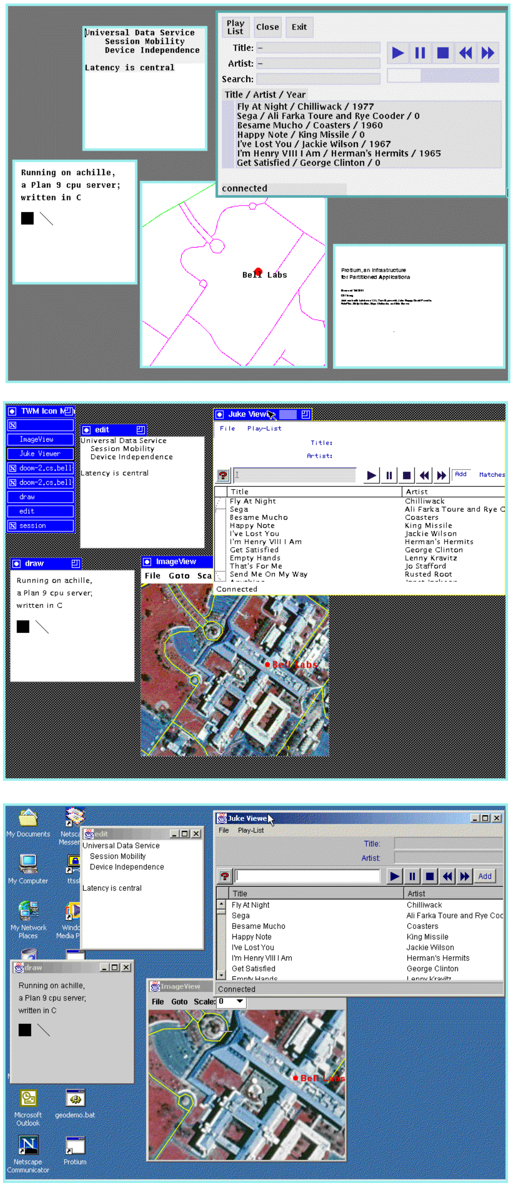
[4] R. Pike. The text editor Sam. *Software Practice and Experience*, 17(11):813–845, 1987.

[5] T. Richardson, et al. Virtual Network Computing. *IEEE Internet Computing*, 2(1): 33-38, Jan/Feb 1998.

[6] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–106, Apr. 1986.

[7] T. Ylonen. The SSH (Secure Shell) Remote Login Protocol. In *Internet Drafts* (November 1995).

[8] J. Waldo, et al. A Note on Distributed Computing. In *Lecture Notes in Comp.Sci. 1222*, Springer, 1997.



```
Universal Data Service
    Session Mobility
    Device Independence

Latency is central
```

```
Running on achille,
a Plan 9 cpu server;
written in C
```

Title: –
Artist: –
Search:

Title / Artist / Year
Fly At Night / Chilliwack / 1977
Sega / Ali Farka Toure and Rye Cooder / 0
Besame Mucho / Coasters / 1960
Happy Note / King Missile / 0
I've Lost You / Jackie Wilson / 1967
I'm Henry VIII I Am / Herman's Hermits / 1965
Get Satisfied / George Clinton / 0

connected

Protium, an Infrastructure for Partitioned Applications

Title	Artist
Fly At Night	Chilliwack
Sega	Ali Farka Toure and Rye C
Besame Mucho	Coasters
Happy Note	King Missile
I've Lost You	Jackie Wilson
I'm Henry VIII I Am	Herman's Hermits
Get Satisfied	George Clinton
Empty Hands	Lenny Kravitz
That's For Me	Jo Stafford
Send Me On My Way	Rusted Root

Connected

Figure 3: Screen shots of Protium applications under Plan 9 (top), Linux/Java (middle), and Windows/Java (bottom). Within each screen, the applications are edit (top left), draw (left), juke (top right), map (bottom), and PDF (bottom right, Plan 9 only).

Probabilistic Modelling of Replica Divergence

Antony I. T. Rowstron, Neil Lawrence and Christopher M. Bishop
Microsoft Research Ltd.
St. George House, 1 Guildhall Street,
Cambridge, CB2 3NH, UK.
antr@microsoft.com

Abstract

It is common in distributed systems to replicate data. In many cases this data evolves in a consistent fashion, and this evolution can be modelled. A probabilistic model *of the evolution allows us to estimate the divergence of the replicas and can be used by the application to alter its behaviour, for example to control synchronisation times, to determine the propagation of writes, and to convey to the user information about how much the data may have evolved.*

In this paper, we describe how the evolution of the data may be modelled and outline how the probabilistic model may be utilised in various applications, concentrating on a news database example.

1. Introduction

In distributed systems the replication of shared mutable data has been widely studied. When mutable data is replicated there is a need to consider the consistency model used to control the level of divergence of the different replicas.

In this paper, we advocate using knowledge of how the shared data evolves to control and manage divergence. Empirical evidence shows that updates to shared data, in many cases, follow systematic patterns. By modelling the way in which the data has been updated in the past, we can provide information to an application on how, the data has evolved since the replicas were last consistent. The basis of this approach is *probabilistic modelling* applied to the distribution of operations performed on the data structure. The approach is novel and preliminary results on a mobile news database and a mobile email reader are encouraging.

In the next section we describe the general approach, in Section 3 a mobile news database case study is detailed, in Section 4 the results for a mobile email reader are presented and then in Section 5 we describe other applications we are currently working on.

2. The General Approach

An application using our approach is provided with probabilistic models that capture how a replicated data structure evolves in time. These probabilistic models allow the application to estimate the number of updates that are likely to have been performed on the data structure, or part of it, during a specified time period, for example between the last time a synchronisation was performed and the current time. The application can then use this to adapt to the data structures' evolution by, for example, controlling when synchronisations should occur, alerting the user to divergence, or controlling when updates to the shared data are propagated. The generation of a single probabilistic model that captures the evolution of the other replicas, is known as *inference*. The application then makes *decisions* based upon the information contained within this single model. This partition of the problem into two stages of inference and decision enables our approach to be applied to a wide variety of applications. The inference stage is decomposed into the generation of models representing the evolution of the replicated data structure, or parts of it, and the subsequent combining of these models as requested by the application. In the For the inference stage a general purpose tool can be used to create the probabilistic models, and combine them, whilst the decision stage is specific to each application.

The probabilistic models are generated by a tool, which requires a log of descriptions of the operations performed on the shared data structure. For each update to the data structure the log contains: information about the operation, the part of the data structure that was affected, the time when the operation was performed and an identifier representing the source of the update (for example a user id). A description of the data structure and its different components is also required by the tool, which allows each component of the data structure to be modelled independently. Once a set of probabilistic models have been created, these can be updated dynamically as new updates are performed.

As a simple example of the data structure decomposition, consider an address database application. Each entry within the database is marked as either being a personal contact or as a business contact. This data structure can be thought of as being composed of two parts, the personal and the business parts, and two models can be generated and provided to the application. A model for the entire address database can then be generated by combining the two models. A further sub-division could exist within the database with, perhaps, the personal database is divided into family and friends. Separate probabilistic models can then also be generated for each of these sub-divisions and again composed.

The application is required to create the logs, provide the information about the decomposition of the data structure, and to perform the decisions based upon the information provided by the models.

Probabilistic Modelling Learning of the probabilistic models can be automated using model selection techniques, and the models may also be changed over time as more updates are made to the replicas.

A probabilistic model is a particularly powerful representation as such models may be combined in a straightforward and principled manner. This means that the data structure can be decomposed and each part modelled individually, as can different sources of data updates. For example, in the address database application, there could be separate models for the secretary and the owner of the address book, reflecting their particular patterns of updates. Hence, when the address book is replicated for the owners use, the probabilistic model generated can describe how the secretarys copy evolves.

It is important to remember that the probabilistic model is a prediction of future behaviour based on the past. The true evolution of the replica may not be consistent with the model. Even if the model is correct, its probabilistic nature means that its individual predictions can err, even if in general it is accurate. As a result we advocate using our approach in ways that will enhance the user experience rather than restrict functionality. The user interface should suggest and advise rather than constrain and command.

The System The System is composed of a tool for creating the probabilistic models, and a library for use in the application for merging the probabilistic models. These models capture the rate at which the operations are performed, and how that rate changes over time. Therefore, the time at which an update to the data structure occurs is the primary information required to create the models. The other information in the log allows multiple models to be created, based on the part of the data structure being updated, or on the user performing the update. In order to achieve this, the tool pre-processes the log, creating a separate log for each entity to be modelled. A probabilistic model is then created for each of these sub-logs independently, and this is now described.

There are a number of factors that effect the creation of the models. For example, the periodicity of the data has to be determined (e.g. hourly, daily, weekly, monthly and so forth). The tool currently creates histogram based models. Such models may be parameterised by widths and starting points for the bins. All the parameters of the model can be learned from the information contained within the log. It should be noted that there are many alternative forms of probabilistic model which can be used, for example wrapped mixtures of Gaussians and circular normals (see [5]). Although in this paper we use histogram based models, we are currently evaluating other approaches.

For each probabilistic model the correct parameters need to be established, and these control the model complexity. The main consideration in the selection of the model complexity is its generalisation ability. In other words, we wish to create a model that not only describes well the updates upon which it is based but also one that will describe future updates. In the address book example above, where the events we are modelling are updates of the database, we could create a histogram model with too many bins so that each update occurs in a single bin. Such a model is unlikely to provide a good predictor of the evolution of the replica because the model complexity is too high. At the other extreme, if we create a histogram model with only one bin we will be predicting a uniform distribution for the future updates, again this is likely to be a poor predictor of the replica's evolution. There is obviously a 'happy medium' and this may be found through *cross-validation* of the model [1]. Cross-validation involves splitting the log into a number parts, for example five. The first four parts are then used to construct a model with a particular parameterisation and the fifth part is used to 'validate' the model. This involves computation of the histogram models likelihood of creating the validating data. The part that is used for validation and one of those used for construction is then inter-changed and the model is re-validated. This procedure is repeated five times so that each part of the data has been used to validate the model once giving five different scores. The validation scores are then combined, for example by averaging, and the final score is associated with the parameters used for constructing the model. A range of parameterisations can be tested in this manner and the one with the highest score is then selected, and utilised to construct a model based on all the data, which is the final model.

Another factor determined during the cross-validation phase is the periodicity of the updates. The tool uses a number of pre-programmed periodicities: a daily cycle, a weekly cycle, weekdays separately generated from weekends, and Saturdays separately generated from Sundays both of which are separately generated from weekends.

More pre-programmed periodicities can easily be added, such as hourly or monthly based periodicities. Note that the set of candidate models includes the uniform distribution, and so the performance of the system should be no worse that that of the uniform model, in the event that one of the pre-programmed periodicities is not appropriate. Currently, we are looking at other techniques to determine the periodicity of the updates.

A *prior distribution* is used, which can either be a *uniform prior* or, in some situations, there may be prior knowledge about when the updates arrive. The prior distribution is combined with the model generated using Bayes's rule. For a histogram model this prior plays an important role of ensuring the final histogram model is non-zero at all points within its range, i.e. even when there are no observed points within a bin's range the histogram has some residual value. The residual value of the histogram is a further cross-validated parameter. If there is no or little information about when the updates occur, this is valuable, because the model is initialised using the prior distribution and as more updates are observed, the model is refined to represent more accurately the underlying distribution of the updates. This process can be extended to allow more recent data to be more influential, thereby allowing the system to deal with non-stationary situations in which the distribution is itself evolving with time.

3. Example Mobile News Database

We now demonstrate the use of our approach in a mobile news database application. We are seeing a proliferation of applications that replicate information on mobile devices, such as the service provided by AvantGo[1]. These allow mobile devices to store replicas of small news databases for access when the mobile device is disconnected.

Our mobile news database application provides, on a mobile device, a list and summary of the current news stories. We assume that the mobile device has wireless connectivity which allows it to synchronise with the master news database. We assume a pull model, where the device initiates the connection.

For this application a database of new articles is required, and we generated one from the BBC News web site. The BBC publishes news articles to their web site 24 hours a day and each of the news items is classified under a category, such as sport, business, health and so forth. For every article appearing on the BBC News website over a three month period we extracted the date and time of publication, and the subject of the news article to create a news database.

We treated the news database as the replicated data structure, and used the information gathered from the website to create the log required to generate the probabilistic models. We decomposed the news database into several parts, where each subject was treated as a separate part. All writes to the news database were considered as being performed by a single user. The mobile news database application allowed the user to select which parts of the news database they wished to have replicated on the mobile device.

[1]http://www.avantgo.com/

The probabilistic models of the news database are created by the tool overviewed in the previous section. The mobile news database uses the probabilistic models to generate a visual cue in the application interface to allow a user to see the number of updates that are likely to have occurred to each part of the news database since the last synchronisation. The application also uses the probabilistic models to control synchronisation times between the device and the master news database. It is likely that, due to the cost of bandwidth, as well as the limited amount of bandwidth, the mobile devices will not be continuously connected. Therefore, the synchronisation times have to be chosen, as this is part of the decision stage.

Optimal synchronisation The obvious approach to choosing when the mobile device should synchronise would be to have a user specify the number of synchronisations per day they were willing to pay for[2], and these would occur uniformly during the day, for example once every four hours.

Our mobile news database makes an adaptive choice of when to synchronise. This aims to find a trade-off between the cost of synchronisation and the average staleness of the data, where staleness is defined as the time between an article appearing on the master news database and appearing on the device.

In order to calculate the synchronisation times, it is necessary to formalise the user's requirements and calculate how to achieve them. In the mobile news database this is achieved either by setting the number of synchronisations per day to achieve a particular average staleness, or by allowing the user to set the number of synchronisations per day and then scheduling the synchronisation times to minimise the staleness.

We express the user's preferences in terms of a *cost function*, which represents mathematically what the user wants. In the news database problem, the simple cost function we have chosen is one which represents the staleness of the news articles. We wish to minimise the time that articles are available in the news database, but are not available on the mobile device. For every article the staleness is the time from when the article was available but not in the replica on the mobile device.

[2]Assuming a per synchronisation charge; other charging models are possible and different cost functions can be created to deal with this in our approach.

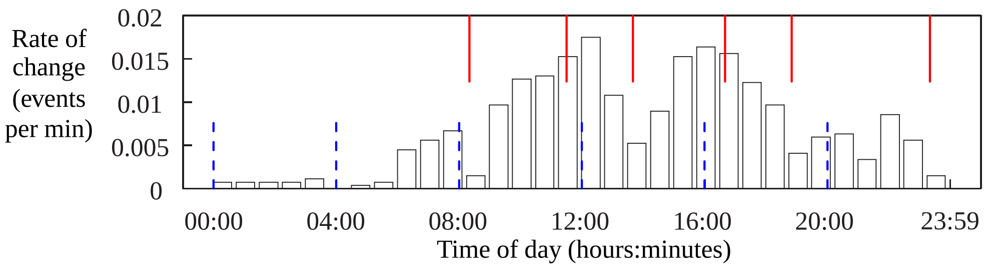


Figure 1. Synchronisation time optimisation for the Business part of the news database, showing the learned histogram model together with uniform synchronisation times (dashed lines) and the optimised synchronisation times (solid lines).

The cost incurred from synchronising at time s_i may be written

$$C = \sum_{n=1}^{N}(s_i - u_n), \tag{1}$$

given that N articles have arrived since the last synchronisation at times u_1 to u_N. We wish to find a synchronisation time which minimises this cost. Unfortunately we don't know when the updates will arrive, we can only estimate the rate of arrival using our probabilistic model, so we need to minimise the *expected cost*.

Consider how the expected cost will depend on three ordered synchronisation times s_{i-1}, s_i and s_{i+1}:

$$\mathcal{C}(s_{i-1}, s_i, s_{i+1}) = \int_{s_{i-1}}^{s_i} \lambda(t)(s_i - t)dt + \int_{s_i}^{s_{i+1}} \lambda(t)(s_{i+1} - t)dt, \tag{2}$$

where $\lambda(t)$ is a time varying function representing the estimated rate at which updates are occuring at the master news database, and is obtained from the probabilistic model. The first term in Equation 2 is the expected cost that will be incurred when we synchronise at time s_i. Note the inclusion of the second term, which is the expected cost that will be incurred when we synchronise at time s_{i+1}. This cost also depends on s_i.

We can now minimise the expected cost with respect to each s_i given the neighbouring synchronisations. This may be done through an iterative algorithm where passes are made through the proposed synchronisation times optimising each one in turn until convergence is achieved.

An alternative to minimising staleness is to maintain the same level of staleness that could be achieved using the uniform approach, but to achieve this using fewer synchronisations per day. This has the benefit of reducing the number of messages required (potentially reducing the financial cost of using the system), and has implications for saving power.

3.1. Results

Figure 1 shows some of the elements of our approach. The histogram based probabilistic model for weekdays for the business part of the news database is shown as boxes on the graph, generated using the updates occurring in May and June 2000. The tool automatically determines the periodicity of the data, and for the business part of the news database this is a weekday and weekend periodicity. Therefore, the weekdays are mapped into one twenty-four hour period and created a histogram to represent that twenty-four hours, and this is shown in Figure 1 (there is a separate model for the weekend which is not shown here). Six synchronisations were requested per day, and the vertical solid lines in Figure 1 show the optimal synchronisation times, to minimise the staleness. The vertical dotted lines in the lower half of the graph identify synchronisation times as taken from a 'uniform' strategy that synchronises every four hours.

Table 1 presents some results for our news database application, showing the staleness achieved when each of the four named databases is replicated individually, and when they are all replicated. It shows the average time in minutes between an article becoming available on the master news database and appearing in the replica on the mobile device, for articles arriving on weekdays in July 2000. The figures show the results when six synchronisations per day were used, with both the uniform and adaptive synchronisation times. The uniform approach started at midnight, as is shown in Figure 1[3]. The percentage decrease in staleness for adaptive over uniform is shown. In the final column the number of synchronisations required by the adaptive approach to achieve a similar average staleness of articles as the uniform approach is given, with the observed average staleness shown in brackets afterwards.

[3]It should be noted the effect of starting the uniform synchronisation at other times does not impact the results significantly.

Classification	Staleness (mins)		% Decrease in staleness for adaptive over uniform	Number of synchronisations for equivalent staleness
	Uniform	Adaptive		
Business	123.3	87.9	29%	4 (130.2)
Entertainment	113.7	78.6	31%	4 (119.4)
Health	131.8	94.6	28%	5 (125.4)
UK	120.2	109.5	9%	5 (127.2)
All	122.3	105.2	14%	5 (132.9)

Table 1. Results for weekdays of the month of July 2000 using six synchronisations and comparing uniform with optimised synchronisation times, together with the number of optimised synchronisations required to achieve comparable levels of staleness as six uniform synchronisations.

4. Example Mobile Email Client

We now demonstrate the use of our approach in a second example, a mobile email client. A central server is being used to store email, and the mobile email client running on a mobile device synchronises with the central email server. We assume that a pull model is used, so the mobile email reader initiates the synchronisation. The mobile email client is similar to the mobile news database, and uses the probabilistic models to indicate to the user the likely divergence between the email cached on the mobile device, and the to control when synchronisations should occur. These are calculated using a similar cost function to that used in the News Database example.

A tool was used to create a log of when email arrived (email folders updated) for six Microsoft Exchange users over the months of January and February 2001, by using information held in the Exchange server. The update log for January was used to create the probabilistic models and the information for February was used to evaluate the performance of the synchronisation times chosen. The probabilistic models were created automatically, with the tool calculating the periodicity of the data being modelled. For the six users, four were modelled using a weekly periodicity, and the other two were modelled using a weekday/weekend periodicity.

Figure 2 presents results for the optimally chosen synchronisation times for the six Exchange users, showing the mean percentage decrease in staleness versus the number of synchronisations per day, with the error bars representing +/- one standard deviation. For the uniform synchronisation, all possible synchronisation times (based on a five minute granularity) were tried. So, for 24 synchronisations per day, the scenarios tried included a synchronisation occurred on every hour, then 5 minutes past every hour, then 10 minutes past every hour, etc. In this example, 11 sets of synchronisation times would be calculated and the average staleness was evaluated, and used to represent the staleness for the uniform approach.

The number of synchronisations was varied between 1 and 24 synchronisations per day. The results show clearly

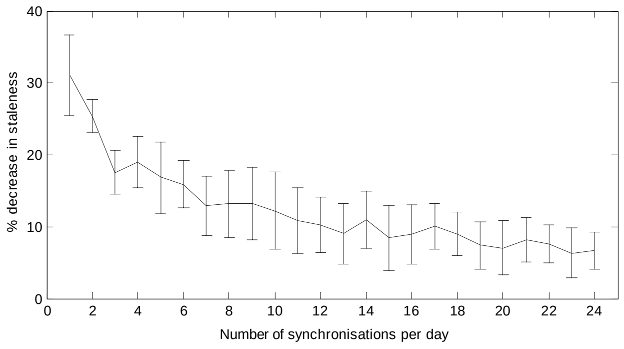


Figure 2. Reduction in staleness of email items for between 1 and 24 synchronisations per day for six users.

that regardless of the number of synchronisations per day the average staleness of email is reduced.

5. Other Applications and Future Work

Web Cache Web caches have become an integral part of the World Wide Web. Caches are embedded within web browsers as well as throughout the general infrastructure of the web. Their role is to replicate web pages thereby reducing latency of web page access, bandwidth usage and web server load. The HTTP protocol [4] provides support for web caches, allowing the life times of object received to be explicitly set, and for fields providing explicit instructions to caches on how to treat a particular page. A number of schemes have been proposed to allow caches to work more efficiently [6].

Many web sites are automatically generated using tools that could generate logs of when the pages are updated. These logs could then be used by our tools to generate the probabilistic models of each page. The models are small (approximately 100 bytes) and can be carried within the HTTP protocol from the web server which generates the web page to web caches and browsers. Enabled web caches and browsers can then use these models to make decisions about when a cached page is still acceptable (under user specified parameters), and inform a user the likelihood that the page has been updated.

Calendar The examples used so far involve data that cannot be modified at the replica. Whilst interesting, clearly the most exciting applications are those that allow the all replicas to be modified. Therefore, we have been looking at a calendar application, where a single user's calendar is replicated, and there are multiple people concurrently accessing and updating the calendar (for example a manager and their secretary).

As with the mobile news database and mobile email reader, the calendar application can calculate synchronisation times. More interestingly, the user interface can use the information to adapt, for example, indicate appointment slots that are less likely to lead to conflicts when synchronisation occurs. Also, potentially, the models can be used to provide *just-in-time update propagation*. Imagine a scenario where a secretary has access to a salesperson's calendar. The salesperson and the secretary are the only people who make appointments and the secretary works only weekdays. If on a Saturday the salesperson makes an appointment for the following week this need not be propagated until Monday morning, when the secretary arrives. However, if on a Tuesday morning the salesperson makes an appointment for the next day this should be propagated immediately because the secretary will be booking appoints on the same day. If the same salesperson also makes an appointment on the Tuesday morning for a month in the future, this might not need to be propagated immediately because, for example, the secretary never makes appointments more than a week in advance. Using the models of how the data evolves, the write update's propagation can be delayed until the system thinks that by delaying any longer the chance of conflict increases significantly. Furthermore, the updates can be propagated in any order. Thus the advantages of delaying propagation are that it may be possible to package the updates in packets more efficiently, saving cost and bandwidth, as well as the potential to delay until a cheaper communication medium becomes available. We are currently working on evaluating the feasibility of *just-in-time update propagation*.

6. Related work

Cho et al. [3, 2] examine techniques for a web crawler to maintain a large repository of web pages. Their work is focused on when each of the web pages should be checked in order to maintain a fresh and consistent repository. This involves estimating the rate of update of web pages, which is assumed to be constant. The main aim of their approach is to obtain an estimate of the rate of page update given that they haven't observed every update of the page. Due to its nature, a web crawler is unable to obtain a complete log of page updates and as a result it is non-trivial to obtain an unbiased estimate of the rate of change of the page. The model they prescribe is too simple, however, to enable decisions on the granularity of a day about when synchronisations should be made.

In contrast, we assume that we have complete logs of updates. Our models can be much richer than an estimation of a constant rate and thus provide more information for decision making. We are also making the information available to the application, allowing it to choose when to synchronise (picking an optimal time to synchronise rather than just picking the order in which to synchronise elements in the database), and also allowing the application to generally alter its behaviour based on the expected divergence.

In TACT [7] a number of metrics are proposed that allow the control of the replica divergence: *Numerical error*, *Order error* and *Staleness*. However, these metrics control the divergence rather than attempt to estimate its probable divergence.

7. Conclusions

This paper has described how probabilistic models can be used to estimate replica divergence and has given examples as to the sort of decisions that can be made based upon these models to improve the end-user experience. We have given a proof-of-concept demonstration of the approach in two simple applications and have suggested further, more complex examples to which the methods can be applied.

References

[1] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[2] J. Cho and H. Garcia-Molina. Estimating frequency of change. Submitted for publication., 2000.

[3] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proc. 2000 ACM Int. Conf. on Management of Data (SIGMOD)*, May 2000.

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Http version 1.1 rfc 2616. http://www.w3.org/Protocols/Specs.html, 1999.

[5] K. V. Mardia and P. E. Jupp. *Directional Statistics*. John Wiley and Sons, Chichester, West Sussex, 2nd edition, 1999.

[6] H. Yu, L. Breslau, and S. Shenker. A scalable web cache consistency architecture. In *Proc. ACM SIGCOMM'99*, Boston, MA, USA, September 1999.

[7] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *4th Symp. on Operating System Design and Implementation (OSDI)*, Oct 2000.

Self-Tuned Remote Execution for Pervasive Computing

Jason Flinn, Dushyanth Narayanan and M. Satyanarayanan
School of Computer Science
Carnegie Mellon University

Abstract

Pervasive computing creates environments saturated with computing and communication capability, yet gracefully integrated with human users. Remote execution has a natural role to play in such environments, since it lets applications simultaneously leverage the mobility of small devices and the greater resources of large devices. In this paper, we describe Spectra, a remote execution system designed for pervasive environments.

Spectra monitors resources such as battery energy and file cache state which are especially important for mobile clients. It also dynamically balances energy use and quality goals with traditional performance concerns to decide where to locate functionality. Finally, Spectra is self-tuning—it does not require applications to explicitly specify intended resource usage. Instead, it monitors application behavior, learns functions predicting their resource usage, and uses the information to anticipate future behavior.

1 Introduction

Remote execution is an old and venerable topic in systems research. Systems such as Condor [3] and Butler [15] have long provided the ability to exploit spare CPU cycles on other machines. Yet, the advent of pervasive computing has created new opportunities and challenges for remote execution. In this paper, we discuss these issues and how we have addressed them in the implementation of Spectra, a remote execution system for pervasive computing.

The need for mobility leads to smaller and smaller computing devices. The size limitations of these devices constrain their compute power, battery energy and storage capacity. Yet, many modern applications are resource-intensive, with demands that often outstrip device capacity. Remote execution using wireless networks to access compute servers thus fills a natural role in pervasive computing, allowing applications to leverage both the mobility of small devices and the greater resources of stationary devices.

Pervasive computing also creates new challenges [19]. When locating functionality, Spectra must balance the traditional goal of minimizing application latency with new goals such as maximizing battery lifetime. It must allow for wider variation in resources such as CPU and network bandwidth and monitor new resources such as energy use and cache state.

Pervasiveness causes additional complexity, and it is unreasonable to leave the burden of handling this complexity to applications. Spectra does not require applications to specify resource requirements for a variety of platforms and output qualities. Instead, it is *self-tuning*—it monitors application resource usage in order to predict future behavior.

2 Design considerations

The design of Spectra has been greatly influenced by the need to address the complexities of pervasive computing.

Spectra weighs several possibly divergent goals when deciding where to execute applications. Performance remains important in mobile environments, but is no longer the sole consideration. It is also vital to conserve energy so as to prolong battery lifetime. Quality is another factor—a resource-poor mobile device may only be able to provide a low fidelity version of a data object [16] or computation [20], while a stationary machine may be able to generate a better version.

Spectra monitors environmental conditions and adjusts the relative importance of each goal. For example, energy use is paramount when a device's battery is low. However, when the battery is charged, performance considerations may dominate. Monitoring battery state and expected time to recharge allows Spectra to adjust the relative importance of these goals.

Spectra monitors resources that are uniquely significant in pervasive environments. In addition to battery energy, file cache state is often critical. Consider a mobile client with limited storage running a distributed file system. When there is a choice of remote execution sites, a server with a warmer file cache may often be preferable to one with a faster processor.

Finally, Spectra is self-tuning. Applications need not

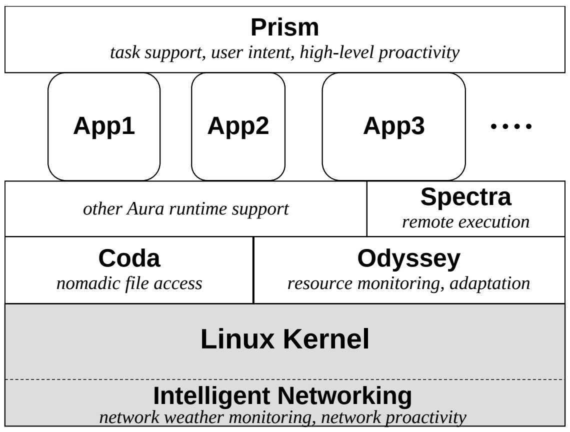

Prism
task support, user intent, high-level proactivity
App1
App2
App3
....
other Aura runtime support
Spectra
remote execution
Coda
nomadic file access
Odyssey
resource monitoring, adaptation
Linux Kernel
Intelligent Networking
network weather monitoring, network proactivity

Figure 1. Aura architecture

specify their expected usage of various resources. Providing estimates for even a single resource such as battery energy is very difficult since energy use depends upon the hardware platform and the degree of power management used. Spectra applications need only specify operations of interest and the input parameters to those operations. Spectra monitors and logs resource usage as applications execute. From logged data, it learns functions relating input parameters to resource usage, allowing it to predict future application resource use.

3 Implementation

3.1 Spectra overview

Spectra is the remote execution component of Aura, a new computing system being built at Carnegie Mellon University. Aura provides users with an invisible halo of computing and information services that persists regardless of location. As shown in Figure 1, an Aura client is composed of many parts. The Coda file system [10] allows mobile nodes to access shared data, even when weakly-connected or disconnected from the network. Odyssey [16] supports applications that vary their fidelity as resource availability changes. Fidelity is an application-specific metric of quality expressed in multiple discrete or continuous dimensions. For instance, dimensions of fidelity for speech recognition are vocabulary size and acoustic model complexity.

To provide a complete solution, Spectra must address several complex issues, including function placement, service discovery, execution mechanism and data consistency. Our initial prototype focuses on the first problem: deciding where and how operations should be executed. It uses existing technology to address the remaining issues. We hope to leverage service discovery protocols which allow attribute-value lookup [1, 23]. Similarly, while we currently use RPC-based remote execution, Spectra could be modified to use other mechanisms such as mobile code. Finally, Coda provides Spectra a single shared file system across multiple machines.

Spectra consists of three main elements:

- an application interface for describing operations.
- monitors that predict resource use and availability.
- a decision engine that selects the best execution option.

3.2 Application interface

Applications use the Odyssey multi-fidelity interface [14] to communicate with Spectra. The fundamental unit of discourse is the *operation:* a code component which may profit from remote execution. Spectra targets applications which perform operations of one second or more in duration—examples are speech recognition, rendering for augmented reality, and document processing.

Applications first register operations with Spectra. A registration lists possible fidelities and methods of dividing computation between local and remote machines. It also lists input parameters that affect operation complexity.

For example, we have modified the Janus speech recognizer [24] to use Spectra. The basic operation is utterance recognition. This operation has two fidelities: full and reduced. Reduced fidelity uses a smaller, more task-specific vocabulary than full fidelity. There are three modes of dividing computation: recognition may be performed on the client (local mode), on a server (remote mode) or on both (hybrid mode). In hybrid mode, the first phase is performed locally, yielding a greatly compressed data set which is shipped remotely for the completion of recognition. The single input parameter is the length of the utterance.

Prior to operation execution, an application invokes Spectra to determine how and where the operation will execute. The application passes in the value of the input parameters—for example, the size of an utterance to be recognized. Spectra chooses the best fidelity level and execution mode as described in Section 3.4 and returns these values to the application. For remote operations, Spectra also chooses the server on which the operation will be executed.

Applications execute operations by making remote procedure calls to the selected server. Direct procedure calls can be used in the local case to optimize performance. Applications inform Spectra when operations complete, at which time Spectra logs resource usage. The logged data allows Spectra to improve resource prediction over time.

3.3 Resource monitoring

Only part of the data needed by Spectra comes from applications—the remainder is supplied by resource monitors. Resource monitors are modular, resource-specific code components that predict resource availability and demand.

Prior to operation execution, each monitor predicts how much of a resource the operation will receive. Monitors make predictions for the local machine and for any remote servers on which the operation may execute. For instance, the network monitor predicts bandwidth and round-trip times between the client and each server. Spectra gathers the predictions in a *resource snapshot*, which provides a consistent view of resource availability for that operation.

Resource monitors observe application behavior to predict future resource demand. While an operation executes, each monitor measures its resource usage. Upon operation completion, these values are logged, along with the operation's input parameters, fidelity, and method of dividing computation. From this data, Spectra learns functions which predict operation resource usage. Thus, the more an operation is executed, the more accurately its resource usage is predicted.

We have built monitors for four resources: CPU, network, battery, and cache state. As CPU and network are well-understood resources, we describe these monitors only briefly here. The CPU monitor, described in [14], predicts availability using a smoothed estimate of recent CPU load, weighted by the maximum speed of the processor. During operation execution, the CPU monitor measures CPU cycles consumed on local and remote machines. The network monitor predicts available bandwidth and round-trip times to remote machines using the algorithm in [16]. For each operation, it measures bytes sent and received, as well as the number of RPCs.

3.3.1 The battery monitor

The battery monitor must provide accurate, detailed information without hindering user mobility. Previous energy measurement approaches are thus insufficient for the task. It is infeasible to use external measurement equipment [7, 21] since such equipment can only be used in a laboratory setting. Alternatively, one can calibrate the energy use of events such as network transmission, and then later approximate energy use by counting event occurrences [4, 13]. However, results will be inaccurate when the calibration does not anticipate the full set of possible events, or when events such as changes in screen brightness are invisible to the monitor.

Our battery monitor takes advantage of the advent of "smart" batteries: chips which report detailed information about battery levels and power drain. The monitor predicts availability by querying the amount of charge left in the battery. It measures operation energy use by periodically polling the chip to sample energy use.

The first platform on which we have implemented our battery monitor is Compaq's Itsy v2.2 [8], an advanced pocket computer with a DS2437 smart battery chip [5]. Since the DS2437 reports average current drawn over a 31.25 ms. period and voltage levels change little, we could measure power by sampling current at 32 Hz. Unfortunately, the DS2437's communication protocol makes the overhead of frequent sampling unacceptably high. The battery monitor balances overhead and accuracy by sampling at 6 Hz during operation execution. This rate accurately measures operation energy use with low (1.8%) CPU overhead. At other times, the monitor samples at 1 Hz—a rate sufficient to accurately measure battery charge and background power drain.

3.3.2 The cache state monitor

Data access can consume significant time and energy when items are unavailable locally. The cache state monitor estimates these costs by predicting which uncached objects will be accessed. It currently provides estimates for one important class of items: files in the Coda file system.

During operation execution, the monitor observes accesses of Coda files. When an operation completes, the monitor logs the name and size of each file accessed.

The cache state monitor currently uses a simple prediction scheme—it assumes the likelihood of a file being accessed during an operation is similar to the percentage of times it was accessed during recent operations of similar type and input parameters. The access likelihood is maintained as a weighted average, allowing the monitor to adjust to changes in application behavior over time. For each file that may be accessed, the monitor queries Coda to determine if the file is cached. If it is uncached, the expected number of bytes to fetch is equal to the file's size multiplied by its access likelihood. The monitor estimates the number of bytes that an operation will fetch by summing individual predictions for each file.

The monitor makes predictions for both local and remote machines. It also estimates the rate at which data will be fetched from Coda servers so that Spectra can calculate the expected time and energy cost of fetching uncached items.

3.4 Selecting the best option

Spectra's decision engine chooses a location and fidelity for each operation. Its inputs are the application's description of the operation and the monitors' snapshot of resource availability. It uses Odyssey's multi-fidelity solver [14] to search the space of possible fidelities, remote servers, and methods of dividing computation. Using gradient-descent heuristics, the solver attempts to find the best execution alternative.

Spectra evaluates alternatives by their impact on *user metrics*. User metrics measure performance or quality perceptible to the end-user—they are thus distinct from resources, which are not directly observable by the user (other than by their effect on metrics). For instance, while battery energy and CPU cycles are resources, execution latency and change in expected battery lifetime are user metrics.

To evaluate an alternative, Spectra first calculates a context-independent value for each metric. It then weights each value with an *importance function* that expresses the current desirability of the metric to the user. Finally, it calculates the product of the weighted metrics to compute a single value for evaluating the alternative. This calculation is a specific instance of the broader concept of "resource-goodness mappings" [17]. Spectra currently considers three user metrics in its evaluation: execution latency, battery lifetime, and application fidelity.

Spectra may use many resource predictions to calculate a metric's context-independent value. For example, execution latency is the sum of the predicted latencies of fetching uncached items, network transmissions, and processing on local and remote machines. Processing latencies are calculated by dividing the predicted cycles needed for execution by the predicted amount of cycles available per second. Network and cache latencies are calculated similarly.

Since importance functions express the current desirability of metrics to the user, they may change over time. For example, we use goal-directed adaptation [6] as the importance function for battery lifetime. The user specifies a duration that the battery should last, and the system attempts to ensure that the battery lasts for this duration. A feedback parameter, c, represents how critical energy use is at the present moment. Spectra adjusts this parameter using estimates of battery charge and recent power usage reported by the battery monitor. Given expected energy use, E, the battery importance function is $(1/E)^c$. As an example, when the computer operates on wall power, c is 0 and energy has no impact in evaluating alternatives.

For execution latency, we use an application-specific importance function that reflects perceptible deadlines for operation completion. For example, the speech recognizer's importance function for latency, L, is simply $1/L$. This function has the intuitive property that a recognition that takes twice as long is half as desirable to the user.

Fidelity is a multidimensional metric of application-specific quality. The importance of fidelity is user-dependent and is often expressed with utility functions that map each user's preferences to a single value. For the speech recognizer, the fidelity importance function gives reduced fidelity the value 0.5 and full fidelity the value 1.0.

4 Preliminary evaluation

Our evaluation measured how well Spectra adapts to changes in resource availability. As a sample application, we used the speech recognizer described in Section 3.2.

We limited execution to two machines. The client was an Itsy v2.2 pocket computer with a 206 MHz SA-1100 processor and 32 MB DRAM. The server was an IBM T20 laptop with a 700 MHz PIII processor and 256 MB DRAM. Since the Itsy lacks a PCMCIA slot (such as is available on the Compaq iPAQ), the two machines were connected with a serial link.

We first recognized 15 utterances so that Spectra could learn the application's resource requirements. We then created several scenarios with varying resource availability and measured how well Spectra adapted application behavior when a new utterance was recognized. Figure 2(a) shows measured execution latency and energy use for each possible combination of fidelity and location. For each scenario, the option that best satisfies the evaluation criteria for the speech application is highlighted. Figure 2(b) shows results when Spectra chooses the alternative to execute.

In the baseline scenario both computers are unloaded and connected to wall power. Spectra correctly chooses the hybrid mode and full vocabulary here. Using the reduced vocabulary in hybrid mode slightly reduces execution time, but not nearly enough to counter the reduction in fidelity.

Each remaining scenario differs from the baseline by varying the availability of a single resource. In the battery scenario, the client is battery-powered with an ambitious battery lifetime goal of 10 hours. Energy use is critical, so Spectra chooses the remote mode. As before, the small energy and latency benefits of using the reduced vocabulary do not outweigh the decrease in fidelity.

The network scenario halves the bandwidth between the client and server. Spectra correctly chooses hybrid execution and the full vocabulary in this scenario. The CPU scenario loads the client processor. Spectra chooses remote execution since the cost of doing the first recognition phase locally outweighs the benefit of reduced network usage.

In the cache scenario, the server is made unavailable and the 277 KB language model for the full vocabulary is flushed from the client's cache. Spectra uses the reduced vocabulary since the cache miss makes full fidelity recognition approximately 3 times slower than the reduced case.

Though preliminary, these results are encouraging, since Spectra chooses the best execution mode in each scenario. Further, the overhead of using Spectra to choose an alternative is within experimental error in all cases.

5 Related work

Spectra's uniqueness derives from its focus on pervasive computing. It is the first remote execution system to monitor battery and cache state, support self-tuning operation, and balance performance goals with battery use and fidelity.

	Local/Reduced		Local/Full		Hybrid/Reduced		Hybrid/Full		Remote/Reduced		Remote/Full	
	(Fidelity = 0.5)		(Fidelity = 1.0)		(Fidelity = 0.5)		(Fidelity = 1.0)		(Fidelity = 0.5)		(Fidelity = 1.0)	
	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy	Time	Energy
Scenario	(s.)	(J.)	(s.)	(J.)	(s.)	(J.)	(s.)	(J.)	(s.)	(J.)	(s.)	(J.)
baseline	37.4 (0.1)		69.2 (0.5)		7.8 (0.6)		8.7 (0.7)		9.3 (0.6)		10.3 (0.3)	
battery	37.4 (0.0)	22.6 (0.2)	69.2 (0.6)	43.5 (0.5)	7.3 (0.2)	3.5 (0.0)	8.6 (0.6)	3.6 (0.1)	9.2 (0.4)	2.4 (0.1)	10.2 (0.5)	2.5 (0.1)
network	37.4 (0.2)		69.8 (0.4)		9.2 (0.1)		10.5 (0.6)		22.2 (3.7)		21.4 (4.3)	N/A
CPU	75.2 (0.4)		137.6 (0.6)		12.4 (1.2)		12.7 (0.1)		10.8 (1.4)		12.0 (2.7)	
cache	36.6 (0.2)		105.4 (0.4)									

(a) Time and energy cost of each possible execution alternative

Scenario	Best Alternative	Chosen Alternative	Time (s.)	Energy (J.)	Fidelity
baseline	Hybrid/Full	Hybrid/Full	8.7 (0.8)		1.0
battery	Remote/Full	Remote/Full	10.6 (1.2)	2.7 (0.3)	1.0
network	Hybrid/Full	Hybrid/Full	10.7 (1.1)		1.0
CPU	Remote/Full	Remote/Full	12.0 (1.2)		1.0
cache	Local/Reduced	Local/Reduced	36.7 (0.2)		0.5

(b) Results of using Spectra to select an alternative

This figure shows how Spectra adapts the behavior of a speech recognizer in the resource availability scenarios described in Section 4. Part (a) shows the value of the three user metrics considered by Spectra (execution time, energy use, and fidelity) for each of the six possible execution alternatives. The highlighted alternative is the one that best satisfies the evaluation criteria for the speech application. Part (b) shows the results of using Spectra to select an alternative—it lists the best possible alternative, the alternative actually chosen by Spectra, and the values of the three metrics. Energy use is only measured in the battery scenario since the client operates on wall power in all other scenarios. Each result shown is the mean of five trials—standard deviations are shown in parentheses.

Figure 2. Spectra speech recognition results

As the field of remote execution is enormous, we restrict our discussion of related work to the most closely related systems. Rudenko's RPF [18] considers both performance and battery life when deciding whether to execute processes remotely. Kunz's toolkit [12] uses similar considerations to locate mobile code. Although both monitor application execution time and RPF also monitors battery use, neither monitors individual resources such as network and cache state, limiting their ability to cope with resource variation.

Kremer et al. [11] propose using compiler techniques to select tasks that might be executed remotely to save energy. At present, this analysis is static, and thus can not adapt to changing resource conditions. Such compiler techniques are complementary to Spectra, in that they could be used to automatically select Spectra operations and insert Spectra calls in executables.

Vahdat [22] notes issues considered in the design of Spectra: the need for application-specific knowledge and the difficulty of monitoring remote resources.

Several systems designed for fixed environments share Spectra's self-tuning nature. Coign [9] statically partitions objects in a distributed system by logging and predicting communication and execution costs. Abacus [2] monitors network and CPU usage to migrate functionality in a storage-area network. Condor monitors goodput [3] to migrate processes in a computing cluster.

6 Conclusion

Remote execution lets pervasive applications leverage both the mobility of small devices and the greater resources of large devices. Our initial results with Spectra show that this benefit can be effectively realized if the system monitors pervasive resources, balances multiple goals in evaluation, and supports self-tuning operation.

Yet, much work remains to be done. Our early experience with Spectra suggests that predictions often involve tradeoffs between speed and accuracy. For example, when estimating remote CPU availability, Spectra might use a slightly stale cached value, or it might query the server to obtain more accurate information. If the difference between possible alternatives is slight, as for example with short-running operations, Spectra would do better to make a "quick and dirty" decision. However, when alternatives differ significantly, Spectra should invest more effort to choose the optimal alternative. This suggests to us that Spectra itself should be adaptive—it should balance the amount of effort used to decide between alternatives against the possible benefit of choosing the best alternative.

Since resource logs can grow quite large for complex operations, we hope to develop methods for compressing log data without sacrificing significant semantic content. We also plan to investigate how the importance functions used in evaluation can be modified with simple user interfaces. Finally, we wish to evaluate Spectra using more dynamic resource scenarios.

Acknowledgements

Many people contributed to this paper. Keith Farkas, Lawrence Brakmo, Deborah Wallach, Bill Hamburgen, and the rest of the Itsy team at Compaq Western Research Lab provided a great deal of software and support that aided us in porting applications to the Itsy platform. Jan Harkes and Shafeeq Sinnamohideen assisted with the Coda file system. Rajesh Balan, Keith Farkas, David Petrou, and the anonymous reviewers gave us several helpful comments that improved the paper.

This research was supported by the National Science Foundation (NSF) under contracts CCR-9901696 and ANI-0081396, the Defense Advanced Projects Research Agency (DARPA) and the U.S. Navy (USN) under contract N660019928918, IBM Corporation, Nokia Corporation, Intel Corporation, and Compaq Corporation. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the NSF, DARPA, USN, IBM, Nokia, Intel, Compaq, or the U.S. government.

References

[1] Adjie-Winoto, W., Schartz, E., Balakrishnan, H., and Lilley, J. The Design and Implementation of an Intentional Naming System. *17th ACM Symp. on Op. Syst. and Princ.*, pages 202–16, Kiawah Island, SC, Dec. 1999.

[2] Amiri, K., Petrou, D., Ganger, G., and Gibson, G. Dynamic Function Placement for Data-Intensive Cluster Computing. *USENIX Annual Tech. Conf.*, San Diego, CA, June 2000.

[3] Basney, J. and Livny, M. Improving Goodput by Co-scheduling CPU and Network Capacity. *Intl. Journal of High Performance Computing Applications*, 13(3), Fall 1999.

[4] Bellosa, F. The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems. *9th ACM SIGOPS European Workshop*, Kolding, Denmark, Sept. 2000.

[5] Dallas Semiconductor Corp., 4401 South Beltwood Parkway, Dallas, TX. *DS2437 Smart Battery Monitor*, 1999.

[6] Flinn, J. and Satyanarayanan, M. Energy-Aware Adaptation for Mobile Applications. *17th ACM Symp. on Op. Syst. and Princ.*, pages 48–63, Kiawah Island, SC, Dec. 1999.

[7] Flinn, J. and Satyanarayanan, M. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. *2nd IEEE Workshop on Mobile Comp. Syst. and Apps.*, pages 2–10, New Orleans, LA, Feb. 1999.

[8] Hamburgen, W. R., Wallach, D. A., Viredaz, M. A., Brakmo, L. S., Waldspurger, C. A., Bartlett, J. F., Mann, T., and Farkas, K. I. Itsy: Stretching the Bounds of Mobile Computing. *IEEE Computer*, 13(3):28–35, Apr. 2001.

[9] Hunt, G. C. and Scott, M. L. The Coign Automatic Distributed Partitioning System. *3rd Symposium on Operating System Design and Implemetation*, New Orleans, LA, Feb. 1999.

[10] Kistler, J. and Satyanarayanan, M. Disconnected Operation in the Coda File System. *ACM Trans. Comput. Syst.*, 10(1), Feb. 1992.

[11] Kremer, U., Hicks, J., and Rehg, J. M. Compiler-Directed Remote Task Execution for Power Management. *Workshop on Compilers and Operating Systems for Low Power*, Philadelphia, PA, Oct. 2000.

[12] Kunz, T. and Omar, S. A Mobile Code Toolkit for Adaptive Mobile Applications. *3rd IEEE Workshop on Mobile Comp. Syst. and Apps.*, pages 51–9, Monterey, CA, Dec. 2000.

[13] Lorch, J. R. and Smith, A. J. Apple Macintosh's Energy Consumption. *IEEE Micro*, 18(6):54–63, Nov./Dec. 1998.

[14] Narayanan, D., Flinn, J., and Satyanarayanan, M. Using History to Improve Mobile Application Adaptation. *3rd IEEE Workshop on Mobile Comp. Syst. and Apps.*, pages 30–41, Monterey, CA, Aug. 2000.

[15] Nichols, D. Using Idle Workstations in a Shared Computing Environment. *11th ACM Symp. on Op. Syst. and Princ.*, pages 5–12, Austin, TX, Nov. 1987.

[16] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile Application-Aware Adaptation for Mobility. *16th ACM Symp. on Op. Syst. and Princ.*, pages 276–87, Saint-Malo, France, Oct. 1997.

[17] Petrou, D., Narayanan, D., Ganger, G., Gibson, G., and Shriver, E. Hinting for Goodness' Sake. *8th Workshop on Hot Topics in OS*, May 2001.

[18] Rudenko, A., Reiher, P., Popek, G., and Kuenning, G. The Remote Processing Framework for Portable Computer Power Saving. *ACM Symp. Appl. Comp.*, San Antonio, TX, Feb. 1999.

[19] Satyanarayanan, M. Pervasive Computing: Vision and Challenges. *To appear in IEEE Personal Communications*.

[20] Satyanarayanan, M. and Narayanan, D. Multi-Fidelity Algorithms for Interactive Mobile Applications. *3rd Intl. Workshop on Discrete Alg. and Methods in Mobile Comp. and Comm.*, pages 1–6, Seattle, WA, Aug. 1999.

[21] Stemm, M. and Katz, R. H. Measuring and Reducing Energy Consumption of Network Interfaces in Hand-Held Devices. *IEICE Trans. Fundamentals of Electr., Comm. and Comp. Sci.*, 80(8):1125–31, Aug. 1997.

[22] Vahdat, A., Lebeck, A., and Ellis, C. Every Joule is Precious: The Case for Revisiting Operating System Design for Energy Efficiency. *9th ACM SIGOPS European Workshop*, Kolding, Denmark, Sept. 2000.

[23] Viezades, J., Guttman, E., Perkins, C., and Kaplan, S. *Service Location Protocol*. IETF RFC 2165, June 1997.

[24] Waibel, A. Interactive Translation of Conversational Speech. *IEEE Computer*, 29(7):41–8, July 1996.

Energy is just another resource: Energy accounting and energy pricing in the Nemesis OS

Rolf Neugebauer[*]
Department of Computing Science
University of Glasgow
Glasgow, G12 8QQ, Scotland, U.K.
neugebar@dcs.gla.ac.uk

Derek McAuley
Marconi Laboratory
10 Downing Street
Cambridge CB2 3DS, U.K.
derek.mcauley@marconi.com

Abstract

In this position paper, we argue that, with an appropriate operating system structure, energy in mobile computers can be treated and managed as just another resource. *In particular, we investigate how energy management could be added to the Nemesis OS which provides detailed and accurate resource accounting capabilities in order to provide Quality of Service (QoS) guarantees for all resources to applications. We argue that, with such an operating system, accounting of energy to individual processes can be achieved. Furthermore, we investigate how an economic model, proposed for congestion avoidance in computer network, and recently applied to CPU resource management, can be used as a dynamic, decentralised energy management system, forming a collaborative environment between operating system and applications.*

1 Motivation

Recently, there has been increased research activity in energy management in operating systems. The main motivation is the increased use of mobile devices such as laptop computers or PDAs, but environmental issues, such as overall power consumption and the noise generated by active cooling also play a role. A general consensus of this research is that applications should be involved in the management of energy, as the different modes of operation they might offer can have a significant impact on overall power consumption of the system (e.g., [6, 8, 24]). In this paper, we argue that, with an appropriate operating system structure, energy can be managed just like any other resource. Furthermore, we investigate how a decentralised resource management architecture can be used to manage energy consumption in mobile computers.

[*]Rolf Neugebauer has been supported by a Marie Curie Fellowship from the European Union, Contract-No.: ERBFMBICT972363.

This paper is motivated by two observations: First, recent research efforts to move energy management into the operating system, and indeed, the application domain, exhibit similar problems to those we have observed while working on the Nemesis operating system [19], an OS providing applications with Quality of Service (QoS) guarantees for all physical resources. An essential prerequisite for providing QoS guarantees (and more generally, managing resource allocations to competing clients) is accurate accounting of resource usage to individual applications and users. We observed that the mechanisms, deployed in Nemesis for accounting for the usage of traditional resources (e.g., CPU, network, disks, and displays), can be applied to accurately account for the energy consumption of individual applications as well. In section 2 we detail how Nemesis' resource accounting mechanisms can be utilised and extended to provide accounting of energy consumption.

The second observation is related to our ongoing research where we apply pricing and charging mechanisms to the area of resource management in multimedia operating systems [22]. We view charges, indicating the current level of resource contention, as feedback signals to processes. By limiting the amount of credits available, processes are given the incentive to adapt to these feedback signals in an application specific fashion. This forms a simple *decentralised* model for resource management, capable of yielding resource allocations proportional to the credit allocation for each process. In section 3 we investigate how this model, which previously has been successfully applied to both network congestion control [21, 17, 18] and CPU resource allocation [22], can be applied to manage energy consumption.

This paper is rounded off by a comparison of related work and our conclusions.

2 Energy Accounting

The Nemesis operating system was designed to provide accurate accounting of all resources consumed by individual applications. In traditional operating systems a significant amount of resources are consumed anonymously, i.e., unaccounted for, because a significant proportion of code is executed in the kernel, or in shared servers, on behalf of processes. In a multi-media operating system this may lead to an undesired effect, termed *QoS-Crosstalk* [19], where one process could influence the performance of other processes by causing contention for shared resources. In Nemesis, this problem has been addressed by multiplexing shared physical resources only once and at the lowest possible level, facilitating accurate accounting of resource consumption. The resulting operating system is vertically structured [1], with most of the functionality provided by traditional operating systems instead being executed by the applications themselves, implemented as user-level shared libraries[1].

An effect similar to QoS-crosstalk has been observed in the context of energy management, most notably in [8]: a significant amount of energy is consumed by shared resources such as the networking stack, the kernel, or the X-server, and by shared devices, such as the display, disk or network card. We argue, that if the operating system is already designed to accurately account for traditional resource usage, then it is possible to accurately account for energy consumption as well.

2.1 Resource Accounting in Nemesis

Nemesis provides QoS guarantees, and therefore accurate resource accounting, for the following traditional resources: CPU [19], memory [12], I/O devices such as the network interface [4] and disk drives [2], and framebuffer devices [1]. Processes can make reservations of slices of CPU which are then scheduled using an Earliest Deadline First (EDF) based real-time scheduler. CPU resource usage is accounted for with cycle accuracy. For memory, individual processes can request ranges of virtual memory which are guaranteed to be backed by a specified number of physical pages. Processes are then responsible for their own virtual memory management. Device drivers for I/O devices are implemented as privileged, user-level processes which register interrupt handlers with the system. The interrupt handler typically only clears the interrupt condition, and sends an event to the device driver process, effectively decoupling interrupt notification from interrupt servicing. The device driver process only implements infrequent out-of-band management functions and performs the single demultiplexing function for the hardware device (e.g., packet filtering for network devices)[2]. All higher level functionality, such as network stack processing, is performed at the user level utilising (shared) libraries. Similarly, processes *own* individual pixels or regions of pixels of the framebuffer device and all higher level drawing primitives are performed by the processes themselves. Again, protection and access control is managed by the device driver.

As a result of this OS architecture, most activities typically performed by an operating system kernel or services are performed by the applications themselves and virtually all resources consumed can be accounted to individual processes, i.e., there is no significant anonymous resource consumption.

2.2 Energy Accounting to Processes

In this section we investigate how this model can be extended to provide per-process accounting of energy. Consider a number of processes P (indexed by i) and a number of devices and resources J (indexed by j). Each active device consumes an amount E_j of energy. As Nemesis provides accurate accounting information for the usage of the resources J by the processes P, the proportion x_{ij} of the individual resources a process is using can be determined. For example, consider a display device with process P_i owning p_i pixels. Then the proportion $x_{iDisplay}$ of the display resource belonging to P_i is $x_{iDisplay} = p_i / \sum_i p_i$. Similarly, the proportions for the network device or disk device can be determined based either on the number of bytes transferred or on the proportions of time processes access the device[3]. Using the proportions of the resources used and the energy E_j consumed by each device, the overall energy consumption for each process can be determined: $E_i = \sum_j (E_j \times x_{ij})$. The system should inform each process about their total energy consumption E_i as well as its breakdown for each device (i.e., $E_j \times x_{ij}$).

2.3 Implementation Issues

Unfortunately, the energy E_j each device is consuming is difficult to determine on typical laptops or other mobile devices. Ideally, there would be hardware support measuring the power consumption, i.e., voltage and current drawn, for each device individually. Then, the device driver could

[1]This structure is comparable to Exokernel systems [15], though the motivation behind the design is different. The principal motivation for the Exokernel design was to allow applications to optimise the implementation of various system components using application-specific knowledge.

[2]With appropriate hardware support, as provided, for example, by some network cards, de-multiplexing can be mainly performed in hardware, thus reducing the resources needed by the device driver. A software mechanism, known as *call-privs* [4, 1], also allows some of the de-multiplexing costs to be accounted to the clients.

[3]For network devices, access times may be the more appropriate basis, as processes receiving data require the device to be active without necessarily transferring data.

frequently sample the current power consumption of the device and account the energy to its client processes accordingly. However, we are not aware of any system providing such built-in online measurement facilities.

Rather than requiring the provisioning of such facilities, we propose a mechanism, similar to PowerMeasure [20], to estimate energy consumption of the individual devices. Modern laptops provide advanced power management features through the Advanced Configuration and Power Interface (ACPI) [14]. Unlike its predecessor Advanced Power Management (APM) [13], ACPI encourages collaborative energy management between BIOS and operating system, leaving power management policy decisions to the OS while providing detailed information about each device and its possible power states through a hierarchical namespace. The namespace contains control methods which the OS can use to manage the devices under the control of ACPI. An ACPI compliant BIOS also provides access to detailed information about batteries by including a variant of the Smart Battery interface [10].

Using the Smart Battery interface, the overall current power consumption of the system can be measured; the interface exports queries on the current voltage and current rate of discharge (either in mA or in mW). During a calibration process, individual devices can be systematically placed into their supported power states using ACPI control methods, and the resulting change in the system's power consumption can be observed. For I/O devices, especially for disks, where state transitions can consume considerable amounts of energy, the power consumption during state transitions should also be measured during the calibration process.

From this information, energy vectors $\vec{E}_j$, containing the estimated energy consumption for each device in their respective power saving states and state transitions can be derived. For example, the energy vector for a display would contain energy consumption values for the different brightness levels the display supports. During normal operation, the operating systems keeps track of the state each device is currently operating in and uses the corresponding E_j value when calculating a process' energy consumption. The energy consumed during state transitions, such as a disk spinning up, can be accounted to processes using the device in a time window after the state transition.

Alternatively, it is also conceivable that laptop vendors would provide detailed energy profiles for each device. This information, similar to the one obtainable during the calibration process, could be stored in the ACPI namespace.

In either case, using energy consumption values for discrete device states and state transitions would only lead to *estimates* of the individual processes' energy consumption. Furthermore, it has been reported [3, 7], that non device related activities, such as cache misses, may have a significant impact on the overall energy consumption of the system, especially in small, hand-held devices. Should this significantly impact the accuracy of the per process energy accounting, the calibration process could be extended to include micro-benchmarks; process performance counter samples could be used during accounting, similar to [3, 7].

3 Energy Management

Accounting for a process' energy consumption forms only the basis on which advanced energy management can be performed. Centralised, passive energy management policies, such as provided by most APM BIOS implementations or the DPMS extensions in X-Servers, only provide static policies where the user can specify timeouts, after which parts of or the entire system are put into energy saving modes. It is now widely accepted [6, 8, 24] that energy management should be performed at a higher level and may involve applications themselves. In [8] it is impressively demonstrated how a variety of applications, executing in different modes of operation, can have a significant impact on a system's energy consumption. The authors of [8] therefore argue that processes should form a collaborative relationship with the operating system.

In our ongoing research [22] we are applying microeconomic ideas to the area of resource management in operating systems in a similar approach to [23]. Informally, one can assume that basic resource consumption is free if a resource has no resource contention (fixed costs for the provisioning of the resource should be covered by fixed charges). However, if a resource is congested (i.e., demand exceeds the maximum supply) then everyone responsible for the contention should be charged, and the charges should be proportional to the users' individual responsibility for the contention. Prices, capturing this *external* congestion cost, are known as *shadow prices* and provide a *meaningful* feedback signal to applications, as they convey information about the level of contention and the user's responsibility for it. Applications can react to these feedback signals and dynamically adapt their behaviour in an application and resource-specific way. Adaption is encouraged by limiting the amount of credits available to individual processes. Thus, operating system and applications together form a *decentralised* resource management system, with the OS determining current resource prices and applications adapting their behaviour accordingly. In this section, we investigate how this model can be applied to energy management to form the "collaborative relationship between OS and applications" also advocated by others [6, 8, 24].

3.1 The Pricing Model

First, we briefly summarise the theoretical framework presented in [17, 18, 22]. In general, a user or process i of a resource (i.e., energy) attempts to maximise its utility $u_i(x_i)$ obtained from the amount of the resource x_i consumed. As the user is charged some cost $C(\cdot)$ for the resource usage, the user seeks to maximise $U_i(x_i) = u_i(x_i) - C(\cdot)$. A social planner, on the other hand, attempts to achieve a *socially optimal* resource allocation which maximises the sum of all the users' utility minus the cost of the overall system load (externalities):

$$\max \sum_i u_i(x_i) - C(\sum_i x_i) \tag{1}$$

As this desirable optimisation problem requires knowledge of the utility functions, which are typically not explicitly known, Kelly et al. [16] suggest decomposing the optimisation problem into a user and a system problem and demonstrate that the decomposed system also yields the socially optimal resource allocation.

For this decomposition, suppose that a user is charged a rate t_i proportional to the amount of the resource x_i the user receives. Then the user faces the optimisation problem:

$$\max U_i(x_i) = u_i(x_i) - t_i x_i \tag{2}$$

For a monotonically increasing, concave, and continously differentiable utility function the unique solution is:

$$u_i'(x_i) = t_i$$

If the resource manager seeks to achieve a socially optimal resource allocation according to equation 1, it will set the charge t_i to the shadow price $p(y)$ depending on the load y of the resource giving:

$$t_i = p(y) = \frac{d}{dy} C(y) \tag{3}$$

with $C(y)$ being the rate at which cost is incurred at overall load y. Thus, the feedback signal in the form of the charge $x_i p(y)$ is both proportional to the user's resource allocation and the congestion cost it incurs.

3.2 Charging for Energy

The decomposed approach requires the resource manager to be able to assess the external cost of resource contention $C(y)$. It is straightforward to identify external costs of resource contention in communication networks ($\rightsquigarrow$ dropped packets) and soft real-time systems ($\rightsquigarrow$ missed deadlines). For battery energy we adopt the *goal directed* approach, proposed in [8]: to a user the primary meaningful, energy related performance metric is the lifetime expectancy of the current battery charge. In other words, the user should be able to specify how long the current battery charge should last, and the energy management system should strive to achieve this goal while maximising the utility provided to the user[4].

Access to the Smart Battery interface allows us to determine the remaining battery capacity. Thus, with the user specified battery life expectancy, we can calculate the maximum average discharge rate of the battery acceptable to achieve the user's goal. If the current discharge rate exceeds this average discharge rate, the system runs the risk of not being able to meet the user's expectation[5]. This can be interpreted as energy contention and the processes responsible for the excess energy consumption should be charged proportional to their current energy consumption E_i.

More specifically, in intervals Δt the reduction of battery capacity ΔE is measured (courtesy of ACPI). If ΔE exceeds the maximum amount of energy E_{max} the system is allowed to use in that interval, we charge every process i proportional to the energy ΔE_i it consumed during that interval, thus $t_i x_i$ from equation 2 equals $\Delta E_i / \Delta E$. As the battery capacity is unlikely to decrease linearly, even under constant load, E_{max} needs to be recalculated periodically (e.g., the Odyssey prototype [8], implementing a similar mechanism, uses adaption intervals Δt of half a second). The model of identifying shadow prices for energy consumption is similar to the simple "slotted time" model discussed in [11] for network congestion prices.

Applications interpret the charges as feedback signals and may adapt their behaviour. In [8], the authors give a number of good examples of how applications can change their energy demands in application specific ways. Applications can adapt in various ways, making different trade-offs with respect to energy consumption. While the charges indicate the amount of adaption required, the detailed energy information provided to processes (i.e., the overall energy consumption E_i and its per resource breakdown E_{ij}) can be used to aid the adaption strategy. By limiting the amount of credits available to individual processes, applications are encouraged to perform adaption and the user may use different credit allocations to prioritise processes (e.g., to stop unimportant background processes from running when energy is scarce). Our experience with applying a similar resource management mechanism to more tradi-

[4]For desktop computers, the "goal" could instead be for the system not to need active cooling. An external cost of contention can then be identified, if active cooling is required. For server systems, a system administrator could set a target energy consumption and resource contention can be identified if this target is exceeded.

[5]If, however, even an idle or lightly loaded system cannot meet the user's expectation, the user should be informed, so that the user can reconsider the goal or his or her activities.

tion resources (network [18] and CPU [22]) shows that such a decentralised resource management can lead to resource allocations proportional to the credit allocation. We expect that similar results can be achieved for energy allocations.

Not all applications may be capable of adapting their energy requirements. For these applications a user has two options. Either the user decides that the application is important enough to run, in which case enough credits should be allocated to it to offset the maximum charging rate the application can incur; or the user does not value the application enough for it to be run at all. For applications, where no source code is available, application proxies, as in Puppeteer [5], can be deployed to transcode input data streams based on the feedback signals to manipulate the energy consumption of such applications.

It is worth pointing out that, in addition to the application specific energy adaption, basic policies can also be used by the system. For example, unused devices would be switched off by their device driver, the user could select a brightness level for the display appropriate for the environment the user is working in, and device drivers could put devices in low power states, potentially trading off energy consumption and performance based on stated user preferences and by observing device activity. These policies are orthogonal to the resource management system discussed in this paper.

4 Related Work

The energy accounting methodology, proposed in this paper is based on previous work by researchers investigating the detailed characteristics of energy consumption of mobile devices. Our proposed calibration process is very similar to the PowerMeasure tool, described in [20], which is used to characterise the energy consumption of Apple PowerBook laptops. The tool places each component of the laptop in different power states and observes the change in its instantaneous power consumption using the built-in battery hardware. The authors show that such an approach is feasible and that the observed values are within 0.1% to 5.6% of the externally measured power consumption. Ellis [6] describes a similar approach to determine the power consumption of a Palm Pilot in known power states.

The PowerScope [9] and Farkas et al. [7] utilise an external digital multimeter to measure the power consumption, while manipulating the target system. PowerScope then associates power samples with program counter samples to determine an energy profile of different software components and processes. In addition, Farkas et al. are using a set of micro-benchmarks to determine the energy consumption of the memory subsystem. A similar methodology is described in [3], which correlates various processor performance counter events with energy consumption.

Unlike these systems, we propose to use the power characteristics, gathered by such a methodology, to account the entire energy consumption of a system to individual processes. By taking advantage of the structure and the resource accounting mechanisms provided by the Nemesis operating system, we believe that it is possible to directly attribute the energy consumed by the individual devices to the processes using them.

Our proposal for the resource management has some similarities with Odyssey [8]. In Odyssey, a central resource manager monitors the system's energy consumption and attempts to meet a user-specified battery lifetime. To achieve this, Odyssey periodically samples the residual energy of the battery, predicts future energy demand, and notifies applications with an upcall if adaption is required. If more than one application is active, a simple priority scheme determines which applications are notified. In contrast, our economic based approach provides a more expressive feedback mechanism than the simple upcall mechanism used in Odyssey. Furthermore, the per process energy accounting provides more input into an application's adaption decisions.

The Milli Watt project [6, 24] seeks to raise the management of energy to first-class status among performance goals for operating system design and that applications should be involved in the management of energy. We believe, that to an extent, the approach outlined in this paper satisfies these aims.

Recently, the application of economic models to resource management in computer systems has received renewed attention, especially as a mechanism for congestion control and provisioning of QoS in communication networks. Significant contributions in this area have been made by economists (e.g., [21]), mathematicians (e.g., [17]), and computer scientists (e.g., [18]). In our previous work, we have applied similar techniques to CPU resource management [22]. In this paper, we extended this work by applying it to energy management in mobile computers.

5 Conclusions

We have argued that energy can be treated as just another resource provided that the operating system structure is appropriate. We have used the Nemesis OS as an example of an existing resource-centric operating system, and proposed a mechanism which allows us to account virtually all energy consumed by the system to individual processes. We believe that this can be achieved by accounting the energy consumption of individual devices, as determined by a calibration process, to the processes in proportion to their device usage. We argued that, as with any resource management system, accurate energy accounting to processes has to form the basis for an energy management system.

We have also investigated how a pricing mechanism, previously only applied to network congestion pricing and CPU resource management, can be applied to energy management. In such a system, the operating system charges individual processes for their energy consumption if the overall demand for energy exceeds the amount necessary to achieve a user-specified goal (e.g., a specified battery lifetime). These charges can be interpreted by applications as meaningful feedback signals, to which they can perform application-specific adaption, leading to a *decentralised* form of energy management.

While there are still many open implementation issues, e.g., the accuracy of the energy accounting process or details of the implementation of the pricing mechanism, we believe that we have made a strong case to support our claim that energy is just another resource.

Acknowledgement

We would like to thank Huw Evans and Peter Dickman for their helpful comments on earlier versions of this paper and the anonymous reviewers for the detailed feedback they provided.

References

[1] P. Barham. *Devices in a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, Oct. 1996.

[2] P. Barham. A fresh approach to File System Quality of Service. In *Proc. of the 7th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, St. Louis, MO, May 1997.

[3] F. Bellosa. The Benefits of Event-Driven Energy Accounting in Power Sensitive Systems. In *Proc. of the 9th ACM SIGOPS European Workshop*, pages 37–42, Kolding, Denmark, Sept. 2000.

[4] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol Implementation in a Vertically Structured Operating System. In *Proc. of the 22nd Annual Conference on Local Computer Networks*, pages 179–188, Nov. 1997.

[5] E. de Lara, D. Wallach, and W. Zwaenepoel. Puppeteer: Component-based Adaptation for Mobile Computing. In *Proceedings of the third USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, Mar. 2001.

[6] C. Ellis. The Case for Higher-Level Power Management. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, Mar. 1999.

[7] K. Farkas, J. Flinn, G. Back, D. Grunwald, and J. Anderson. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. In *Proc. of the Int. Conference on Measurements and Modeling of Computer Systems*, Santa Clara, CA, June 2000.

[8] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. of the 17th ACM SIGOPS Symposium on Operating Systems Principles*, pages 48–79, Kiawah Island Resort, SC, Dec. 1999.

[9] J. Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *Proc. of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, New Orleans, LA, Jan. 1999.

[10] S. I. Forum. *Smart Battery System Specifications: Smart Battery Data Specification*, Dec. 1998. Rev. 1.1.

[11] R. Gibbens and F. Kelly. Resource pricing and the evolution of congestion control. *Automatica*, 35(12), 1999.

[12] S. Hand. Self-Paging in the Nemesis Operating System. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, pages 73–86, New Orleans, LA, Feb. 1999.

[13] Intel and Microsoft. *Advanced Power Management (APM) BIOS Interface Specification*, Feb. 1996. Rev. 1.2.

[14] Intel, Microsoft and Toshiba. *Advanced Configuration and Power Interface Specification (ACPI)*, Feb. 1999. Rev. 1.0b.

[15] F. Kaashoek, D. Engler, G. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibilty on Exokernel Systems. In *Proc. of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, pages 52–65, Saint-Malo, France, Oct. 1997.

[16] F. Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8:33–37, 1997.

[17] F. Kelly, A. Maulloo, and D. Tan. Rate control in Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49(3):237–252, Mar. 1998.

[18] P. Key, D. McAuley, P. Barham, and K. Laevens. Congestion pricing for congestion avoidance. Technical Report MSR-TR-99-15, Microsoft Research, Feb. 1999.

[19] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, Sept. 1996.

[20] J. Lorch and A. Smith. Energy consumption of Apple Macintosh computers. *IEEE Micro*, 18(6):54–63, Nov. 1998.

[21] J. MacKie-Mason and H. Varian. Pricing Congestable Network Resources. *IEEE Journal on Selected Areas In Communications*, 13(7):1141–1149, Sept. 1995.

[22] R. Neugebauer and D. McAuley. Congestion Prices as Feedback Signals: An Approach to QoS Management. In *Proc. of the 9th ACM SIGOPS European Workshop*, pages 91–96, Kolding, Denmark, Sept. 2000.

[23] N. Stratford and R. Mortier. An Economic Approach to Adaptive Resource Management. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, Mar. 1999.

[24] A. Vahdat, A. Lebeck, and C. Ellis. Every Joule is Precious: The Case for Revisiting Operating System Design for Energy Efficiency. In *Proc. of the 9th ACM SIGOPS European Workshop*, pages 31–36, Kolding, Denmark, Sept. 2000.

PAST: A large-scale, persistent peer-to-peer storage utility

Peter Druschel
Rice University
Houston, TX 77005-1892, USA[*]
druschel@cs.rice.edu

Antony Rowstron
Microsoft Research Ltd.
Cambridge, CB2 3NH, UK
antr@microsoft.com

Abstract

This paper describes PAST, a large-scale, Internet based, global storage utility that provides high availability, persistence and protects the anonymity of clients and storage providers. PAST is a peer-to-peer Internet application and is entirely self-organizing. PAST nodes serve as access points for clients, participate in the routing of client requests, and contribute storage to the system. Nodes are not trusted, they may join the system at any time and may silently leave the system without warning. Yet, the system is able to provide strong assurances, efficient storage access, load balancing and scalability.

Among the most interesting aspects of PAST's design are (1) the Pastry location and routing scheme, which reliably and efficiently routes client requests between any pair of PAST nodes, has good network locality properties and automatically resolves node failures and node additions; (2) the use of randomization to ensure diversity in the set of nodes that store a file's replicas and to provide load balancing; and (3) the use of smartcards, which are held by each PAST user and issued by a third party called a broker. The smartcards support a quota system that balances supply and demand of storage in the system.

1. Introduction

There are currently many projects aimed at constructing peer-to-peer applications and understanding more of the issues and requirements of such applications and systems [1, 5, 2, 4]. Peer-to-peer systems can be characterized as distributed systems in which all nodes have identical capabilities and responsibilities and all communication is symmetric. We are developing PAST, an Internet based, peer-to-peer global storage utility, which aims to provide strong persistence, high availability, scalability, and anonymity of clients and storage providers.

While PAST offers persistent storage services, its access semantics differ from that of a conventional filesystem. Files stored in PAST are associated with a *fileId* that is quasi-uniquely associated with the file's content, name and owner[1]. This implies that files stored in PAST are *immutable* since a modified version of a file cannot be written with the same fileId as its original. Files can be shared at the owner's discretion by distributing (potentially anonymously) the fileId and, if necessary, a decryption key.

PAST does not support a *delete* operation for files. Instead, the owner of a file may *reclaim* the storage associated with a file. While the semantics of file deletion require that the file is removed when the operation completes, reclaim has weaker semantics, and simply means that a user can reuse the space, and the system no longer provides any guarantees about the availability of the file.

The PAST system is composed of nodes, where each node is capable of initiating and routing client requests to insert, retrieve, or reclaim a file. Optionally, nodes may also contribute storage to the system. The PAST nodes form a self-organizing network. After a new node arrival or a node failure, the system automatically restores all invariants after exchanging $O(logN)$ messages, where N is the total number of nodes in the system.

Inserted files are replicated across multiple nodes to ensure persistence and availability. The system ensures, with high probability, that the set of nodes over which a file is replicated is diverse in terms of geographic location, ownership, administrative entity, network connectivity, rule of law and so forth. Additional copies of popular files may be cached in any PAST node.

An efficient routing scheme, called *Pastry* [10], ensures that client requests to *insert* or *reclaim* a file are routed to each node that must store a replica of the file. Client requests to *retrieve* a file are reliably routed to a node that is "close in the network"[2] to the client that issued the request,

[*]Work done while visiting Microsoft Research, Cambridge, UK.

[1]The fileId is based on a secure hash of the file's content, name and owner. Therefore, it is extremely unlikely that files that differ in content, name, or owner have the same fileId.

[2]The notion of network proximity may be based on geographic loca-

among all live nodes that store the requested file. The number of PAST nodes traversed, as well as the number of messages exchanged while routing a client request is at most logarithmic in the total number of PAST nodes in the system.

A storage management scheme in PAST ensures that the global storage utilization in the system can approach 100%, despite widely differing file sizes and storage node capacities, and the lack of centralized control [11]. In a storage system where nodes are not trusted, a secure mechanism is also required that ensures a balance of storage supply and demand. In PAST, a third party (broker) issues smartcards for all users of the system. The smartcards support a quota system, which balances storage supply and demand and can be used by a broker to trade storage. The broker is not directly involved in the operation of the PAST network, and its knowledge about the system is limited to the number of smartcards it has circulated, their quotas and expiration dates.

Another issue in peer-to-peer systems, and particularly in storage and file-sharing systems, is privacy and anonymity. A provider of storage space used by others does not want to risk prosecution for content it stores, and clients inserting or retrieving a file may not wish to reveal their identity. Anderson [3] describes the ”the Gutenberg Inheritance” and motivates why such levels of privacy and anonymity are desirable.

PAST clients and storage providers need not trust each other, and place only limited trust in the broker. In particular, all nodes trust the broker to facilitate the operation of a secure PAST network by assigning and protecting appropriate keys for the smartcards, and by balancing storage supply and demand via responsible use of the quota system. On the other hand, users do not reveal to the broker (or anyone else) their identity, the files they are retrieving, inserting or storing. Each user holds an *initially unlinkable pseudonym* [7] associated with their smartcard. The pseudonym remains unlinkable to the user's identity, unless the user voluntarily reveals the binding. In addition, if desired a user may use multiple pseudonyms (i.e., smartcards) to obscure knowledge that certain operations were initiated by the same user.

In the following sections, we shall present an overview of PAST's design.

2. PAST architecture

Some of the key aspects of PAST's architecture are (1) the Pastry routing scheme, which routes client requests in less than $\lceil logN \rceil$ steps on average within a self-configuring, fault tolerant overlay network; (2) the use of randomization to ensure (probabilistic) storage load balancing and diversity of nodes that store replicas of a file, without the need for centralized control or expensive distributed agreement protocols; (3) a decentralized storage management and caching scheme that balances the storage utilization among the nodes as the total utilization of the system approaches 100%, and balances query load by caching copies of popular files close to interested clients; (4) the broker, which performs key management and ensures the integrity of the system; and, (5) the use of smartcards, which support a quota system to control storage supply and demand.

PAST is composed of nodes, where, in general, each node can act as a storage node and a client. The smartcard issued to the node provides a node identifier (nodeId), which is a 128 bit number chosen randomly from a uniform distribution and signed by the broker (using a public key cryptosystem maintained by the broker).

Files that are inserted into the PAST system are each assigned a fileId. A fileId is 160 bits in length, and is the secure hash (SHA-1) of the following data: a textual file name, a secure hash of the content, and the creator's smartcard id. Before a file is inserted, a write certificate is generated, which contains the fileId, file expiry date, the replication factor, the creation date and a secure hash of the content. The write certificate is signed by the smartcard of the file's creator.

When a file is inserted in PAST, the network routes the file to the k nodes whose node identifiers are numerically closest to the first 128 bits of the file identifier (fileId). Each of these nodes then stores a copy of the file. The replication factor k depends on the availability and persistence requirements of the file and may vary between files. A lookup request for a file is routed towards the live node with a nodeId that is numerically closest to the requested fileId.

This procedure ensures that (1) a file remains available as long as one of the k nodes that store the file is alive and reachable via the Internet; (2) the set of nodes that store the file is diverse in geographic location, administration, ownership, network connectivity, rule of law, etc.; and, (3) the number of files assigned to each node is roughly balanced. (1) follows from the properties of the PAST routing algorithm described in Section 2.2. (2) and (3) follow from the uniformly distributed random nodeId assigned to each storage site and the properties of a secure hash function (uniform distribution of hash values, regardless of the set of files).

2.1 Security

In discussing PAST's security, we make the following assumptions. We assume that it is computationally infeasible for an attacker to break the public-key cryptosystem and the secure hash function used by the broker and smartcards. It is assumed that an attacker can control individual

tion, number of network hops, bandwidth, delay, or a combination of these and other factors.

PAST nodes, but that they cannot control the behavior of the smartcard.

Smartcards are issued by a PAST broker. Each card holds a nodeId, assigned and signed by the broker, a private/public key pair unique to the smartcard, and the broker's public key. The smartcard's public key is signed by the broker for certification purposes. A smartcard generates and verifies various certificates used during insert and reclaim operations. It also maintains client storage quotas. In the following, we sketch the main security related functions.

Generation of nodeIds The smartcard provides a nodeId for its associated PAST node. This nodeId is generated using a secure random number generator and signed by the broker as part of the smartcard's manufacture. The random assignment of nodeIds probabilistically ensures uniform coverage of the space of nodeIds, and a random spread of nodeIds across geographic locations, countries, node operators, etc. Furthermore, the use of signed nodeIds prevents attacks involving malicious node operators trying to choose particular values for their nodeIds (for instance, to control all nodes that store a particular file).

Generation of write certificates and receipts The smartcard of a user wishing to insert a file into PAST issues a *write certificate*. The certificate contains a secure hash of the file's contents (computed by the client node, not the smartcard), the fileId (computed by the smartcard), a replication factor, a file expiration date, and is signed by the smartcard. During an insert operation, the write certificate allows each storing node to verify that (1) the user is authorized to insert the file into the system, (2) the content of the file arriving at the storing node have not been corrupted en route from the user's node, and (3) the fileId is valid (i.e., it is consistent with the content arriving at the node).

Each storage node that has successfully stored a copy of the file then issues and returns a *write receipt* to the client, which allows the client to (4) verify that k copies of the file have been created on nodes with adjacent nodeIds. (1) prevents clients from exceeding their storage quotas, (2) renders ineffective attacks by malicious nodes involved in the routing of an insert request that change the content, (3) prevents denial-of-service attacks where malicious clients try to exhaust storage at a subset of PAST nodes by generating bogus fileIds with nearby values, and (4) prevents a malicious node from suppressing the creation of k diverse replicas. During a retrieve operation, the write certificate is returned along with the file, and allows the client to verify that the content has not been corrupted.

Generation of reclaim certificates and receipts Prior to issuing a reclaim operation, the user's smartcard generates a *reclaim certificate*. The certificate contains the fileId, is signed by the smartcard and is included with the reclaim request that is routed to the nodes that store the file. When processing a reclaim request, the smartcard of a storage node first verifies that the signature in the reclaim certificate matches that in the write certificate stored with the file. This prevents users other than the owner of the file from reclaiming the file's storage. If the reclaim operation is accepted, the smartcard of the storage node generates a *reclaim receipt*. The receipt contains the reclaim certificate and the amount of storage reclaimed; it is signed by the smartcard and returned to the client.

Storage quotas The smartcard maintains storage quotas. Each user's smartcard is issued with a usage quota, depending on how much storage the client is allowed to use. When a write certificate is issued, an amount corresponding to the file size times the replication factor is debited against the quota. When the client presents an appropriate reclaim receipt issued by a storage node, the amount reclaimed is credited against the client's quota. This prevents clients from exceeding the storage quota they have paid for. A smartcard also specifies the amount of storage contributed by the associated node (possibly zero). Nodes are randomly audited to see if they can produce files they are supposed to store, thus exposing nodes that cheat by offering less storage than indicated by their smartcard.

In the following, we briefly discuss how some of the system's key properties are maintained.

Providing system integrity Several conditions underly the basic integrity of a PAST system. Firstly, to maintain load balancing among storage nodes, the nodeIds and fileIds must be uniformly distributed. The procedure for generating and verifying nodeIds and fileIds ensures that malicious nodes cannot bias this distribution. Secondly, there must be a balance between the sum of all client quotas (potential demand) and the total available storage in the system (supply). The broker ensures that balance, potentially using the monetary price of storage to regulate supply and demand. Thirdly, individual malicious nodes must be incapable of persistently denying service to a client. A randomized routing protocol, described in Section 2.2, ensures that a retried operation will eventually be routed around the malicious node.

Providing Persistence File persistence in PAST depends primarily on three conditions. (1) Unauthorized users are prevented from reclaiming a file's storage, (2) the file is initially stored on k storage nodes, and (3) there is sufficient diversity in the set of storage nodes that store a file. By issuing and requiring reclaim certificates, the smartcards ensure condition (1). (2) is enforced through the use of write receipts and (3) is ensured by the random distribution of nodeIds, which can't be biased by an attacker. The choice of a replication factor k must take into account the expected rate of transient storage node failures to ensure sufficient availability. In the event of storage node failures that involve loss of the stored files, the system automatically restores k copies of a file as part of the failure recovery process.

Providing data privacy and integrity Users may use encryption to protect the privacy of their data, using a cryptosystem of their choice. Data encryption does not involve the smartcards. Data integrity is ensured by means of the write certificates issued by the smartcards.

Providing anonymity A user's smartcard signature is the only information associating a stored file or a request with the responsible user. The association between a smartcard and the user's identity is only known to the user, unless the user voluntarily releases this information. Anonymity of storage nodes is similarly ensured because the node's smartcard signature is not linkable to the identity of the node operator.

Space limitations prevent us from a full discussion of PAST's security model. Next, we briefly reflect on the role of smartcards in PAST. Many of the functions performed by the smartcards in the current design could be safely performed by the (untrusted) node software. Indeed, the use of smartcards is not fundamental to PAST's design. However, the smartcards serve two important purposes, given today's technology.

First, the smartcards maintain storage quotas. Without the trusted hardware provided by the smartcard, it is difficult to prevent a malicious node from cheating against its quota. Second, the smartcards are a convenient medium through which a user can obtain necessary credentials from the broker in an anonymous fashion. A user can obtain a smartcard with the desired quota from a retail outlet in exchange for cash, without any risk of revealing their identity. Obtaining the credentials on-line carries the risk of revealing the user's identity or leaking sensitive information to third parties.

There are disadvantages to the use of smartcards. First, clients need to obtain a physical device and periodically replace it (e.g., every year) to ensure key freshness. Second, sophisticated, resource-rich attackers could compromise a smartcard. However, since the maximal gain is to cheat against the storage quote for a limited time, we believe there is insufficient incentive for a high-tech attack.

Finally, there are performance costs due to the limited processing speed and I/O performance of smartcards. Fortunately, read operations involve no smartcard operations. (In fact, read-only users don't need a smartcard). Write operations require a write certificate verification and a write confirmation generation, and we expect that a smartcard keeps up with the speed of a single disk. Larger storage nodes use multiple smartcards, and very large storage nodes may require more powerful tamperproof hardware. Professionally managed storage sites also have the option of contracting with a broker, thus obviating the need for trusted hardware in exchange for a loss in anonymity.

Future Internet technologies like an anonymous payment and micropayment infrastructure could obviate the need for smartcards in PAST. For instance, micro-payments could be used to balance the supply and demand of storage without quotas, and anonymous payment transactions could make it possible for a user to safely and anonymously obtain necessary credentials from the broker. We plan to explore emerging technologies and reevaluate the use of smartcards as alternatives become available.

It is to be noted that multiple PAST systems, with separate brokers, can co-exist in the Internet. In fact, we envision many competing brokers, where a client can access files in the entire system, but can only store files on storage nodes affiliated with the client's broker. Furthermore, it is possible to operate isolated PAST systems that serve a mutually trusting community without a broker or smartcards. In these cases, a virtual private network (VPN) can be used to interconnect the system's nodes.

In the remainder of this paper, we give a brief overview of other interesting aspects of PAST, namely its routing, self-configuration schemes, storage management and caching schemes.

2.2 Pastry

We now briefly describe Pastry, the location and routing scheme used by PAST. Given a fileId, Pastry routes the associated message towards the node whose nodeId is numerically closest to the 128 most significant bits (msb) of the fileId, among all live nodes. Given the invariant that a file is stored on the k nodes whose nodeIds are numerically closest to the 128 msbs of the fileId, it follows that a file can be located unless all k nodes have failed simultaneously (i.e., within a recovery period).

Pastry is highly efficient, scalable, fault resilient and self-configuring. Assuming a PAST network consisting of N nodes, Pastry can route to the numerically closest node to a given fileId in less than $\lceil log_{2^b} N \rceil$ steps on average (b is a configuration parameter with typical value 4). With concurrent node failures, eventual delivery is guaranteed unless $\lfloor l/2 \rfloor$ nodes with *adjacent* nodeIds fail simultaneously (l is a configuration parameter with typical value 32).

The tables required in each PAST node have only $(2^b - 1) * \lceil log_{2^b} N \rceil + 2l$ entries, where each entry maps a nodeId to the associated node's IP address. Moreover, after a node failure or the arrival of a new node, the invariants in all affected routing tables can be restored by performing $O(log_{2^b} N)$ remote procedure calls (RPCs). In the following, we give a brief overview of the Pastry routing scheme.

For the purpose of routing, nodeIds and fileIds are thought of as a sequence of digits with base 2^b. A node's routing table is organized into levels with $2^b - 1$ entries each. The $2^b - 1$ entries at level n of the routing table each refer to a node whose nodeId shares the present node's nodeId in the first n digits, but whose $n + 1$th digit has one of the $2^b - 1$ possible values other than the $n + 1$th digit in the present node's id. Note that an entry in the routing table points to one of potentially many nodes whose nodeId have the appropriate prefix. Among such nodes, the one closest to the present node (according to the proximity metric) is chosen in practice.

In addition to the routing table, each node maintains pointers to the set of l nodes whose nodeIds are numerically closest to the present node's nodeId, irrespective of prefix. (More precisely, the set contains $l/2$ nodes with larger and $l/2$ with smaller nodeIds). This set is called the *leaf set*.

In each routing step, a node normally forwards the message to a node whose nodeId shares with the fileId a prefix that is at least one digit (or b bits) longer than the prefix that the fileId shares with the present node's id. If no such node exists, the message is forwarded to a node whose nodeId shares a prefix with the fileId as long as the current node, but is numerically closer to the fileId than the present node's id. It follows from the definition of the leaf set that such a node exists in the leaf set unless $\lfloor l/2 \rfloor$ adjacent nodes in the leaf set have failed simultaneously.

Locality In the following, we turn our attention to the properties of the Pastry routing scheme with respect to the network proximity metric. Pastry can normally route messages to any node in $\lceil log_{2^b} N \rceil$ steps. Another question is what distance (in terms of the proximity metric) a message is traveling. Recall that the entries in the node routing tables are chosen to refer to the nearest node with the appropriate nodeId prefix. As a result, in each step a message is routed to the nearest node with a longer prefix match (by one digit). While this local decision process clearly can't achieve globally shortest routes, simulations have shown the the average distance travelled by a message is only 40% higher than the distance of the source and destination in the underlying network [10].

Moreover, since Pastry always takes the locally shortest step towards a node that shares a longer prefix with the fileId, messages have a tendency to first reach a node, among the k nodes that store the requested file, that is near the client (according to the proximity metric). One experiment shows that among 5 replicated copies, Pastry is able to find the nearest copy in 76% of all lookups and it finds one of the two nearest copied in 92% of all lookups [10].

Node addition and failure A key design issue in Pastry is how to efficiently and dynamically maintain the node state, i.e., the routing table, leaf set and neighborhood sets, in the presence of node failures, node recoveries, and new node arrivals. The protocol is described and evaluated in [10].

Briefly, an arriving node with the new nodeId X can initialize its state by contacting a nearby node A (according to the proximity metric) and asking A to route a special message to the existing node Z with nodeId numerically closest to X. X then obtains the leaf set from Z, the neighborhood set from A, and the ith row of the routing table from the ith node encountered along the route from A to Z. One can show that using this information, X can correctly initialize it state and notify interested nodes that need to know of its arrival, thereby restoring all of Pastry's invariants.

To handle node failures, neighboring nodes in the nodeId space (which are aware of each other by virtue of being in each other's leaf set) periodically exchange keep-alive messages. If a node is unresponsive for a period T, it is presumed failed. All members of the failed node's leaf set are then notified and they update their leaf sets to restore the invariant. Since the leaf sets of nodes with adjacent nodeIds overlap, this update is trivial. A recovering node contacts the nodes in its last known leaf set, obtains their current leafs sets, updates its own leaf set and and then notifies the members of its new leaf set of its presence. Routing table entries that refer to failed nodes are repaired lazily; the details are described in [10].

Fault-tolerance The routing scheme as described so far is deterministic, and thus vulnerable to malicious or failed nodes along the route that accept messages but do not correctly forward them. Repeated queries could thus fail each time, since they are likely to take the same route.

To overcome this problem, the routing is actually randomized. To avoid routing loops, a message must always be forwarded to a node that shares at least as long a prefix with, but is numerically closer to the destination node in the namespace than the current node. The choice among multiple such nodes is random. In practice, the probability distribution is heavily biased towards the best choice to ensure low average route delay. In the event of a malicious or failed node along the path, the query may have to be repeated several times by the client, until a route is chosen that avoids the bad node.

2.3 Storage management and caching

The statistical assignment of files to storage nodes in PAST approximately balances the number of files stored at each node. However, non-uniform storage node capacities and file sizes require more explicit storage load balancing to permit graceful behavior under high global storage utilization; and, non-uniform popularity of files requires caching to minimize fetch distance and to balance the query load.

PAST employs a storage management scheme that achieves high global storage utilization while rejecting few file insert requests. The scheme relies only on local coordination among the nodes in a leaf set, and imposes little overhead. Experimental results show that PAST can achieve global storage utilization in excess of 95%, while the rate of rejected file insertions remains below 5% and failed insertions are heavily biased towards large files [11].

Caching in PAST allows any node to retain an additional copy of a file. This caching is effective in achieving query load balancing, high throughput for popular files, and it reduces fetch distance and network traffic. PAST's storage management and caching are described in detail in [11].

3. Related work and conclusion

There are currently many peer-to-peer systems under development. Among the most prominent are file sharing facilities, such as Gnutella [2], Freenet [5], and Napster [1]. These systems are intended for the large-scale sharing of music; persistence and reliable content location are not required in this environment. PAST instead is a large-scale storage utility that aims at combining scalability and self-configuration with strong persistence. In this regard, it is more closely related with projects like OceanStore [6], FarSite [4], FreeHaven [9], Publius [13] and Eternity [3].

Like PAST, OceanStore provides a global, persistent storage utility on top of an untrusted, unreliable infrastructure. However, PAST focuses on providing a simple, lean storage abstraction for persistent, immutable files with the intention that more sophisticated storage semantics be build on top of PAST if needed. OceanStore provides a more general storage abstraction that supports serializable updates on widely replicated and nomadic data.

FarSite and SFS have more traditional filesystem semantics, while PAST is more targeted towards global, archival storage. Farsite uses a distributed directory service to locate content; this is very different from PAST's Pastry scheme, which integrates content location and routing. FreeHaven, Publius and Eternity are more focused on providing anonymity and survivability of data in the presence of a variety of threats.

Pastry, PAST's routing scheme, bears some similarity to the work by Plaxton et al. [8]. The general approach of routing using prefix matching on the fileId is used in both systems, which can be seen as a generalization of hypercube routing. However, in the Plaxton scheme there is a special node associated with each file, which forms a single point of failure. Also, Plaxton does not handle automatic node integration and failure recovery.

Oceanstore uses a two phase approach to content location and routing. The first stage is probabilistic, using a generalization of Bloom filters. If that stage fails to find an object, then a location and routing scheme called Tapestry is used [14]. Tapestry is based on Plaxton et al. but extends that earlier work in several dimensions. Like Pastry, Tapestry replicates objects for fault resilience and availability and supports dynamic node addition and recovery from node failures. However, Pastry and Tapestry differ in the approach they take for replicating files and in the way they achieve locality. Another closely related scheme is location and routing scheme is Chord [12].

A prototype of PAST operates in a simulated network environment. We have performed tests with up to 100,000 PAST nodes and several million files. Routing, self-configuration, file storage/retrieval, caching and storage load balancing are fully functional; early experience and performance results are very encouraging [10, 11]. Plans for the immediate future are to verify PAST's security model more formally, and to complete an implementation that can be deployed in the Internet.

References

[1] Napster. http://www.napster.com/.

[2] The Gnutella protocol specification, 2000. http://dss.clip2.com/GnutellaProtocol04.pdf.

[3] R. Anderson. The eternity service. In *Proc. PRAGOCRYPT'96*, pages 242–252. CTU Publishing House, 1996. Prague, Czech Republic.

[4] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proc. SIGMETRICS'2000*, pages 34–43, 2000.

[5] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, July 2000. ICSI, Berkeley, CA, USA.

[6] J. K. et al. Oceanstore: An architecture for global-scale persistent store. In *Proc. ASPLOS'2000*, November 2000.

[7] A. Pfitzmann and M. Köhntopp. Anonymity, unobservability, and pseudonymity - a proposal for terminology, Apr. 2001. http://www.koehntopp.de/marit/pub/anon/Anon_Terminology_IHW.pdf.

[8] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–280, 1999.

[9] D. M. Roger Dingledine, Michael J. Freedman. The Free Haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[10] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, Jan. 2001. http://www.research.microsoft.com/ antr/PAST/.

[11] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility, Mar. 2001. http://www.research.microsoft.com/ antr/PAST/.

[12] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. Technical Report TR-819, MIT, March 2001.

[13] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[14] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service

Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger
Robert Morris, Ion Stoica[*], Hari Balakrishnan
MIT Laboratory for Computer Science
{*fdabek, emma, kaashoek, karger, rtm, hari*}*@lcs.mit.edu*
`http://pdos.lcs.mit.edu/chord`

Abstract

We argue that the core problem facing peer-to-peer systems is locating documents in a decentralized network and propose Chord, a distributed lookup primitive. Chord provides an efficient method of locating documents while placing few constraints on the applications that use it. As proof that Chord's functionality is useful in the development of peer-to-peer applications, we outline the implementation of a peer-to-peer file sharing system based on Chord.

1 Introduction

The peer-to-peer architecture offers the promise of harnessing the resources of vast numbers of Internet hosts. The primary challenge facing this architecture, we argue, is efficiently locating information distributed across these hosts in a decentralized way. In this paper we present Chord, a distributed lookup service that is both scalable and decentralized and can be used as the basis for general purpose peer-to-peer systems.

A review of the features included in recent peer-to-peer systems yields a long list. These include redundant storage, permanence, efficient data location, selection of nearby servers, anonymity, search, authentication, and hierarchical naming. Chord does not implement these services directly but rather provides a flexible, high-performance lookup primitive upon which such functionality can be efficiently layered. Our design philosophy is to separate the lookup problem from additional functionality. By layering additional features on top of a core lookup service, we believe overall systems will gain robustness and scalability.

In contrast, when these application-level features are an integral part of the lookup service the cost is often limited scalability and diminished robustness. For example, Freenet [5] [6] is designed to make it hard to detect which hosts store a particular piece of data, but this feature prevents Freenet from guaranteeing the ability to retrieve data.

Chord is designed to offer the functionality necessary to implement general-purpose systems while preserving maximum flexibility. Chord is an efficient distributed lookup system based on consistent hashing [10]. It provides a unique mapping between an identifier space and a set of nodes. A node can be a host or a process identified by an IP address and a port number; each node is associated with a Chord identifer. Chord maps each identifier a to the node with the smallest identifier greater than a. This node is called the *successor* of a.

By using an additional layer that translates high level names into Chord identifiers, Chord may be used as a powerful lookup service. We will outline the design of a distributed hash table (DHASH) layer and of a peer-to-peer storage application based on the Chord primitive. Figure 1 shows the distribution of functionality in the storage application.

Layer	Function
Chord	Maps identifiers to successor nodes
DHASH	Associates values (blocks) with identifiers
Application	Provides a file system interface

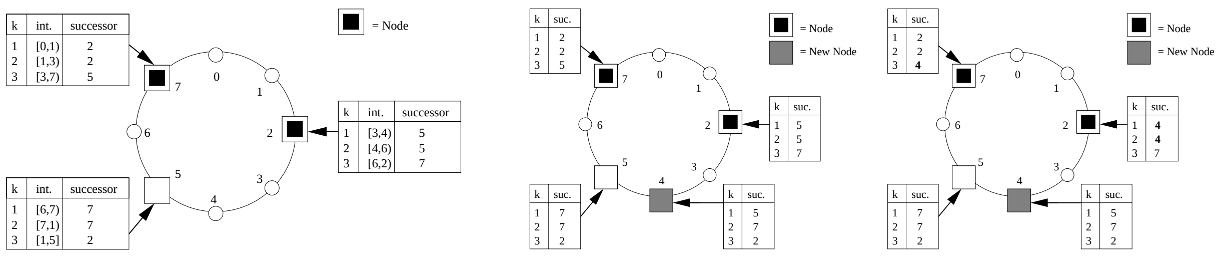
Figure 1: A layered Chord application

Chord is efficient: determining the successor of an identifier requires that $O(\log N)$ messages be exchanged with high probability where N is the number of servers in the Chord network. Adding or removing a server from the network can be accomplished, with high probability, at a cost of $O(\log^2 N)$ messages.

The rest of this position paper outlines the algorithms used to implement the Chord primitive (Section 2), describes how Chord can be used to build peer-to-peer storage systems (Section 3), summarizes the current implementation status of the system (Section 4), identifies some open research problems (Section 5), relates Chord to other work (Section 6), and summarizes our conclusions (Section 7).

[*] University of California, Berkeley. istoica@cs.berkeley.edu

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933.



(a) Find successor (b) Initialize routing tables (c) Update other routing tables

Figure 2: The Chord algorithm in a three-bit identifier space

2 Chord

Chord uses consistent hashing [10] to map nodes onto an m-bit circular identifer space. In particular, each identifier a is mapped to the node with the least identifier greater or equal to a in the circular identifier space. This node is called the *successor* of a.

To implement the *successor* function, all nodes maintain an m-entry routing table called the *finger* table. This table stores information about other nodes in the system; each entry contains a node identifier and its network address (consisting of an IP address and a port number). The k-th entry in the finger table of node r is the smallest node s that is greater than $r + 2^{k-1}$. Node s is also termed the order-k successor of node r. The number of unique entries in the finger table is $O(\log N)$. The finger table can also be thought of in terms of m identifier intervals corresponding to the m entries in the table: the order-k interval of a node r is defined as $((r + 2^{k-1}) \bmod 2^m, (r + 2^k) \bmod 2^m]$. Figure 2(a) shows a simple example in which m=3 and three nodes 2, 5, and 7 are present. The immediate successor of node 5 is the successor of $(5 + 2^0) \bmod 2^3 = 6$ or node 7.

Each node also maintains a pointer to its immediate predecessor. For symmetry, we also define the corresponding immediate successor (identical to the first entry in the finger table). In total, each node must maintain a finger table entry for up to $O(\log N)$ other nodes; this represents a significant advantage over standard consistent hashing which requires each node to track almost all other nodes.

2.1 Evaluating the successor function

Since each node maintains information about only a small subset of the nodes in the system, evaluating the successor function requires communication between nodes at each step of the protocol. The search for a node moves progressively closer to identifying the successor with each step.

A search for the successor of f initiated at node r begins by determining if f is between r and the immediate successor of r. If so, the search terminates and the successor of r is returned. Otherwise, r forwards the search request to the largest node in its finger table that precedes f; call this node s. The same procedure is repeated by s until the search terminates.

For example, assume that the system is in a stable state (all routing tables contain correct information) and a search is initiated at node 2 of Figure 2(a) for the successor of identifier 6. The largest node with an identifier smaller than 6 is 5. The target of the search, 6, is in the interval defined by 5 and its successor (7); therefore 7 is returned value.

The algorithm is outlined above in *recursive* form: if a search request requires multiple steps to complete, the n^{th} step is initiated by the $(n-1)^{th}$ node on behalf of the initiator. The successor function may also be implemented *iteratively*. In an iterative implementation, the initiating node is responsible for making requests for finger table information at each stage of the protocol. Both implementation styles offer advantages: an iterative approach is easier to implement and relies less on intermediary nodes, while the recursive approach lends itself more naturally to caching and server selection (described in Section 3).

2.2 Node insertion

When a new node r joins the network it must initialize its finger table; existing nodes must also update their tables to reflect the existence of r (see Figure 2(b) and Figure 2(c)).

If the system is in a stable state, a new node r can initialize its finger table by querying an existing node for the respective successors of the lower endpoints of the k intervals in r's table. Although we omit the details here, nodes whose routing information is invalidated by r's addition can be determined using r's finger table and by following predecessor pointers: these nodes are instructed by r to update their tables.

2.3 Additional algorithm details

Several additional details of the Chord protocol are merely mentioned here in the interest of brevity; a complete description of the Chord primitive is given by Stoica et al. [17]. Removing a node from the network involves a sim-

```
void event_register ((fn)(int))
ID next_hop (ID j, ID k)
```

Figure 3: Exposing Chord layer information. The `event_register` function arranges for `fn` to be called when a node with an ID near the registrant's joins or leaves the network. `next_hop` performs one step of the evaluation of the successor function and returns the intermediate result (a finger table entry).

ilar series of steps as adding a node. Parallel joins, parallel exits, and failures are handled by maintaining the invariant that all nodes are aware of their immediate successor and predecessor, and by allowing the remaining entries of nodes' finger tables to converge to the stable state over time. Handling failures also requires that nodes store k successors in addition to the immediate successor.

2.4 The chord library API

The Chord library is intended to be used in a layered design where it provides the base location functionality. Two design principles facilitate the the use of Chord in a layered architecture: minimum functionality and exposed information. By minimizing the amount of functionality embedded in Chord, we minimize the constraints we place on higher levels which depend on Chord.

In our initial experiments with systems based on Chord, we found that larger systems were constrained not because Chord provides an inflexible feature set, but because higher layers desired access to the internal state of Chord during its computation.

To provide this access while still preserving the abstraction barrier we allow layers to register callback functions for events they are interested in (see Figure 3) and to evaluate the successor function one step at a time. `next_hop(j, k)` sends a message to node `j` asking `j` for the smallest entry in its finger table greater than `k`. This allows callers to control the step-by-step execution of the Chord lookup algorithm.

For example, the `DHASH` layer (described in section 3.1) uses the callback interface to move values when nodes join or leave the system. `DHASH` also evaluates the successor function step by step to perform caching on search paths.

3 Building on Chord

To illustrate the usefulness of the Chord API we will outline the design of layers that could be built on the basic Chord primitive. These layers would be useful in a larger peer-to-peer file sharing application. This application should allow a group of cooperating users to share their network and disk resources. Possible users of the application might be a group of open source developers who wish to make a software distribution available, but individually do not have network resources to meet demand.

```
err_t insert(void *key, void *value)
void * lookup(void *key)
```

Figure 4: The DHASH API (a) Inserts value under key (b) returns value associated with key or NULL if key does not exist

3.1 Distributed hash service

Chord is not a storage system: it associates keys with nodes rather than with values. A useful initial extension to this system is a distributed hash table (DHASH). The API for this layer is shown in Figure 4.

`DHASH::insert` can be implemented by hashing `key` to produce a 160-bit Chord identifier k, and storing `value` at the successor of k. A `DHASH::lookup` request is handled analogously: `key` is hashed to form k and the successor of k is queried for the value associated with `key`. The transfer of value data to and from nodes is accomplished by an additional RPC interface which is separate from that exported by Chord.

Values introduce a complication: when nodes leave or join the system, the successor node of a given key may change. To preserve the invariant that values are stored at the successor of their associated keys, `DHASH` monitors the arrival and departure of nodes using the callback interface provided by Chord and moves values appropriately. For example, if the value associated with key 7 is stored on node 10 and node 9 joins the system, that value will be transferred to node 9.

Because it is based on Chord, `DHASH` inherits Chord's desirable properties: performing a lookup operation requires $O(\log N)$ RPCs to be issued and does not require any centralized control. The `DHASH` layer imposes an additional cost of transferring $O(\frac{1}{N})$ of the keys in the system each time a node joins or leaves the system.

3.2 Achieving reliability

The `DHASH` layer can also exploit the properties of Chord to achieve greater reliability and performance. To ensure that lookup operations succeed in the face of unexpected node failures, `DHASH` stores the value associated with a given key not only at the immediate successor of that key, but also at the next r successors. The parameter r may be varied to achieve the desired level of redundant storage.

The tight coupling between `DHASH`'s approach to replication and Chord's (both use knowledge of a node's immediate successors) is typical of the interaction we hope to see between Chord and higher layers.

3.3 Improving performance

To improve `DHASH` lookup performance, we exploit a property of the Chord lookup algorithm: the paths that searches for a given successor (from different initiating nodes) take through the Chord ring are likely to intersect. These intersections are more likely to occur near the target of the search where each step of the search algorithm makes a smaller 'hop' through the identifier space and provide an opportunity to cache data. On every successful lookup operation of a pair (k, v), the target value, v, is cached at each node in the path of nodes traversed to determine the successor of k (this path is returned by Chord's successor function).

Subsequent lookup operations evaluate the successor function step by step using the provided `next_hop` method and query each intermediate node for v; the search is terminated early if one of these nodes is able to return a previously cached v.

As a result, values are "smeared" around the Chord ring near corresponding successor nodes. Because the act of retrieving a document caches it, popular documents are cached more widely than unpopular documents; this is a desirable side-effect of the cache design. Caching reduces the path length required to fetch a value and therefore the number of messages per operation: such a reduction is important given that we expect that latency of communication between nodes to be a serious performance bottleneck facing this system.

3.4 Denial of service

The highly distributed nature of Chord helps it resist many but not all denial of service attacks. For instance, Chord is resistant to attacks that take out some network links since nodes nearby in identifier space are unlikely to have any network locality. Additional steps are taken to preclude other attacks.

A Chord-based storage system could be attacked by inserting such a large volume of useless data into the system that legitimate documents are flushed from storage. By observing that the density of nodes nearby any given node provides an estimate of the number of nodes in the system we can partially defend against this attack by limiting the number of blocks any one node can store in the system. We make a local decision to fix a block quota based on the number of nodes in the system, effectively enforcing a fixed quota for each user on the whole system.

Nodes that could pick their own identifiers could effectively delete a piece of data from the system by positioning themselves as the data's successor and then failing to store it when asked to. This attack can be prevented by requiring that node identifiers correspond to a hash of a node's IP address, a fact which can be verified by other nodes in the system.

Malicious nodes could fail to execute the Chord protocol properly resulting in arbitrarily incorrect behavior. A single misbehaving node can be detected by verifying its responses with those of other, presumably cooperative, nodes. For instance, if a node l reports that its successor is s, we can query s for its predecessor which should be l. A group of such nodes could cooperate to make a collection of nodes appear to be a self-consistent Chord network while excluding legitimate nodes. We have no decentralized solution to this problem and rely instead on the legitimacy of the initial 'bootstrap' node to avoid this attack.

3.5 Designing a storage system: balancing load

In using Chord as the core of a peer-to-peer storage system we are faced with the problem of efficiently distributing load among nodes despite wide variations in the popularity of documents. In building this system we must consider how to map documents to nodes and at what granularity to store documents.

One might consider using DHASH directly as a peer-to-peer storage system. In this design, the contents of a document are directly inserted into the DHASH system keyed by the hash of either the contents of the document or, perhaps, a human readable name. If one document becomes highly popular, however, the burden of delivering that document will not be distributed. The caching scheme described in Section 3.3 helps for small documents, but is not practical for very large documents.

An alternate approach uses DHASH as a layer of indirection: DHASH maps document identifiers to a list of IP addresses where that document was available. In this design DHASH functions analogously to the DNS system but does not depend on a special set of root servers as DNS does. Once an IP address is selected, documents are retrieved using some other transfer protocol (HTTP, SSL, SFS etc.).

Maintaining a dynamically updated list of potential servers for any document solves the problem of popular documents by distributing load among all of the servers in the list. However, this design requires that optimizations such as caching and redundant storage be implemented twice: once in the Chord stack and again in the transfer protocol. We desire a tighter coupling between the solution to the popular document problem and mechanisms of the Chord protocol.

This coupling can be achieved by using Chord to map pieces of documents (blocks), rather than whole documents, to servers. In this scheme, files are broken into blocks and each block is inserted into the `DHASH` layer using the cryptographic hash of the block's contents as a key. A piece of meta-data, equivalent to an inode in a traditional file system, is also inserted into the system to provide a single name for the file. The equivalence to a file system can be extending to include directories as well; in our prototype implementation, names map to a directory of documents which is mapped into the user's local namespace when accessed.

This approach aggressively spreads a single large document across many servers, thus distributing the load of serving it. It also inherits the reliability and performance enhancements of the DHASH layer with little or no additional effort. One might note that documents smaller than the block size are still served by a single node: we count on our caching scheme to distribute these documents and the load of serving them if they become popular.

The major drawback of this scheme derives from the same property that made it desirable: because we spread a single document across many servers, for each document we fetch we must pay the cost of several DHASH lookups (and thus several evaluations of the successor function). A naive implementation might require $\frac{S \times L \times \log N}{B}$ seconds to fetch an S byte document where N is the number of servers in the network, B is the block size and L is the average latency of the network. We hope to hide most of this latency through aggressive prefetching of data and by selecting a server from the redundant set which is near (in the network) the requesting node.

3.6 Authenticity

A Chord-based file system could achieve authenticity guarantees through the mechanisms of the SFS read-only server [9]. In SFSRO, file system blocks are named by the cryptographic hash of their contents, an inherently unforgeable identifier. To name file systems we adopt self-certifying pathnames [14]: The block containing the root inode of a file system is named by the public key of the publisher and signed by that public key. The DHASH layer can verify that the root inode is correctly signed by the key under which it is inserted. This prevents unauthorized updates to a file system. Naming file systems by public key does not produce easily human readable file names; this is not a serious shortcoming, however, in a hypertext environment, or one that is indexed or provides symbolic links.

4 Status

The system described is under development. The Chord protocol has been designed, implemented, and tested[1]. Results of testing with up to 1,000 nodes on the Berkeley Millennium Cluster demonstrate that Chord's performance scales well with the size of the system. We have also implemented the DHASH layer and a file system; in the same testing environment and on a geographically diverse network both demonstrated good load balancing properties.

5 Open problems

A number of open problems face applications built on the Chord framework.

[1]The delete operation has not been implemented yet

Our design deliberately separates questions of anonymity and deniability from the location primitive. These properties are difficult to add to the Chord system given the strong mapping between a document and the node which is responsible for serving that document. We speculate than overlaying a mix-network [4] on Chord might allow for anonymous publishing and reading.

Collecting an index of all documents stored in Chord is a straightforward operation: an indexer might visit every node in the Chord system by following successor pointers. Storing an index and servicing queries without resort to a central authority remains an open question, however. Alternatively we could provide a Chord to WWW gateway and rely on existing WWW indexing services.

Directing requests to servers nearby in the network topology is important to reducing the latency of requests. To do so requires measuring the performance of servers in the system. However, because Chord aggressively distributes documents to unrelated servers, in a large network we are not likely to visit the same server multiple times; this makes maintaining server performance metrics difficult.

6 Related work

There has been previous work in the area of decentralized location systems. Chord is based on consistent hashing [10]; its routing information may be thought of as a one-dimensional analogue of the GRID [12] location system. OceanStore [11] uses a distributed data location system described by Plaxton et al. [7], which is more complicated than Chord but offers proximity guarantees. CAN uses a d-dimensional Cartesian coordinate space to implement a distributed hash table data structure [16]. CAN operations are easy to implement, but an aditional maintenance protocol is required to periodically remap the identifier space onto nodes. The Chord algorithm is also very similar to the location algorithm in PAST [15].

Anonymous storage systems such as Freenet [5], Publius [13] and the Free Haven Project [8] use encryption, probabilistic routing, or secret-sharing schemes to guarantee clients and publishers anonymity. This anonymity guarantee often leads to design compromises that limit reliability and performance. Chord separates problems like these from the design of routing and file transfer protocols.

Napster [2], Gnutella [1], and Ohaha [3] provide a non-anonymous file sharing service similar to that of the sharing application presented here. Chord's location algorithm is more efficient than Gnutella's broadcast based routing; the decentralized nature of Chord eliminates a single point of failure present in Napster. The Ohaha system [3] uses a consistent hashing-like algorithm for ID mapping, and a Freenet-style method of document retrieval; it shares some of the weaknesses of Freenet.

7 Conclusions

The performance and reliability of existing peer-to-peer systems have been limited by inflexible architectures that attempt to find one solution for many problems. By using the Chord primitive to separate the problem of location from the problems of data distribution, authentication and anonymity, peer-to-peer systems are able to decide where to compromise and as a result offer better performance, reliability and authenticity.

References

[1] Gnutella website. http://gnutella.wego.com.

[2] Napster. http://www.napster.com.

[3] Ohaha. http://www.ohaha.com/design.html.

[4] David Chaum. Untraceable electronic mail, return addresses and digital pseudonyms. *Communications of the A.C.M.*, 24(2):84–88, 1981.

[5] Ian Clarke. A distributed decentralised information storage and retrieval system. Master's thesis, University of Edinburgh, 1999.

[6] Ian Clarke, Oscar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, California, June 2000. http://freenet.sourceforge.net.

[7] C.Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA*, pages 311–320, Newport, Rhode Island, June 1997.

[8] Roger Dingledine, David Molnar, and Michael J. Freedman. The Free Haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[9] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 181–196, San Diego, California, October 2000.

[10] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[11] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Boston, MA, November 2000.

[12] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (MobiCom '00)*, pages 120–130, Boston, Massachusetts, August 2000.

[13] Aviel D. Rubin Marc Waldman and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[14] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 124–139, Kiawah Island, South Carolina, December 1999.

[15] Antony Rowstron Peter Druschel. Past: Persistent and anonymous storage in a peer-to-peer networking environment. In *Proceedings of the 8th Conference on Hot Topics in Operating Systems (HotOS 2001)*, May 2001.

[16] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM 2001*, August 2001.

[17] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM 2001*, August 2001. An early version appeared as LCS TR-819 available at `http://www.pdos.lcs.mit.edu/chord/papers`.

Herald: Achieving a Global Event Notification Service

Luis Felipe Cabrera, Michael B. Jones, Marvin Theimer

Microsoft Research, Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
USA

{cabrera, mbj, theimer}@microsoft.com
http://research.microsoft.com/~mbj/, http://research.microsoft.com/~theimer/

Abstract

This paper presents the design philosophy and initial design decisions of Herald: a highly scalable global event notification system that is being designed and built at Microsoft Research. Herald is a distributed system designed to transparently scale in all respects, including numbers of subscribers and publishers, numbers of event subscription points, and event delivery rates. Event delivery can occur within a single machine, within a local network or Intranet, and throughout the Internet.

Herald tries to take into account the lessons learned from the successes of both the Internet and the Web. Most notably, Herald is being designed, like the Internet, to operate correctly in the presence of numerous broken and disconnected components. The Herald service will be constructed as a set of protocols governing a federation of machines within cooperating but mutually suspicious domains of trust. Like the Web, Herald will try to avoid, to the extent possible, the maintenance of globally consistent state and will make failures part of the client-visible interface.

1. Introduction

The function of event notification systems is to deliver information sent by event publishers to clients who have subscribed to that information. Event notification is a primary capability for building distributed applications. It underlies such now-popular applications as “instant messenger” systems, “friends on line”, stock price tracking, and many others [7, 3, 13, 16, 10, 15].

Until recently, most event notification systems were intended to be used as part of specific applications or in localized settings, such as a single machine, a building, or a campus. With the advent of generalized eCommerce frameworks, interest has developed in the provision of global event notification systems that can interconnect dynamically changing sets of clients and services, as well as to enable the construction of Internet-scale distributed applications. Such global event notification systems will need to scale to millions and eventually billions of users.

To date, general event notification middleware implementations are only able to scale to relatively small numbers of clients. For instance, Talarian, one of the more-scalable systems, according to published documentation, only claims that its “SmartSockets system was designed to scale to thousands of users (or processes)” [14, p. 17].

While scalability to millions of users has been demonstrated by centralized systems, such as the MSN and AOL Instant Messenger systems, we do not believe that a truly global general-purpose system can be achieved by means of such centralized solutions, if for no other reason than that there are already multiple competing systems in use. We believe that the only realistic approach to providing a global event notification system is via a federated approach, in which multiple, mutually suspicious parties existing in different domains of trust interoperate with each other.

A federated approach, in turn, implies that the defining aspect of the design will be the interaction protocols between federated peers rather than the specific architectures of client and server nodes. In particular, one can imagine scenarios where one node in the federation is someone’s private PC, serving primarily its owner’s needs, and another node is a mega-service, such as the MSN Instant Messenger service, which serves millions of subscribers within its confines.

The primary goal of the Herald project is to explore the scalability issues involved with building a global event notification system. The rest of this paper describes our design criteria and philosophy in Section 2, provides an overview of our initial design decisions in Section 3, discusses some of the research issues we are exploring in Section 4, presents related work in Section 5, and concludes in Section 6.

2. Goals, Non-Goals, and Design Strategy

The topic of event notification systems is a broad one, covering everything from basic message delivery issues to questions about the semantic richness of client subscription interests. Our focus is on the scalability of the basic message delivery and distributed state management capabilities that must underlie any distributed event notification system. We assume, at least until proven otherwise, that an event notification system can be decomposed into a highly-scalable base layer that has relatively simple se-

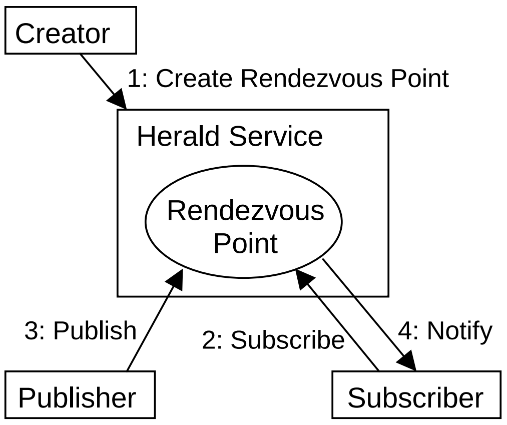


Figure 1: Herald Event Notification Model

mantics and multiple higher-level layers whose primary purposes are to provide richer semantics and functionality.

Consequently, we are starting with a very basic event notification model, as illustrated in Figure 1. In Herald, the term ***Event*** refers to a set of data items provided at a particular point in time by a *publisher* for a set of *subscribers*. Each subscriber receives a private copy of the data items by means of a *notification* message. Herald does not interpret the contents of the event data.

A ***Rendezvous Point*** is a Herald abstraction to which event publications are sent and to which clients subscribe in order to request that they be notified when events are published to the Rendezvous Point. An illustrative sequence of operations is: (1) a Herald client creates a new Rendezvous Point, (2) a client subscribes to the Rendezvous Point, (3) another client publishes an event to the Rendezvous Point, (4) Herald sends the subscriber the event received from the publisher in step 3.

This model remains the same in both the local and the distributed case. Figure 1 could be entirely upon a single machine, each of the communicating entities could be on separate machines, or each of them could even have distributed implementations, with presence on multiple machines.

2.1 Herald Design Criteria

Even with this simple model, there are still a variety of design criteria we consider important to try to meet:

- **Heterogeneous Federation:** Herald will be constructed as a federation of machines within cooperating but mutually suspicious domains of trust. We think it important to allow the coexistence of both small and large domains, containing both impoverished small device nodes and large mega-services.
- **Scalability:** The implementation should scale along all dimensions, including numbers of subscribers and publishers, numbers of event subscription points, rates of event delivery, and number of federated domains.
- **Resilience:** Herald should operate correctly in the presence of numerous broken and disconnected components. It should also be able to survive the presence of malicious or corrupted participants.
- **Self-Administration:** The system itself should make decisions about where data will be placed and how information should be propagated from publishers to subscribers. Herald should dynamically adapt to changing load patterns and resource availability, requiring no manual tuning or system administration.
- **Timeliness:** Events should be delivered to connected clients in a timely enough manner to support human-to-human interactions.
- **Support for Disconnection:** Herald should support event delivery to clients that are sometimes disconnected, queuing events for disconnected clients until they reconnect.
- **Partitioned operation:** In the presence of network partitions, publishers and subscribers within each partition should be able to continue communicating, with events being delivered to subscribers in previously separated partitions upon reconnection.
- **Security:** It should be possible to restrict the use of each Herald operation via access control to authenticated authorized parties.

2.2 Herald Non-Goals

As important as deciding what a system *will* do is deciding what it *will not* do. Until the need is proven, Herald will avoid placing separable functionality into its base model. Excluded functionality includes:

- **Naming:** Services for naming and locating Rendezvous Points are not part of Herald. Instead, client programs are free to choose any appropriate methods for determining which Rendezvous Points to use and how to locate one or more specific Herald nodes hosting those Rendezvous Points. Of course, Herald will need to export means by which one or more external name services can learn about the existence of Rendezvous Points, and interact with them.
- **Filtering:** Herald will not allow subscribers to request delivery of only some of the events sent to a Rendezvous Point. A service that filters events, for instance, by leveraging existing regular expression or query language tools, such as SQL or Quilt engines, and only delivering those matching some specified criteria, could be built as a separate service, but will not be directly supported by Herald.
- **Complex Subscription Queries: Herald** has no notion of supporting notification to clients interested in complex event conditions. Instead, we assume that complex subscription queries can be built by deploying agent processes that subscribe to the relevant Rendezvous Points for simple events and then publish

an event to a Rendezvous Point corresponding to the complex event when the relevant conditions over the simple event inputs become true.

- **In-Order Delivery:** Because Herald allows delivery during network partitions—a normal condition for a globally scaled system—different subscribers may observe events being delivered in different orders.

2.3 Applying the Lessons of the Internet and the Web

Distributed systems seem to fall into one of two categories—those that become more brittle with the addition of each new component and those that become more resilient. All too many systems are built assuming that component failure or corruption is unusual and therefore a special case—often poorly handled. The result is brittle behavior as the number of components in the system becomes large. In contrast, the Internet was designed assuming many of its components would be down at any given time. Therefore its core algorithms had to be tolerant of this normal state of affairs. As an old adage states: "The best way to build a reliable system is out of presumed-to-be-broken parts." We believe this to be a crucial design methodology for building any large system.

Another design methodology of great interest to us is derived from the Web, wherein failures are basically thrown up to users to be handled, be they dangling URL references or failed retrieval requests. Stated somewhat flippantly: "If it's broken then don't bother trying to fix it." This minimalist approach allows the basic Web operations to be very simple—and hence scalable—making it easy for arbitrary clients and servers to participate, even if they reside on resource-impoverished hardware.

Applied to Herald, these two design methodologies have led us to the following decisions:

- *Herald peers treat each other with mutual suspicion and do not depend on the correct behavior of any given, single peer*. Rather, they depend on replication and the presence of sufficiently many well-behaved peers to achieve their distributed systems goals.
- *All distributed state is maintained in a weakly consistent soft-state manner and is aged, so that everything will eventually be reclaimed unless explicitly refreshed by clients*. We plan to explore the implications of making clients responsible for dealing with weakly consistent semantics and with refreshing the distributed state that is pertinent to them.
- *All distributed state is incomplete and is often inaccurate*. We plan to explore how far the use of partial, sometimes inaccurate information can take us. This is in contrast to employing more accurate, but also more expensive, approaches to distributed state management.

Another area of Internet experience that we plan to exploit is the use of overlay networks for content delivery [5, 6]. The success of these systems implies that overlay networks are an effective means for distributing content to large numbers of interested parties. We plan to explore the use of dynamically generated overlay networks among Herald nodes to distribute events from publishers to subscribers.

3. Design Overview

This section describes the basic mechanisms that we are planning to use to build Herald. These include replication, overlay networks, ageing of soft state via time contracts, limited event histories, and use of administrative Rendezvous Points for maintenance of system metastate. While none of these are new in isolation, we believe that their combination in the manner employed by Herald is both novel, and useful for building a scalable event notification system with the desired properties. We hypothesize that these mechanisms will enable us to build the rest of Herald as a set of distributed policy modules.

3.1 Replication for Scaling

When a Rendezvous Point starts causing too much traffic at a particular machine, Herald's response is to move some or all of the work for that Rendezvous Point to another machine, when possible. Figure 2 shows a possible state of three Herald server machines at locations L1, L2, and L3, that maintain state about two Rendezvous Points, RP1 and RP2. Subscriptions to the Rendezvous Points are shown as Sub*n* and publishers to the Rendezvous Points are shown as Pub*n*.

The implementation of RP1 has been distributed among all three server locations. The Herald design allows potential clients (both publishers and subscribers) to interact with any of the replicas of a Rendezvous Point for any operations, since the replicas are intended to be functionally equivalent. However, we expect that clients will typically interact with the same replica repeatedly, unless directed to change locations.

3.2 Replication for Fault-Tolerance

Individual replicas do not contain state about all clients. In Figure 2, for instance, Sub5's subscription is recorded only by RP1@L3 and Pub2's right to publish is recorded only by RP2@L1. This means that event notifications to these subscriptions would be disrupted should the Herald servers on these machines (or the machines themselves) become unavailable.

For some applications this is perfectly acceptable, while for others additional replication of state will be necessary. For example, both RP1@L1 and RP1@L2 record knowledge of Sub2's subscription to RP1, providing a degree of fault-tolerance that allows it to continue receiving notifications should one of those servers become unavailable.

Since RP1 has a replica on each machine, it is tolerant of faults caused by network partitions. Suppose L3

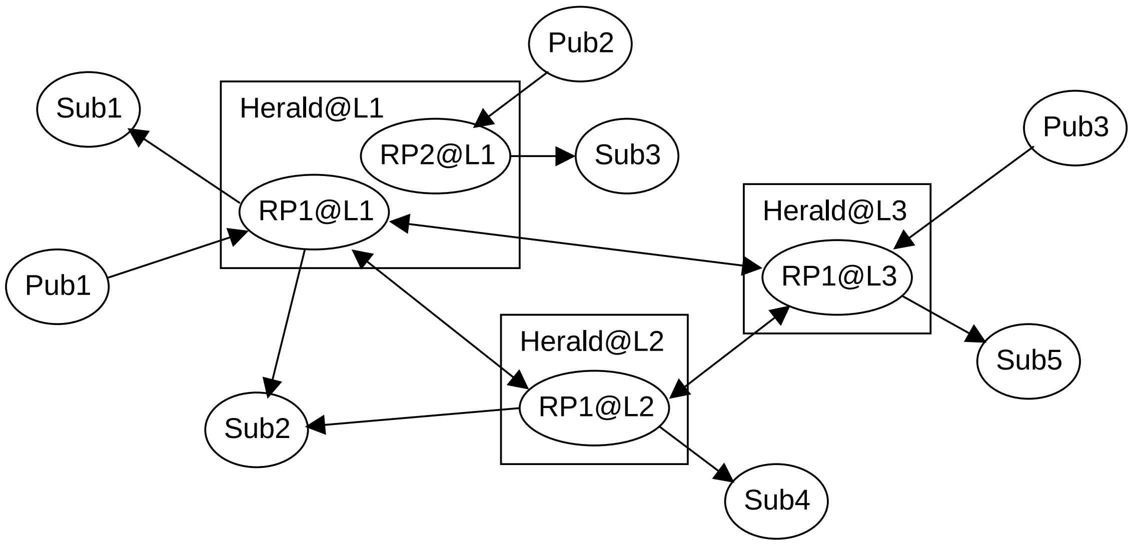


Figure 2: Replicated Rendezvous Point RP1 and Fault-Tolerant Subscription Sub2

became partitioned from L1 and L2. In this case, Pub1 could continue publishing events to Sub1, Sub2, and Sub4 and Pub3 could continue publishing to Sub5. Upon reconnection, these events would be sent across the partition to the subscribers that hadn't yet seen them.

Finally, note that since it isn't (yet) replicated, should Herald@L1 go down, then RP2 will cease to function, in contrast to RP1, which will continue to function at locations L2 and L3.

3.3 Overlay Distribution Networks

Delivery of an event notification message to many different subscribers must avoid repeated transmission of the same message over a given network link if it is to be scalable. Herald implements event notification by means of multicast-style overlay distribution networks.

The distribution network for a given Rendezvous Point consists of all the Herald servers that maintain state about publishers and/or subscribers of the Rendezvous Point. Unicast communications are used to forward event notification messages among these Herald servers in much the same way that content delivery networks do among their interior nodes. However, unlike most content delivery networks, Herald expects to allow multiple geographically distributed publishers. Delivery of an event notification message to the subscribers known to a Herald server is done with either unicast or local reliable multicast communications, depending on which is available and more efficient.

In order to implement fault tolerant subscriptions, subsets of the Herald servers implementing a Rendezvous Point will need to coordinate with each other so as to avoid delivering redundant event notifications to subscribers. Because state can be replicated or migrated between servers, the distribution network for a Rendezvous Point can grow or shrink dynamically in response to changing system state.

3.4 Time Contracts

When distributed state is being maintained on behalf of a remote party, we associate a time contract with the state, whose duration is specified by the remote party on whose behalf the state is being maintained. If that party does not explicitly refresh the time contract, the data associated with it is reclaimed. Thus, knowledge of and state about subscribers, publishers, Rendezvous Point replicas, and even the existence of Rendezvous Points themselves, is maintained in a soft-state manner and disappears when not explicitly refreshed.

Soft state may, however, be maintained in a persistent manner by Herald servers in order to survive machine crashes and reboots. Such soft state will persist at a server until it is reclaimed at the expiration of its associated time contract.

3.5 Event History

Herald allows subscribers to request that a history of published events be kept in case they have been disconnected. Subscribers can indicate how much history they want kept and Herald servers are free to either accept or reject requests.

History support imposes a storage burden upon Herald servers, which we bound in two ways. First, the creator of a Rendezvous Point can inform Herald of the maximum amount of history storage that may be allocated at creation time. As with subscriptions, servers are free to reject creation requests requiring more storage than their policies or resources allow.

Second, because clients and servers both maintain only ageing soft state about one another, event history information kept for dead or long-unreachable subscribers will eventually be reclaimed.

While we recognize that some clients might need only a synopsis or summary of the event history upon reconnection, we leave any such filtering to a layer that can be built over the basic Herald system, in keeping with our Internet-style philosophy of providing primitives on which other services are built. Of course, if the last event sent will suffice for a summary, Herald directly supports that.

3.6 Administrative Rendezvous Points

One consequence of name services being outside Herald is that when Herald changes the locations at which a Rendezvous Point is hosted, it will need to inform the relevant name servers of the changes. In general, there may be a variety of parties that are interested in learning about changes occurring to Rendezvous Points and the replicas that implement them.

Herald notifies interested parties about changes to a Rendezvous Point by means of an *administrative* Rendezvous Point that is associated with it. By this means we plan to employ a single, uniform mechanism for all client-server and server-server notifications.

Administrative Rendezvous Points do not themselves have other Administrative Rendezvous Points associated with them. Information about their status is communicated via themselves.

4. Research Issues

In order to successfully build Herald using the mechanisms described above, we will have to tackle a number of research issues. We list a few of the most notable ones below.

The primary research problem we face will be to develop effective policies for deciding when and how much Rendezvous Point state information to move or replicate between servers, and to which servers. These policies will need to take into account load balancing and fault-tolerance concerns, as well as network topology considerations, for both message delivery and avoidance of unwanted partitioning situations. Some of the specific topics we expect to address are:

- determining when to dynamically add or delete servers from the list of those maintaining a given Rendezvous Point,
- dynamic placement of Rendezvous Point state—especially event histories—to minimize the effects of potential network partitions,
- dynamically reconfiguring distributed Rendezvous Point state in response to global system state changes,
- dealing with "sudden fame", where an Internet-based application's popularity may increase by several orders of magnitude literally overnight, implying that our algorithms must stand up to rapid changes in load.

Since we plan to rely heavily on partial, weakly consistent, sometimes inaccurate, distributed state information, a key challenge will be to explore how well one can manage a global service with such state. Equally important will be to understand what the cost of disseminating information in this fashion is.

It is an open question exactly how scalable a reliable multicast-style delivery system can be, especially when multiple geographically dispersed event publishers are allowed and when the aggregate behavior of large numbers of dynamically changing Rendezvous Points is considered. In addition, Herald requires that event notifications continue to be delivered to reachable parties during partitions of the system and be delivered "after the fact" to subscribers who have been "disconnected" from one or more event publication sources. To our knowledge, operation of delivery systems under these circumstances has not yet been studied in any detail.

Herald's model of a federated world in which foreign servers are not necessarily trustworthy implies that information exchange between servers may need to be secured by means such as Byzantine communication protocols or statistical methods that rely on obtaining redundant information from multiple sources. Event notification messages may need to be secured by means such as digital signatures and "message chains", as described, for example, in [12].

Another scaling issue is how to deal with access control for large numbers of clients to a Rendezvous Point. For example, consider the problem of allowing all 280 million U.S. citizens access to a particular Rendezvous Point, but no one else in the world.

Finally, Herald pushes a number of things often provided by event notification systems, such as event ordering and filtering, to higher layers. It is an open question how well that will work in practice.

5. Related Work

The Netnews distribution system [8] has a number of attributes in common with Herald. Both must operate at Internet scale. Both propagate information through a sparsely connected graph of distribution servers. The biggest difference is that for Netnews, human beings design and maintain the interconnection topology, whereas for Herald, a primary research goal is to have the system automatically generate and maintain the interconnection topology. The time scales are quite different as well. Netnews propagates over time scales of hours to weeks, whereas Herald events are intended to be delivered nearly instantaneously to connected clients.

A number of peer-to-peer computing systems, such as Gnutella [2], have emerged recently. Like Herald, they are intended to be entirely self-organizing, utilizing resources on federated client computers to collectively provide a global service. A difference between these services and

Herald is that the former typically use non-scalable algorithms, including broadcasts. Unlike Herald, with the exception of Farsite [1], these services also typically ignore security issues and are ill prepared to handle malicious participants.

Using overlay networks for routing content over the underlying Internet has proven to be an effective methodology. Examples include the MBONE [11] for multicast, the 6BONE [4] for IPv6 traffic, plus content distribution networks such as Overcast [6] and Inktomi's broadcast overlay network [5]. They have demonstrated the same load-reducing benefits for information dissemination to large numbers of clients needed for Herald. However, most work has focused on single-sender dissemination networks. Furthermore, they have not investigated mechanisms and appropriate semantics for continued operation during partitions.

The OceanStore [9] project is building a global-scale storage system using many of the same principles and techniques planned for Herald. Both systems are built using unreliable servers, and provide reliability through replication and caching. Both intend to be self-monitoring and self-tuning.

6. Conclusions

Global event notification is emerging as a key technology underlying numerous distributed applications. With the requirements imposed by use of these applications at Internet scale, the need for a highly scalable event notification system is clear.

We have presented the requirements and design overview for Herald, a new event notification system designed to fill this need. We are currently implementing the mechanisms described in this paper and are planning to then experiment with a variety of different algorithms and policies to explore the research issues we have identified.

Acknowledgments

The authors would like to thank Yi-Min Wang, Dan Ling, Jim Kajiya, and Rich Draves for their useful feedback on the design of Herald.

References

[1] Bill Bolosky, John Douceur, David Ely, and Marvin Theimer. Evaluation of Desktop PCs as Candidates for a Serverless Distributed File System. In *Proceedings of Sigmetrics 2000*, Santa Clara, CA, pp. 34-43, ACM, June 2000.

[2] *Gnutella: To the Bandwidth Barrier and Beyond*. Clip2.com, November 6th, 2000. http://www.gnutellahosts.com/gnutella.html.

[3] David Garlan and David Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. In *Proceedings of Fourth International Symposium of VDM Europe: Formal Software Development Methods*, Noordwijkerhout, Netherlands, pp. 31-44, October, 1991. Also appears as Springer-Verlag *Lecture Notes in Computer Science 551*.

[4] Ivano Guardini, Paolo Fasano, and Guglielmo Girardi. *IPv6 operational experience within the 6bone*. January 2000. http://carmen.cselt.it/papers/inet2000/index.htm.

[5] *The Inktomi Overlay Solution for Streaming Media Broadcasts*. Technical Report, Inktomi, 2000. http://www.inktomi.com/products/media/docs/whtpapr.pdf.

[6] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, pp. 197-212. USENIX Association, October 2000.

[7] Astrid M. Julienne and Brian Holtz. *ToolTalk and Open Protocols: Inter-Application Communication*. Prentice Hall, 1994.

[8] Brian Kantor and Phil Lapsley. *Network News Transfer Protocol* (NNTP). Network Working Group Request for Comments 977 (RFC 977), February 1986. http://www.ietf.org/rfc/rfc0977.txt.

[9] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.

[10] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus – An Architecture for Extensible Distributed Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Asheville, NC, pp. 58-68, December 1993.

[11] Kevin Savetz, Neil Randall, and Yves Lepage. *MBONE: Multicasting Tomorrow's Internet*. IDG, 1996. http://www.savetz.com/mbone/.

[12] Mike J. Spreitzer, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Douglas B. Terry. Dealing with Server Corruption in Weakly Consistent, Replicated Data Systems, In *Wireless Networks*, ACM/Baltzer, 5(5), 1999, pp. 357-371. A shorter version appears in *Proceedings of the Third Conference on Mobile Computing and Networking*, ACM/IEEE, Budapest, Hungary, September 1997.

[13] Rob Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *Proceedings of International Symposium on Software Reliability Engineering '98*, Fat Abstract, 1998.

[14] Talarian Inc. *Talarian: Everything You Need To Know About Middleware*. http://www.talarian.com/industry/middleware/whitepaper.pdf.

[15] TIBCO Inc. *Rendezvous Information Bus*. http://www.rv.tibco.com/datasheet.html, 2001.

[16] David Wong, Noemi Paciorek, Tom Walsh, Joe DiCelie, Mike Young, Bill Peet. Concordia: An Infrastructure for Collaborating Mobile Agents. In *Proceedings of the First International Workshop on Mobile Agents*, April, 1997.

HeRMES: High-Performance Reliable MRAM-Enabled Storage

Ethan L. Miller *elm@cs.ucsc.edu*
Scott A. Brandt *sbrandt@cs.ucsc.edu*
Darrell D. E. Long *darrell@cs.ucsc.edu*
Computer Science Department
University of California, Santa Cruz

Abstract

Magnetic RAM (MRAM) is a new memory technology with access and cost characteristics comparable to those of conventional dynamic RAM (DRAM) and the non-volatility of magnetic media such as disk. Simply replacing DRAM with MRAM will make main memory non-volatile, but it will not improve file system performance. However, effective use of MRAM in a file system has the potential to significantly improve performance over existing file systems. The HeRMES file system will use MRAM to dramatically improve file system performance by using it as a permanent store for both file system data and metadata. In particular, metadata operations, which make up over 50% of all file system requests [14], are nearly free in HeRMES because they do not require any disk accesses. Data requests will also be faster, both because of increased metadata request speed and because using MRAM as a non-volatile cache will allow HeRMES to better optimize data placement on disk. Though MRAM capacity is too small to replace disk entirely, HeRMES will use MRAM to provide high-speed access to relatively small units of data and metadata, leaving most file data stored on disk.

1. Introduction

Current file systems are optimized for the assumption that the only stable storage in the system is a block-oriented, high-latency device such as a disk. As a result, existing file systems use data structures and algorithms that transfer data in large units and take great pains to ensure that the file system's image on disk remains internally consistent. If the file system includes any non-volatile memory (NVRAM), there is usually a limited amount used as a temporary storage area to facilitate staging data to disk.

Magnetic RAM (MRAM) [4] is a new memory technology, currently in development, with the speed, density, and cost of DRAM and the non-volatility of disk. We are investigating the use of MRAM in the HeRMES (**H**igh-performanc**e**, **R**eliable, **M**RAM-**E**nabled **S**torage) file system to dramatically improve file system performance by storing metadata and some data in MRAM. Since MRAM will have cost comparable to that of DRAM, it cannot totally replace disk or other types of secondary storage such as MEMS [9]. Rather, we are researching the most effective ways to use limited amounts of MRAM in a file system.

An MRAM-based file system such as HeRMES has several major advantages over existing file systems in both performance and reliability. As we discuss in this paper, using MRAM in the file system can reduce the cost of metadata operations to nearly zero, leaving them limited solely by CPU speed. It also increases the speed of file reads and writes both by reducing metadata overhead and by allowing the file system to better lay out data on disk by buffering writes longer in safe MRAM. File system reliability is also greatly improved. Simplifying metadata structures results in less complex and more reliable software. Keeping metadata in MRAM also allows HeRMES to run consistency checks on the file system in the background during normal operation, allowing errors to be caught early, before they spread.

2. HeRMES design

The HeRMES file system is built from the ground up using two assumptions that differ from current file systems: metadata accesses need not be in large contiguous blocks, and metadata accesses take microseconds (at most) rather than milliseconds. These assumptions differ from those underlying disk-based file systems, which require milliseconds to access blocks of data.

2.1. Metadata management

HeRMES maintains all of its metadata in MRAM, avoiding the need to access the disk for metadata requests. The ability of MRAM to handle single-word reads and writes further benefits HeRMES by allowing it to use much simpler data structures. For example, the B+-trees used in XFS [16] make efficient use of large blocks at the expense of file system complexity. HeRMES, on the other hand, can use simpler data structures such as binary trees and hash tables with in-memory semantics because it does not need to allocate and reference structures in large blocks.

Keeping all metadata in MRAM could be prohibitive for traditional file systems, which can require up to a 1–2% overhead for metadata; 600 MB of DRAM for a 60 GB disk may be too expensive, with memory costs exceeding those of disk. HeRMES, in contrast, will make extensive use of compression and variable-sized allocations to drastically reduce needed space, avoiding this problem. For example, an inode in Unix might require 128 bytes; there would be little benefit to reducing its size on disk because retrieval time is dominated by access latency which would not be reduced for smaller objects. It might be possible to save small amounts of DRAM at the expense of transforming the inode when transferring it between disk and memory, but using information from other inodes to do the compression would be difficult. HeRMES, however, can use commonalities between inodes to reduce required space. For example, each file's inode can contain a pointer to an access control list; since many of a user's files have identical permissions, their inodes can share a single list. File index pointers can also benefit from compression and variable-sized memory-style allocation. Many file systems use extents to compress index lists; by storing lists of extents in variable-sized blocks of MRAM, HeRMES can eliminate wasted space.

One potential problem with keeping metadata in MRAM is that it may be *too* easy to modify data structures, potentially causing file system inconsistency. Wild references in the file system (or elsewhere in the operating system) could overwrite valid metadata in MRAM, corrupting the file system. HeRMES will avoid this problems using techniques similar to those in Rio [12]. By keeping file system MRAM protected except when explicitly necessary, HeRMES will ensure that only desired changes are made to MRAM. The process of switching a page from read-only to read-write in the page table is fast, and will not significantly slow down HeRMES MRAM operations, particularly since it is only necessary when metadata is modified.

2.2. MRAM write buffer

Like most file systems, HeRMES will buffer writes in memory for several reasons: allowing a process to continue without waiting for a write to go to disk, reordering writes to minimize disk latency, and waiting in the hope that a file will be deleted. Unlike many file systems, however, writes with HeRMES are safe once they are written to MRAM. This allows HeRMES to postpone writes as long as desired without fear of data loss due to a system crash.

The write buffer in HeRMES is similar to that in systems with NVRAM, with two important differences: MRAM is considerably faster than NVRAM, and metadata updates accompanying a write are done immediately in MRAM. Writes to MRAM are considerably faster than writes to flash RAM, which can require more than two milliseconds. MRAM's faster write time reduces the window of vulnerability during which data can be lost from a system failure.

Because MRAM is a long-term stable store, data written there can be kept as long as necessary. This allows HeRMES to optimize data placement on disk, reducing time wasted to disk access latency. Existing file systems do this as well, but they run the risk of data loss if they hold data in the write buffer too long. Many systems with "non-volatile" RAM actually use battery-backed RAM, which can lose data because of dead batteries in addition to the usual dangers of storing data in RAM.

2.3. MRAM file storage

MRAM may also be useful for disk reads, particularly if there is a relatively large amount of MRAM in the system. Disk latencies are currently around 5–10 ms; in that time, a disk can transfer 64–128 KB of data. The file system can keep the first few blocks of each file in MRAM, transferring the data out of MRAM while the disk seek is completed. Combining this technique with file access prediction and clustering on secondary storage [1] will further improve performance by reserving the scarce MRAM resource for "live" data. As probe-based storage [9] becomes available, this technique will become more effective because the latency to data on secondary storage will be lower, reducing the amount of file data that must be buffered in MRAM and increasing the number of files for which such buffering is possible.

As with write buffering, caching file headers (or entire files, if they are small) is not a new technique. However, MRAM makes this technique more attractive because it allows the structures to survive power loss and system reboot, enabling the file system to build such a cache over time without the need to preserve it on disk or reload it after a system restart.

3. Performance

HeRMES can significantly outperform existing file systems for several reasons. First, metadata operations in HeRMES are nearly free because they only require memory-type accesses. Table 1 shows several common file system request types [14], noting the disk operations needed to satisfy each one. Existing file systems cache metadata in DRAM, updating the original on disk when changes occur. Though they can eliminate many (but not all) disk reads by caching, metadata writes must go through to disk to ensure consistency, and writes often have a partial order enforced on them to maintain file system consistency [13]. HeRMES, on the other hand, handles disk requests in the shaded columns entirely in MRAM, leaving only file data reads and writes to use the disk. This results in dramatically faster metadata operations, requiring microseconds rather than milliseconds to complete. Moreover, data writes can be safely buffered in MRAM indefinitely, as described in Section 2.2, further decreasing latency from user write to "safe" commit of the data.

Table 1. Disk I/O needed for file system requests.

Request	**Type of disk requests needed**			
	Global metadata	**File metadata**	**File index**	**File data**
File stat (50%)	–	read	–	–
File read (20%)	–	read write	read	read
File write (5%)	read write	read write	read write	read write

Because HeRMES metadata operations are limited only by CPU speed, the file system can satisfy them in the time it takes to execute the metadata operation in the CPU. For existing file systems, 20,000 – 40,000 operations are sufficient to execute a file system request; this is 40 to 80 μs on a modern processor, allowing a single processor file server to handle about 25,000 metadata operations per second; HeRMES will likely be able to do more operations per second because it can use simpler data structures (and thus fewer instructions to manipulate them) and has no need to spend instructions on managing disk I/O. If a file server provides, on average, one 4 KB file block for every two metadata operations, such a server could sustain 50 MB per second using a single commodity CPU.

The simple MRAM-resident data structures in HeRMES can provide added speed in another way: reduced lock contention. Disk-based file systems must use fine-grained locking to ensure high levels of concurrency in the face of relatively long metadata operations. In particular, operations that require reading data from disk can hold locks for milliseconds, potentially causing contention for locks. HeRMES, in contrast, can complete metadata reads or updates in less than 100 microseconds. This time is shorter than the scheduling quantum on many systems, and is thus less likely to result in high levels of lock contention. The contention problem is exacerbated on symmetric multiprocessor systems; again, HeRMES can use relatively course-grained locking and still maintain low levels of lock contention.

4. Reliability

File system reliability is, for many users, more important than performance: getting the correct data later is better than getting erroneous data now. HeRMES can provide high performance, as seen in Section 3, without sacrificing reliability. Moreover, HeRMES will be more reliable than existing file systems for several reasons, including lower software complexity and the ability to continuously check the system for consistency.

4.1. Reducing software complexity

By using relatively simple structures in MRAM, HeRMES reduces software complexity, making file system software more reliable. Simple data structures are well-understood and less prone to programming errors, reducing the likelihood that a bug will be hidden in thousands of lines of complex code. Because MRAM is so much faster than disk, there will be less temptation for programmers to take shortcuts that save a few microseconds, making it less likely that such a shortcut will malfunction.

The lower number of locks needed in HeRMES also increase software reliability. With metadata operations locking up structures for around 50 μs, there is no need for thousands of locks in the file system. On a uniprocessor system, in fact, a single lock for the entire metadata structure is sufficient because operations are CPU-bound and thus gain minimal benefit from interleaved requests. Even in multiprocessor file servers, a relatively small number of locks—at most one per file (for metadata), one for disk allocation, and one for memory allocation—will be sufficient to guarantee that processors are not waiting on file system locks. The net result is a lower probability of deadlock as well as less chance that data will be improperly modified.

4.2. Metadata checking

HeRMES will also take an active approach to protecting file system consistency by continuously checking the metadata structures while the system is running. A background process checking 2,000 files per second can fully check a system with ten million files in less than 90 minutes, yet it demands less than 10% of the system's resources to do so.

Checking the file system's metadata while the system is operating increases reliability in several ways. First, it is often easier to write a program that *detects* an error than it is to write a file system that doesn't produce errors in the first place. Merely detecting the error may be sufficient to attempt correcting it, or at least to prevent it from spreading to the rest of the file system. Second, most existing file systems *never* have their metadata checked. They rely on logging [10] and other techniques to recover quickly from a crash, but they do not examine metadata written during normal operation. This is necessary because a full check of the metadata on a large file system with ten million files might take hours, if not days, and would consume most of the disk bandwidth during that time. Third, extremely large file systems are now encountering a new problem: disk unreliability due to firmware errors and undetectable bit errors is becoming a concern. A bit error rate of 10^{-12} becomes a problem when file systems store a terabyte of data because bit errors may go unnoticed for days. Rather than do continuous checks, though current file systems must assume that their code does not contain any bugs and that the underlying media is reliable, assumptions that are increasingly less likely as file systems grow larger and more complex.

4.3. Backing up metadata

MRAM, like any other part of a computer, will be subject to component failure. Because MRAM is the only place metadata is stored, HeRMES must guard against MRAM failure. It does so by logging metadata changes to a location other than that holding the MRAM. This can be done in several ways. The first option is to write metadata changes to disk. This is very similar to logging, but does not involve the same ordering issues that metadata updates in conventional systems suffer. The second option is to keep the metadata log in a different bank of MRAM than that holding the original metadata. If MRAM can be removed from a computer, placed in a new one, and its contents read, this solution is sufficient to back up metadata at very little cost.

In either case, metadata update logging requires very little space. The majority of metadata updates are timestamp modifications, which can be recorded in a few bytes. More complex modifications take more space; however, MRAM can buffer changes and flush them to disk several times per minute. Using this mechanism means that total MRAM failure (chip failure) can lose small amounts of data, but that consistency is not affected. It is important to remember that chip failure is not a common source of computer failure, and that chip failure affects *all* file systems that use memory for caching and buffering.

5. Related work

Our work builds on many areas of file system research, but research into non-volatile RAM (NVRAM) systems and schemes to reduce latency for disk accesses, particularly metadata, is most relevant.

Douglis [6] and Wu [17] proposed the use of NVRAM to hold an entire file system. This approach is acceptable for relatively small file systems, but MRAM (like NVRAM) is too expensive to replace disk for general purpose file systems. Additionally, the flash RAM used in these systems does not support single word writes; instead, it requires 1–2 ms (or more) to write a relatively large block of data. This prevents fine-grained modification of data in non-volatile memory. In eNVy [17], copy-on-write and buffering were used to get around the long erase latency of flash RAM; this approach required extensive garbage collection similar to that used in log-structured file systems [3,15].

NVRAM has long been used for recovery and file system reliability [2], again with the restrictions of small size and coarse-grained write access. In such systems, NVRAM is used as a non-volatile cache for disk, but data "lives" on disk. This design improves file system reliability by reducing the window of vulnerability for written data and improves performance by relaxing metadata write constraints. However, it does not allow the rich metadata structures possible when metadata is permanently resident in MRAM, and writes must still be sent to disk, requiring disk seeks and consuming disk bandwidth.

Techniques for reducing disk latency and improving reliability for metadata include writing data to the nearest free disk blocks [7,11], logging [10], and soft updates [13]. All of these techniques reduce access latency for writes, but none reduces the *number* of blocks that must be written. Additionally, these techniques use little beyond caching to speed up metadata read access. Another technique, combining metadata with file data [8], allows data and metadata for small files to be read and written in a single contiguous request. However, this technique was only tried with relatively small files.

6. Current research

Our research into using MRAM for file systems, specifically HeRMES, has just begun. In this paper, we described several ways in which MRAM can be used to improve file system performance, but many questions remain. For example, what happens if MRAM is limited? If insufficient MRAM is available for all of the metadata, how can HeRMES efficiently transform in-memory structures to on-disk structures for infrequently used files? What is the correct tradeoff between using MRAM for metadata, write buffering, and other uses such as caching the first few blocks of a file to reduce access latency?

We are also exploring issues related to using MRAM across a distributed file system. Clearly, some form of sharing, perhaps similar to cooperative caching [5], will be necessary to fully utilize MRAM in such a system. However, there will be differences as well—the access latency across a network, while lower than that of disk, is considerably higher than that of MRAM.

We are just at the beginning of research into using the new technology of MRAM in file systems, and there are many avenues of research that we will pursue.

7. Conclusions

Magnetic RAM will be available commercially within a few years; it is crucially important that file system designers incorporate it into file systems and use it effectively. We have shown how magnetic RAM can be used to dramatically improve file system performance and reliability. Our file system, HeRMES, will keep metadata in MRAM, allowing nearly free metadata operations limited only by CPU speed. Because MRAM is non-volatile, there is never a need to flush metadata to disk, also improving file system data bandwidth by freeing disk from the need to handle frequent metadata accesses.

File system reliability also benefits from the use of MRAM. The simpler metadata structures possible using MRAM will reduce file system complexity, and thus increase software reliability. Background metadata consistency checking, likewise, will increase the chance than an error will be found, increasing file system reliability by snuffing out errors as soon as they happen. It is this combination of performance and reliability that makes MRAM attractive as a technology for incorporation into file systems.

References

[1] A. Amer and D. Long, "Noah: Low-cost file access prediction through pairs," *20th IEEE International Performance, Computing, and Communications Conference (IPCCC 2001)*, pp. 27–33, 2001.

[2] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-volatile memory for fast, reliable file systems," *5th Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 10–22, 1992.

[3] T. Blackwell, J. Harris, and M. Seltzer, "Heuristic cleaning algorithms in log-structured file systems," *Proceedings of the 1995 USENIX Technical Conference*, pages 277–288, January 1995.

[4] H. Boeve, C. Bruynseraede, J. Das, K. Dessein, G. Borghs, J. De Boeck, R. Sousa, L. Melo, and P. Freitas, "Technology assessment for the implementation of magnetoresistive elements with semiconductor components in magnetic random access memory (MRAM) architectures," *IEEE Trans. on Magnetics* **35**(5), pp. 2820–2825, 1999.

[5] M. Dahlin, R. Wang, T. Anderson, and D. Patterson, "Cooperative caching: using remote client memory to improve file system performance," *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, 1994.

[6] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage alternatives for mobile computers," *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 25–37, 1994.

[7] R. English and A. Stepanov, "Loge: a self-organizing disk controller," *Winter 1992 USENIX Technical Conference*, pp. 237–252, 1992.

[8] G. Ganger and M. Kaashoek, "Embedded inodes and explicit groupings: exploiting disk bandwidth for small files," *1997 USENIX Technical Conference*, pp. 1–17, 1997.

[9] J. Griffin, S. Schlosser, G. Ganger, and D. Nagle, "Operating system management of MEMS-based storage devices," *4th Symp. on Operating Systems Design and Implementation (OSDI)*, pp. 227–242, 2000.

[10] R. Hagmann, "Reimplementing the Cedar File System using logging and group commit," *11th ACM Symposium on Operating Systems Principles*, pp. 155–162, 1987.

[11] D. Hitz, J. Lau, and M. Malcom, "File system design for an NFS file server appliance," *Winter 1994 USENIX Conference*, 1994.

[12] D. Lowell and P. Chen, "Free transactions with Rio Vista," *16th ACM Symposium on Operating Systems Principles*, pp. 92–101, 1997.

[13] M. McKusick and G. Ganger, "Soft updates: a technique for eliminating most synchronous writes in the Fast Filesystem," *FREENIX Track: 1999 USENIX Technical Conference*, pp. 1–18, 1999.

[14] D. Roselli, J. Lorch, and T. Anderson, "A comparison of file system workloads," *2000 USENIX Technical Conference*, pp. 41–54, 2000.

[15] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems* **10**(1), pages 26–52, 1992.

[16] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system," *Proceedings of the 1996 USENIX Conference*, pages 1–14, 1996.

[17] M. Wu and W. Zwaenepoel, "eNVy: a non-volatile, main memory storage system," *6th Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 86–97, 1994.

Better Security via Smarter Devices

Gregory R. Ganger and David F. Nagle
Carnegie Mellon University

E-mail: {ganger,bassoon}@ece.cmu.edu

Abstract

This white paper promotes a new approach to network security in which each individual device erects its own security perimeter and defends its own critical resources (e.g., network link or storage media). Together with conventional border defenses, such self-securing devices *could provide a flexible infrastructure for dynamic prevention, detection, diagnosis, isolation, and repair of successful breaches in borders and device security perimeters. We overview the self-securing devices approach and the siege warfare analogy that inspired it. We also describe several examples of how different devices might be extended with embedded security functionality and outline some challenges of designing and managing self-securing devices.*

1. Overview

From all indications, assured OS security seems to be an impossible goal. Worse, conventional security architectures are brittle by design, because a small number of border protections (e.g., firewalls and/or host OSs) are used to protect a large number of resources and services. For example, an attacker who compromises a machine's OS gains complete control over all resources of that machine. Thus, such an intruder gains the ability to transmit anything onto the network, modify anything on the disk, and examine all input device signals (e.g., typing patterns and video feeds). Likewise, an attacker who circumvents firewall-based protection has free reign within the "protected" environment.

Having shared border protections for large sets of resources creates three fundamental difficulties: (1) the many interfaces and functionalities for the many resources (e.g., consider most multi-purpose OSs) make correct implementation and administration extremely difficult; the practical implications are daily security alerts for popular OSs (e.g., Windows NT and Linux) and network applications (e.g., e-mail and web); (2) the ability of successful attackers to freely manipulate everything beyond the border protection greatly complicates most phases of security management, including intrusion detection, isolation, diagnosis, and recovery; (3) having a central point of security checks creates performance, fault-tolerance, and flexibility limitations for large-scale environments.

This position paper promotes an alternative architecture in which individual system components erect their own security perimeters and protect their resources (e.g., network, storage, or video feed) from intruder tampering. This "self-securing devices" architecture distributes security functionality amongst *physically distinct* components, avoiding much of the fragility and unmanageability inherent in today's border-based security. Specifically, this architecture addresses the three fundamental difficulties by: (1) simplifying each security perimeter (e.g., consider NIC or disk interfaces), (2) reducing the power that an intruder gains from compromising just one of the perimeters, and (3) distributing security enforcement checks among the many components of the system.

Conventional application-executing CPUs will still run application programs, but they won't dictate which packets are transferred onto network wires and they won't dictate which disk blocks are overwritten. Instead, self-securing NICs will provide firewall and proxy server functionality for a given host, as well as throttling or labelling its outbound traffic when necessary. Likewise, self-securing storage devices will protect their data from compromised client systems, and self-securing graphics cards will display warning messages even when the window manager is compromised. In a system of self-securing devices, compromising the OS of an application-executing CPU won't give a malicious party complete control over all system resources — to gain complete power, an intruder must also compromise the disk's OS, the network card's OS, etc.

Augmenting current border protections with self-securing devices promises much greater flexibility for security administrators. By having each device erect an independent security perimeter, the network environment gains many outposts from which to act when under attack. Devices not only protect their own resources, but they can observe, log, and react to the actions of other nearby devices. Infiltration of one security perimeter will compromise only

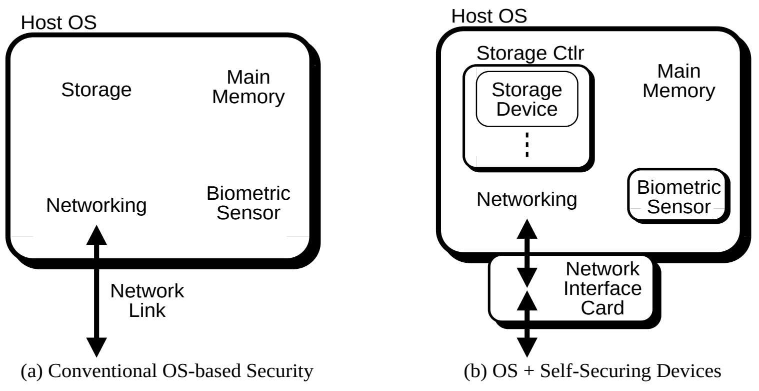


Figure 1. Two security approaches for a computer system. On the left, (a) shows the conventional approach, which is based on a single perimeter around the set of system resources. On the right, (b) shows our new approach, which augments the conventional security perimeter with perimeters around each self-securing device. These additional perimeters offer additional protection and flexibility for defense against attackers. Firewall-enforced network security fits a similar picture, with the new architecture providing numerous new security perimeters within each system on the internal network.

a small fraction of the environment, allowing other devices to dynamically identify the problem, alert still-secured devices about the compromised components, raise the security levels of the environment, and so forth.

Self-securing devices will require more computational resources in each device. However, with rapidly shrinking hardware costs, growing software development costs, and astronomical security costs, it makes no sense to not be throwing hardware at security problems. A main challenge for we OS folks is to figure out how to best partition (and replicate) functionality across self-securing components in order to enhance security and robustness. A corollary challenge is to re-marshall the distributed functionality to achieve acceptable levels of performance and manageability. After describing our inspiration for this architecture (medieval siege warfare), this position paper outlines some of our thoughts on these challenges.

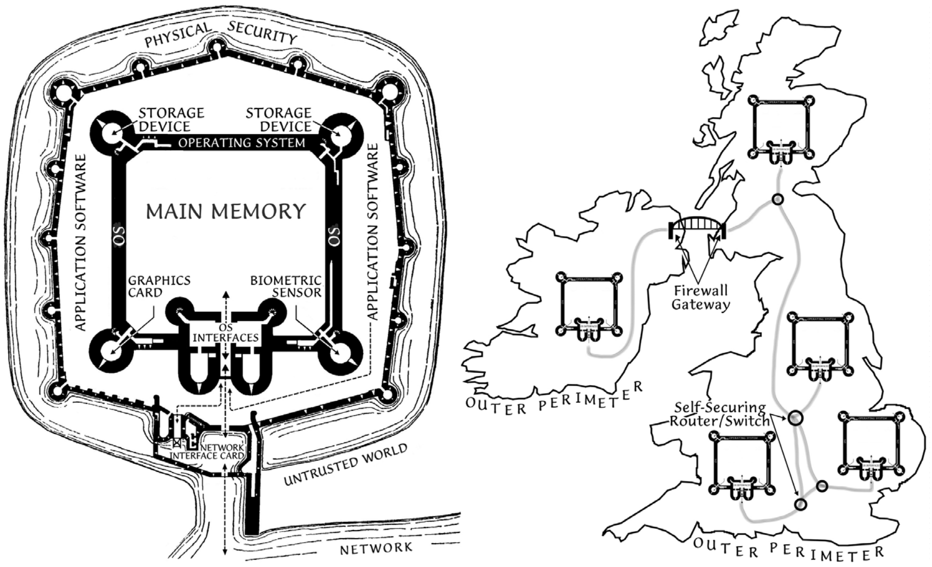
2. Siege Warfare in the Internet Age

Despite enormous effort and investment, it has proven nearly impossible to prevent computer security breaches. To protect our critical information infrastructures, we need defensive strategies that can survive determined and successful attacks, allowing security managers to dynamically detect, diagnose, and recover from breaches in security perimeters. Borrowing from lessons learned in pre-gun warfare, we propose a new network security architecture analogous to medieval defense constructs.

Current security mechanisms are based largely on singular border protections. This roughly corresponds to defense practices during Roman times, when defenders erected walls around their camps and homes to provide protective cover during attacks. Once inside the walls, however, attackers faced few obstacles to gaining access to all parts of the enclosed area. Likewise, a cracker who successfully compromises a firewall or OS has complete access to the resources protected by these border defenses—no additional obstacles are faced.[1] Of course, border defenses were a large improvement over open camps, but they proved difficult to maintain against determined attackers — border protections can be worn down over time and defenders of large encampments are often spread thin at the outer wall.

As the size and sophistication of attacking forces grew, so did the sophistication of defensive structures. The most impressive such structures, constructed to withstand determined sieges in medieval times, used multiple tiers of defenses. Further, tiers were not strictly hierarchical in nature — rather, some structures could be defended independently of others. This major advancement in defense capabilities provided defenders with significant flexibility in defense strategy, the ability to observe attacker activities, and the ability to force attackers to deal with multiple independent defensive forces.

[1]This is not quite correct in the case of a firewall protecting a set of hosts that each run a multi-program OS, such as Linux. Such an environment is more like a town of many houses surrounded by a guarded wall. Each house affords some protection beyond that provided by the guarded wall, but not as much in practice as might be hoped. In particular, most people in such an environment will simply open the door when they hear a knock, assuming that the wall keeps out attackers. Worse, in the computer environment, homogeneity among systems results in a single set of keys (attacks) that give access to any house in the town.



(a) a siege-ready computer system (b) 2 siege-ready intranets

Figure 2. The self-securing device architecture illustrated via the siege warfare constructs that inspired it. On the left, (a) shows a siege-ready system with layered and independent tiers of defense enabled by device-embedded security perimeters. On the right, (b) shows two small intranets of such systems, separated by firewall-guarded entry points. Also note the self-securing routers/switches connecting the machines within each intranet.

Applying the same ideas to computer and network security, border protections (i.e., firewalls and host OSs) can be augmented with security perimeters erected at many points within the borders. Enabled by low-cost computation (e.g., embedded processors, ASICs), security functionality can be embedded in most device microcontrollers, yielding "better security via smarter devices." We refer to devices with embedded security functionality as *self-securing devices*.

Self-securing devices can significantly increase network security and manageability, enabling capabilities that are difficult or impossible to implement in current systems. For example, independent device-embedded security perimeters guarantee that a penetrated boundary does not compromise the entire system. Uncompromised components continue their security functions even when other system components are compromised. Further, when attackers penetrate one boundary and then attempt to penetrate another, uncompromised components can observe and react to the intruder's attack; from behind their intact security perimeters, they can send alerts to the security administrator, actively quarantine or immobilize the attacker, and wall-off or migrate critical data and resources. Pragmatically, each self-securing device's security perimeter is simpler because of specialization, which should make correct implementations more likely. Further, distributing security checks among many devices reduces their performance impact and allows more checks to be made.

By augmenting conventional border protections with self-securing devices, this new security architecture promises substantial increases in both network security and security manageability. As with medieval fortresses, well-defended systems conforming to this architecture could survive protracted sieges by organized attackers.

3. Device-embedded security examples

To make our new security architecture more concrete, this section gives several examples of how different devices might be extended with embedded security functionality. In each case, there are difficulties and research questions to be explored; here, we focus mainly on conveying the potential.

Network interface cards (NICs): The role of NICs in computer systems is to move packets between the system's components and the network. Thus, the natural security extension is to enforce security policies on packets forwarded in each direction [2]. Like a firewall, a self-securing NIC does this by examining packet headers and simply not forwarding unacceptable packets into or out of the computer system. A self-securing NIC can also act as a machine-specific gateway proxy, achieving the corresponding protections without scalability or identification problems; by performing such functions at each system's NIC, one avoids the bottleneck imposed by current centralized approaches. NIC-based firewalls and proxies can also protect systems from insider attacks as well as Internet attacks, since only the one host system is inside the NIC's boundary. Further, self-securing NICs offer a powerful control to network administrators: the ability to throttle or tag network traffic at its sources. So, for example, a host whose security status is questionable could have its network access blocked or limited. Security administrators manage and configure self-securing NICs over the network, since they must obviously be connected directly to it — this allows an administrator to use the NIC to protect the network from its host system. By embedding this traffic management functionality inside the NIC, one enjoys its benefits even when the host OS or other machines inside the LAN border are compromised.

Storage devices: The role of storage devices in computer systems is to persistently store data. Thus, the natural security extension is to protect stored data from attackers, preventing undetectable tampering and permanent deletion [6]. A self-securing storage device does this by managing storage space from behind its security perimeter, keeping an audit log of all requests, and keeping previous versions of data modified by attackers. Since a storage device cannot distinguish compromised user accounts from legitimate users, the latter requires keeping all versions of all data. Finite capacities will limit how long such comprehensive versioning can be maintained, but 100% per year storage capacity growth will allow modern disks to keep several weeks of all versions. If intrusion detection mechanisms reveal an intrusion within this multi-week *detection window*, security administrators will have this valuable audit and version information for diagnosis and recovery. This information will simplify diagnosis, as well as detection, by not allowing system audit logs to be doctored, exploit tools to be deleted, or back doors to be hidden — the common steps taken by intruders to disguise their presence. This information will simplify recovery by allowing rapid restoration of pre-intrusion versions and incremental examination of intermediate versions for legitimate updates. By embedding this data protection functionality inside the storage device, one enjoys its benefits even when the network, user accounts, or host OSs are compromised.

Biometric sensors: The role of biometric sensors in computer systems is to provide input to biometric-enhanced authentication processes, which promise to distinguish between users based on measurements of their physical features. Thus, the natural security extension is to ensure the authenticity of the information provided to these processes. A self-securing sensor can do this by timestamping and digitally signing its sensor information. Such evidence of when and where readings were taken is needed because, unlike passwords, biometrics are not secrets [4]. For example, anyone can lift fingerprints from a laptop with the right tools or download facial images from a web page. Thus, the evidence is needed to prevent straightforward forgery and replay attacks. Powerful self-securing sensors may also be able to increase security and privacy by performing the identity verification step from within their security perimeter and only exposing the results (with the evidence). By embedding mechanisms for demonstrating authenticity and timeliness inside sensor devices, one can verify sensor information (even over a network) even when intruders gain the ability to offer their own "sensor" data.

Graphical displays: The role of graphical displays in computer systems is to visually present information to users. Thus, a natural security extension would be to ensure that critical information is displayed. A self-securing display could do this by allowing high-privilege entities to display data that cannot be overwritten or blocked by less-privileged entities. So, for example, a security administrator could display a warning message when there is a problem in the system (e.g., a suspected trojan horse or a new e-mail virus that must not be opened). By embedding this screen control inside the display device, one gains the ability to ensure information visibility even when an intruder gains control over the window manager.

Routers and switches: The role of routers and switches in a network environment is to forward packets from one link to an appropriate next link. Thus, one natural security extension for such devices is to provide firewall and proxy functionality; many current routers provide exactly this. Some routers/switches also enhance security by isolating separate virtual LANs (VLANs). More dynamic defensive actions could provide even more defensive flexibility and strength. For example, the ability to dynamically change VLAN configurations would give security administrators the ability to create protected command and control channels in times of crisis or to quarantine areas suspected of compromise. When under attack, self-securing routers/switches could also initiate transparent replication of data services, greatly reducing the impact of denial-of-service attacks. Further, essential data sites could be replicated on-the-fly to "safe locations" (e.g., write-once storage devices) or immediately isolated via VLANs to ensure security. Self-securing routers/switches can also take an active role in intrusion detection and tracking, by monitoring and mining network traffic. When an attack is suspected, alerts can be sent to administrators and to other self-securing devices to increase security protections. By embedding traffic monitoring and isolation functionality in self-securing routers/switches, one can enjoy its benefits even when firewalls and systems on the internal network are compromised.

Application-only CPUs: Though not strictly devices, most future host systems are likely to have multiple CPUs. They already have multiple functions, including OS-level resource management and various application-level tasks. Rather than trying to correctly implement and use a sandbox to safely host iffy code, we again suggest using physical boundaries — that is, run untrusted code on a separate *application-only CPU* that has no kernel (in the traditional sense) and no kernel-like capabilities. An application-only CPU should be physically locked away from its virtual memory mappings and device communication. The mappings, permissions, and external communication should be controlled by separate *management CPUs*, with which the application-only CPU communicates via a well-defined protocol. With such an organization, the safety of the hosted code becomes less critical, and the boundaries between it and more trusted components become more explicit.

4. Newly-enabled dynamic actions

Many new dynamic network security actions are enabled by the more numerous and heterogenous security perimeters inherent to the self-securing device architecture. To illustrate the potential, this section describes a few such actions:

Network DefCon Levels: Often, there is a trade-off between security and performance. For example, the more detailed and numerous the firewall rules, the greater the overhead introduced. Likewise, the more detailed the event logging, the greater the overhead. One use of the many new security perimeters is to support dynamic increases of security level based on network-wide status. For example, if an attack can be detected after only a small number of perimeters are compromised, the security levels at all other self-securing devices can be dynamically raised. As suggested above, this might take the form of more detailed firewalling at NICs, logging of network traffic to storage, and dynamic partitioning of the network into distinct VLANs.

Email Virus Stomping: One commonly observed security problem is the rapidly-disseminated e-mail virus. Even after detecting the existence of a new virus, it often takes a significant amount of time to root it out of systems. Ironically, the common approach to spreading the word about such a virus is via an e-mail message (e.g., "don't open unexpected e-mail that says 'here is the document you wanted'"). By the time a user reads this message, it is often too late. An alternative, enabled by self-securing NICs, is for the system administrator to immediately send a new rule to all NICs: check all in-bound and out-bound e-mail for the new virus's patterns. This would immediately stop further spread of the virus within the intranet, as well as quickly identifying many of the infected systems.

Traffic Throttling at the Source: As the previous example suggests, self-securing NICs allow network traffic to be throttled at its sources. Thus, a system that is deemed "bad" could have its network traffic slowed or cut off completely. Also, such malicious network activity as "SYN bombs" and IP address spoofing can be detected, terminated at its source, and even automatically repaired by the source's NIC (e.g., sending RST packets to clear SYN bomb connections).

Biometric Identity Verification: A more exotic use of self-securing devices is auxiliary identity checks on users. For example, imagine that an authenticated user does something that triggers an intrusion detection alarm. There are many possible explanations, one of which is that someone else is using the real user's session (e.g., while the real user is away at lunch). To check for this, a network security administrator could silently consult a nearby (or attached) self-securing video camera and perform face or iris recognition. Many other biometrics could also be used. The intrusion detection system could even trigger this check automatically and terminate the corresponding system's network and storage access, if the user is deemed to be an imposter.

Migration of Critical Data from Compromised Systems: If a system is compromised, one important action is trying to save and retain access to its user data. In our new architecture, this can be done by having the self-securing storage device (appropriately and authoritatively directed) encrypt and send the relevant contents over the network via the self-securing NIC. The self-securing router can forward the data to one or more alternate locations and route subsequent accesses to the data appropriately. In fact, different user bases could be routed to distinct replicas. With emerging device-to-device I/O interconnects, the storage-to-network transfer can be done with no host OS involvement at all, leaving the successful intruder with no way to stop it. Going back to the first example, another use of this support would be to frequently transfer the audit logs from various self-securing devices to on-line intrusion detection systems during perceived siege situations.

Displaying Trojan-defeating Messages: In perhaps the simplest example, a security administrator could direct a self-securing graphics card to override system directives and display a warning message. Such support would be particularly useful when users need to be warned to discontinue (or not start) using a system suspected of housing Trojan horses. Again, device-to-device communication allows this to happen over the network without host OS interference.

5. Research challenges

This change in network security architecture raises two major research questions, each with a number of sub-questions. First, “what should each device do behind its security perimeter?” Answering this question will require exploration of cost, performance, and flexibility trade-offs, as well as exploring what is possible with the limited information available at any given device. Section 3 outlines potential functionalities for a number of devices. Second, “how does one effectively manage a large collection of independent security perimeters?” Answering this question will require exploration of tools and techniques for marshaling sets of self-securing devices, monitoring their current state, and dynamically updating their policies in the face of changes to and attacks upon the network environment’s state.

The second question raises several complex sub-questions that must be answered in order to realize dynamic and robust network security environments from large collections of distinct security perimeters. The clearest sub-questions center on administrative control over the various devices, where security and convenience must be balanced. Research is also needed into how to reason about global network security given the set of local insights provided by distinct host systems and self-securing devices. Many other sub-questions exist, including those related to local policy configuration, robust reconfiguration, coordinated intrusion diagnosis, and avoidance of internally-imposed denial-of-service.

6. Related Work

Several researchers have used the siege warfare analogy to promote more comprehensive information security defenses [1, 3, 5]. Usually, the associated proposals are only loosely connected to the analogy, simply referring to the strengths (e.g., many parts), weaknesses (e.g., traitors), or eventual replacement of siege defenses. We use the analogy to inspire a specific defense strategy: use of physically-distinct barriers that monitor one another, defend collectively, and must be penetrated independently.

The concept of using physical separation of functionality for security is also not new. Perhaps the simplest examples are physically-secured machines with no network connections. Perhaps the best examples are firewalls and proxies, which enforce rules on network traffic entering and leaving an intranet via hardware specifically dedicated to this purpose. Here, we propose using physical component boundaries as the core of a security architecture rather than as a bandaid on inherently insecure network environments. The references below identify and discuss more related work.

7 Summary

This white paper promotes a new security architecture in which traditional boundary protections are coupled with security functionality embedded into self-securing devices. The resulting collection of independent security perimeters could provide a flexible infrastructure for dynamic prevention, detection, diagnosis, isolation, and repair of successful intrusions. Although many research challenges arise, we believe that the new architecture has great potential.

References

[1] Bill Cheswick. Security Lessons From All Over. Keynote Address, USENIX Security Symposium, 1998.

[2] David Friedman and David F. Nagle. *Building Scalable Firewalls with Intelligent Network Interface Cards*. CMU-CS-00-173. Technical Report, Carnegie Mellon Univeristy School of Computer Science, December 2000.

[3] Jr. John L. Woodward. Information Assurance Through Defense in Depth. U.S. Department of Defense brochure, February 2000.

[4] Andrew J. Klosterman and Gregory R. Ganger. *Secure Continuous Biometric-Enhanced Authentication*. CMU-CS-00-134. Technical Report, Carnegie Mellon Univeristy School of Computer Science, May 2000.

[5] Gary McGraw and Greg Morrisett. Attacking Malicious Code: A Report to the Infosec Research Council. *IEEE Software*, pages 33–41, September/October 2000.

[6] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised systems. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 165–180. USENIX Association, 2000.

Research Issues in No-Futz Computing

David A. Holland, William Josephson,
Kostas Magoutis, Margo I. Seltzer, Christopher A. Stein
Harvard University
Ada Lim
University of New South Wales

Abstract

At the 1999 Workshop on Hot Topics in Operating Systems (HotOS VII), the attendees reached consensus that the most important issue facing the OS research community was "No-Futz" computing; eliminating the ongoing "futzing" that characterizes most systems today. To date, little research has been accomplished in this area. Our goal in writing this paper is to focus the research community on the challenges we face if we are to design systems that are truly futz-free, or even low-futz.

1 Introduction

The high cost of system administration is well known. In addition to the official costs (such as salaries for system administrators), countless additional dollars are wasted as individual users tinker with the systems on their desktops. The goal of "no-futz" computing is to slash these costs and reduce the day-to-day frustration that futzing causes users and administrators.

We define "futz" to mean "tinkering or fiddling experimentally with something." That is, futzing refers specifically to making changes to the state of the system, while observing the resulting behavior in order to determine how these relate and what combination of state values is needed to achieve the desired behavior. When we refer to "no-futz" computing, we mean that futzing should be allowed, but should never be required. We interpret "low-futz" in this way as well.

It should be noted that reducing futz is not the same as making a system easy to use. It is also not the same as hiding or reducing complexity: it is about *managing* complexity and *managing* difficulty. Computer systems involve intrinsically complex and difficult things. These are not going to go away. The goal is to make it as easy as possible to cope with that complexity and difficulty.

Systems can be easy to use but still require unnecessary futzing: TCP/IP configuration on older Macintoshes was easy to adjust, but was difficult to set properly. One can also imagine a (purely hypothetical) system that hides all its complexity: it appears to need almost no futzing at all, until it breaks. Then, extensive futzing is required, to figure out what happened.

The goal of No-Futz computing is to eliminate the futzing due to poor design or poor presentation, not to try to find a silver bullet for software complexity; no-futz computing attacks areas that are needlessly complicated, not those that are inherently complicated.

Let's begin with an example of a good, hi-tech, low-futz device, and understand its basic characteristics. While reading the rest of this section, keep in mind the computer systems you use regularly (particularly the ones you dislike) and how they differ from the example.

Our Xerox 256ST copier is a no-futz device. It performs just about every function imaginable for a copier: it collates, staples, copies between different sizes of paper, will copy single-sided originals to double-sided copies and vice versa, etc., and it even sits on the network and accepts print jobs like a printer. However, it demands no futzing. It has instructions printed on the case that describe how to accomplish common tasks. Its user interface makes it impossible to ask it to do something it cannot. It keeps track of its operating state, continuously monitoring itself, and communicates in a simple fashion with its operators. When there is a problem it can diagnose, it displays a clear message on its console. (For example, "Paper tray 2 is empty.") When it detects a problem it cannot diagnose, it begins a question-and-answer dialog with the operator to diagnose the problem and determine an appropriate course of action - and then, in most cases, it guides the operator through that course of action. The questions it asks are simple, and can be answered by a novice, such as "Did paper come out when you tried to copy?" The key factors that make this device no-futz are:

- Ease of use: The user documentation and user interface are organized in terms of the user's tasks, not in terms of the system's internal characteristics.
- It is unusual to encounter a situation where it is not clear what to do next, even in the presence of various failures.
- Self-diagnostics: When a failure occurs, the copier diagnoses it and offers instructions for fixing things.
- Simple, clear communication: It never asks the user a question that the user cannot answer.

What makes this such an interesting example is that only a decade ago, photocopiers required much futzing, mostly by expert servicemen, and were extremely frustrating for all concerned. Since then, not only have copiers become vastly faster and more powerful, but both the use and maintenance of them has become vastly easier. Today's copiers have one-tenth the components of their predecessors, significantly more functionality, and dramatically reduced futzing [6]. How can we make similar strides forward in computing?

That which works for a photocopier may not be sufficient for computers: the copier is a relatively straightforward device with well-defined function and state, whereas general-purpose computer systems have a wide variety of functions, have essentially infinitely mutable state, and are subjected to complicated and often ill-understood interconnections both within themselves and with other computers.

In the rest of this paper, we first discuss some current approaches to futz reduction, arguing that these do not attack the problem directly and have negative side-effects. We then discuss how futz arises in computer systems and describe what we believe is the key to a real solution: understanding and managing system state. Then we outline some directions for future research, discuss briefly some existing related work, and conclude.

2 Current Approaches to Futz Reduction

The cost and frustration associated with futzing has led to three common approaches to futz reduction: (1) limiting the scope of functionality, (2) homogeneity, and (3) centralization. These approaches are not mutually exclusive and are frequently used together.

The copier described above is an example of the first approach: it is a special-purpose device. Relative to a general-purpose computer, its functionality is quite limited. In this context, it has addressed the futz problem quite well. Since futzing involves state changes, special purpose systems, which have relatively limited state spaces, can offer correspondingly reduced futz. Other low-futz, limited scope devices include dedicated file servers (e.g., Network Appliance's filers) and special purpose web or mail servers (e.g., Sun's Cobalt servers) among others.

Homogeneity is the second approach to futz avoidance. This approach is most often seen in large installations. In order to reduce total installation-wide futzing, a single standard machine configuration is deployed everywhere. If there is a problem, any machine can be replaced with any other machine. Systems can be reinstalled quickly from a master copy. Maintenance requirements are reduced drastically. Custom management tools need only interact with one kind of system, and are thus much cheaper to build. The administrators see the same problems over and over again and can prepare solutions in advance; nobody besides the administrators needs to futz with anything.

This approach can reduce global futz drastically; however, it does not address the underlying problem: the amount of futzing required by a single machine is constant. Furthermore, it has other flaws: first, it is inherently incompatible with letting users control their computers. While this is fine or even desirable in some environments (e.g., the terminals bank tellers use), it is unacceptable in others (e.g., research labs). Second, it is a security risk. The same homogeneity that makes system administration easier also makes break-ins and virus propagation easier: if you can get into one system, you can get into all of them the same way [1]. Third, most organizations grow incrementally. Adding new computers to a collection of identical existing ones is difficult: the new ones are rarely truly identical, which inevitably cuts into the economy of scale.

The third approach to futz reduction is centralization. Centralization moves state and its accompanying requirements for futzing, away from the systems with which people interact directly and into places where it is more conveniently managed. This gives administrators tight and efficient control over each system. This makes it more convenient for system administrators to futz and lets system administrators do more of the futzing and users less of it. While this does reduce cost, there is no actual reduction in total futz. For that, another approach is required.

These three approaches are capable of reducing the futz of, or at least the cost of maintenance for, computer systems and networks. However, all of them are limiting and/or have negative consequences. This is a result of attempting to reduce the total futzable state, instead of the futz problem directly. We advocate the direct attack.

3 The Source of Futz

One definition of "futz" is in terms of state manipulation. Thus, the more state there is to manipulate, the more futzing a system allows. Mandatory futzing arises when it is not clear by inspection or documentation what manipulations are required or when the supposedly correct manipulations fail to produce the correct result. At this point, one must experiment (or call for help).

If one can manipulate the system state without resorting to experimentation, futzing has not occurred. For instance, seasoned Unix administrators do not have to futz to add accounts to their systems. But beginners generally do. And even seasoned administrators usually have to futz to get printing to work.

Note that the degree of futz depends on the level of expertise of the user. A premise of no-futz computing, however, is that one should not have to be an expert, or the cost of being an expert should be quite low. Unix systems are already quite low-futz for hard-core experts, but it takes years and years of apprenticeship to reach that level. Reducing futz for a select few is not a solution, so we need to examine sources of futz as they appear to a casual user.

The mutable state of a computer system can be broken down into the following categories (this may not be a complete list):

- **Derived state:** State automatically derived or generated from other state.
- **Policy state**: Configuration state that reflects policy of a site or user.
- **Autoconfig data**: Data to be served in some manner by the system in order to enable autoconfiguration for other systems. For example, /etc/bootptab.
- **Cached state**: Cached results from autoconfiguration protocols.

- **Manual config state**: Configuration state that reflects the setup of the operating environment or hardware, and needs to be set manually.
- **OS file state**: files (programs or data) that are part of the operating system, as well as their organizational meta-data.
- **Application file state**: files (programs or data) that are part of installed applications, as well as their organizational meta-data.
- **User file state**: user files and their organizational meta-data. For example, a secretary's word processor files, or web pages.
- **Application context**: persistent saved application state that is not user data. For instance, many environments try to automatically recreate on startup where you were when you left the last time.
- **System context**: persistent OS state that is not in any of the above categories. For example, file system meta-data.
- **Cryptographic keys**.

Policy state is a source of futz: the system acts on its policy settings, and if it acts incorrectly, somebody needs to tinker with the settings until it behaves properly. Unfortunately, policy state cannot be avoided in a general-purpose computer system: policy decisions need to be made by humans and the computer needs to know what they were. One can reduce futz in this area by cutting back the amount of state, and building special-purpose systems, but that inherently reduces the amount of functionality as well. Reducing futz in this area without cutting back functionality is feasible as we outline in the next section.

Autoconfig data is another source of futz. This category reflects futz that has been "centralized away" from other systems. It is not necessarily the case that all autoconfig mechanisms require a server to serve data, but many of the existing ones do. It is not unreasonable to suppose that development of more sophisticated autoconfiguration can reduce or eliminate most of the state and thus the futz in this category.

Cached state is not normally a source of futz. Cached results can be purged or updated as necessary without any manual intervention. Similarly, derived state is a solved problem: if it goes out of date, it needs only to be regenerated. The Unix make utility is already routinely used for this.

Manual config state is a tremendous source of futz in most systems today. Worse, it is the most difficult kind of futz possible: unlike policy state, where various alternatives work but may not be desired, most of the questions answered by manual config state have only one or two right answers and plenty of wrong answers, and wrong answers generally render the system or components of it completely inoperative. Ultimately, this is the category of futz that is most seriously in need of reduction. Fortunately, it is possible to accomplish this: to the extent that there are right answers, in almost all cases, with sufficient engineering of components, those right answers can be probed or determined from context. For instance, the only reason we need video card and monitor information in /etc/XF86Config is that on PC-based systems it is not possible in many cases to safely or reliably interrogate the hardware to find out what it is. In a hypothetical world where you could query this hardware, which is easy to imagine, this major source of futz could be abolished.

OS file state and application file state are an area in which many current systems fall down: it is quite easy, in general, to install new application software that breaks the system, or to update the system and thereby break applications. It is also possible to delete or rename important files inadvertently (or lose in a power failure) thereby breaking the system. At present, recovering from these problems is generally quite difficult. In this area, for most people, futzing at all tends to equate to reinstallation.

Reducing this category of futz requires taking more care in analyzing the dependencies among software components, and improving the mechanisms with which software components are bound to one another at runtime. We need several things: automated analysis of runtime dependencies (a hard problem), better systems for preventing accidental version skew, and mechanisms for cross-checking that can be performed at runtime to allow failures to occur gracefully. Reinstallation as a failure recovery mechanism is unacceptable.

User file state is inevitably a source of futz as things become disorganized and users mislay their data. We see no immediate prospects of cutting back on the futzing this requires, although developing a good model for how applications should choose default save directories and the like would be a good start. Content indexing techniques may be of help as well.

Application context is normally automatically maintained, and only becomes a source of futz when it becomes corrupted or saves an undesired application state. This problem is easily solved: check it for consistency when loaded, be able to withstand it being deleted, and store it in a known location so users can delete it if they so desire. In many cases, simply not keeping such context is an adequate solution.

System context is essentially the same, except that it is sometimes not possible (or meaningful) to erase it and start over. It is much more important to check it for consistency and repair any problems. With some engineering, failures that require expert attention to repair can be made quite rare, as they generally are with most Unix implementations of fsck.

Cryptographic keys are listed separately because they have their own unique requirements for management, and because they are mandatory for the use of secure autoconfiguration protocols. In our experience, these are not large sources of futz. Furthermore, a lot of attention has already been paid to key management in the security literature.

All the above assumes that a user is changing state in order to make some kind of desired configuration change, either as ongoing maintenance or at system installation time. There are two other cases in which one needs to interact in intimate detail with the state of a system: to diagnose and repair a system failure and to monitor the system for signs of upcoming failure.

Properly speaking, as we have defined futzing, diagnosis is not futzing; rather than experimentally adjusting state to achieve a result, diagnosis properly involves analyzing existing state. Sometimes, however, one needs to experiment to interpret the existing state. And additionally, a common method for recovering from a system failure is to futz until the obvious signs of the failure have disappeared and the system appears to be working again. (Rebooting is a drastic example of this technique, and it works because much system state is not persistent across reboot.)

The reason this method works is that many system problems involve the failure of supposedly automatic state management mechanisms; tweaking the state tickles the state management mechanism, and with some luck it will start functioning again. The reason it is common is that actual diagnosis by inspection usually amounts to debugging and requires an extremely high level of expertise.

If the system can diagnose problems itself, like our copier can, this futzing becomes unnecessary. Even if it can only diagnose a small number of the most common problems, a good deal of mandatory futzing can be eliminated. Self-diagnosis in software systems is an important research area. We believe a good deal of progress is possible.

Monitoring for signs of upcoming failure, including monitoring for security problems, does not, itself, involve futzing. However, failure to perform monitoring can lead to huge amounts of futzing later on - recovering from a server dying can easily take as much futzing as installing a new one, whether the death took place because of hardware failure or because of hackers. Therefore, automatic monitoring is also crucial to building true no-futz systems. This is another important research area.

Ultimately, all of these things - monitoring, diagnosis, and configuration - involve interaction with the system state. We believe that research and engineering in the areas outlined above can tame a good proportion of the typical system state space. However, policy state, cryptographic keys, and probably some leftover bits of state in the other categories, are not going away. More is required; we need to be able to manage this state.

4 Futz and State Management

The less state a system has, the easier it is to organize and present to users in a coherent manner.

As outlined in the previous section, one can design out some state and automate the handling of a lot more. This will take care of a good deal of futz. However, a great deal of state remains, and it requires editing, and undoubtedly, futzing. One cannot eliminate the editing. But one may be able to eliminate the futzing.

The leftover state consists mostly of policy state, manual config state, and autoconfig state. This state can be thought of as a list of configuration questions and their answers. The ultimate goal is to allow a user to type in answers to these questions, or change the answers to suit changed circumstances, without needing much training or specialized knowledge.

It should now be clear that question formulation is crucial — not just their wording, although that is significant, but what questions are asked, how interconnected they are with each other, how they're grouped, etc.

What this means is that, once all the easier issues are addressed, the organization of the state space of the system is the most significant factor determining how much futzing the system will demand.

It is crucial to analyze this state space in detail and determine how to best decompose it into a set of variables (and thus questions). In the best such decomposition, the variables will be as simple and as orthogonal to each other as possible. It will be clear what answering each question entails and who, in any of several typical environments, ought to decide the answer. Then the questions need to be written in such a manner that the people who typically fill these roles can, in fact, answer the questions without needing an excessive amount of training, and the software needs to be written so that questions will not be posed to the wrong people.

For example, in almost all cases, the person sitting at the computer should be the one to choose the desktop background. However, it is not necessarily the case that this person should be asked "What is the IP address of your web proxy?" — this question may need to be posed, but if so it should be posed in a context where it is clear that the answer is the local network administrator's responsibility.

We believe this is the key. It is not an easy problem; in the absence of any useful decomposition theorems for state spaces or state machines, it must be solved by manual inspection and ad-hoc heuristic analysis. Worse, one has to address the complete state space of the entire system at once; if one leaves some state out of the analysis and tacks it on later, it is almost guaranteed to be a poor fit.

At first glance this might seem to mean that all application software must be designed into the system. This is not the case. However, what *is* necessary is for the sorts of state applications may need to use to be anticipated; that is, one needs an abstract model of what an application is and does. Such a model should be reasonably general without going overboard: applications that fail to fit will still work, but may require increased amounts of futzing. Allowing for these applications in the general design might result in even more futzing in the common case. There will be a trade-off, and that trade-off will need to be explored.

5 Research Directions in No-Futz Computing

If the systems community is to ever build no-futz systems, we must embark on a research program that addresses the key issues in no-futz computing. This section defines those areas.

The first step on the path to no-futz computing is determining how to measure a system's futz. We wholeheartedly endorse the term “FutzMark” coined at the last HotOS and challenge researchers to define it.

We believe the central issue in no-futz computing is state management. We must reduce system state to a manageable level, isolate each state variable so that it is orthogonal to other state variables, and make it impossible to specify invalid states. Where possible, we should replace state with dynamic discovery. Where possible, we should devise ways to turn static state into dynamically discoverable state (e.g., autoconf data, manual config state). Achieving orthogonality is perhaps the most difficult aspect of this task, but also the most essential. Without orthogonality, the problems of management and testing grow factorially. If we can achieve orthogonality, it becomes a manageable linear problem.

In lieu of total orthogonality, we need better mechanisms to identify inconsistent state and remedy it. We need to identify (or avoid) version skew among software components and do more extensive runtime cross checking and analysis.

Coping with failure requires a great deal of futzing; thus we need to achieve cleaner failure models. In the fault-tolerance community, “failstop” behavior (ceasing operation as soon as a fault occurs) is considered desirable so that failing systems do not corrupt state or data. In the context of no-futz, failstop behavior could permit the precise identification of failure causes. If systems can diagnose their own failures, it’s conceivable that they can then direct users to perform recovery, as our copier does. In general, we need to make progress in the areas of self-diagnosis and automatic monitoring.

Finally, there are areas outside of systems research where progress is necessary. In particular, improvements in user interfaces and data presentation will reduce futz. Collaborative interfaces, which act as intermediaries between users and their machines that enable them to work together, hold great potential if applied to no-futz computing. Security management is sometimes considered outside the realm of systems, but insecurity is a major contributor to current futz and improvement is needed. Improvements in content indexing will reduce the futz associated with user data management.

6 Related Work

There have been a number of efforts to reduce futz in computer systems. In a distributed setting, Sun’s Sunray [4], as well as Microsoft’s Zero Administration initiative and the associated IntelliMirror [7] product, are projects to centralize futzing.

The Sunray system’s desktop machines are simple, stateless I/O devices with no administration needs. Sunray relies on modern off-the-shelf interconnection technology and a simple display update protocol (SLIM) to support good interactive performance. In addition to eliminating client administration, the Sunray model offers client mobility. Client session state is entirely stored on the server and can be associated with a smart card that can be inserted in any Sunray client connected to the same server. Sunrays are anonymous commodities. However, this does not eliminate the administration cost. Sunray servers are complicated systems and not easy to administer: once, in our department, one of the junior system administrators broke all the Sunrays for three days just by trying to install a new utility on the Sunray server.

Microsoft’s Zero Administration initiative is an effort to reduce the administration needs of Windows installations and thus the cost of ownership. Central to Zero Administration is the IntelliMirror product, which helps an administrator (a) manage user data, (b) install and maintain software throughout an organization, and (c) manage user settings. Management of user data requires knowledge of properties and locations of users’ files so that the data is available both online and offline from any computer. Manual installation, configuration, upgrades, repair and removal of software across an organization requires large management effort. IntelliMirror automates this: it offers remote OS installation, a service allowing a computer connected on a LAN to request installation of a fresh copy of the Windows OS, appropriately configured with applications for that user and that computer.

Sun’s Jini [5] for Java is an example of a system that tries to eliminate administration in a decentralized (“federated”) manner. Jini provides a distributed infrastructure for services to register with the network and clients to find and use them.

7 Conclusion

Leading systems researchers identified no-futz computing as an important research area two years ago [3], but to the best of our knowledge, there has been no significant research activity in this area. We believe one reason is that the problem is enormously complex and may not be solvable within the constraints of legacy systems. Regardless, until we identify the important research questions, no progress can be made. In this paper, we have identified some, if not all, of the important areas in which research must be conducted if we are ever to “solve” the problem of high-futz systems.

8 References

[1] Forrest, S., Somayaji, A., and Ackley, D., “Building diverse computer systems,” *In Sixth Workshop on Hot Topics in Operating Systems*, 1997.

[2] Dan Plastina, “Microsoft Zero Administration Windows”, invited talk given at the 11th USENIX Systems Administration Conference (LISA ‘97), October 26-31, 1997, San Diego, California, USA

[3] Satyanarayanan, M., “Digest of Proceedings”, Seventh IEEE Workshop on Hot Topics in Operating Systems, March 29-30 1999, Rio Rico, AZ, USA.

[4] Schmidt, B. et al., “The interactive performance of SLIM: a stateless, thin-client architecture”, in Proceedings of the 17th SOSP, December 1999, Kiawah Island, SC, USA.

[5] Waldo, J., “The Jini Architecture for Network-centric Computing” *Communications of the ACM*, pp 76-82, July 1999.

[6] Conversation with Xerox Technical Representative. January 18, 2001.

[7] http://www.microsoft.com/WINDOWS2000/library/howitworks/management/intellimirror.asp as of April 23, 2001.

Don’t Trust Your File Server

David Mazières and Dennis Shasha
NYU Department of Computer Science
{dm,shasha}@cs.nyu.edu

Abstract

All too often, decisions about whom to trust in computer systems are driven by the needs of system management rather than data security. In particular, data storage is often entrusted to people who have no role in creating or using the data—through outsourcing of data management, hiring of outside consultants to administer servers, or even collocation servers in physically insecure machine rooms to gain better network connectivity.

This paper outlines the design of SUNDR, a network file system designed to run on untrusted servers. SUNDR servers can safely be managed by people who have no permission to read or write data stored in the file system. Thus, people can base their trust decisions on who needs to use data and their administrative decisions on how best to manage the data. Moreover, with SUNDR, attackers will no longer be able to wreak havoc by compromising servers and tampering with data. They will need to compromise clients while legitimate users are logged on. Since clients do not need to accept incoming network connections, they can more easily be firewalled and protected from compromise than servers.

1 Motivation and significance

People are increasingly reliant on outside organizations to manage their data. From data warehouses that store a company’s crucial data to consultants performing system administrative duties like backups, more and more data is being left in the hands of people who have no role in creating or using the data. Unfortunately, those who manage data also have the ability to read it or even change it in subtle, difficult to detect ways. Thus, organizations end up placing a great deal of trust in outsiders. Worse yet, the decision to trust outsiders is often driven as much by system management needs as by concern for data security.

This research was supported in part by National Science Foundation Career award CCR–0093361.

At the same time, another trend is developing: people increasingly need to share information with a population of users that cannot be confined by firewalls. Web servers outside of firewalls distribute software and information to the general population. People collocate servers in foreign machine rooms to get better network connectivity. Such outside servers have proven quite vulnerable to attack. While many attacks merely involve defacing web pages to prove a machine has been penetrated, the consequences of a compromise could be much more serious—for instance when a server distributes software many people download and run.

To address these problems, we are building SUNDR—the secure untrusted data repository. SUNDR is a novel distributed data storage system in which the functions of data storage and management can be performed by someone who has no ability to read or modify the data. On client machines, SUNDR behaves like an ordinary network file system, providing reasonable performance and semantics. However, the client cryptographically protects and verifies all data it exchanges with the server. An attacker with complete control of a SUNDR server can accomplish little more than a simple and very noticeable denial of service attack. For any attack to go undetected, the perpetrator would need to compromise a client while an authorized user is logged in.

SUNDR will be the first general-purpose network file system to provide strong data integrity guarantees without trusting the server. Previous work has shown how to protect data secrecy on untrusted servers, or how to protect the integrity of data that is read and written by the same person. In practice, however, many files on a system are readable but not writable by ordinary users. Thus, to replace an existing, general-purpose file system with one that eliminates trust in the server, one must support read-only access to shared data.

SUNDR’s design exploits the fact that on modern processors, digital signatures have become cheap enough to compute on every file close. Whenever a client commits a change to the file server, the client effectively digitally signs the entire contents of the file system (though in an efficient, incremental way).

SUNDR will not only protect data integrity, but also make assurances about the recentness of data. Ensuring that users see each others' updates to a file system is a challenging problem when the server is untrusted and the users may not simultaneously be on-line. SUNDR introduces a new notion of data consistency called *relative freshness*: Two SUNDR users will either see all of each others changes to a file system or none. If two users can communicate with each other (for instance, if they are on the same local Ethernet when accessing a remote SUNDR server), they will be guaranteed traditional close-to-open consistency. Otherwise, if the server violates consistency, the users can never again see each others' updates to the file system, thus maximizing the chances of their detecting the attack.

SUNDR's security model makes remote data storage useful where it previously was impractical. However, remote network bandwidth is typically lower than LAN bandwidth. SUNDR will therefore employ large client caches and exploit commonality between files to reduce network traffic. Thus, one can easily "outsource" file service to organizations across the network without heavily penalizing performance, but while also achieving several benefits. The local organization will no longer need to worry about backups (no harm is done if the local cache is lost). Moreover, when several sites share a file system over the wide area network, they will see considerably better performance than with traditional file systems. Finally, because SUNDR does not trust servers, it will allow cooperative caching amongst mutually distrustful clients, and will allow client caches to be used in the reconstruction of server state in the event of a failure.

2 Related work

While a number of file systems have used cryptographic storage in the past, none has provided strong integrity guarantees in the face of a compromised server. The swallow [13] distributed file system used client-side cryptography to enforce access control. Clients encrypted files before writing them to the server. Any client could read any file, but could only decrypt the file given the appropriate key. Unfortunately, one could not grant read-only access to a file. An attacker with read access could, by controlling the network or file server, substitute arbitrary data for any version of a file.

CFS [2] allows users to keep directories of files that get transparently encrypted before being written to disk. CFS does not allow sharing of files between users, nor does it guarantee freshness or integrity of data. It is intended for users to protect their most sensitive files from prying eyes, not as a general-purpose file system. Cepheus [5] adds integrity and file sharing to a CFS-like file system, but trusts the server for the integrity of read-shared data.

The Byzantine fault-tolerant file system, BFS [3], uses replication to ensure the integrity of a network file system. As long as more than $2/3$ of a server's replicas are uncompromised, any data read from the file system will have been written by a legitimate user. SUNDR, in contrast, will not require any replication or place any trust in machines other than a user's client. However, SUNDR and BFS provide different freshness guarantees.

The proposed OceanStore file system [1] names file objects with secure "GUID" handles. For archival objects, which are immutable over all time, the GUID is simply a collision-resistant cryptographic hash of a file's contents, allowing clients to verify the contents directly from the GUID. For mutable objects, the GUID is the hash of a public key and username. Data returned as part of the file must be digitally signed by the private key corresponding to the GUID. File names are mapped to GUIDs using SDSI [14] (effectively using SDSI as a file system).

We actually built a file system (the old read-only file system, described in [10]) in which files are individually signed like mutable OceanStore objects. We subsequently rejected the design for several reasons, however, the most important of which was security. There is no way to guarantee the freshness of files when they are individually signed. If a file was signed a year ago, is that because an attacker wants us to accept a year-old version, or has the user user simply not resigned the file in the past year? To address this problem SUNDR signs file systems rather than individual files, and it introduces the notion of relative freshness.

SUNDR uses hash trees, introduced in [12], to verify a file block's integrity without touching the entire file system. Duchamp [4], BFS[3], SFSRO [6] and TDB [9] have all made use of hash trees for comparing data or checking the integrity of part of a larger collection of data.

The CODA file system [7] saves network bandwidth by operating in a disconnected mode, saving changes locally and later reconciling the changes over a fast network link. SUNDR, too, must save bandwidth for people to use servers over the wide area network. However, SUNDR is designed for constant, lower-bandwidth connectivity (e.g., a T1), rather than an intermittent high-bandwidth connection. Like CODA, SUNDR reduces read bandwidth with a large client-site cache. In addition, SUNDR exploits commonality between files to compress data exchanged between clients and servers. The approach is similar to one used by the rsync [16] file transfer utility.

3 Design

SUNDR has several design goals. The server must not see any private file data in the clear. Clients should never be tricked into believing a file contains data that wasn't written there by an authorized user. Furthermore, SUNDR should provide an easy way to recover if a server is indeed compromised. Finally, SUNDR should provide the best guarantees it can on the freshness of data.

Unfortunately, because SUNDR assumes so little about security, there are two attacks it cannot protect against. An attacker can destroy data, wiping or even physically destroying a disk. We call this a *smashing attack*. Second, if users only ever communicate with each other through the file system, an attacker can clone the server, show each user a separate copy, and prevent each user from finding out that the other has updated the file system. We call this a *forking attack*. These vulnerabilities inherently result from letting attackers control a file server. Fortunately, SUNDR can make the attacks easy to detect and recover from.

3.1 Architectural overview

SUNDR will use a two-level architecture reminiscent of Frangipani [15] on Petal [8]. At the lowest level is the SUNDR block server, a network server that stores chunks of data for users. The block server neither understands nor interprets the blocks it is stores for users. This, in large part, is what lets the block server be managed by someone who does not have access to the file system itself. However, the maintainer of the block server can still perform traditional administrative tasks such as making backups and adding more disk space.

The actual file system in SUNDR is built on top of the block server and is implemented entirely by clients. A SUNDR file system is effectively a collection of hash trees—one per user—mapping per-user inode numbers to the contents of files. The nodes of the hash trees are stored at the block server. Given the root directory and the proper decryption keys, a client can, using the block server, fetch and verify the contents of any file in the system.

Every SUNDR file system has a public key, known to all clients accessing the server. SUNDR will use the SFS [11] file naming scheme, which embeds file system public keys in the file namespace. Thus, users can employ any of SFS's key management techniques to obtain a file server's public key. Unlike the current SFS file system, however, a SUNDR server will not know its own private key. The private key will be known only to those with superuser privileges on the file system.

Each user of a SUNDR file system also has a public key. Users' public keys serve two purposes. They authenticate users to the block server, to prevent an unauthorized person from consuming space or deleting blocks still in use. They are also used to digitally sign any changes to the file system, so that other clients can verify the updated contents. The superuser's public key is the one embedded in the file namespace.

Each SUNDR block server also has a public key, called the server key, used for authentication of the server to clients. It may seem odd to authenticate a block server that is supposedly untrusted anyway. However, SUNDR does not guarantee that servers will not misbehave. It only guarantees that bad behavior will be detected, and facilitates recovery. The operator of a block server can then be held accountable, and any damage repaired from backups and client caches. The server key's role is therefore to assure clients they are communicating with a responsible server. Other parties cannot impersonate the server, so users can squarely pin the blame on the operator if anything goes wrong.

3.2 The SUNDR block server

Both for efficiency and because it is not trusted to operate on high-level file constructs, the SUNDR server implements a simple block protocol. The server stores two types of state: user metadata, and file system blocks. The user metadata consists of a table of user public keys and a signed *version structure* for each user, described further in Section 3.3. The superuser's public key is the public key of the file system (the one embedded in the file system's pathname on clients). Note that the term "user" here really designates a user or a group in the traditional UNIX sense. A group is just a SUNDR user with a zero-sized quota (so it cannot allocate space) and a private key known to several people.

The main function of the block server is to store and serve blocks of data that clients can interpret as a file system. The server indexes blocks of data by their cryptographic hashes. A client can store a block at the server and later request the contents of the block by its cryptographic hash. For each block stored, the block server also records a list of users "requiring" the block, and a reference count for each user. When no one requires a block any more, the server garbage collects it and recycles the disk space. Users may have a quota limiting the amount of data they can require a server to store.

Clients and servers communicate over an authenticated link. They negotiate a session key using the server key and the user's public key, and authenticate requests and replies using the session key for a symmetric message authentication code (MAC). Thus, the server knows that a request to store or delete a block really comes from a particular user, and the user knows that an acknowledgment of a store request comes from a server that knows the private half of the appropriate server key.

While the client can send the server an entire block to store, it can also send a description of the block based on existing blocks stored by the server—for instance, to store a new block that is identical to an existing block but for a 20 byte region, or to create a new block by appending the beginning of one block to the end of another. This protocol makes it particularly bandwidth efficient to update all the blocks in a hash tree stored at the server, as a client can simply transmit a new leaf node along with a bunch of small deltas to the parent blocks.

3.3 The SUNDR file system

The SUNDR file system is implemented by clients on top of the block server. The fundamental data structure in the file system is the *virtual inode*, shown in Figure 3.3, with which one can efficiently retrieve and verify any portion of a file. The virtual inode contains a file's metadata and the size and cryptographic hashes of its blocks. If the file is not world-readable, the metadata also includes an index into a table of symmetric encryptions keys to designate the key with which file contents has been encrypted. The table entry can only be decrypted by users with the appropriate private keys. For large files, the inode also contains the hash of an *indirect block*, which in turn contains hashes and sizes of file blocks. The inode further contains the size of all data pointed to by the indirect block. For larger files, an inode can point to double-, triple-, or even quadruple-indirect blocks.

For every user there is an ordered list, known as the *i-table*, mapping 64-bit per-user inode numbers to the cryptographic hashes of virtual inodes. The i-table is broken into blocks, converted to a hash tree, and each node stored at the block server. The hash of the tree's root is known as the user's *i-handle*. The data pointed to by a directory inode is a list of $\langle \text{file name}, \text{user}, \text{inode number}\rangle$ triples, sorted by file name.

Each user's version structure consists of that user's i-handle a set of $\langle \text{user}, \text{version}\rangle$ pairs, one for each user of the system. The entire version structure is digitally signed using the user's private key. Thus, given a user's signed version structure, a client can obtain the file contents of any inode number by repeatedly requesting blocks by their cryptographic hash. Inode number 2 in the superuser's i-table has special significance. It is the root directory of the file system.

To update the file system, a client first uploads new blocks to the file server (sending only deltas for blocks in which only a hash has changed). It then brings the user's version structure up to date, putting in the new i-handle and updating version numbers, signing the new structure and sending it to the server. Finally, the client decrements the reference count on any blocks no longer needed.

3.4 Consistency protocol

The goal of SUNDR's consistency protocol is to make it as easy as possible to detect if the server has not faithfully provided a consistent view of the file system to all clients. The SUNDR server is responsible for assigning an order to all file open and close operations (excluding closes of unmodified files). The order assigned by the server must preserve the order of each individual client's operations.

Each user of a SUNDR file system has a version number, incremented each time the user updates the file system. The basic principle behind the SUNDR consistency protocol is that each user signs not only his own i-handle and version number, but also what he believes to be the version numbers of all other users. Thus, each user's version structure contains a version number for every other user as well.

When a user u opens a file, the client increments u's own version number v and signs an *update certificate* with the user's private key, $\{\text{UPDATE}, v+1\}_{K_u^{-1}}$. When a client closes one or more modified files (more in the case of directory operations such as rename), the update certificate contains the inode numbers of the modified files, $\{\text{UPDATE}, v+1, I_1, \ldots\}_{K_u^{-1}}$. The client sends the update certificate to the server. The server assigns the operation a place in the order of operations, and sends back any new signed version structures the client has not yet seen. It also sends back other users' update certificates if they are more recent that those users' version structures.

Definition 1 *If x and y are two version structures, we say that $x \geq y$ iff for all users u, $\langle u, v_x\rangle \in x$ and $\langle u, v_y\rangle \in y$ implies $v_x \geq v_y$.*

The client uses the signed messages it receives to update its own version structure s. For each version structure s' that the client sees, it verifies that either $s \geq s'$ or $s' \geq s$. If not, the client declares that a forking attack has occurred. If no attack is detected, the client signs the updated version structure and uploads it to the server.

When a client is determining each user's latest version number, it considers both signed version structures and update certificates forwarded by the server. The client accepts an update certificate so long as the file declared modified in the certificate is not the one being opened. Otherwise, it must wait for the version structure (thereby also obtaining the file owner's new i-handle, and ultimately the new file contents).

Definition 2 *User u_1 has* ***close-to-open consistency*** *with respect to u_2 iff:*

1. *There exists a partial order,* happens before, *on all open and close operations such that any two operations by the same user are ordered, and any close of a file is ordered with respect to all opens and other closes of that same file.*

2. *When u_1 opens file F, and the last close of F to happen before the open was performed by u_2, u_1 will see the contents written by u_2.*

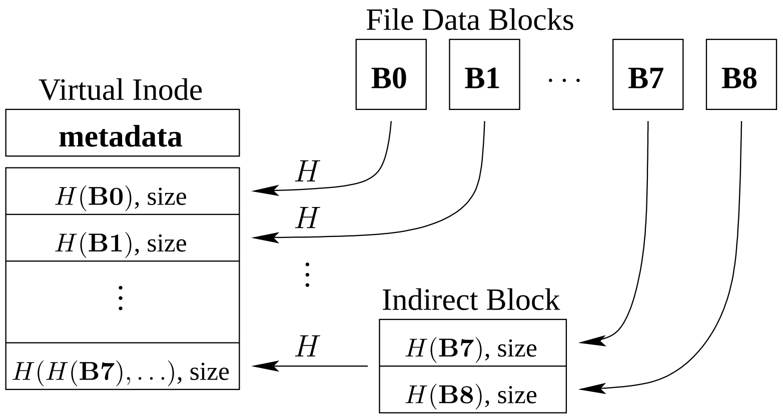


Figure 1. The SUNDR virtual inode structure

Application	savings
dvips	84%
g++	59%
ar	97%

Figure 2. Redundancy in files written by various programs after small changes in the source.

Definition 3 *A file system provides **relative freshness** iff, whenever user u_1 sees the effects of an open or close operation $\mathcal{O}$ by u_2, then at least until u_2 performed $\mathcal{O}$, u_1 had close-to-open consistency with respect to u_2.*

The SUNDR consistency protocol guarantees relative freshness (proof omitted). Given this property, when the clients of two users can communicate on-line, they can achieve close-to-open consistency with respect to each other. The clients need only keep each other up to date about their users' latest version numbers. When each of two users does not know if the other will be on-line, a corrupt SUNDR server can mount a forking attack. However, once such an attack has occurred, the server can never again show either user another update by the other. The corrupt server can therefore partition the set of users and give each partition its own clone of the file system, but when partitioned users have any out-of-band communication—for instance one user in person telling the other to look at a new file—the forking attack will be detected.

3.5 Low-bandwidth protocol

The SUNDR block protocol lets clients use delta-compression to avoid sending an entire metadata block to the server when the new block is identical to an old one but for a small hash value. The same protocol can be used to achieve dramatic compression of file blocks sent to the server under many realistic workloads. The most obvious workload is that of a text editor. When one edits and saves a large file, the editor generally creates and writes a new file substantially similar to the old one.

Many other workloads generate substantially redundant file system traffic, however. To give a sense for this, Figure 2 lists the savings in bandwidth one could obtain from using delta-compression on three typical workloads. The dvips workload consisted of generating a postscript file for this paper twice, with some minor edits in between. 84% (by space) of the write calls made by dvips the second time around were for 8KB blocks that already existed in the first output file (albeit at different offsets). The g++ workload consisted of recompiling and linking a 1.3MB C++ program after adding a debugging print statement. 59% of the writes could have saved using delta compression between the two versions. ar consisted of regenerating a 3.9MB C++ library after modifying one of the source files. 97% of the writes could have been saved.

SUNDR can use the rsync algorithm [16] to reduce bandwidth when one file is very similar to another. However, that still leaves the problem of actually pairing up old files with new ones. In some cases this is easy. For example, dvips just truncates the old file and writes out the new one. Since allocating and freeing of blocks in SUNDR is controlled by the client and decoupled from the file system's structure, the client can, space permitting, temporarily delay removing truncated or deleted files from the server and its local cache in order to use them for delta compression. In other cases, however, the relation between old and new files may be less obvious. The emacs text editor, for instance, saves a file by creating and writing out new file with a different name, forcing that file's contents to be committed to disk, and only then renaming the new file to replace the old.

Fortunately, the SUNDR client can index its local file cache. Using Rabin fingerprints, the client can efficiently compute a running checksum of every (overlapping) 8K block in a file. The client can then index cached files by their n lowest valued checksums for some small n. Many of these checksums will likely appear in a new version of the file to be written out to the server. Thus, the client can pick a file against which to run the rsync algorithm by looking up a small number of fingerprints in its index.

The encryption of blocks slightly complicates delta compression. SUNDR will encrypt data with a block cipher using the combination OFB/ECB mode of [2], but each block stored at the server will have a randomly chosen IV prefixed to the block. When updating internal nodes of the file system, this encryption scheme allows a 20 byte hash in an encrypted block to be updated by changing only 32 bytes of ciphertext.

The situation is more complicated on leaf nodes, when the client is attempting to apply differences between two files in its local cache to an encrypted version of the file on the server. It is for this reason that the SUNDR virtual inode structure permits variable size data blocks and contains the sizes of those blocks. The client can preserve intact any blocks from the old file that are present in the new file, and glue those blocks together with odd-sized fragments. Performing write compression in this way will automatically give other clients the same factor in read compression, as it will maximize the number of the new file's blocks that already exist in their local caches.

4 Summary

The SUNDR file system securely stores data on untrusted servers. Thus, people can base trust decisions on who needs to use data and administrative decisions on how best to manage the data. While some attacks simply cannot be prevented—for instance physical destruction of the hard disk—SUNDR makes it easy to detect and recover from such problems. For instance, after restoring a server from backup, recent changes can securely be merged in from the caches of untrusted clients. SUNDR also introduces the notion of relative freshness—the guarantee that users will see all of each other's changes or none. While weaker than traditional file system consistency guarantees, relative freshness easily lets clients verify tighter consistency guarantees through client-to-client communication.

References

[1] D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, W. Weimer, C. Wells, B. Zhao, and J. Kubiatowicz. Oceanstore: An exteremely wide-area storage system. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000. (to appear).

[2] M. Blaze. A cryptographic file system for unix. In *1st ACM Conference on Communications and Computing Security*, pages 9–16, November 1993.

[3] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, February 1999.

[4] D. Duchamp. A toolkit approach to partially disconnected operation. In *Proceedings of the 1997 USENIX*, pages 305–318. USENIX, Jan. 1997.

[5] K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, May 1999.

[6] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[7] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[8] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92. ACM, October 1996.

[9] U. Maheshwari and R. Vingralek. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000. (to appear).

[10] D. Mazières. Security and decentralized control in the SFS distributed file system. Master's thesis, Massachusetts Institute of Technology, August 1997.

[11] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawa Island, SC, 1999. ACM.

[12] R. C. Merkle. A digital signature based on a conventional encryption function. In C. Pomerance, editor, *Advances in Cryptology—CRYPTO '87*, number 293 in Lecture Notes in Computer Science, pages 369–378, Berlin, 1987. Springer-Verlag.

[13] D. Reed and L. Svobodova. Swallow: A distributed data storage system for a local network. In A. West and P. Janson, editors, *Local Networks for Computer Communications*, pages 355–373. North-Holland Publ., Amsterdam, 1981.

[14] R. L. Rivest and B. Lampson. SDSI—a simple distributed security infrastructure. Working document from `http://theory.lcs.mit.edu/~cis/sdsi.html`.

[15] C. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, Saint-Malo, France, 1997. ACM.

[16] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, 1997.

Active Protocols for Agile, Censor-Resistant Networks

Robert Ricci   *Jay Lepreau*

University of Utah

{ricci,lepreau}@cs.utah.edu   http://www.cs.utah.edu/flux/

January 22, 2001

Abstract

In this paper we argue that *content distribution in the face of censorship* is a compelling and feasible application of active networking. In the face of a determined and powerful adversary, every fixed protocol can become known and subsequently monitored, blocked, or its member nodes identified and attacked. *Rapid and diverse protocol change* is key to allowing information to continue to flow. Typically, *decentralized protocol evolution* is also an important aspect in providing censor-resistance.

A programmable overlay network can provide these two features. We have prototyped such an extension to Freenet, a storage and retrieval system whose goals include censor resistance and anonymity for information publishers and consumers.

1 Introduction

The ability to communicate effectively—even in the face of censorship attempts by a hostile party, such as a repressive government—is important to maintaining the values held by many societies. As The New York Times reported [17], just last week a corrupt head of state was toppled from power, "due in no small part" to 100,000 people responding to a "blizzard" of wireless text messages summoning them to demonstrations. But what if the government had deployed a powerful jamming signal, or simply taken over the cell phone company?

The fundamental rationale for active networking [16]—allowing the network itself to be programmable or extensible by less than completely trusted users—is to ease the deployment of new protocols throughout the network infrastructure. Active networking's flexibility is its only real virtue, since any one protocol can be more efficient, more robust, and have fewer troubling side effects if it is part of the fixed network infrastructure. Thus it has proven difficult to demonstrate even a single compelling application of active networking, as it is really the *space* of applications that is compelling.

Rob Ricci, a fulltime undergraduate student, is the primary author. Jay Lepreau is the contact author.

Many active networking publications discuss the benefits of facilitating deployment of new protocols over existing networking infrastructure. Overlay networks such as Freenet [6, 7] that aim to provide a censorship-resistant document publishing mechanism seem a good fit for such a protocol update system: once in place, it is possible that hostile governments, ISPs, or network administrators might attempt to monitor or block Freenet links. One way to make this task more difficult is to diversify the protocols by which two peer nodes can communicate. Ideally, rather than expanding the size of the set of protocols spoken from one to a small finite number, the size of the protocol set would be theoretically unbounded, to prevent an attacker from learning every member of the set.

The keys to making this strategy successful are to allow and encourage the deployment of new hop-by-hop protocols at any time, even after the system is in wide use, and to allow any user of the system to introduce new protocols. The system should be able to evolve rapidly to react to changes in its environment. There should be no central source of new protocols to become vulnerable. Any member of the network should be able to decide upon a new protocol to use, and "teach" its neighbors to use the new protocol when communicating with it. Thus, any node is able quickly to take action if it deems that a protocol change is desirable. Such adaptive, evasive protocols may—and probably will—be inefficient, but above some threshold, that is not a significant concern. Like other projects (e.g., Oceanstore) we explicitly choose to exploit the ever-growing supply of network bandwidth and processing power for benefits other than speed.

If the publishing system's core is implemented in Java [1] or similar typesafe language, or if the core can interface with such typesafe code, then such evolution can be implemented by using mobile bytecode that implements a new node-to-node protocol. When a node wishes to change the protocol spoken with one of its peers (either because of suspected attack, or as a matter of course), it can send the peer bytecode that implements the new protocol. We call this passed bytecode a Protocol Object. In our experiment, we call the result of extending Freenet with Protocol Objects, "Agile Freenet."

2 Related Work

Outside of active networks, mobile code has often been used to support heterogeneous environments and platforms (pervasive computing [12] and Sun's Jini), data transcoding and proxies [15], moving computation to data (Mobile Agents), or towards more abundant computational or I/O resources (e.g., applets in Web browsers).

In the wireless realm there is a long history of electronic response to jamming, either accidental or purposeful, often using spread-spectrum techniques. Software radios [3] and other more traditional approaches provide adaptive physical-layer protocols. All of these wireless efforts emphasize improved performance by seeking less-used parts of the spectrum, or by using spectrum in a more sophisticated manner. Ad-hoc networks, whether mobile or not, apply adaptive protocols in a more extensive manner [14], and although they must sometimes consider issues of trust, have so far also focused on efficiency and performance.

"Radioactive networks" [4], in which active networking is proposed as a way to extend a software radio infrastructure, comes closest to the ideas in this paper. The authors' goals are primarily the traditional goals of adaptive wireless protocols: better performance through better use of spectrum and energy. However, they do mention security as a potential benefit, and suggest a software radio system that can vary its spectrum spreading codes to avoid jamming.

It is interesting that the first known active network, Linkoping Technical Institute's Softnet [21], also involved radio, though in a different manner. Softnet, in 1983, implemented a programmable packet radio network, building upon a special Forth environment. It allowed users to extend the network with their own services and protocols above the physical layer.

In terms of censor-resistance, a user-programmable collection of wireless nodes would have strengths that a wired network does not possess. In the latter, typical users are almost entirely vulnerable to their sole ISP. Wireless nodes, particularly if they have software-defined waveforms and a multitude of accessible peer nodes, provide a large set of diverse paths to the broader network.

In our content-distribution application area, there are an increasing number of censor-resistant and anonymous publishing efforts, some of which are outlined in the next section. To our knowledge, none of them use active code.

3 Censor-resistant Publishing Networks

Freenet employs a variety of techniques aimed at creating a censorship-free, anonymous environment. Central to Freenet's strategy is the distribution of data across a large number of independently-administered nodes. Freenet is decentralized; requests are propagated through a series of peer-to-peer links. When a file is transferred, the file is cached by all nodes on the route between the requesting node and the node where a copy of the file is found. Frequently requested files are thus replicated at many points on the network, making the removal or censorship of files infeasible. A peer forwarding a Freenet message from one of its peers is indistinguishable from a peer originating the message, providing a degree of anonymity for the suppliers and requesters of data. Freenet is only as good, however, as the network it is built upon; underlying networks hostile to Freenet can potentially block or monitor its connections, preventing Freenet from fulfilling its goals. Agile protocols, therefore, can provide many potential benefits.

Other systems incorporate similar ideas in different contexts. Gnutella [11] provides a file sharing and search system that is decentralized across widely dispersed nodes, but does not maintain endpoint anonymity. Publius [19] offers anonymity for publishers, and plausible deniability for servers. The FreeHaven [8] design—so far unimplemented and known to be inefficient—has similar goals to Freenet, but uses a wider variety of techniques to offer stronger guarantees of anonymity and document persistence.

In all of the above systems, monitoring can reveal information, even if it cannot directly discover the contents of a message, or identify its endpoints [5]. Large quantities of cover traffic, many participating nodes, and widespread routine use by others of end-to-end encryption are required for many of the publishing networks to function effectively. Recognizing that a given data stream belongs to one of these networks is not necessarily difficult, and can give an attacker information on the usage, behaviors or identities of network users. In addition, once such communications are recognized, they can be selectively blocked. Using agile protocols for communication can make this task much more difficult for an attacker.

4 Agile Protocols

The Case for Agile Protocols

An agile protocol, as we define it, is a protocol whose details can be changed arbitrarily and quickly while the system is running. The most flexible way to do this is through mobile code.

If the goal of Agile Freenet were simply to *obscure* Freenet connections, then it would probably be sufficient simply to encrypt them, headers and all. It is desirable, however, to be able to *disguise* Freenet connections as well, by communicating over a protocol that is similar in appearance to some well-known application-level protocol, such as HTTP, SMTP, etc. Statically including some set of these protocols in each release of the software would be a solution, but would give a potential attacker a small set of protocols to understand. Additionally, this would create problems with peer nodes running different versions of the software, as they would all be under separate administrative control. Instead, by allowing nodes to exchange protocol implementations, we make the set of protocols spoken on the network dynamic, regardless of the version of the software nodes are running, and making an attacker's job very difficult.

New Protocol Objects could implement steganography [13], proxying through third parties, tunneling through firewalls and proxy servers, and other techniques to make them difficult to monitor and block. Most importantly, using agile protocols allows us to take advantage of such technologies and others yet to be discovered *as soon* as they are developed. Users can write Protocol Objects to suit their own network situation (for example, if they are behind a firewall that allows only certain ports through, or wish to tunnel their connections through some unusual proxy service) and distribute it easily to their Freenet peers and other users without having to go through any central or even local authority. In response to a determined adversary, new protocols might be written and deployed daily or hourly, all on a decentralized and demand-driven basis.

Issue: Level of Programmability

Allowing any user to introduce new protocols into a censor-resistant network presents clear threats. The censors themselves will certainly have the means and motivation to introduce malicious protocols. We cannot, therefore, allow arbitrary programmability, but must restrict the API available to the active code. There is an obvious and permanent tension between constraining the active code and allowing it space to diversify. In addition, it is useful to draw a distinction between the overall architecture of a content-distribution network, i.e., its global invariants and central algorithms, and the details of its hop–by–hop communication protocol. It is clearly safer to allow programmability of the latter than the former. Wetherall's retrospective on active networking [20] reaches an analogous conclusion: under complete programmability, it is feasible to assure the safety of individual nodes, but not of the overall network.

Comparison to Traditional Active Networking

The issues that confront agile protocols in general, and Agile Freenet in particular, differ in some ways from the problems that traditionally have been the target of active networking research.

First, new protocols need not be spoken along an entire data path, only on individual point-to-point links. In fact, it is desirable to have a very diverse set of different protocols spoken, in order to make the system as dynamic as possible. A given file transfer could be accomplished with as many different Protocol Objects as there are links in the route it takes. This eliminates the necessity for a more complicated "capsule" system like ANTS [20], which ensures that each hop on a given route has the proper (and identical) code to run a given protocol.

Second, file sharing overlay networks such as Freenet tend to be systems where data being searched for is extremely likely to be available from many sources. This lessens the importance of a single point-to-point link, as, even if an individual link goes down or misbehaves, data is likely to be available through some other route. Additionally, such networks involve some user interaction, so some classes of problems can be addressed by the users. Thus, we do not require strong guarantees about the correctness or efficiency of each Protocol Object.

5 Agile Freenet

5.1 Basics

Identification: Identifying protocol objects can be done simply by computing a cryptographic hash of the protocol bytecode—this eliminates any need for a centralized naming scheme, and allows hosts to distinguish between Protocol Objects that they already have, independent of any identifier they may be given. When a node wishes to change the protocol it is speaking on a point-to-point link with a peer, it sends its peer a message containing this hash. If the peer does not have the bytecode matching the hash, it can request that the originating node send it. A similar but more complicated method is employed in the ANTS toolkit.

Bootstrapping: Bootstrapping an agile protocol can be problematic—when a node wishes to contact a new potential peer, it must be assured that there is a common protocol that both it and its peer speak. This problem, however, is shared by the base Freenet system itself—joining the network requires learning the network addresses and ports of potential peers through some out-of-band mechanism. These mechanisms could be extended to encompass more information—users wishing to join the Agile Freenet could obtain files from their prospective peers containing network address, port numbers, public keys (whose purpose will be discussed later), and one or more Protocol Objects that are in use by that node.

5.2 Potential Dangers

Protocol Objects help to thwart listeners or attackers in some ways, but also have the potential to be a tool for them. Here, we outline potential dangers. The next section addresses solutions.

Compromise of Local Information: A malicious Protocol Object inserted into Agile Freenet could attempt to discover information about nodes that it is spread to, such as the files in their cache and their list of peer nodes. Discovery of such information could lead to the compromise of some of Freenet's core goals, such as anonymity.

Disclosure to an outside source: A malicious Protocol Object could contact a third party and disclose addresses of nodes, keys being searched for, data being transferred, etc.

Failure: Protocol Objects could fail, either maliciously or through poor programming. This failure could be total or intermittent.

Selective Failure: This is a more insidious instance of failure, in which a Protocol Object fails only for certain requests or data transfers, to prevent them from serving requests for certain files, or certain types of data.

Corruption: Protocol Objects could corrupt the data passing through them to disrupt the integrity of the information stored in Agile Freenet, or to somehow "tag" transfers for tracing.

Resource Usage: A malicious or poorly written Protocol Object could consume excessive system resources, degrading its performance

5.3 Combating the Dangers

Given the dangers outlined above, it is easy to see that some precautions will be necessary when adopting our Agile Protocol. Here, we outline ways of protecting the core Freenet system from malicious or poorly written Protocol Objects.

Namespace Isolation: With current Java Virtual Machines, it is possible to limit which classes a given object can resolve references to, through ClassLoader objects. This can be used to restrict access to parts of the Freenet node, system classes, etc, that are not necessary for a Protocol Object to use, and prevent compromise of local information.

Network Isolation: Using the namespace isolation technique discussed above, we can force Protocol Objects to use a network API that we define, which can perform checks to insure that a Protocol Object is not making unauthorized network access or contacting a third party.

Rating of Protocol Objects: To combat Protocol Object failure, each node can maintain a rating system for each Protocol Object it uses, evaluating each one's effectiveness based on factors like success rate for searches, dropped connections, and detected corruption (discussed below.) It can then decide not to use Protocol Objects that have too low of a rating. Note that this may make perfectly good Protocol objects look bad—the peers they are used with may not have much data cached, may be unstable, etc. This is acceptable—the primary goal is to prevent the spread of "bad" Protocol Objects.

Data Encryption: Selective failure and corruption can be solved by encrypting data before passing it to the Protocol Object, using a key unknown to it. In this way, we can prevent Protocol Objects from discriminating based on the data being transferred, as they do not have the ability to read it. Also, we can send separately-encrypted checksums or digital signatures to verify that the data has not been corrupted.

Resource Management: Agile Freenet can be run in a Java Virtual Machine that supports resource management between different Java "applications," using features such as those available in KaffeOS [2] or Janos [18]. However, just as in Web browsers running untrusted Java code, simpler mechanisms should suffice for initial deployment. E.g., existing OS mechanisms can limit the JVM to a fixed share of memory and cpu, and the node user or administrator can be notified if a limit is consistently reached.

5.4 Encryption

Encryption is useful at several different points in Agile Freenet. First, as already discussed, it can prevent Protocol Objects from discovering what data they are transmitting or receiving, and can ensure that data transferred has not been tampered with or otherwise altered.

Second, if two peer nodes wish to exchange a new Protocol Object, this transfer should be encrypted, even if the "normal" data they are sending is not. If switching protocols is done because of a suspicion that communications are being monitored, then transferring a new Protocol Object unencrypted would give the listener the bytecode necessary to continue listening when the new protocol is used, or at least a means to determine which protocol is being used. Encrypting Protocol Objects as they are being transferred denies a listener this advantage.

5.5 Experience

We have implemented the basic framework described in this paper by modifying the current Freenet implementation [10] to incorporate Protocol Objects. Since our extensions to Freenet were different than those envisioned by its developers, we found it moderately difficult to extend. However, once the framework was in place, we were pleased with the resulting extensible system.

Our prototype sends the bytecode for Protocol Objects over the network and loads it into a restricted Java execution environment using standard Java ClassLoader mechanisms; sensitive Freenet and system APIs are hidden. We implemented three different Protocol Objects. One implements the standard Freenet protocol, another mimics HTTP syntax to facilitate tunneling through HTTP,[1] and a third implements TCP "port-hopping." Nodes can,

[1]We will soon demonstrate HTTP tunneling via passage through a stock Web proxy. Doing so awaits our extending Freenet internals in a minor way, so that "in-bound" connections can fake the Web's "client-side-initiates" behavior.

at the behest of their peers, change Protocol Objects at any Freenet "message" (file) boundary.[2] This diversity is on a per-peer basis, allowing a node to speak an arbitrarily different protocol, on a different port, at different times, to each of its peers.

The prototype was developed and tested on our scalable Network Emulation facility [9]. We plan to continue development of our prototype as a test platform for research on agile protocols.

6 Conclusion

Censor-resistant content distribution networks provide a compelling application of active networking technology. Agile protocols seem likely to improve significantly such networks' resistance to monitoring and blocking, without an unduly large increase in the potential damage from malicious protocols. We have demonstrated, in the Freenet system, that such an extension is feasible. What remains as future work is evaluating the extent of improvement, increasing the range of protocol variants, and ultimately deploying and evaluating agile protocols in the live Freenet. It should be easy to interest students in such a contest between publishers and censors.

[2]We are extending Freenet to allow changing connection attributes at any byte boundary. These attributes include ports (including spreading a single connection over many ports) and peer host (to allow "bouncing" through another host).

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, second edition, 1998.

[2] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, pages 333–346, San Diego, CA, Oct. 2000. USENIX Association.

[3] V. Bose, M. Ismert, M. Welborn, and J. Guttag. Virtual Radios. *IEEE Journal on Selected Areas in Communications*, 17(4), Apr. 1999.

[4] V. Bose, D. Wetherall, and J. Guttag. Next Century Challenges: RadioActive Networks. In *Proc. of the Fifth Annual ACM/IEEE Internation Conference on Mobile Computing and Networking*, pages 242–248, Aug. 1999.

[5] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 4(2), February 1982.

[6] I. Clarke. A Distributed Decentralized Information Storage And Retrieval System. Master's thesis, University of Edinburgh, 1999. Available at http://-freenet.sourceforge.net/freenet.pdf.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. Berkeley, California, July 2000. International Computer Science Institute. Revised December 18 2000. Available at http://freenet.sourceforge.net/icsi-revised.ps.

[8] R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven Project: Distributed Anonymous Storage Service. Berkeley, California, July 2000. International Computer Science Institute. Revised December 17 2000. Available at http://www.freehaven.net/doc/berk/freehaven-berk.ps.

[9] Flux Research Group, University of Utah. Network Testbed and Emulation Facility Web Site. http://-www.emulab.net/ and http://www.cs.utah.edu/flux/-testbed/.

[10] The Freenet Project. http://freenet.sourceforge.net/.

[11] The Gnutella Project. http://gnutella.wego.com/.

[12] R. Grimm, T. Anderson, B. Bershad, and D. Wetherall. A System Architecture for Pervasive Computing. In *Proc. of the Ninth ACM SIGOPS European Workshop*, pages 177–182, Kolding, Denmark, Sept. 2000.

[13] N. Johnson and S. Jajodia. Exploring Steganography: Seeing the Unseen. *IEEE Computer*, 31(2):26–34, 1998.

[14] J. Kulik, W. Rabiner, and H. Balakrishnan. Adaptive Protocols for Information Dissemination in Wireless Sensor Networks. In *Proc. of the Fifth Annual ACM/IEEE Internation Conference on Mobile Computing and Networking*, Aug. 1999.

[15] Steven D. Gribble et al. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Computer Networks*, 2001. To appear in a special issue on Pervasive Computing. Available at http://ninja.cs.berkeley.edu/dist/-papers/ninja.ps.gz .

[16] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. *Computer Communication Review*, 26(2), Apr. 1996.

[17] The New York Times. Text Messaging is a Blizzard That Could Snarl Manila. *New York Times*. January 20, 2001.

[18] P. Tullmann, M. Hibler, and J. Lepreau. Janos: A Java-oriented OS for Active Network Nodes. In *IEEE Journal on Selected Areas in Communications, Active and Programmable Networks*, 2001. To appear.

[19] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A Robust, Tamper-evident, Censorship-resistant, Web Publishing System. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[20] D. J. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 64–79, Kiawah Island, SC, Dec. 1999.

[21] J. Zander and R. Forchheimer. Softnet – An approach to high level packet communication. In *Proc. Second ARRL Amateur Radio Computer Networking Conference (AMRAD)*, San Francisco, CA, Mar. 1983.

Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel

George Candea Armando Fox
Stanford University
{candea,fox}@cs.stanford.edu

Abstract

Even after decades of software engineering research, complex computer systems still fail, primarily due to nondeterministic bugs that are typically resolved by rebooting. Conceding that Heisenbugs will remain a fact of life, we propose a systematic investigation of restarts as "high availability medicine." In this paper we show how recursive restartability (RR) — the ability of a system to gracefully tolerate restarts at multiple levels — improves fault tolerance, reduces time-to-repair, and enables system designers to build flexible, highly available software infrastructures. Using several examples of widely deployed software systems, we identify properties that are required of RR systems and outline an agenda for turning the recursive restartability philosophy into a practical software structuring tool. Finally, we describe infrastructural support for RR systems, along with initial ideas on how to analyze and benchmark such systems.

1 Introduction

Despite decades of research and practice in software engineering, latent and pseudo-nondeterministic bugs in complex software systems persist; as complexity increases, they multiply further, making it difficult to achieve high availability. It is common for such bugs to cause a system to crash, deadlock, spin in an infinite loop, livelock, or to develop such severe state corruption (memory leaks, dangling pointers, damaged heap) that the only high-confidence way of continuing is to restart the process or reboot the system.

The rebooting "technique" has been around as long as computers themselves, and remains a fact of life for substantially all nontrivial systems today. Rebooting can be applied at various levels: Deadlock resolution in commercial database systems is typically implemented by killing and restarting a deadlocked thread in hopes of avoiding a repeat deadlock [15]. Major Internet portals routinely kill and restart their web server processes after waiting for them to quiesce, in order to deal with known memory leaks that build up quickly under heavy load. A major search engine periodically performs rolling reboots of all nodes in their search engine cluster [3]. Although rebooting is often only a crude "sledgehammer" for maintaining system availability, its use is motivated by two common properties:

1. **Restarting works around Heisenbugs.** Most software bugs in production quality software are Heisenbugs [27, 8, 17, 2]. They are difficult to reproduce, or depend on the timing of external events, and often there is no other way to work around them but by rebooting. Even if the source of such bugs can be tracked down, it may be more cost-effective to simply live with them, as long as they occur sufficiently infrequently and rebooting allows the system to work within acceptable parameters. The time to find and deploy a permanent fix can sometimes be intolerably long. For example, the Patriot missile defense system, used during the Gulf War, had a bug in its control software that could be circumvented only by rebooting every 8 hours. Delays in sending a fix or the reboot workaround to the field led to 28 dead and 98 wounded American soldiers [34].

2. **Restarting can reclaim stale resources and clean up corrupt state.** This returns the system to a known, well-tested state, albeit with possible loss of data integrity. Corrupt or stale state, such as a mangled heap, can lead to some of the nastiest bugs, causing extensive periods of downtime. Even if a buggy process cannot be trusted to clean up its own resources, entities with hierarchically higher supervisory roles (e.g., the operating system) can cleanly reclaim any resources used by the process and restart it.

Rebooting is not usually considered a graceful way to keep a system running – most systems are not designed to tolerate unannounced restarts, hence experiencing extensive and costly downtime when rebooted, as well as potential data loss. Case in point: UNIX systems that are abruptly halted without calling `sync()`.

The Gartner Group [31] estimates that 40% of unplanned downtime in business environments is due to application failures; 20% is due to hardware faults, of which 80% are transient [8, 25], hence resolvable through reboot. Starting from this observation, we argue that in an *appropriately designed* system, we can *improve* overall system availability through a combination of reactively restarting failed components (revival) and prophylactically restarting functioning components (rejuvenation) to prevent state degradation that may lead to unscheduled downtime. Correspondingly, we present initial thoughts on how to design for recursive restartability, and outline a research agenda for systematic investigation of this area.

The paper is organized as follows: In section 2, we explain how the property of being recursively restartable can improve a system's overall availability. In section 3, we present examples of existing restartable and non-restartable systems. Section 4 identifies some required properties for recursively restartable systems and proposes an initial design framework. Finally, in section 5, we outline a research agenda for converting our observations into structured design rules and software tools for building and evaluating recursively restartable systems. Many of the basic ideas we leverage have appeared in the literature, but have not been systematically exploited as a collection of guidelines; we will highlight related work in the context of each idea.

2 Recursive Restartability Can Improve Availability

"Recursive restartability" (RR) is the ability of a system to tolerate restarts at multiple levels. An example would be a software infrastructure that can gracefully tolerate full reboots, subsystem restarts, and component restarts. An alternate definition is provided by the following recursive construction: the simplest, base-case RR system is a restartable software component; a general RR system is a composition of RR systems that obeys the guidelines of section 4. In the present section we describe properties of recursively restartable systems that lead to high availability.

RR improves fault tolerance. The unannounced restart of a software component is seen by all other components as a temporary failure; systems that are designed to tolerate such restarts are inherently tolerant to all transient non-Byzantine failures. Since most manifest software bugs and hardware problems are short lived [25, 27, 8], a strategy of failure-triggered, reactive component restarts will mask most faults from the outside world, thus making the system as a whole more fault tolerant.

RR can make restarts cheap. The fine granularity of recursive restartability allows for a bounded portion of the system to be restarted upon failure, hence reducing the impact on other components. This way, the system's global time-to-repair is minimized (e.g., full reboots are replaced with partial restarts), which increases availability. Similarly, RR allows for components and subsystems to be independently rejuvenated on a rolling basis; such incremental rejuvenation, unlike full application reboots, makes software rejuvenation [21] affordable for a wide range of 24×7 systems.

RR provides a confidence continuum for restarts. The components of a recursively restartable system are tied together in an abstract "restartability tree," in which (a) siblings are well isolated from each other by the use of simple, high-confidence machinery, and (b) a parent can unilaterally start, stop, or reclaim the resources of any of its children, using the same kind of machinery. For example, in a cluster-based network service, the root of the tree would be an administrator, each child of the root would be a node's OS, each grandchild a process on a node, and each great-grandchild a kernel-level process thread. This tree captures the tradeoff that, the closer to the root a restart occurs, the more expensive the ensuing downtime, but the higher the confidence that transient failures will be resolved. In the above example, processes are fault-isolated from each other by the hardware-supported virtual memory system, which is generally a high-confidence field-tested mechanism. The same mechanism also allows parents to reclaim process resources cleanly. Nodes are fault-isolated by virtue of their independent hardware. When a bug manifests, we can use a cost-of-downtime/benefit-of-certainty tradeoff to decide whether to restart threads, processes, nodes, or the entire cluster.

RR enables flexible availability tradeoffs. The proposed rejuvenation/revival regimen can conveniently be tailored to best suit the application and administrators: it can be simple (reboot periodically) or sophisticated (differentiated restart treatment for each subsystem/component). Identical systems can have different revival and rejuvenation policies, depending on the application's requirements and the environment they are in. Scheduled non-uniform rejuvenation can transform unplanned downtime into planned, shorter downtime, and it gives the ability to more often rejuvenate those components that are critical or more prone to failure. For example, a recent history of revival restarts and load characteristics can be used to automatically decide how often each component requires rejuvenation. Simpler, coarse-grained solutions have already been proposed by Huang et al. [21] and are used by IBM's xSeries servers [22].

3 Existing Systems

Very few systems today can be classified as being recursively restartable. Many systems do not tolerate restarts at all, and we provide some examples in this section. Others, though not necessarily designed by following an existing set of RR principles, fortuitously exhibit RR-friendly properties. Our long term goal is to derive a canon of design rules, including tradeoffs and programming model semantics, so that future efforts will be more systematic and deliberate.

3.1 Poorly Restartable Systems

In software systems not designed for restartability, the transient failure of one or more components often ends up being treated as a permanent failure. Depending on the system's design, the results can be anywhere from inconvenient to catastrophic. NFS [30] exhibits a flavor of this problem in its implementation of locking: a crash in the lock subsystem can result in an inconsistent lock state between a client and the server, which sometimes requires manual intervention by an administrator to repair. The result is that many applications requiring file locks test whether they are running on top of NFS and, if so, perform their own locking using the local filesystem, thereby defeating the NFS lock daemon's purpose.

As a more serious example, in July 1998, the USS Yorktown battleship lost control of its propulsion system due to a string of events started by a data overflow. Had the overall system been recursively restartable, its components could have been independently restored, avoiding the need to have the entire missile cruiser towed back to port [10].

Many UNIX applications use the `/tmp` directory for temporary files. Should `/tmp` become unavailable (e.g., due to a disk upgrade), programs will typically hang in the I/O system calls. Consequently, these monolithic, tightly coupled applications become crippled and cannot be restarted without losing all the work in progress.

Tightly coupled operating systems belong in this category as well. For example, Windows NT depends on the presence of certain system libraries (DLLs); accidentally deleting one of them can cause the entire system to hang, requiring a full reboot and the loss of all applications' work in progress. In the ideal case, an administrator would be able to replace the DLL and restart the dependent component, allowing the system to continue running. If the failed component was, say, the user interface on a machine running a web server, RR would allow availability of the web service to be unaffected. The ability to treat operating system services as separate components can avoid these failures, as evidenced by true microkernels [1, 24].

3.2 Restartability Winners

The classic replicated Internet server configuration has n instances of a server for a population of u users, with each server being able to handle in excess of u/n users. In such systems, node reboots result simply in a transient $1/n$ throughput loss. Moreover, read-only databases can be striped across these instances such that each node contributes a fixed fraction of DQ (data/query $\times$ queries/unit time) [4]. Independent node reboots or transient node failures result solely in decreased data/query, while keeping overall queries/unit time constant. Such a design makes "rolling rejuvenation" very affordable [3].

At major Internet portals, it is not uncommon for newly hired engineers to write production code for the system after little more than one week on the job. Simplicity is stressed above all else, and code is often written under the explicit assumption that it will necessarily be killed and restarted frequently. This affords programmers such luxuries as never calling `free()` in their C code, thereby avoiding an entire class of pernicious bugs.

Finally, NASA's Mars Pathfinder illustrates the value of coarse-grained reactive restarts. Shortly after landing on Mars, the spacecraft identified that one of its processes failed to complete execution on time, so the control software decided to restart all the hardware and software [28]. Despite the fact that the software was imperfect — it was later found that the hang had been caused by a hard-to-reproduce priority-inversion deadlock — the watchdog timers and restartable control system saved the mission and helped it exceed its intended lifetime by a factor of three.

4 The Restart Scalpel: Toward Structured Recursive Restartability

In proposing RR, we are inspired by the effect of introducing ACID (atomic, consistent, isolated, durable) transactions [16] as a building block many years ago. Not only did transactions greatly simplify the design of data management systems, but they also provided a clean framework within which to reason about the error behavior of such systems. Our goal is for recursive restartability to offer the same class of benefits for systems where ACID semantics are not required or are expensive to engineer, given the system's availability or performance goals. In particular, we address systems in which weaker-than-ACID requirements can be exploited for tradeoffs that improve availability or simplicity of construction.

In this section we make some observations about the properties of RR-friendly systems, and propose guidelines for how RR subsystems can be assembled into more complex RR systems. The overarching theme is that of designing applications as loosely coupled distributed systems, even if they are not distributed in nature.

Accepting *No* for an answer. Software components should be designed such that they can deny service for any request or call. Then, if an underlying component can say *No*, applications must be designed to take *No* for an answer and decide how to proceed: give up, wait and retry, reduce fidelity, etc. Such components can then gracefully tolerate the temporary unavailability of their peer, as evidenced in the cluster-based distributed hash table described by Gribble et al. [19]. Dealing with *No* answers in the callers, as opposed to trying to cope with them in the server, closely follows the end-to-end argument [29]. Moreover, Lampson observes that such error handling is absolutely necessary for a reliable system anyway [23].

> Subsystems should make their interface guarantees sufficiently weak, so they can occasionally restart with no advance warning, yet not cause their callers to hang/crash.

Using reconstructable soft state with announce/listen protocols. Soft state and announce/listen have been extensively used at the network level [37, 9] as well as the application level [12]. Announce/listen makes the default assumption that a component is unavailable unless it says otherwise; soft state can provide information that will carry a system through a transient failure of the authoritative data source for that state. The use of announce/listen with soft state allows restarts and "cold starts" to be treated as one and the same, using the same code path. Moreover, complex recovery code is no longer required, thus reducing the potential for latent bugs and speeding up recovery.

Unfortunately, sometimes soft state systems cannot react quickly enough to deliver service within their specified time frame. Use of soft state implies tolerance of some state inconsistency, and sometimes the state may never stabilize. For example, in a soft-state load balancer for a prototype scalable network server [14], the instability manifested as alternating saturation and idleness of workers. This was due to load balancing decisions based on worker load data that was too old. Mitzenmacher [26] derives a quantitative analytical model to capture the costs and benefits of using such stale information, and his model's predictions coincide with behavior observed in practice. This type of problem can be addressed by increasing refresh frequency, albeit with additional bandwidth and processing overhead.

State shared among subsystems should be mostly soft. The extent of soft state depends on (a) the application's convergence and response-latency requirements and (b) the refresh frequency supported by the inter-component communication substrate (which is a function not only of "raw" bandwidth and latency but also of "goodput").

Automatically trading precision or consistency for availability. Online aggregation [20], harvest/yield tradeoffs [13], and distributed databases such as Bayou [33] are examples of dynamic or adaptive trading of some property, usually either consistency or precision, for availability. Recently, TACT [36] showed how such tradeoffs could be brought to bear on systems employing replication for high availability, by using a framework in which consistency degradation is measured in application-specific units. The ability to make such tradeoffs dynamically and automatically during transient failures makes a system much more amenable to RR.

Inter-component "glue" protocols should allow components to make dynamic decisions on trading consistency/precision for availability, based on both application-specific consistency/precision measures, and a consistency/precision utility function (e.g., "a perfectly consistent answer is twice as good as one missing the last two updates," or "a 100% precise answer is twice as good as a 90% precise answer").

Structuring applications around fine grain workloads. A primary example of fine grain workload requirements comes from HTTP: the Web's architecture has challenged application architects to design mechanisms for state maintenance and session identification, some more elegant than others. The result is that the Web as a whole exhibits the desirable property that individual server processes can be quiesced rapidly, since HTTP connections are typically short-lived, and servers are extremely loosely bound to their clients, given that the protocol itself is stateless. This makes them highly restartable and leads directly to the simple replication and failover techniques found in large cluster-based Internet services.

"Glue" protocols should enforce fine grain interactions between subsytems. They should provide hooks for computing the cost of a subsystem's restart based on the expected duration of its current task and its children's tasks.

Using orthogonal composition axes. Independent subsystems that do not require an understanding of each other's functionality are said to be mutually orthogonal. Compositions of orthogonal subsystems exhibit high tolerance to component restarts, allowing the system as a whole to continue functioning (perhaps with reduced utility) in spite of temporary failures. There is a strong connection between good modular structure and the ability to exploit orthogonal mechanisms; systems that exploit them well seem to go even further: their control flows are completely decoupled, influencing each other only indirectly through explicit message passing. Examples of orthogonal mechanisms include deadlock resolution in databases [15], software-based fault isolation [35], as well as heartbeats and watchdogs used by process peers that monitor each others' liveness [14, 7].

Split functionality along orthogonal axes. Each corresponding subsystem should be centered around an independent locus of control, and interact with other subsystems via events posted using an asynchronous mechanism.

5 Research Agenda and Evaluation

After refining the above design guidelines, evaluation of a RR research agenda will consist of answering at least three major categories of questions:

- What classes of applications are amenable to RR? What model would capture the behavior of these applications and allow them to be compared directly?
- How do we quantify the improvements in availability and the possible losses in performance, consistency or other functionality that may result from the application of RR?
- What software infrastructure and tools are necessary to execute the proposed automatic revival/rejuvenation policy?

5.1 Building RR Systems

Some existing applications, most notably Internet services, are already incorporating a subset of these techniques (usually in an ad hoc fashion) and are primary candidates for systematic RR. Similarly, many geographically dispersed systems can benefit if they tolerate weakened consistency, due to the potential lack of reliability in their communication medium. We suspect the spectrum of applications that are amenable to RR is much wider, but still needs to be explored.

Loosely coupled architectures often exhibit emergent properties that can lead to instability (e.g., noticed in Internet routing [11]) and investigating them is important for RR. There is also a natural tension between the cost of restructuring a system for RR and the cost (in downtime) of restarting it. Fine module granularity improves the system's ability to tolerate partial restarts, but requires the implementation of a larger number of internal, asynchronous interfaces. The paradigm shift required of system developers could make RR too expensive in practice and, when affordable, may lead to buggier software. In some cases RR is simply not feasible, such as for systems with inherent tight coupling (e.g., real-time closed-loop feedback control systems).

Finally, the key to wide adoption of recursive restartability are tools that can aid the software architect in deciding when to use a RR structure and how to apply the RR guidelines.

5.2 Quantifying Availability and the Effects of Recursive Restartability

A major contribution of the transaction concept was the emergence of a model, TP systems, that allowed different implementations of data management systems to be directly compared (e.g., using TPC benchmarks [18]). We are seeking an analogous model that characterizes applications possessing RR properties, and that can serve in quantifying availability.

Availability benchmarking has been of interest only for the past decade [32, 5]. It is considerably more difficult than performance benchmarking, because a fault model is required in addition to a workload, and certain aspects, such as software aging, cannot even be captured reliably. Performance benchmark results that ignore availability measurements, such as "our system obtained 300,000 tpmC", are dishonest — a fast system that is hung or crashed is simply an infinitely slow system. The converse holds for avalability benchmarks as well, so we seek a unified approach to the measurement of RR systems.

Given an application amenable to RR, a model, and a suitable benchmark, we must quantify the improvement in availability and the decrease in functionality (reduced precision, weaker consistency, etc.) when specific RR rules are applied. We expect that work such as TACT [36] and Mitzenmacher's models for usefulness of stale information [26] will provide a starting point for quantitative validation of RR.

We will identify application classes that, compared to their current implementations, are more tolerant of our guidelines (e.g., trading precision for availability). We will restructure the applications incrementally, while maintaining their semantics largely intact. Availability will be evaluated at different stages: (1) initial application; (2) recursively restartable version of the application; (3) RR version using our execution infrastructure (described below), with revival restarts; (4) RR version using the execution infrastructure with both revival and rejuvenation restarts.

5.3 RR Infrastructure Support

Recursively restartable systems rely on a generic execution infrastructure (EI) which is charged with instantiating the restartability tree mentioned in section 2, monitoring each individual component and/or subsystem, and prompting restarts when necessary. In existing restartable systems, the EI homologue is usually application-specific and built into the system itself.

The execution infrastructure relies on a combination of periodic application-specific probes and end-to-end checks (such as verifying the response to a well-known query) to determine whether a component is making progress or not. In most cases, application-specific probes are implemented by the components themselves via callbacks. When the EI detects an anomaly, it advises the faulty component that it should clean up any pending state because it is about to be restarted by its immediate ancestor in the restartability tree. An analogy would be UNIX daemons that understand the "`kill -TERM; sleep 5; kill -9`" idiom. If restarting does not eliminate the anomaly, a restart at a higher level of the hierarchy is attempted, similar to the return up a recursive call structure.

Note how the availability problem itself becomes recursive: we now need a highly available infrastructure that cares for the RR system. Medusa [6], our EI prototype, is functionally much simpler than most applications, making it possible to design and implement it with care. Medusa is built out of simple, highly restartable segments that run on different hosts, use multicast heartbeats to keep track of each other and their activity, and self-reinstantiate to replace dead segments.

6 Conclusion

In this paper we took the view that transient failures will continue plaguing the software infrastructures we depend on, and thus reboots are here to stay. We proposed turning the reboot from a demonic concept into a reliable partner in the fight against system downtime, given that it is a time-tested, effective technique for circumventing Heisenbugs.

We defined recursively restartable (RR) systems as being those systems that tolerate successive restarts at multiple levels. Such systems possess a number of valuable properties that by themselves improve availability. For instance, a RR system's fine granularity permits partial restarts to be used as a form of bounded healing, reducing the overall time-to-repair, and hence increasing availability. On top of these desirable intrinsic properties, we can employ an automated, recursive policy of component revival/rejuvenation to further reduce downtime.

Building RR systems in a systematic way requires a framework consisting of well-understood design rules. A first attempt at formulating such a framework was presented here, advocating the paradigm of building applications as distributed systems, even if they are not distributed in nature. We set forth a research agenda aimed at validating these proposals and verifying that recursive restartability can be an effective supplement to existing high availability mechanisms. With recursive restartability, we hope to add a useful item to every system architect's toolbox.

7 Acknowledgments

We thank Peter Chen, David Cheriton, Jim Gray, Steve Gribble, Butler Lampson, David Lowell, Udi Manber, Dejan Milojicic, Milyn Moy, and Stanford's Mosquitonet and SWIG groups for helpful and stimulating comments on the ideas presented here.

References

[1] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. R. A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, pages 93–113, 1986.

[2] E. Adams. Optimizing preventative service of software products. *IBM J. Res. Dev.*, 28(1):2–14, 1984.

[3] E. Brewer. Personal communication. 2000.

[4] E. Brewer. Lessons from giant-scale services (draft). Submitted for publication, 2001.

[5] A. Brown and D. A. Patterson. Towards availability benchmarks: A case study of software RAID systems. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[6] G. Candea. Medusa: A platform for highly available execution. CS244C (Distributed Systems) course project, Stanford University, `http://stanford.edu/~candea/papers/medusa`, June 2000.

[7] Y. Chawathe and E. A. Brewer. System support for scalable and fault tolerant internet service. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, Lake District, UK, Sep 1998.

[8] T. C. Chou. Beyond fault tolerance. *IEEE Computer*, 30(4):31–36, 1997.

[9] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei, P. Sharma, and A. Helmy. Protocol independent multicast (PIM), sparse mode protocol: Specification, March 1996. Internet Draft.

[10] A. DiGiorgio. The smart ship is not enough. *Naval Institute Proceedings*, 124(6), June 1998.

[11] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Transactions on Networking*, 2(2):122–136, Apr. 1994.

[12] S. Floyd, V. Jacobson, C. Liu, and S. McCanne. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. In *ACM SIGCOMM '95*, pages 342–356, Boston, MA, Aug 1995.

[13] A. Fox and E. A. Brewer. ACID confronts its discontents: Harvest, yield, and scalable tolerant systems. In *Seventh Workshop on Hot Topics In Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[14] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16)*, St.-Malo, France, October 1997.

[15] J. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, J. H. Saltzer, and G. Seegmüller, editors, *Operating Systems, An Advanced Course*, volume 60, pages 393–481. Springer, 1978.

[16] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of VLDB*, Cannes, France, September 1981.

[17] J. Gray. Why do computers stop and what can be done about it? In *Proc. Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[18] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufman, 2 edition, 1993.

[19] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, October 2000.

[20] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *ACM–SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

[21] Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *International Symposium on Fault-Tolerant Computing*, pages 381–390, 1995.

[22] International Business Machines. IBM director software rejuvenation. White paper, Jan. 2001.

[23] B. W. Lampson. Hints for computer systems design. *ACM Operating Systems Review*, 15(5):33–48, 1983.

[24] J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[25] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing relevance of memory hardware errors. a case for recoverable programming models. In *ACM SIGOPS European Workshop "Beyond the PC: New Challenges for the Operating System"*, Kolding, Denmark, Sept. 2000.

[26] M. Mitzenmacher. How useful is old information? In *Principles of Distributed Computing (PODC) 97*, pages 83–91, 1997.

[27] B. Murphy and N. Davies. System reliability and availability drivers of Tru64 UNIX. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, Madison, WI, February 1999. IEEE Computer Society. Tutorial.

[28] G. Reeves. What really happened on Mars? RISKS-19.49, Jan. 1998.

[29] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[30] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the USENIX Summer Conference*, pages 119–130, Portland, OR, 1985.

[31] D. Scott. Making smart investments to reduce unplanned downtime. Tactical Guidelines Research Note TG-07-4033, Gartner Group, Stamford, CT, 1999.

[32] D. P. Siewiorek, J. J. Hudak, B.-H. Suh, and Z. Segall. Development of a benchmark to measure system robustness. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 88–97, 1993.

[33] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, TX, Sept. 1994.

[34] U.S. General Accounting Office. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Technical Report GAO/IMTEC-92-26, 1992.

[35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP-14)*, 1993.

[36] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, Oct. 2000.

[37] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource Reservation Protocol. *IEEE Network*, 7(5), Sept. 1993.

When Virtual Is Better Than Real

Peter M. Chen and Brian D. Noble
Department of Electrical Engineering and Computer Science
University of Michigan
pmchen@umich.edu, bnoble@umich.edu

Abstract

This position paper argues that the operating system and applications currently running on a real machine should relocate into a virtual machine. This structure enables services to be added below the operating system and to do so without trusting or modifying the operating system or applications. To demonstrate the usefulness of this structure, we describe three services that take advantage of it: secure logging, intrusion prevention and detection, and environment migration.

1. Introduction

First proposed and used in the 1960s, virtual machines are experiencing a revival in the commercial and research communities. Recent commercial products such as VMware and VirtualPC faithfully emulate complete x86-based computers. These products are widely used (e.g. VMware has more than 500,000 registered users) for purposes such as running Windows applications on Linux and testing software compatibility on different operating systems. At least two recent research projects also use virtual machines: Disco uses virtual machines to run multiple commodity operating systems on large-scale multiprocessors [4]; Hypervisor uses virtual machines to replicate the execution of one computer onto a backup [3].

Our position is that the operating system and applications that currently run directly on real machines should relocate into a virtual machine running on a real machine (Figure 1). The only programs that run directly on the real machine would be the host operating system, the virtual machine monitor, programs that provide local administration, and additional services enabled by this virtual-machine-centric structure. Most network services would run in the virtual machine; the real machine would merely forward network packets for the virtual machine.

This virtual-machine-centric model allows us to provide services *below* most code running on the computer, similar to providing services in the hardware of a real machine. Because these services are implemented in a layer of software (the virtual machine monitor or the host operating system), they can be provided more easily and flexibly than they could if they were implemented by modifying the hardware. In particular, we can provide services below the guest operating system without trusting or modifying it. We believe providing services at this layer is especially useful for enhancing security and mobility.

This position paper describes the general benefits and challenges that arise from running most applications in a virtual machine, then describes some example services and alternative ways to provide those services.

2. Benefits

Providing services by modifying a virtual machine has similar benefits to providing services by modifying a real machine. These services run separately from all processes in the virtual machine, including the guest operating system. This separation benefits security and portability. Security is enhanced because the services do not have to trust the guest operating system; they have only to trust the virtual machine monitor, which is considerably smaller and simpler. Trusting the virtual machine monitor is akin to trusting a real processor; both expose a narrow interface (the instruction set architecture). In contrast, services in an operating system are more vulnerable to malicious and random faults, because operating systems are larger and more prone to security and reliability holes. Separating the services from the guest operating system also enhances portability. We can implement the services without needing to change the operating system, so they can work across multiple operating system vendors and versions.

While providing services in a virtual machine gains similar benefits to providing services in a real machine, virtual machines have some advantages over the physical machines they emulate. First, a virtual machine can be modified more easily than a physical machine, because the virtual machine monitor that creates the virtual machine

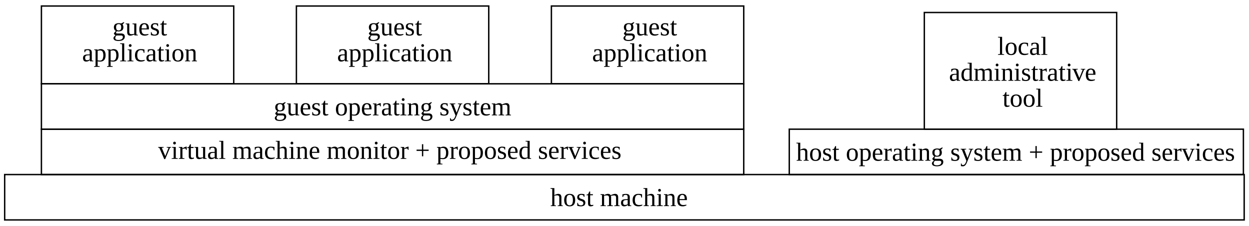

guest application
guest application
guest application
local administrative tool
guest operating system
virtual machine monitor + proposed services
host operating system + proposed services
host machine

Figure 1: Virtual-machine structure. In this model, most applications that currently run on real machines relocate into a virtual machine running on the host machine. The virtual machine monitor and local administrative programs run directly on the host machine. In VMware, the virtual machine monitor issues I/O through the host operating system, so services that manipulate I/O events can be implemented in the host operating system [2].

abstraction is a layer of software. Second, it is much easier to manipulate the state of a virtual machine than the state of a physical machine. The state of the virtual machine can be saved, cloned, encrypted, moved, or restored, none of which is easy to do with physical machines. Third, a virtual machine has a very fast connection to another computing system, that is, the host machine on which the virtual machine monitor is running. In contrast, physical machines are separated by physical networks, which are slower than the memory bus that connects a virtual machine with its host.

3. Challenges

Providing services at the virtual-machine level holds two challenges. The first is performance. Running all applications above the virtual machine hurts performance due to virtualization overhead. For example, system calls in a virtual machine must be trapped by the virtual machine monitor and re-directed to the guest operating system. Hardware operations issued by the guest must be trapped by the virtual machine monitor, translated, and reissued. Some overhead is unavoidable in a virtual machine; the services enabled by that machine must outweigh this performance cost. Virtualizing an x86-based machine incurs additional overheads because x86 processors don't trap on some instructions that must be virtualized (e.g. reads of certain system registers). One way to implement a virtual machine in the presence of these "non-virtualizable" instructions is to re-write the binaries at run time to force these instructions to trap [13], but this incurs significant overhead.

The second challenge of virtual-machine services is the semantic gap between the virtual machine and the service. Services in the virtual machine operate below the abstractions provided by the guest operating system and applications. This can make it difficult to provide services. For example, it is difficult to provide a service that checks file system integrity without knowledge of on-disk structures. Some services do not need any operating system abstractions; secure logging (Section 4.1) is an example of such a service. For services that require higher-level information, one must re-create this information in some form. Full semantic information requires re-implementing guest OS abstractions in or below the virtual machine. However, there are several abstractions—virtual address spaces, threads of control, network protocols, and file system formats—that are shared across many operating systems. By observing manipulations of virtualized hardware, one can reconstruct these *generic* abstractions, enabling services that require semantic information.

4. Example services

In this section, we describe three services that can be provided at the virtual-machine level. Others have used virtual machines for many other purposes, such as preventing one server from monopolizing machine resources, education, easing the development of privileged software, and software development for different operating systems [10].

4.1. Secure logging

Most operating systems log interesting events as part of their security strategy. For example, a system might keep a record of login attempts and received/sent mail. System administrators use the logged information for a variety of purposes. For example, the log may help administrators understand how a network intruder gained access to the system, or it may help administrators know what damage the intruder inflicted after he gained access. Unfortunately, the logging used in current systems has two important shortcomings: integrity and completeness. First, an attacker can easily turn off logging after he takes over the system; thus the contents of the log cannot be trusted after the point of compromise. Second, it is difficult to anticipate what information may be needed during the post-attack analysis; thus the log may lack information needed to discern how the intruder gained access or what actions he took after gaining access.

Virtual machines provide an opportunity to correct both shortcomings of current logging. To improve the integrity of logging, we can move the logging software out of the operating system and into the virtual machine monitor. The virtual machine monitor is much smaller and simpler than the guest operating system and hence is less vulnerable to attack. By moving the logging software into the virtual machine monitor, we move it out of the domain that an intruder can control. Even if the intruder gains root access or completely replaces the guest operating system, he cannot affect the logging software or the logged data. Logged data can be written quickly to the host file system, taking advantage of the fast connection between the virtual machine monitor and the host computer.

To improve the completeness of logging, we propose logging enough data to replay the complete execution of the virtual machine [3]. The information needed to accomplish a faithful replay is limited to a checkpoint with which to initialize the replaying virtual machine, plus the non-deterministic events that affected the original execution of the virtual machine since the time of the saved checkpoint. These non-deterministic events fall into two categories: external input and time. External input refers to data sent by a non-logged entity, such as a human user or an external computer (e.g. a web server). Time refers to the exact point in the execution stream at which an event takes place. For example, to replay the interleaving pattern between threads, we must log which instruction is preempted by a timer interrupt [17] (we assume the virtual machine monitor is not running on a multi-processor). Note that most instructions executed by the virtual machine do not need to be logged; only the relatively infrequent non-deterministic events need to be logged.

Using the virtual machine monitor to perform secure logging raises a number of research questions. The first question regards the volume of log data needed to support replay. We believe that the volume of data that needs to be logged will not be prohibitive. Local non-deterministic events, such as thread scheduling events and user inputs, are all small. Data from disk reads can be large, but these are deterministic (though the time of the disk interrupts are non-deterministic). The largest producer of log data is likely to be incoming network packets. We can reduce the volume of logged network data greatly by using message-logging techniques developed in the fault-tolerance community. For example, there is no need to log message data received from computers that are themselves being logged, because these computers can be replayed to reproduce the sent message data [11]. If all computers on the same local network cooperate during logging and replay, then only messages received from external sites need to be logged. For an important class of servers (e.g. web servers), the volume of data received in messages is relatively small (HTTP GET and POST requests). Last, as disk prices continue to plummet, more computers (especially servers worthy of being logged) will be able to devote many gigabytes to store log data [20].

A second research direction is designing tools to analyze the behavior of a virtual machine during replay. Writing useful analysis tools in this domain is challenging because of the semantic gap between virtual machine events and the corresponding operating system actions. The analysis tool may have to duplicate some operating system functionality to distill the log into useful information. For example, the analysis tool may need to understand the on-disk file system format to translate the disk transfers seen by the virtual machine monitor into file-system transfers issued by the operating system. Translating virtual machine events into operating system events becomes especially challenging (and perhaps impossible) if the intruder modifies the operating system. One family of analysis tools we hope to develop trace the flow of information in the system, so that administrators can ask questions like “What network connections caused the password file to change?”.

4.2. Intrusion prevention and detection

Another important component to a security strategy is detecting and thwarting intruders. Ideally, these systems *prevent* intrusions by identifying intruders as they attack the system [9]. These systems also try to *detect* intrusions after the fact by monitoring the events and state of the computer for signs that a computer has been compromised [8, 12]. Virtual machines offer the potential for improving both intrusion prevention and intrusion detection.

Intrusion preventers work by monitoring events that enter or occur on the system, such as incoming network packets. Signature-based preventers match these input events against a database of known attacks; anomaly-based preventers look for input events that differ from the norm. Both these types of intrusion preventers have flaws, however. Signature-based systems can only thwart attacks that have occurred in the past, been analyzed, and been integrated into the attack database. Anomaly-based systems can raise too many false alarms and may be susceptible to re-training attacks.

A more trustworthy method of recognizing an attack is to simply run the input event on the real system and seeing how the system responds. Of course, running suspicious events on the real system risks compromising the system.

However, we can safely conduct this type of test on a *clone* of the real system. Virtual machines make it easy to clone a running system, and an intrusion preventer can use this clone to test how a suspicious input event would affect the real system. The clone can be run as a hot standby by keeping it synchronized with the real system (using primary-backup techniques), or it can be created on the fly in response to suspicious events. In either case, clones admit more powerful intrusion preventers by looking at the response of the system to the input event rather than looking only at the input event. Because clones are isolated from the real system, they also allow an intrusion preventer to run potentially destructive tests to verify the system's health. For example, an intrusion preventer could forward a suspicious packet to a clone and see if it crashes any running processes. Or it could process suspicious input on the clone, then see if the clone still responds to shutdown commands.

A potential obstacle to using clone-based intrusion prevention is the effect of clone creation or maintenance on the processing of innocent events. To avoid blocking the processing of innocent events, an intrusion preventer would ideally run the clone in the background. Allowing innocent events to go forward while evaluating suspicious events implies that these events have loose ordering constraints. For example, a clone-based preventer could be used to test e-mail messages for viruses, because ordering constraints between e-mail messages are very loose.

Intrusion detectors try to detect the actions of intruders after they have compromised a system. Signs of an intruder might include bursts of outgoing network packets (perhaps indicating a compromised computer launching a denial-of-service attack), modified system files [12], or abnormal system-call patterns from utility programs [8]. As with system logging, these intrusion detectors fall short in integrity or completeness. Host-based intrusion detectors (such as those that monitor system calls) may be turned off by intruders after they compromise the system, so they are primarily useful only for detecting the act of an intruder breaking into a system. If an intruder evades detection at the time of entry, he can often disarm a host-based intrusion detector to avoid detection in the future. Network-based intrusion detectors can provide better integrity by being separate from the host operating system (e.g. in a standalone network router), but they suffer from a lack of completeness. Network intrusion detectors can see only network packets; they cannot see the myriad other events occurring in a computer system, such as disk traffic, keyboard events, memory usage, and CPU usage.

Implementing post-intrusion detection at the level of a virtual machine offers the potential for providing both integrity and completeness. Like a network-based intrusion detector, virtual-machine-based intrusion detectors are separate from the guest operating system and applications. Unlike network intrusion detectors, however, virtual-machine intrusion detectors can see all events occurring in the virtual machine they monitor. Virtual-machine intrusion detectors can use this additional information to implement new detection policies. For example, it could detect if the virtual machine reads certain disk blocks (e.g. containing passwords), then issues a burst of CPU activity (e.g. cracking the passwords). Or it could detect if the virtual machine has intense CPU activity with no corresponding keyboard activity.

As with secure logging, a key challenge in post-intrusion detection in a virtual machine is how to bridge the semantic gap between virtual machine events and operating system events. This challenge is similar to that encountered by network-based intrusion detectors, which must parse the contents of IP packets.

4.3. Environment migration

Process migration has been a topic of interest from the early days of distributed computing. Migration allows one to package a running computation—either a process or collection of processes—and move it to a different physical machine. Using migration, a user's computations can move as he does, taking advantage of hardware that is more convenient to the user's current location.

The earliest systems, including Butler [15], Condor [14], and Sprite [6], focused on load sharing across machines rather than supporting mobile users. These load-sharing systems typically left residual dependencies on the source machine for transparency, and considered an individual process as the unit of migration. This view differs from that of mobile users, who consider the unit of migration to be the collection of all applications running on their current machine.

Recently, migration systems have begun to address the needs of mobile users. Examples of systems supporting mobility include the Teleporting system [16] and SLIM [18]. These systems migrate the user interface of a machine, leaving the entire set of applications to run on their host machine. In the limit, the display device can be a stateless, thin client. This approach provides a better match to the expectations of a migrating user, and need not deal with residual dependencies. However, these systems are intolerant of even moderate latency between the interface device and the cycle server, and thus support only a limited form of user mobility.

Migration based on virtual machines solves these problems. Since the entire (virtual) machine moves, there are no residual dependencies. A user's environment is moved en masse, which matches a user's expectations. By taking advantage of the narrow interface provided by the virtual machine, very simple migration code can relocate a guest operating system and its applications.

There are several challenges that must be overcome to provide migration at the virtual-machine level. The first is that a machine has substantial state that must move with it. It would be infeasible to move this state synchronously on migration. Fortunately, most of this state is not needed immediately, and much may never be needed at all. We can predict which state is needed soon by taking advantage of temporal locality in disk and memory accesses. This prediction is complicated by the guest operating system's virtual memory abstraction, because the physical addresses seen by a virtual machine monitor are related only indirectly to accesses issued by applications. We can reconstruct information about virtual to physical mappings by observing manipulation of virtualized hardware elements such as the TLB.

After identifying the state likely to be needed soon, we need a mechanism to support migration of that state to the new virtual machine. If migration times are exposed, one can take advantage of efficient, wide-area consistency control schemes, such as that provided by Fluid Replication [5]. Fluid Replication provides safety, visibility, and performance across the wide area identical to that offered by local-area file systems such as NFS. It depends on typical file system access patterns, in particular a low incidence of concurrent data sharing. Machine migration, with coarse-grained, sequential sharing, fits this pattern well, allowing for migration without undue performance penalty.

To provide the most benefit, we must also support migration between physical machines that are not entirely identical. This is difficult because most virtual machine monitors improve performance by accessing some hardware components directly (e.g. the video frame buffer). This direct access complicates matters for the guest operating system when migrating between machines with different components. There are two approaches to solving this kind of problem. The first is to further virtualize the component, at a performance cost. The second is to modify the guest operating system to adapt to the new component on the fly. The right alternative depends on the resource in question, the performance penalty of virtualization, and the complexity of dynamic adaptation.

Migration is only one of several services that leverage the easy packaging, storage, and shipment of virtual machines. Clone-based intrusion detection is one example. One can also extend services that apply to individual resources across an entire virtual machine. For example, cryptographic file systems protect only file data; once an application reads sensitive data, it cannot be made secure. However, suspending a virtual machine to disk when its user is away provides process-level protection using only the virtual machine services plus file system mechanisms.

5. Alternative structures

Each of the above services can be implemented in other ways. One alternative is to include these services in the operating system. This structure makes it easier for the service to access information in terms of operating system abstractions. For example, an intrusion detector at the operating system level may be able to detect when one user modifies files owned by another user. A virtual machine service, in contrast, operates below the notions of users and files and would have to reconstruct these abstractions. In addition, including these services in the operating system reduces the number of layers and redirections, which will likely improve performance relative to a virtual machine.

However, including services in the operating system has some disadvantages. First, such services are limited to a single operating system (and perhaps a single operating system version), whereas virtual-machine services can support multiple operating systems. For example, a secure logging service in a virtual machine can replay any operating system. Second, for security services such as secure logging and intrusion detection, including the service in the operating system depends critically on the integrity of the operating system. Because operating systems are typically large, complex, and monolithic, they usually contain security and reliability vulnerabilities. For example, the Linux 2.2.16 kernel contained at least 7 security holes [1]. In particular, secure logging is challenging to provide in the operating system, because an intruder may try to crash the system to prevent the log tail from being written to stable storage.

Some of the disadvantages of including services in the operating system can be mitigated by re-structuring the operating system into multiple protection domains [19] and placing security-related services in the most-privileged ring. This approach is similar to kernels that include only the minimum set of services [7]. However, this approach requires re-writing the entire operating system, and frequent crossings between multiple protection domains degrade performance.

A different approach is to add services to a language-specific virtual machine such as Java. Language-specific virtual machines potentially have more information than the operating system, which may be helpful for some services. However, these services would be available only for applications written in the target language. For the system-wide services described above, the entire system would have be written in the target language.

6. Conclusions

Running an operating system and most applications inside a virtual machine enables a system designer to add services below the guest operating system. This structure enables services to be provided without trusting or modifying the guest operating system or the applications. We have described three services that take advantage of this structure: secure logging, intrusion prevention and detection, and environment migration.

Adding services via a virtual machine is analogous to adding network services via a firewall. Both virtual machines and firewalls intercept actions at a universal, low-level interface, and both must overcome performance and semantic-gap problems. Just as network firewalls have proven useful for adding network services, we believe virtual machines will prove useful for adding services for the entire computer.

7. References

[1] Linux Kernel Version 2.2.16 Security Fixes, 2000. http://www.linuxsecurity.com/advisories/slackware_advisory-481.html.

[2] VMware Virtual Machine Technology. Technical report, VMware, Inc., September 2000.

[3] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-Based Fault-Tolerance. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 1–11, December 1995.

[4] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[5] Landon P. Cox and Brian D. Noble. Fluid Replication. In *Proceedings of the 2001 International Conference on Distributed Computing Systems*, April 2001.

[6] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software Practice and Experience*, 21(7), July 1991.

[7] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 1995 Symposium on Operating Systems Principles*, pages 251–266, December 1995.

[8] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedings of 1996 IEEE Symposium on Computer Security and Privacy*, 1996.

[9] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 1996 USENIX Technical Conference*, July 1996.

[10] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–45, June 1974.

[11] David B. Johnson and Willy Zwaenepoel. Sender-Based Message Logging. In *Proceedings of the 1987 International Symposium on Fault-Tolerant Computing*, pages 14–19, July 1987.

[12] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity checker. In *Proceedings of 1994 ACM Conference on Computer and Communications Security*, November 1994.

[13] Kevin Lawton. Running multiple operating systems concurrently on an IA32 PC using virtualization techniques, 1999. http://plex86.org/research/paper.txt.

[14] M. J. Litzkow. Remote UNIX: turning idle workstations into cycle servers. In *Proceedings of the Summer 1987 USENIX Technical Conference*, pages 381–384, June 1987.

[15] D. A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the 1987 Symposium on Operating System Principles*, pages 5–12, November 1987.

[16] T. Richardson, F. Bennet, G. Mapp, and A. Hopper. Teleporting in an X window system environment. *IEEE Personal Communications*, 1(3):6–12, 1994.

[17] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the 1996 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–266, May 1996.

[18] Brian K. Schmidt, Monica S. Lam, and J. Duane Northcutt. The interactive performance of SLIM: a stateless, thin-client architecture. In *Proceedings of the 1999 Symposium on Operating Systems Principles*, pages 32–47, December 1999.

[19] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, March 1972.

[20] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

Virtualization Considered Harmful: OS Design Directions for Well-Conditioned Services

Matt Welsh and David Culler

Computer Science Division
University of California, Berkeley
{mdw,culler}@cs.berkeley.edu

Abstract

We argue that existing OS designs are ill-suited for the needs of Internet service applications. These applications demand massive concurrency (supporting a large number of requests per second) and must be well-conditioned to load (avoiding degradation of performance and predictability when demand exceeds capacity). The transparency and virtualization provided by existing operating systems leads to limited concurrency and lack of control over resource usage. We claim that Internet services would be far better supported by operating systems by reconsidering the role of resource virtualization. We propose a new design for server applications, the staged event-driven architecture *(SEDA). In SEDA, applications are constructed as a set of event-driven* stages *separated by* queues*. We present the SEDA architecture and its consequences for operating system design.*

1. Introduction

The design of existing operating systems is primarily derived from a heritage of multiprogramming: allowing multiple applications, each with distinct resource demands, to safely and efficiently share a single set of resources. As such, existing OSs strive to virtualize hardware resources, and do so in a way which is transparent to applications. Applications are rarely, if ever, given the opportunity to participate in system-wide resource management decisions, or given indication of resource availability in order to adapt their behavior to changing conditions. Virtualization fundamentally hides the fact that resources are limited and shared.

Internet services are a relatively new application domain which presents unique challenges for OS design. In contrast to the batch-processing and interactive workloads for which existing operating systems have been designed, Internet services support a large number of concurrent operations and exhibit enormous variations in load. The number of concurrent sessions and hits per day to Internet sites translates into an even higher number of I/O and network requests, placing great demands on underlying resources. Microsoft's web sites receive over 300 million hits with 4.1 million users a day; Yahoo has over 900 million page views daily. The peak load experienced by a service may be many times that of the average, and services must deal gracefully with unexpected increases in demand.

A number of systems have attempted to remedy the problems with OS virtualization by exposing more control to applications. Scheduler activations [1], application-specific handlers [29], and operating systems such as SPIN [3], Exokernel [12], and Nemesis [17] are attempts to augment limited operating system interfaces by giving applications the ability to specialize the policy decisions made by the kernel. However, the design of these systems is still based on the multiprogramming mindset, as the focus continues to be on safe and efficient resource virtualization.

We argue that the design of most existing operating systems fails to address the needs of Internet services. Our key premise is that supporting concurrency for a few tens of users is fundamentally different than for many thousands of service requests. This paper proposes a new architecture for services, which we call the *staged event-driven architecture* (SEDA). SEDA departs from the traditional multiprogramming approach provided by existing OSs, decomposing applications into a set of *stages* connected by explicit *event queues*. This design avoids the high overhead associated with thread-based concurrency, and allows applications to be well-conditioned to load by making informed decisions based on the inspection of pending requests. To mitigate the effects of resource virtualization, SEDA employs a set of *dynamic controllers* which manage the resource allocation and scheduling of applications.

In this paper, we discuss the shortcomings of existing OS designs for Internet services, and present the SEDA architecture, arguing that it is the right way to construct these applications. In addition, we present a set of OS design directions for Internet services. We argue that server operating systems should eliminate the abstraction of transparent resource virtualization, a shift which enables support for high concurrency, fine-grained scheduling, scalable I/O, and application-controlled resource management.

2. Why Internet Services and Existing OS Designs Don't Match

This section highlights four main reasons that existing OS designs fail to mesh well with the needs of Internet services: inefficient concurrency mechanisms, lack of scalable I/O interfaces, transparent resource management, and coarse-grained control over scheduling.

2.1. Existing OS Design Issues

Concurrency limitations: Internet services must efficiently multiplex many computational and I/O flows over a limited set of resources. Given the extreme degree of concurrency required, services are often willing to sacrifice transparent virtualization in order to obtain higher performance. However, contemporary operating systems typically support concurrency using the process or thread model: each process/thread embodies a virtual machine with its own CPU, memory, disk, and network, and the O/S multiplexes these virtual machines over hardware. Providing this abstraction entails a high overhead in terms of context switch time and memory footprint, thereby limiting concurrency. A number of studies have shown the scalability limitations of thread-based concurrency models [6, 11, 21, 32], even in the context of so-called "lightweight" threads.

I/O Scalability limitations: The I/O interfaces exported by existing OSs are generally designed to provide maximum transparency to applications, often at the cost of scalability and predictability. Most I/O interfaces employ blocking semantics, in which the calling thread is suspended during a pending I/O operation. Obtaining high concurrency requires a large number of threads, resulting in high overhead. Traditional I/O interfaces also tend to degrade in performance as the number of simultaneous I/O flows increases [2, 23]. In addition, data copies on the I/O path (themselves an artifact of virtualization) have long been known to be a performance limitation in network stacks [24, 27, 28].

Transparent resource management: Internet services must be in control of resource usage in order to make informed decisions affecting performance. Virtualization implies that the OS will attempt to satisfy any application request regardless of cost (e.g., a request to allocate a page of virtual memory which requires other pages to be swapped out to disk). However, services do not have the luxury of paying an arbitrary penalty for processing such requests under heavy resource contention. Most operating systems hide the performance aspects of their interfaces; for instance, the existence of (or control over) the underlying file system buffer cache is typically not exposed to applications. Stonebraker [26] cites this aspect of OS design as a problem for database implementations as well.

Coarse-grained scheduling: The thread-based concurrency model yields a coarse degree of control over resource management and scheduling decisions. While it is possible to control the prioritization or runnable status of an individual thread, this is often too blunt of a tool to implement effective load conditioning policies. Instead, it is desirable to control the flow of requests through a particular resource.

As an example consider the page cache for a Web server. To maximize throughput and minimize latency, the server might prioritize requests for cache hits over cache misses; this is a decision which is being made at the level of the cache by inspecting the stream of pending requests. Such a policy would be difficult (although not impossible) to implement by changing the scheduling parameters for a pool of threads each representing a different request in the server pipeline. The problem is that this model only provides control over scheduling of individual threads, rather than over the ordering of requests for a particular resource.

2.2. Traditional Event-Driven Programming

The limitations of existing OS designs have led many developers to favor an event-driven programming approach, in which each concurrent request in the system is modeled as a finite state machine. A single thread (or small number of threads) is responsible for scheduling each state machine based on events originating from the OS or within the application itself, such as I/O readiness and completion notifications.

Event-driven systems are generally built from scratch for particular applications, and depend on mechanisms not well-supported by most operating systems. Because the underlying OS is structured to provide thread-based concurrency using blocking I/O, event-driven applications are at a disadvantage to obtain the desired behavior over this imperfect interface. Consequently, obtaining high performance requires that the application designer carefully manage event and thread scheduling, memory allocation, and I/O streams [4, 9, 10, 21]. This "monolithic" event-driven design is also difficult to modularize, as the code implementing each state is directly linked with others in the flow of execution.

Nonblocking I/O is provided by most OSs, but these interfaces typically do not scale well as the number of I/O flows grows very large [2, 14, 18]. Much prior work has investigated scalable I/O primitives for servers [2, 5, 13, 16, 22, 23, 25], but these solutions are often an afterthought lashed onto a process-based model, and do not always perform well. To demonstrate this fact, we have measured the performance of the nonblocking socket interface in Linux using the `/dev/poll` [23] event-delivery mechanism, which is known to scale better than the standard UNIX *select()* and *poll()* interfaces [14]. As Figure 1 shows, the performance of the nonblocking socket layer degrades when a large number of connections are established; despite the use of an efficient event-delivery mechanism, the underlying network stack does not scale as the number of connections grows large.

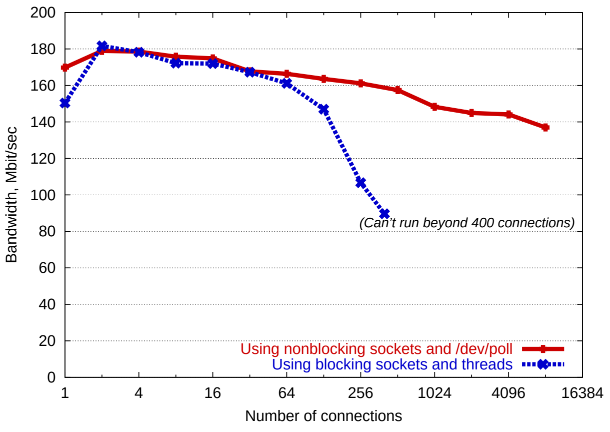


Figure 1: **Linux socket layer performance:** *This graph shows the aggregate bandwidth through a server making use of either asynchronous or blocking socket interfaces. Each client opens a connection to the server and issues bursts of 1000 8 KB packets; the server responds with a single 32-byte ACK for each burst. All machines are 4-way Pentium III systems running Linux 2.2.14 interconnected by Gigabit Ethernet. Two implementations of the server are shown: one makes use of nonblocking sockets along with the* `/dev/poll` *mechanism for event delivery, and the other emulates asynchronous behavior over blocking sockets by using threads. The latter implementation allocates one thread per socket for reading packets, and uses a fixed-size thread pool of 120 threads for writing packets. The threaded implementation could not support more than 400 simultaneous connections due to thread limitations under Linux, while the nonblocking implementation degrades somewhat due to lack of scalability in the network stack.*

3. The Staged Event-Driven Architecture

In this section we propose a structured approach to event-driven programming that addresses some of the challenges of implementing Internet services over commodity operating systems. This approach, the *staged event-driven architecture* (SEDA) [30], is designed to manage the high concurrency and load conditioning demands of these applications.

3.1. SEDA Design

As discussed in the previous section, the use of event-driven programming can be used to overcome some (but not all) of the shortcomings of conventional OS interfaces. SEDA refines the monolithic event-driven approach by structuring applications in a way which enables load conditioning, increases code modularity, and facilitates debugging.

SEDA makes use of a set of design patterns, first described in [32], which break the control flow of an event-driven system into a series of *stages* separated by *queues*. Each task in the system is processed by a sequence of stages each representing some set of states in the traditional event-driven design. SEDA relies upon asynchronous I/O primitives that expose I/O completion and readiness events directly to applications by placing those events onto the queue for the appropriate stage.

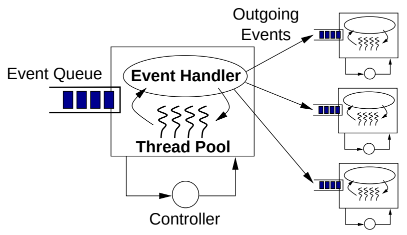


Figure 2: **A SEDA Stage:** *A stage consists of an* incoming event queue*, a* thread pool*, and an application-supplied* event handler*. The stage's operation is managed by the* controller*, which adjusts resource allocations and scheduling.*

A stage is a self-contained application component consisting of an *event handler*, an *incoming event queue*, and a *thread pool*, as shown in Figure 2. Each stage is managed by a *controller* which affects scheduling and resource allocation. Threads operate by pulling events off of the stage's incoming event queue and invoking the application-supplied event handler. The event handler processes each task, and dispatches zero or more tasks by enqueuing them on the event queues of other stages. Figure 3 depicts a simple HTTP server implementation using the SEDA design.

Event handlers do not have direct control over queue operations and threads. By separating core application logic from thread management and scheduling, the stage's controller is able to manage the execution of the event handler to implement various resource-management policies. For example, the number of threads in the stage's thread pool is adjusted dynamically by the controller, based on an observation of the event queue and thread behavior. Details are beyond the scope of this paper; more information is provided in [30].

3.2. SEDA Benefits

The SEDA design yields a number of benefits which directly address the needs of Internet services:

High concurrency: As with the traditional event-driven design, SEDA makes use of a small number of threads to process stages, avoiding the performance overhead of using a large number of threads for managing concurrency. The use of asynchronous I/O facilitates high concurrency by eliminating the need for multiple threads to overlap pending I/O requests.

In SEDA, the number of threads can be chosen at a per-stage level, rather than for the application as a whole; this approach avoids wasting threads on stages which do not need them. For example, UNIX filesystems can usually handle a fixed number (between 40 and 50) concurrent read/write requests before becoming saturated [6]. In this case there is no benefit to devoting more than this number

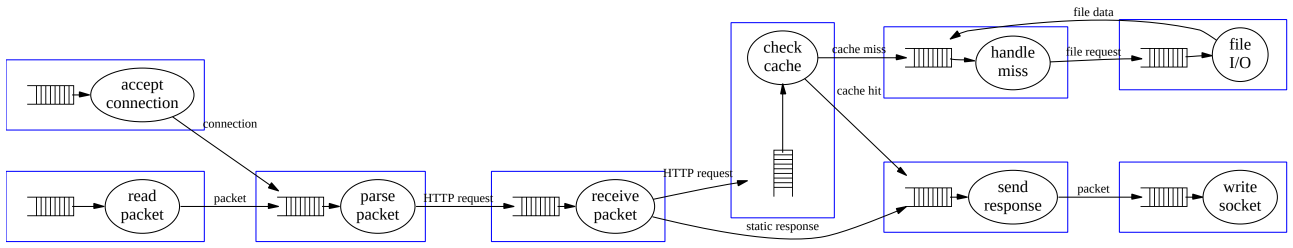


Figure 3: **Staged event-driven (SEDA) HTTP server:** *The application is decomposed into a set of* stages *separated by* queues*. Edges represent the flow of events between stages. Each stage can be independently managed, and stages can be run in sequence or in parallel, or a combination of the two. The use of event queues allows each stage to be individually load-conditioned, for example, by thresholding its event queue.*

of threads to a stage which performs filesystem access. To shield the application programmer from the complexity of managing thread pools, the stage's controller is responsible for determining the number of threads executing within each stage.

Application-specific load conditioning: The use of explicit event queues allows applications to implement load conditioning policies based on the observation of pending events. Backpressure can be implemented by having a queue reject new entries (e.g., by raising an error condition) when it becomes full. This is important as it allows excess load to be rejected by the system, rather than buffering an arbitrary amount of work. Alternately, a stage can drop, filter, or reorder incoming events in its queue to implement other policies, such as event prioritization. During overload, a stage may prioritize requests requiring few resources over those which involve expensive computation or I/O. These policies can be tailored to the specific application, rather than imposed by the system in a generic way.

Code modularity and debugging support: The SEDA design allows stages to be developed and maintained independently. A SEDA-based application consists of a network of interconnected stages; each stage can be implemented as a separate code module in isolation from other stages. The operation of two stages is composed by inserting a queue between them, thereby allowing events to pass from one to the other. This is in contrast to the "monolithic" event-driven design, in which the states of the request-processing state machine are often highly interdependent.

Few tools exist for understanding and debugging a complex event-driven system, as stack traces do not represent the control flow for the processing of a particular request. SEDA facilitates debugging and performance analysis, as the decomposition of application code into stages and explicit event delivery mechanisms provide a means for direct inspection of application behavior. For example, a debugging tool can trace the flow of events through the system and visualize the interactions between stages. Our prototype of SEDA is capable of generating a graph depicting the set of application stages and their relationship.

4. Operating System Design Directions

While SEDA aids the construction of highly-concurrent applications over conventional OS interfaces, these interfaces still present a number of design challenges for Internet services. In particular, we argue that the goal of transparent resource virtualization is undesirable in this context, and that server operating systems should eliminate this abstraction in favor of an approach which gives applications more control over resource usage. This fundamental shift in ideology makes it possible to implement a number of features which support Internet services:

Concurrency and scheduling: Because SEDA uses a small number of threads for driving the execution of stages, much of the scalability limitation of threads is avoided. Ideally, the code for each stage should never block, requiring just one thread per CPU. However, for this approach to be feasible every OS interface must be nonblocking. This is unproblematic for I/O, but may be more challenging for other interfaces, such as demand paging or memory synchronization. The goal of a SEDA-oriented operating system is not to eliminate threads altogether, but rather to support interfaces which allows their use to be minimized.

A SEDA-based OS should allow applications to specify their own thread scheduling policy. For example, during overload the application may wish to give priority to stages which consume fewer resources. Another policy would be to delay the scheduling of a stage until it has accumulated enough work to amortize the startup cost of that work, such as aggregating multiple disk accesses and performing them all at once. The SEDA approach can simplify the mechanism used to implement application-specific scheduling, since the concerns raised by "safe" scheduling in a multiprogrammed environment can be avoided. Specifically, the system can trust the algorithm provided by the application, and need not support multiple competing applications with their own scheduling policies.

Scalable I/O: SEDA's design should make it easier to construct scalable I/O interfaces, since the goal is to support a large number of I/O streams through a single application, rather than to fairly multiplex I/O resources across multiple applications. A SEDA-oriented asynchronous I/O layer would closely follow the internal implementation of contemporary filesystems and network stacks, but do away with the complexity of safe virtualization of the I/O interface. For example, rather than exporting a high-level socket layer, the OS could expose the event-driven nature of the network stack directly to applications. This approach also facilitates the implementation of zero-copy I/O, a mechanism which is difficult to virtualize for a number of reasons, such as safe sharing of pinned network buffers [31].

Application-controlled resource management: A SEDA-based operating system need not be designed to allow multiple applications to transparently share resources. Internet services are highly specialized and are not designed to share the machine with other applications: it is plainly undesirable for, say, a Web server to run on the same machine as a database engine (not to mention a scientific computation or a word processor!). While the OS may enforce protection (to prevent one stage from corrupting the state of the kernel or another stage), the system should not virtualize resources in a way which masks their availability from applications.

For instance, rather than hiding a file system buffer cache within the OS, a SEDA-based system should expose a low-level disk interface and allow applications to implement their own caching mechanism. In this way, SEDA follows the philosophy of systems such as Exokernel [12], which promotes the implementation of OS components as libraries under application control. Likewise, a SEDA-based OS should expose a virtual memory interface which makes physical memory availability explicit; this approach is similar to that of application-controlled paging [7, 8].

5. Related Work

The SEDA design was derived from approaches to managing high concurrency and unpredictable load in a variety of systems. The Flash web server [21] and the Harvest web cache [4] are based on an asynchronous, event-driven model which closely resembles the SEDA architecture. In Flash, each component of the web server responds to particular events, such as socket connections or filesystem access requests. The main server process is responsible for continually dispatching events to each of these components. This design typifies the "monolithic" event-driven architecture described earlier. Because certain I/O operations (in this case, filesystem accesses) do not have asynchronous interfaces, the main server process handles these events by dispatching them to *helper processes* via IPC.

StagedServer [15] is a platform which bears some resemblance to SEDA, in which application components are decomposed into stages separated by queues. In this case, the goal is to maximize processor cache locality by carefully scheduling threads and events within the application. By aggregating the execution of multiple similar events within a queue, locality is enhanced leading to greater performance.

The Click modular packet router [19] and the Scout operating system [20] use a software architecture similar to that of SEDA; packet processing stages are implemented by separate code modules with their own private state. Click modules communicate using either queues or function calls, while Scout modules are composed into a *path* which is used to implement vertical resource management and integrated layer processing. Click and Scout are optimized to improve per-packet latency, allowing a single thread to call directly through multiple stages. In SEDA, threads are isolated to their own stage for reasons of safety and load conditioning.

Extensible operating systems such as Exokernel [12] and SPIN [3] share our desire to expose greater resource control to applications. However, these systems have primarily focused on safe application-specific resource virtualization, rather than support for extreme concurrency and load. For instance, Exokernel's I/O primitives are blocking, necessitating a thread-based approach to concurrency. Our proposal is in some sense more radical than extensible operating systems: we claim that the right approach to supporting scalable servers is to eliminate resource virtualization, rather than to augment it with application-specific functionality.

6. Conclusion

We argue that traditional OS designs, intended primarily for safe and efficient multiprogramming, do not mesh well with the needs of highly-concurrent server applications. The large body of work that has addressed aspects of this problem suggests that the ubiquitous process model, along with the attendant requirement of transparent resource virtualization, is fundamentally wrong for these applications. Rather, we propose the *staged event-driven architecture*, which decomposes applications into stages connected by explicit event queues. This model enables high concurrency and fine-grained load conditioning, two essential requirements for Internet services.

We have implemented a prototype of a SEDA-based system, described in [30]. Space limitations prevent us from providing details here, although our experience with the SEDA prototype (implemented in Java on top of UNIX interfaces) has demonstrated the viability of this design for implementing scalable Internet service applications over commodity OSs. Still, Internet services necessitate a fundamental shift in operating system design ideology. We believe that the time has come to reevaluate OS architecture in support of this new class of applications.

References

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999.

[3] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, 1995.

[4] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, pages 153–163, January 1996.

[5] P. Druschel and L. Peterson. Fbufs: A high bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, 1993.

[6] S. D. Gribble. *A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction*. PhD thesis, UC Berkeley, September 2000.

[7] S. M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of OSDI '99*, February 1999.

[8] K. Harty and D. Cheriton. Application controlled physical memory using external page cache management, October 1992.

[9] J. C. Hu, I. Pyarali, and D. C. Schmidt. High performance Web servers on Windows NT: Design and performance. In *Proceedings of the USENIX Windows NT Workshop 1997*, August 1997.

[10] J. C. Hu, I. Pyarali, and D. C. Schmidt. Applying the Proactor pattern to high-performance Web servers. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, October 1998.

[11] S. Inohara, K. Kato, and T. Masuda. 'Unstable Threads' kernel interface for minimizing the overhead of thread switching. In *Proceedings of the 7th IEEE International Parallel Processing Symposium*, pages 149–155, April 1993.

[12] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.

[13] M. F. Kaashoek, D. R. Engler, G. R. Ganger, and D. A. Wallach. Server operating systems. In *Proceedings of the 1996 SIGOPS European Workshop*, September 1996.

[14] D. Kegel. The C10K problem. `http://www.kegel.com/c10k.html`.

[15] J. Larus. Enhancing server performance with StagedServer. `http://www.research.microsoft.com/~larus/Talks/StagedServer.ppt`, October 2000.

[16] J. Lemon. FreeBSD kernel event queue patch. `http://www.flugsvamp.com/~jlemon/fbsd/`.

[17] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14:1280–1297, September 1996.

[18] J. Mogul. Operating systems support for busy internet services. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

[19] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 217–231, Kiawah Island, South Carolina, December 1999.

[20] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of OSDI '96*, October 1996.

[21] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[22] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd Usenix Symposium on Operating Systems Design and Implementation (OSDI'99)*, February 1999.

[23] N. Provos and C. Lever. Scalable network I/O in Linux. Technical Report CITI-TR-00-4, University of Michigan Center for Information Technology Integration, May 2000. `http://www.citi.umich.edu/techreports/reports/citi-tr-00-4.ps.gz`.

[24] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-performance local area communication with fast sockets. In *Proceedings of the USENIX 1997 Annual Technical Conference*, 1997.

[25] M. Russinovich. Inside I/O Completion Ports. `http://www.sysinternals.com/comport.htm`.

[26] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[27] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th Annual Symposium on Operating System Principles*, December 1995.

[28] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A mechanism for integrating communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[29] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *Proceedings of the ACM SIGCOMM '96 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 40–52, Stanford, California, August 1996.

[30] M. Welsh. The Staged Event-Driven Architecture for highly concurrent server applications. Ph.D. Qualifying Examination Proposal, UC Berkeley, December 2000. `http://www.cs.berkeley.edu/~mdw/papers/quals-seda.pdf`.

[31] M. Welsh, A. Basu, and T. von Eicken. Incorporating memory management into user-level network interfaces. In *Proceedings of Hot Interconnects V*, August 1997.

[32] M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler. A design framework for highly concurrent systems. Technical Report UCB/CSD-00-1108, U.C. Berkeley Computer Science Division, April 2000.

Systems Directions for Pervasive Computing

Robert Grimm, Janet Davis, Ben Hendrickson, Eric Lemar, Adam MacBeth,
Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello,
Steven Gribble, David Wetherall
University of Washington
one@cs.washington.edu

Abstract

Pervasive computing, with its focus on users and their tasks rather than on computing devices and technology, provides an attractive vision for the future of computing. But, while hardware and networking infrastructure to realize this vision are becoming a reality, precious few applications run in this infrastructure. We believe that this lack of applications stems largely from the fact that it is currently too hard to design, build, and deploy applications in the pervasive computing space.

In this paper, we argue that existing approaches to distributed computing are flawed along three axes when applied to pervasive computing; we sketch out alternatives that are better suited for this space. First, application data and functionality need to be kept separate, so that they can evolve gracefully in a global computing infrastructure. Second, applications need to be able to acquire any resource they need at any time, so that they can continuously provide their services in a highly dynamic environment. Third, pervasive computing requires a common system platform, allowing applications to be run across the range of devices and to be automatically distributed and installed.

1. Introduction

Pervasive computing [10, 26] promises a computing infrastructure that seamlessly and ubiquitously aids users in accomplishing their tasks and that renders the actual computing devices and technology largely invisible. The basic idea behind pervasive computing is to deploy a wide variety of smart devices throughout our working and living spaces. These devices coordinate with each other to provide users with universal and immediate access to information and support users in completing their tasks. The hardware devices and networking infrastructure necessary to realize this vision are increasingly becoming a reality, yet precious few applications run in this infrastructure. Notable exceptions are email for communication and the World Wide Web as a medium for electronic publishing and as a client interface to multi-tier applications.

This lack of applications is directly related to the fact that it is difficult to design, build, and deploy applications in a pervasive computing environment. The pervasive computing space has been mapped as a combination of mobile and stationary devices that draw on powerful services embedded in the network to achieve users' tasks [9]. The result is a giant, ad-hoc distributed system, with tens of thousands of devices and services coming and going. Consequently, the key challenge for developers is to build applications that continue to provide useful services, even if devices are roaming across the infrastructure and if the network provides only limited services, or none at all.

As part of our research into pervasive computing, we are building *one.world*, a system architecture for pervasive computing [14]. Based on our experiences with this architecture, we believe that existing distributed computing technologies are ill-suited to meet this challenge. This is not to say that discovery services [1, 2, 8] or application-aware adaptation [19] are not useful in a pervasive computing environment. On the contrary, we consider them clearly beneficial for pervasive computing applications. However, they are not sufficient to successfully design, build, and deploy applications in the pervasive computing space.

Moreover, we argue that current approaches to building distributed applications are deeply flawed along three axes, which — to express their depth — we call fault lines. In the rest of this paper, we explore the three fault lines in detail; they are summarized in Table 1. First, Section 2 makes our case against distributed objects and outlines a more appropriate approach to integrating application data and functionality. Next, Section 3 discusses the need to write applications that continuously adapt in a highly dynamic environment. Finally, Section 4 argues for a common pervasive computing platform that spans the different classes of devices. We conclude this paper in Section 5.

Problem	Cause	Proposed Solution
Objects do not scale well across large, wide-area distributed systems	Encapsulation of data and functionality within a single abstraction	Keep data and functionality separate
Availability of application services is limited or intermittent	Transparency in a highly dynamic environment	Programming for change: Applications need to be able to acquire any resource they need at any time
Programming and distributing applications is increasingly unmanageable	Heterogeneity of devices and system platforms	Common system platform with an integrated API and a single binary format

Table 1. Overview of the three fault lines discussed in this paper, listing the problem, cause, and proposed solution for each fault line.

2. Data and Functionality

The first fault line concerns the relationship between data and functionality and how they are represented. Several distributed systems, such as Legion [16] or Globe [25], are targeted at a global computing environment and have explored the use of objects as the unifying abstraction for both data and functionality. We are skeptical about this use of objects for distributed computing for two reasons.

First, objects as an encapsulation mechanism are based on two assumptions: (1) Implementation and data layout change more frequently than an object's interface, and (2) it is indeed possible to design interfaces that accommodate different implementations and hold up as a system evolves. However, these assumptions do not hold for a global distributed computing environment. Increasingly, common data formats, such as HTML or PNG, are specified by industry groups or standard bodies, notably the World Wide Web Consortium, and evolve at a relatively slow pace. In contrast, application vendors compete on functionality, leading to considerable differences in application interfaces and implementations and a much faster pace of innovation.

Second, it is preferable to store and communicate data instead of objects, as it is generally easier to access passive data rather than active objects. In particular, safe access to active objects in a distributed system raises important issues, notably system security and resource control, that are less difficult to address when accessing passive data. This is clearly reflected in today's Internet: Access to regular HTML or PDF documents works well, while active content results in an ever continuing string of security breaches [17]. Based on these two realizations, we argue that data and functionality should be kept separate rather than being encapsulated within objects.

At the same time, data and functionality depend on each other, especially when considering data storage and mobile code. On one hand, data management systems already rely on mobile code for their services. For example, Bayou propagates updates as procedures and not simply as data [23]. The Oracle8*i* database not only supports SQL stored procedures, but also includes a fully featured Java virtual machine [11]. On the other hand, mobile code systems have seen limited success in the absence of a standard data model and the corresponding data management solutions. For example, while many projects have explored mobile agents [18], they have not been widely adopted, in part because they lack storage management. Java, which was originally marketed as a mobile code platform for the Internet, has been most successful in the enterprise, where access to databases is universal [21].

The result is considerable tension between integrating data and functionality too tightly — in the form of objects — and not integrating them tightly enough. *one.world* resolves this tension by keeping data and functionality separate and by introducing a new, higher-level abstraction to group the two. In our architecture, data is represented by tuples, which essentially are records with named and optionally typed fields, while functionality is provided by components, which implement units of functionaly. Environments serve as the new unifying abstraction: They are containers for stored tuples, components, and other environments, providing a combination of the roles served by file system directories and nested processes [5, 12, 24] in more traditional operating systems. Environments make it possible to group data and functionality when necessary. At the same time, they allow for data and functionality to evolve separately and for applications to store and exchange just data, thus avoiding the two problems associated with objects discussed above.

To summarize, we are arguing that data and functionality need to be supported equally well in large distributed systems, yet also need to be kept separate. We are not arguing that object-oriented programming is not useful. *one.world* is implemented mostly in Java and makes liberal use of object-oriented language features such as inheritance to provide its functionality.[1] At the same time, our architecture clearly separates data and functionality, using tuples to represent data and components to express functionality.

3. Programming for Change

The second fault line is caused by transparent access to remote resources. By building on distributed file systems or remote procedure call packages, many existing distributed systems mask remote resources as local resources. This transparency certainly simplifies application development. From the programmer's viewpoint, accessing a remote resource is as simple as a local operation. However, this comes at a cost in failure resilience and service availability. Network connections and remote servers may fail. Some services may not be available at all in a given environment. As a result, if a remote service is inaccessible or unavailable, distributed applications cannot provide their services, because they were written without the expectation of change.

We believe that this transparency is misleading in a pervasive computing environment, because it encourages a programming style in which a failure or the unavailability of a resource is viewed as an extreme case. But in an environment where tens of thousands of devices and services come and go, the unavailability of some resource may be the common (or at least frequent) case. We are thus advocating a programming style that forces applications to explicitly acquire all resources, be they local or remote, and to be prepared to reacquire them or equivalent resources at any time.

In *one.world*, applications need to explicitly bind all resources they use, including storage and communication channels. Leases are used to control such bindings and, by forcing applications to periodically renew them, provide timeouts for inaccesible or unavailable resources. While leases have been used in other distributed systems, such as Jini [2], to control access to remote resources, we take them one step further by requiring that *all* resources be explicitly bound and leased. Furthermore, resource discovery in *one.world* can use late binding, which effectively binds resources on every use and thus reduces applications' exposure to failures or changes in the environment [1].

This style of programming for change imposes a strict discipline on applications and their developers. Yet, programming for change also presents an opportunity by enabling system services that make it easier to build applications. *one.world* provides support for saving and restoring application checkpoints and for migrating applications and their data between nodes. Checkpointing and migration are useful primitives for building failure resilient applications and for improving performance in a distributed system. Furthermore, migration is attractive for applications that follow a user as she moves through the physical world.

Checkpointing and migration affect an environment and its contents, including all nested environments. Checkpointing captures the execution state of all components in an environment tree and saves that state in form of a tuple, making it possible to later restore the saved state. Migration moves an environment tree, including all components and stored tuples, from one device to another. Since applications already need to be able to dynamically acquire resources they need, both checkpointing and migration eschew transparency and are limited to the resources contained in the environment tree being checkpointed or migrated. As a result, their implementation in *one.world* can avoid the complexities typically associated with full process checkpointing and migration [18], and migration in the wide area becomes practical.

To summarize, the main idea behind programming for change is to force developers to build applications that better cope with a highly dynamic environment, while also providing primitives that make it easier to implement applications.

4. The Need for a Common Platform

The third fault line is rooted in the considerable and inherent heterogeneity of devices in a pervasive computing environment. Computing devices already cover a wide range of platforms, computing power, storage capacity, form factors, and user interfaces. We expect this heterogeneity to increase over time rather than decrease, as new classes of devices such as pads or car computers become widely used.

Today, applications are typically developed for specific classes of devices or system platforms, leading to separate versions of the same application for handhelds, desktops, or cluster-based servers. Furthermore, applications typically need to be distributed and installed separately for each class of devices and processor family. As heterogeneity increases, developing applications that run across all platforms will become exceedingly difficult. As the number of devices grows, explicitly distributing and installing applications for each class of devices and processor family will become unmanageable, especially in the face of migration across the wide area.

We thus argue for a single application programming interface (API) and a single binary distribution format, including a single instruction set, that can be implemented across the range of devices in a pervasive computing environment. A single, common API makes it possible to develop applications once, and a single, common binary format enables

[1] Though, for several features, including the implementation of tuples, mixin-based inheritance [4] and multiple dispatch as provided by MultiJava [7] would have provided a better match than Java's single inheritance and single dispatching of methods.

the automatic distribution and installation of applications. It is important to note that Java does not provide this common platform. While the Java virtual machine is attractive as a virtual execution platform (and used for this purpose by *one.world*), Java as an application platform does not meet the needs of the pervasive computing space. In particular, Java's platform libraries are rather large, loosely integrated, and often targeted at conventional computers. Furthermore, Java, by itself, fails to separate data and functionality and does not encourage programming for change, as discussed in Sections 2 and 3 respectively.

Given current hardware trends and advances in virtual execution platforms, such as the Java virtual machine or Microsoft's common language runtime [22], we can reasonably expect that most devices can implement such a pervasive computing platform. Devices that do not have the capacity to implement the full platform, such as small sensors [15], can still interact with it by using proxies or emulating the platform's networking protocols. Furthermore, legacy applications can be integrated by communicating through standard networking protocols, such as HTTP or SOAP [3], and by exchanging data in standard formats, such as XML.

A pervasive computing platform that runs across a wide range of devices does impose a least common denominator on the core APIs. Applications can only assume the services defined by the core APIs; they must implement their basic functionality within this framework. At the same time, a common platform does not prevent individual devices from exposing additional services to applications. It simply demands that additional services be treated as optional and dynamically discovered by applications.

As part of our research on *one.world*, we are exploring how to scale a common platform across the range of devices. Taking a cue from other research projects [6, 13, 15, 20], which have successfully used asynchronous events at very different points of the device space, our architecture also relies on asynchronous events to express control flow. All system interfaces are asynchronous, and application components interact by exchanging asynchronous events. The hope behind this design decision is that it will considerably aid with the scalability of the architecture. *one.world*'s implementation currently runs on Windows and Linux computers, and a port to Compaq's iPAQ handheld computer is under way.

5. Outlook

In this paper, we have argued that current approaches to distributed computing are ill-suited for the pervasive computing space and have identified three fault lines of existing distributed systems. First, while object-oriented programming continues to provide an attractive paradigm for application development, data and functionality should be kept separate for pervasive computing applications as they typically need to evolve independently. Second, applications need to be explicitly programmed to gracefully handle change. While this style of programming imposes a strict discipline on application developers, it also enables system services, such as checkpointing and migration, previously not available in distributed systems of this scale. Third, pervasive computing requires a common system platform, so that applications can run across (almost) all devices in this infrastructure and can be automatically distributed and installed.

We are exploring how to address these fault lines with *one.world*, a system architecture for pervasive computing. In an effort to better understand the needs of application developers, we have taught an undergraduate course that leverages *one.world* as the basis for students' projects. We are also building pervasive applications within our architecture and are collaborating with other researchers in the department to implement additional infrastructure services on top of it. Further information on *one.world*, including a source distribution, is available at `http://one.cs.washington.edu/`.

Acknowledgments

We thank David Notkin for helping us to refine our observations and Brendon Macmillan as well as the anonymous reviewers for their comments on an earlier version of this paper.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island Resort, South Carolina, Dec. 1999.

[2] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

[3] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1. W3C note, World Wide Web Consortium, Cambridge, Massachusetts, May 2000.

[4] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications '90*, pages 303–311, Ottawa, Canada, Oct. 1990.

[5] P. Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 250, Apr. 1970.

[6] P. Chou, R. Ortega, K. Hines, K. Partridge, and G. Borriello. ipChinook: An integrated IP-based design framework for distributed embedded systems. In *Proceedings of the 36th*

ACM/IEEE Design Automation Conference, pages 44–49, New Orleans, Louisiana, June 1999.

[7] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications '00*, pages 130–145, Minneapolis, Minnesota, Oct. 2000.

[8] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 24–35, Seattle, Washington, Aug. 1999.

[9] M. L. Dertouzos. The future of computing. *Scientific American*, 281(2):52–55, Aug. 1999.

[10] M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next century challenges: Data-centric networking for invisible computing. In *Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 256–262, Seattle, Washington, Aug. 1999.

[11] S. Feuerstein. *Guide to Oracle8i Features*. O'Reilly, Oct. 1999.

[12] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 137–151, Seattle, Washington, Oct. 1996.

[13] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 319–332, San Diego, California, Oct. 2000.

[14] R. Grimm, T. Anderson, B. Bershad, and D. Wetherall. A system architecture for pervasive computing. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 177–182, Kolding, Denmark, Sept. 2000.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Cambridge, Massachusetts, Nov. 2000.

[16] M. Lewis and A. Grimshaw. The core Legion object model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 551–561, Syracuse, New York, Aug. 1996.

[17] G. McGraw and E. W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. Wiley Computer Publishing, John Wiley & Sons, 1999.

[18] D. Milojičić, F. Douglis, and R. Wheeler, editors. *Mobility—Processes, Computers, and Agents*. ACM Press. Addison-Wesley, Feb. 1999.

[19] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint-Malo, France, Oct. 1997.

[20] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 199–212, Monterey, California, June 1999.

[21] A. Radding. Java emerges as server-side standard. *InformationWeek*, (987):121–128, May 22, 2000.

[22] J. Richter. Microsoft .NET framework delivers the platform for an integrated, service-oriented web. *MSDN Magazine*, 15(9):60–69, Sept. 2000.

[23] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 172–183, Copper Mountain Resort, Colorado, Dec. 1995.

[24] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Proceedings of the 8th ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, Sept. 1998.

[25] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, 1999.

[26] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, Sept. 1991.

The Case for Resilient Overlay Networks

David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris
MIT Laboratory for Computer Science
Cambridge, MA 02139
{dga, hari, kaashoek, rtm}@lcs.mit.edu
`http://nms.lcs.mit.edu/ron/`

Abstract

This paper makes the case for Resilient Overlay Networks (RONs), an application-level routing and packet forwarding service that gives end-hosts and applications the ability to take advantage of network paths that traditional Internet routing *cannot* make use of, thereby improving their end-to-end reliability and performance. Using RON, nodes participating in a distributed Internet application configure themselves into an overlay network and cooperatively forward packets for each other. Each RON node monitors the quality of the links in the underlying Internet and propagates this information to the other nodes; this enables a RON to detect and react to path failures within several seconds rather than several minutes, and allows it to select application-specific paths based on performance. We argue that RON has the potential to substantially improve the resilience of distributed Internet applications to path outages and sustained overload.

1 Introduction

Today's wide-area Internet routing architecture organizes the Internet into autonomous systems (ASes) that peer with each other and exchange information using the Border Gateway Protocol (BGP), Version 4 [10]. This approach scales well to a large number of connected networks, but this scalability comes at the cost of the increased vulnerability to link or router failures. Various recent studies have found that path failures are common and that the convergence time after a problem is detected is usually on the order of several minutes [5], and that path outages, routing anomalies, and active denial-of-service attacks cause significant disruptions in end-to-end communication [1, 8]. This reduces the reliability of end-to-end communication over Internet paths and therefore adversely affects the reliability of distributed Internet applications and services.

We propose *Resilient Overlay Networks* (RONs) as an architecture to improve the reliability of distributed applications on the Internet. Each application creates a RON from its participating nodes. These nodes are typically spread across multiple ASes, and see different routes through the Internet. RON nodes cooperate with each other to forward data on behalf of any pair of communicating nodes in the RON, thus forming an *overlay network*. Because ASes are independently administered and configured, underlying path failures between communicating nodes usually occur independently. Thus, if the underlying topology has physical path redundancy, it is often possible for a RON to find paths between RON nodes even if Internet routing protocols such as BGP (that are optimized for scalability) cannot find them.

Nodes in a RON self-configure into the overlay by exchanging information across the underlying Internet paths. Each RON node has "virtual links" to all other RON nodes, which it uses to maintain connectivity and exploit the underlying IP network's redundancy. When a RON node receives a packet destined for another, it looks for the destination in an application-specific forwarding table, encapsulates the packet in a RON packet, and ships it to the next RON node. In this way, the packet moves across the overlay via a sequence of RON nodes until it reaches the intended destination.

To find and use alternate paths, a RON monitors the health of the underlying Internet paths between its nodes, dynamically selecting paths that avoid faulty or overloaded areas. The goal is to ensure continued communication between RON nodes despite failures due to outages, operational errors, or attacks in the underlying network. RON nodes infer the quality of virtual links using active probing and passive observation of traffic, and exchange this information using a routing protocol. Each node can use a variety of performance metrics, such as packet loss rate, path latency, or available bandwidth to select an appropriate application-specific path. This approach has potential because each RON is small in size (less than fifty nodes), which allows aggressive path monitoring and maintenance.

A RON ensures that as long as there is *an* available path in the underlying Internet between two RON nodes, the RON application can communicate robustly even in the face of problems with the "direct" (BGP-chosen) path between them. The limited size of each independent RON is not a serious limitation for many applications and services. A video conferencing program may link against a RON library, forming a routing overlay between the participants in the con-

This research was sponsored by Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center San Diego under contract N66001-00-1-8933.

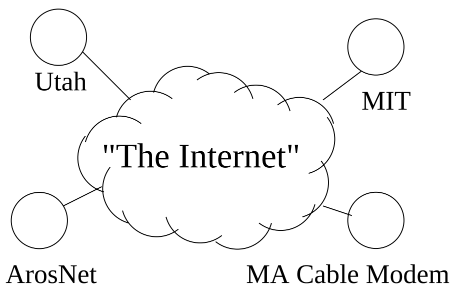


Figure 1: A common (mis)conception of Internet interconnections.

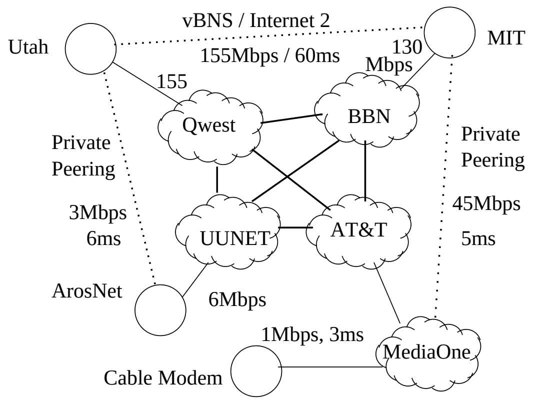


Figure 2: The details of Internet interconnections. Dotted links are *private* and are not announced globally.

ference. Alternatively, a RON-based application-aware IP packet forwarder may be located at points-of-presence in different ASes, forming an "Overlay ISP" that improves the reliability of Internet connectivity for its customers.

This paper presents the case for developing distributed applications using RON (Section 2), outlines an approach by which RONs may be architected (Section 3), relates RON to previous work (Section 4), and concludes with a research agenda for future work (Section 5).

2 The Case for RONs

A common, but incorrect, view of the topology of the Internet is that institutions and companies connect to "The Great Internet Cloud." Figure 1 illustrates an example of four sites, **MIT**, **Utah**, **ArosNet**, and **MediaOne**, connected to the Internet cloud. In this view, the Internet is very robust, rapidly routing packets around failures and traffic overload, and providing near-perfect service.

Unfortunately, this ideal view of the Internet cloud is far from reality. The Internet Service Providers (ISPs) constituting the Internet exchange routing information using BGP, which is designed to scale well at the expense of maintaining detailed information about alternate paths between networks. To avoid frequent route changes that may propagate through many other ASes, frequent route announcements and withdrawals are damped; furthermore, convergence times on route changes take many minutes [5] with currently deployed BGP implementations. Last but not least, there are numerous financial, political, and policy considerations that influence the routes announced via BGP.

ISPs typically provide two types of connectivity: "transit" and "peering." If the ISP provides *transit* for a customer A, it tells other ISPs that they may reach A through the ISP. If an ISP has a *peering* relationship with A, it keeps this knowledge to itself; the ISP and its customers can reach A via this link, but the rest of the Internet may not. Peering relationships are often free, because they enable the more efficient exchange of packets without placing the burden of hauling packets on either partner, but globally announced transit relationships almost always involve some form of settlement.

Figure 2 redraws Figure 1 to reflect reality. MIT is connected to the Internet via BBN, and to Internet2. It has a private peering link to MediaOne in Cambridge (MA), so students can quickly connect to their MIT machines from home. Utah is connected to the Internet via Qwest, to Internet2, and to a local ISP, ArosNet, via a private peering link. ArosNet is connected to the Internet via UUNET, and MediaOne is connected to the Internet via AT&T. In this example, several desirable paths are unavailable globally: the private peering links for financial reasons (the parties have no apparent incentive to provide transit for each other) and the Internet2 connections because it is a research network.

These interconnections show two reasons BGP is unable to ensure "best"—or sometimes even "good"—routes, and route around problems even when different physical paths are available. The first reason, explained above, is a consequence of the economics and administration of peering relationships. The second relates to scalability.

For communication costs to scale well, BGP must simplify routing data enormously; for computational scalability, its decision metrics must be both simple and stable. BGP primarily uses its own hop-counting mechanism to determine routes and it exports a single "best" route for forwarding packets. This causes three important problems: first, as noted in the Detour study [11], BGP may make suboptimal routing decisions. Second, BGP does not consider path performance when making routing decisions, and so cannot route around a path outage caused by traffic overload. The result is that path outages can lead to significant disruptions in communication [1]. Third, BGP may take several minutes to stabilize in the event of a route change or link failure [5]. The result is that today's Internet is easily vulnerable to router faults, link failures, configuration or operational errors, and malice—hardly a week goes by without some serious prob-

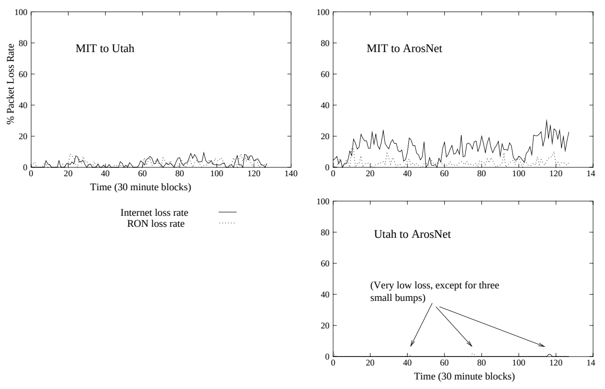


Figure 3: The upper right figure shows the loss rate with and without RON between MIT and ArosNet. RON was able to improve the loss rate considerably by routing through Utah. The upper left figure shows the MIT to Utah loss rate, and the lower right shows the Utah to ArosNet loss rate.

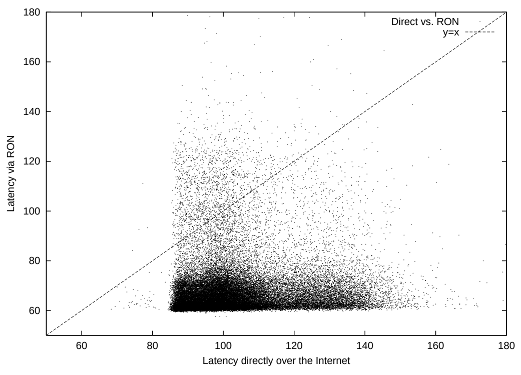


Figure 4: RON vs. direct samples. The samples are temporally correlated; the latency via RON is plotted on the Y axis, and the latency via the Internet is on the X axis. 0.5% of the outlying samples (215 / 51395) are not shown for readability. The dataset represents 62 hours of probes taken roughly 4 seconds apart.

lem affecting one or more backbone service providers [6].

Many of the restrictions of peering can be overcome. An organization that has Internet service in multiple ASes can run an application that is distributed across nodes located in the ASses, and use a RON to provide routing between these nodes. By explicitly constraining the size of any given RON to be small (under, say, 50 nodes), the aggressive exploration of alternate paths and performance-based path selection can be accomplished. Thus, RON's routing and path selection schemes emphasize failure detection and recovery over scalability, improving both reliability and performance of the RON application.

To obtain a preliminary understanding of the benefits of using RON, we evaluated the effects of indirect RON-based packet forwarding between the four sites mentioned in our examples: The University of Utah, MIT, ArosNet, and a MediaOne cable modem in Cambridge, MA. The interconnections between these nodes are as shown in Figure 2. In this topology, RON is able to provide both reliability and performance benefits for some of the communicating pairs.

2.1 Reliability

Figure 3 shows the 30-minute average packet loss rates between MIT and ArosNet. In these samples, the loss rate between MIT and ArosNet ranged up to 30%, but RON was able to correct this loss rate to well below 10% by routing data through Utah (and occasionally through the cable modem site). This shows that situations of non-transitive Internet routing do occur in practice, and can be leveraged by a RON to improve the reliability of end-to-end application communication.

2.2 Performance

We took measurements between the four sites using `tcping`, a TCP-based ping utility that we created. We sent one `tcping` flow over the direct Internet and another through the lowest-latency indirect path as estimated by the results of recent `tcping` probes. If the direct IP path had lower latency that the best indirect path, then the direct one was used since that is what RON would do as well.

Figure 4 shows the latency results between MIT and ArosNet, gathered over 62 hours between January 8 and January 11 2001. In 92% of the samples, the latency of the packets sent over a RON-like path was better than the Internet latency. The average latency over the measurement period decreased from 97ms to 68ms; indirect hops through both MediaOne and Utah were used, and some packets were sent directly. The benefit in this case arose partly from using the high-speed Internet2 connection, but more from avoiding the exchange between MediaOne and Qwest, which frequently went through Seattle!

2.3 Case Summary

These observations argue for a framework that allows small numbers of nodes to form an overlay that can take advantage of these improved paths. By pushing control towards the endpoints, or even directly to the application, the RON architecture achieves four significant advantages. (1) More efficient end-system detection and correction of faults in the underlying routes, even when the underlying network layer incorrectly thinks all is well. (2) Better reliability for applications, since each RON can have an independent, application-specific definition of what constitutes a fault. (3) Better performance, since a RON's limited size allows it to use more

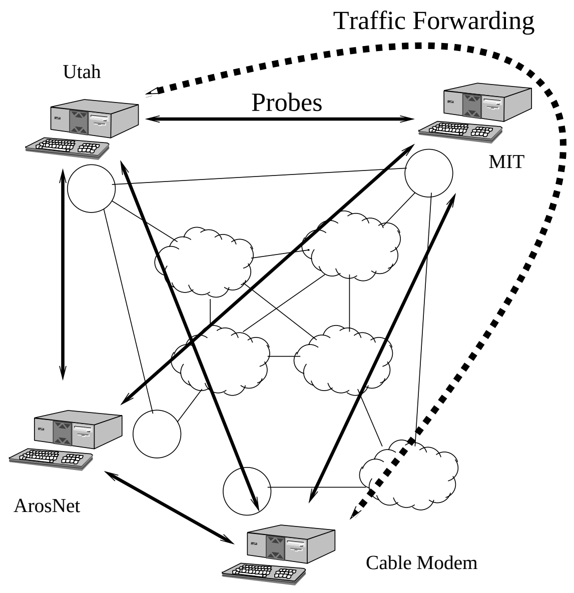


Figure 5: The general approach used in the RON system. Nodes send probes to determine the network characteristics between each other. Using their knowledge of the network, they potentially route traffic through other nodes. In this example, traffic from Utah to the Cable Modem site is sent indirectly via MIT.

aggressive path computation algorithms than the Internet. (4) Application-specific path selection, since RON applications can define their own routing metrics.

3 Approach

The RON approach is conceptually simple. Figure 5 outlines this approach: The RON software sends *probes* between RON nodes to determine the network characteristics between them. Application-layer *RON routers* share this information with the other RON nodes, and decide on next hops for packets. When appropriate, the traffic between two RON nodes is sent indirectly through other RON nodes, instead of going directly over the Internet.

We designed the RON software as libraries, usable by unprivileged user programs. The components of the RON software provide the mechanisms necessary for application-layer indirect routing. RON needs (1) methods to *measure* the properties of the paths between nodes and *aggregate* this information; (2) an algorithm to *route* based on this information; and (3) a mechanism to *send data* via the overlay. We describe each of these components below.

3.1 Monitoring Path Quality

RON nodes measure path quality using a combination of active probing by sending packets across virtual links, and passive measurement of the results achieved by data transfers over the virtual links. Because our goal is to provide better service than the default paths, we must measure links that may not be in use by data transmissions, necessitating the use of active probes. Passive measurements, however, can provide more information with less bandwidth cost by using traffic that must already flow over the network. This is why we use both forms of monitoring.

Measurements may either be *system-defined*, e.g., “the latency between two hosts,” or they may be *application-defined*, e.g., “the time to download this object from a mirror site,” similar to the approach taken in SPAND [12]. The designers of an overlay network cannot be omniscient about the desires and metrics that are important to future users; a well-designed system must provide both a rich set of system-defined metrics for ease of use, and the ability to import and route based on application-defined metrics to accommodate unforeseen applications.

It is impractical to send days of detailed performance history to all other participants in the network so that they can decide on the best path over which to transfer data. Furthermore, a reliable system must handle participating nodes that crash, reboot, or rejoin the RON. Measurement data, particularly network probe data, is often extremely noisy and must be smoothed before it is of use to clients. The RON system must therefore have a mechanism for *summarizing* the performance data it collects, before transmitting it across wide-area network paths to other RON nodes.

Hosts on the same LAN will frequently experience similar network conditions when communicating with other hosts. To reduce the impact of network probe traffic and increase the base of information available to the routing system, hosts on the same LAN should be able to share information about the performance of remote paths and sites. From these requirements, we conclude that the RON system should support a shared *performance database* that local hosts can use to share and aggregate performance data. To avoid introducing more points of failure into the system, both the performance database and its clients must treat the data stored in it as *soft state*. Clients must not fail if information they want is not in the database. The correct functioning of the database must not depend on the presence of information about particular clients or remote networks.

3.2 Routing and Forwarding

Indirect hops through the network require additional bandwidth, time, and computation. We believe that we can achieve the major benefits of an overlay using only a few indirect hops. Our design currently calls for computing paths only with single indirect hops. To send packets indirectly, the RON architecture should use UDP, not IP or some new protocol, to permit implementation as an unprivileged process. The small size of each RON allows us to exchange topology and performance information using a link-state routing protocol.

Intermediate forwarding nodes should not require application-specific knowledge about the packets they handle. We take the idea of *flow labels* from IPv6 [7] and MPLS [4]: The RON endpoints should tag their flows with an appropriate routing hint ("Minimize latency") and with a flow identifier, permitting downstream routers to handle the packets without needing to understand the protocols contained in the encapsulated packets. For instance, a video conferencing application may send its audio and video data as logically separate streams of data, but may want them to be routed along the same path to keep them synchronized. By pushing flow labeling as close to the application as possible, these decisions can be made at the right place. Early flow labeling also reduces the load on the intermediate nodes, by simplifying their routing lookups.

3.3 Sending Data

The basic RON data transmission API is simple: The conduit that provides the input and output for the RON must provide a function to call when there is data to be delivered, and must either notify the RON forwarder explicitly *or* provide a `select`-like mechanism for notifying the forwarder when data is available for insertion into the RON. Both of these alternatives are appropriate for use in the libraries implementing the RON functionality; the needs of the application should determine which is used.

3.4 Applications and Extensions

The components of RON described thus far are necessary for a basic user-level packet forwarding system, but applications that integrate more tightly with the routing and forwarding decisions are capable of more complex behavior. We discuss a few usage scenarios below, considering how they interact with the base RON functionality.

RONs can be deployed on a per-application basis, but they may also be deployed at a border router. There, they can be used to link entire networks with Overlay Virtual Private Networks. An Overlay ISP might even buy bandwidth from a number of conventional ISPs, paying them according to a Service-Level Agreement, and selling "value-added" robust networking services to its own customers.

When used to encapsulate network-level traffic, RONs can be combined with Network Address Translation (NAT) to permit the tunneling of traffic from remote sites not enabled with overlay functionality. For example, consider the network from Figure 2. A RON node located in the EECS department at MIT could be used by the other sites to proxy HTTP requests to `www.mit.edu`, accelerating Web browsing for off-site collaborators. Traffic would flow through the overlay to the MIT RON node, from which an HTTP request would be sent to the Web server. The HTTP response would be sent to the MIT RON node, and from there, relayed to the requesting host over the overlay.

Another use of RONs is to implement multi-path forwarding of flows. TCP performs poorly when subject to the large jitter and packet reordering that is often imposed by splitting one flow between multiple paths, but sending *different* TCP flows between the same two hosts (or two networks) poses few problems. The flow labeling component of a RON provides the first handle necessary to achieve this goal, and a routing component that performs flow assignment would provide the other part.

When a cooperating RON system either controls the majority of the available bandwidth on its links, or is given quality of service (QoS) guarantees on individual links of the network within a single ISP, it may be possible to then use the overlay network to provide global QoS guarantees to individual flows that traverse the overlay[1].

3.5 Routing Policies and Deployment

As with any overlay or tunneling technique, RONs create the possibility of misuse, violation of Acceptable Use Policies (AUPs), or violation of BGP transit policies. At the same time, RONs also provide more flexible routing that can *enhance* the ability of organizations to implement sophisticated *policy routing*, which is the ability to make routing decisions based upon the *source* or *type* of traffic, not just its destination address. This is an old idea [2], but its use in backbone routers have been scarce because of the increased CPU load it frequently imposes.

RONs interact with network policies in two ways. Because RONs are deployed only between small groups of cooperating entities who have already purchased the Internet bandwidth they use, they cannot be used to find "back-doors" into networks without the permission of an authorized user of those networks. The upholding of an organization's AUP is primarily due to cooperation of its employees, and this remains unchanged with the deployment of RONs.

More importantly, the smaller nature of RONs running atop powerful desktop computers can be used to implement policy routing on a per-application basis. One of our goals is the creation of a policy-routing aware forwarder with which administrators can easily implement policies that dictate. For instance, one policy is that only RON traffic from a particular research group may be tunneled over Internet2; traffic from the commercial POPs must traverse the commercial Internet.

3.6 Status

We have implemented a basic RON system to demonstrate the feasibility of our end-host based approach and are continuing to refine our design and implementation. We are deploying our prototype at a few nodes across the Internet and

[1]This possibility was suggested by Ion Stoica.

are measuring outages, loss rates, latency, and throughput to quantify the benefits of RON. We have built one RON application, an IP forwarder that interconnects with other such clients to provide an Overlay ISP service.

4 Related Work

The Detour study made several observations of suboptimal Internet routing [11]. Their study of traceroute-based measurements and post-analysis of Paxson's [8, 9] data shows that alternate paths may have superior latency or loss rates. These studies used traceroutes scheduled from a central server, which may undercount network outages when the scheduler is disconnected. Our research builds on their analysis by elucidating an approach for an architecture to exploit these properties. The Detour framework [3] is an in-kernel packet encapsulation and routing architecture designed to support alternate-hop IP packet routing for improved performance. In contrast, RON advocates tighter integration of the application and the overlay, which permits "pure application" overlays and allows the use of application-defined quality metrics and routing decisions. Furthermore, the main objective of RON is reliability, not performance.

Content Delivery Networks (CDNs) use overlay techniques and caching to improve the performance of specific applications, such as HTTP and streaming video. The functionality provided by the RON libraries may ease the development of future CDNs by providing some basic routing components.

The X-Bone is designed to speed the deployment of IP-based overlay networks [13]. It provides a GUI for automated configuration of IP addresses and DNS names, simple overlay routing configurations, and remote maintenance of the overlays via secure HTTP. The X-Bone does not yet support fault-tolerant operation or metric-based route optimization. Its management functions are complementary to our work.

5 Summary and Research Agenda

This paper made the case for developing reliable distributed Internet services and applications using Resilient Overlay Networks (RONs), an application-level routing and packet forwarding system. A RON improves the end-to-end reliability of Internet communication by taking advantage of alternate paths and enabling application-controlled path selection in a way that traditional BGP-based Internet routing cannot.

While measurements collected by us and others suggest that RONs might work well in practice, several key research questions need to be addressed. Some of these are:

1. *How many intermediate hops?* We hypothesize that, in practice, it is sufficient to consider paths that include at most one intermediate RON node to obtain the benefits of improved reliability and performance. If this is true, it will simplify RON's path selection mechanisms and allow the implementation of a variety of application-controlled metrics.

2. *How do we choose routes?* Route selection involves summarizing link metrics, combining them into a path metric, and applying hysteresis to come up with an estimate of the route quality. How do we best perform these actions for different link metrics? How do we filter out bad measurements, and perform good predictions? How do we combine link metrics (such as loss and latency) to meet application needs?

3. *How frequently do we probe?* The frequency of probing trades off responsiveness and bandwidth consumption. The speed with which failed routes can be detected will determine how well RONs will improve end-to-end reliability.

4. *What routing policies can RON express?* RONs may allow more expressive routing policies than current approaches, in part because of their application-specific architecture.

5. *How do RONs interact?* What happens if RONs become wildly popular in the Internet? How do independent RONs sharing network links interact with one another and would the resulting network be stable? Understanding these interactions is a long-term goal of our future research.

References

[1] CHANDRA, B., DAHLIN, M., GAO, L., AND NAYATE, A. End-to-end WAN Service Availability. In *Proc. 3rd USITS* (San Francisco, CA, 2001), pp. 97–108.

[2] CLARK, D. *Policy Routing in Internet Protocols.* Internet Engineering Task Force, May 1989. RFC 1102.

[3] COLLINS, A. The Detour Framework for Packet Rerouting. Master's thesis, University of Washington, Oct. 1998.

[4] DAVIE, B., AND REKHTER, Y. *MPLS: Technology and Applications.* Academic Press, San Diego, CA, 2000.

[5] LABOVITZ, C., AHUJA, A., BOSE, A., AND JAHANIAN, F. Delayed internet routing convergence. In *Proc. ACM SIGCOMM '00* (Stockholm, Sweden, 2000), pp. 175–187.

[6] The North American Network Operators' Group (NANOG) mailing list archive. `http://www.cctec.com/maillists/nanog/index.html`, Nov. 1999.

[7] PARTRIDGE, C. *Using the Flow Label Field in IPv6.* Internet Engineering Task Force, 1995. RFC 1809.

[8] PAXSON, V. End-to-End Routing Behavior in the Internet. In *Proc. ACM SIGCOMM '96* (Stanford, CA, Aug. 1996).

[9] PAXSON, V. End-to-End Internet Packet Dynamics. In *Proc. ACM SIGCOMM '97* (Cannes, France, Sept. 1997).

[10] REKHTER, Y., AND LI, T. *A Border Gateway Protocol 4 (BGP-4).* Internet Engineering Task Force, 1995. RFC 1771.

[11] SAVAGE, S., COLLINS, A., HOFFMAN, E., SNELL, J., AND ANDERSON, T. The end-to-end effects of Internet path selection. In *Proc. ACM SIGCOMM '99* (1999), pp. 289–299.

[12] SESHAN, S., STEMM, M., AND KATZ, R. H. SPAND: Shared Passive Network Performance Discovery. In *Proc. 1st USITS* (Monterey, CA, December 1997).

[13] TOUCH, J., AND HOTZ, S. The X-Bone. In *Proc. Third Global Internet Mini-Conference in conjunction with Globecom '98* (Sydney, Australia, Nov. 1998).

Position Summary: Toward a rigorous data type model for HTTP

Jeffrey C. Mogul (Jeffrey.Mogul@Compaq.com)
Compaq Computer Corp. Western Research Lab., 250 University Ave., Palo Alto, CA 94301

Abstract

The HTTP protocol depends on a structure of several data types, such as messages and resources. The current ad hoc data type model has served to support a huge variety of HTTP-based applications, but its weaknesses have been exposed in attempts to formalize and (especially) to extend the protocol. These weaknesses particularly affect the semantics of caching within the HTTP distributed system.

1. Introduction

HTTP is a network protocol, but it is also the basis of a large and complex distributed system, with the possibility of caches at many points. An unambiguous and extensible specification of HTTP caching has proved difficult, because HTTP lacks a clear and consistent data type model for the primitive structures of the protocol itself. This is partly a consequence of a conceptual faultline between “protocol designers” and “distributed system designers,” and a failure to meld the expertise of both camps.

2. Problems with the current data model

Every HTTP request operates on a *resource* and results in a *response*. HTTP adopted the MIME term *entity*, defined as “The information transferred as the payload [headers and body] of a request or response ...” HTTP/1.1 added *entity tags*, used in cache validation. The server may attach an entity tag to a response; a client can then validate the corresponding cache entry by including this entity tag in its re-request to the server If it matches the current entity tag, the server can respond with a “Not Modified” message instead of sending the entire entity.

What is the data type of the result of a simple HTTP GET operation? Is it an entity? The attempted analogy between MIME messages and HTTP data types treats the message as the central concern, which is true for MIME (an email protocol that transfers messages) but not for HTTP (a protocol for remote operations on resources). Also, HTTP allows the transmission of subranges of the bytes of a result, or of just the metainformation without the associated body, so the result might span several HTTP-layer messages. Therefore, an HTTP “entity” is merely an ephemeral, and perhaps partial, representation of one aspect of a resource.

So while HTTP has reasonably well-defined terms and concepts for resources and messages, it has no clearly defined term to describe the result of applying an operation to a resource. This might seem like a mere terminology quibble, but the lack of such a term, and the failure to recognize the concept's importance, has led to a several difficult problems.

In particular, what does an HTTP cache entry store? Clearly not the resource itself (think of a CGI-generated resource). Not a Web “document,” since these are often composites of multiple resources with differing cachability properties. Instead, HTTP caches are currently defined as storing “response messages.” (I.e., an HTTP cache entry does not store what a resource is; it stores what the resource says.) As a result, it is difficult to define precisely what an HTTP cache must do in many circumstances, since the same resource could say two different things in response to two apparently identical requests. The lack of a clear formal specification for caching causes implementors to make guesses. This leads to non-interoperability, because content providers cannot predict what caches do.

It also makes it very hard to extend the protocol to handle partial updates (e.g., *delta encoding*) or even to define precisely how to combine existing HTTP/1.1 features (e.g., the ability to request a *range* of bytes and also to apply compression). The current model does not even provide a useful framework to discuss these questions.

3. A better model

We could solve these problems by adding a new data type, the *instance*. One can think of an instance as a complete snapshot of the current result of applying a GET to the resource. The instance can then be the input to a series of *instance manipulations*, which can include range selection, delta encoding, and compression.

In this model, HTTP cache entries are defined to store instances (or partial instances). An entity tag is tied to an instance, because it must be assigned prior to any instance manipulations. It is clearly not tied to the “entity” (and would better have been called an “instance tag”). Therefore, a cache can tell that two partial pieces of the same instance may be combined, because they have the same entity tag.

The implications of the new model (necessary protocol changes; the ability to more rigorously define existing and new HTTP features) require a longer writeup. (See `research.compaq.com/wrl/people/mogul/hotos8`). But it should be clear that the long-term success of a protocol such as HTTP depends on clear definitions that address distributed-systems issues, and on a better dialog between protocol designers and operating systems people.

Position Summary: The *Conquest* File System—Life after Disks

An-I A. Wang, Peter Reiher, and Gerald J. Popek[✧]
Computer Science Department
University of California, Los Angeles
{awang, reiher, popek}@fmg.cs.ucla.edu

Geoffrey H. Kuenning
Computer Science Department
Harvey Mudd College
geoff@cs.hmc.edu

The cost of paper and film has been a critical barrier to cross for a storage technology to achieve wide deployment and better economy of scale. By 2003, the declining cost of persistent RAM (e.g., battery-backed DRAM) will break this barrier, signifying the arrival of the persistent-RAM-based storage era.

Persistent RAM will not fully replace disks for many years. However, as RAM becomes cheap, memory can assume more roles of file systems. In particular, by 2005 high-end desktops can afford to be equipped with 4 to 10 Gbytes of persistent RAM for storage; this is sufficient to deliver nearly all aspects of file-system services, with the single exception of high-capacity storage.

The *Conquest* file system is designed to provide a transition from disk- to persistent-RAM-based storage. Initially, we assume 2 to 4 Gbytes of persistent RAM and the popular single-user desktop environment. Unlike other memory file systems, *Conquest* can incrementally assume more responsibility for in-core storage as memory prices decline. The *Conquest* approach realizes most of the benefits of persistent-RAM-based file systems before persistent RAM becomes cheaply abundant. *Conquest* also benefits from the removal of disks as the primary storage by identifying disk-related complexities and isolating them from the critical path where possible.

Unlike cache, which treats main memory as a scarce resource, *Conquest* anticipates the abundance of cheap persistent RAM. *Conquest* uses disk to store only the data well suited for disk characteristics. Reducing the range of access patterns and characteristics anticipated by the file system translates into simpler disk optimizations.

Our initial *Conquest* implementation uses core memory to store all metadata, small files (currently based on a size threshold), executables, and dynamically linked libraries, leaving only the content of the large files on disk. All accesses to in-core data and metadata incur no data duplication or disk-related overhead, and executions are in-place. For the large-file-only disk storage, we can use a larger access granularity to reduce the seek-time overhead. Because most accesses to large files are sequential, we can relax many historical disk design constraints, such as complex layout heuristics intended to reduce fragmentation or average seek times.

Conquest also speeds up computing by allowing easy reuse of previously computed results. With an expanded API, *Conquest* allows direct storage of runtime data structures, states, or even processes that interact with the environment in constrained ways. Unlike memory-mapped files, storing runtime states under *Conquest* requires no compaction or alignment to page boundaries, which benefits many data representations. Direct storage of runtime states relieves developers of the need for serialization and deserialization. Applications can also take advantage of storing runtime data in the most appropriate form for processing. For example, network applications can store outbound data in the format of network packets to bypass both disk-to-memory and memory-to-network translations.

Storing data in core inevitably invites the question of reliability and data integrity. However, conventional techniques of sandboxing, access control, checkpointing, fsck, and object-oriented self-verification still apply. For example, *Conquest* still needs to perform frequent system backups. *Conquest* uses common memory protection mechanisms by having a dedicated memory address space for storage (assuming a 64-bit address space). A periodic fsck is still necessary, but it runs at memory speed. We are also exploring the object-store approach of having a "typed" memory area, so a pointer can be verified to be of a certain type before dereferencing.

Various areas of *Conquest* are under investigation. Memory under *Conquest* is a shared resource among execution, storage, and buffering for disk access. Finding the "sweet spot" for system performance requires both modeling and empirical investigation. The ability for *Conquest* to store runtime states has the flavor of wide-address-space computing, which can be applied and extended to the distributed environment and database systems.

The *Conquest* prototype is operational under Linux 2.4.2. It is POSIX compliant and supports both in-core and on-disk storage. The source consists of 3,800 lines of kernel code, 1,400 lines of file-system-creation code, and 3,600 lines of testing code. Initial deployment and performance measurements are under way.

[✧] Gerald Popek is also associated with NetZero, Inc.

Position Statement: Supporting Coordinated Adaptation in Networked Systems

Patrick G. Bridges Wen-Ke Chen
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Matti A. Hiltunen Richard D. Schlichting
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932

While adaptation is widely recognized as valuable, adaptations in most existing systems are limited to changing execution parameters in a single software module or on a single host. Our position is that the true potential of adaptation can only be realized if support is provided for more general solutions, including adaptations that span multiple hosts and multiple system components, and *algorithmic adaptations* that involve changing the underlying algorithms used by the system at runtime. Such a general solution must, however, address the difficult issues related to these types of adaptations. Adaptation by multiple related components, for example, must be coordinated so that these adaptations work together to implement consistent adaptation policies. Likewise, large-scale algorithmic adaptations need to be coordinated using graceful adaptation strategies in which as much normal processing as possible continues during the changeover. Here, we summarize our approach to addressing these problems in Cactus, a system for constructing highly-configurable distributed services and protocols [2].

When multiple related system components can adapt to changes in the system state, the adaptations performed by these components must be coordinated to achieve a consistent adaptation policy. To achieve this, we have implemented an *adaptation controller* architecture that is responsible for making adaptation decisions for related adaptive components. Adaptation policies are specified on a component-by-component basis using sets of fuzzy logic rules, and then composed along with rules to coordinate the actions of different components to form a single controller. The challenge, of course, is designing a set of fuzzy rules that reflect the best adaptation strategies for a given situation.

Even when coordinated adaptation decisions are made, large-scale algorithmic adaptations still present a difficult challenge. Without special provisions, for example, an adaptive system may be unable to process normal application traffic while it is changing between different algorithms. To alleviate this problem, we have designed and implemented a *graceful adaptation protocol* that coordinates changes across hosts and gracefully switches between algorithmic alternatives on each host. This protocol uses agreement, barrier synchronization, and message tagging to ensure that hosts reach consistent adaptation decisions and change between alternative algorithms with minimal disruption.

These techniques are being prototyped using Cactus, a design and implementation framework for constructing configurable services in networked systems. The graceful adaptation protocol and adaptation controller are currently being prototyped separately using different versions of Cactus. The controller is being implemented using the C version of Cactus 2.0. The initial focus is on coordinating layers for a test configuration consisting of a streaming video application layered on a configurable transport protocol, CTP [3]. Initial experimental results suggest that the controller is indeed successful in coordinating adaptation between multiple components. An initial version of the graceful adaptation protocol has been completed using the C++ version of Cactus 1.1. Preliminary experimental results using an adaptive group communication service suggest that the protocol does indeed provide a graceful transition from one adaptation aware module to another, and demontrate the overall value of adaptive strategies. Further details on the graceful adaptation protocol can be found in [1].

*This work supported in part by DARPA under grant N66001-97-C-8518 and NSF under grants ANI-9979438 and CDA-9500991.

References

[1] W.-K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, Phoenix, AZ, Apr 2001.

[2] M. Hiltunen, R. Schlichting, and G. Wong. Cactus system software release. http://www.cs.arizona.edu/cactus/software.html, Dec 2000.

[3] G. Wong, M. Hiltunen, and R. Schlichting. CTP: A configurable and extensible transport protocol. In *Proceedings of the 20th Annual Conference of IEEE Communications and Computer Societies (INFOCOM 2001)*, Anchorage, Alaska, Apr 2001.

Position Summary. Middleware for Mobile Computing: Awareness vs. Transparency

Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo
Department of Computer Science
University College London
Gower Street, London WC1E 6BT, UK
{L.Capra|W.Emmerich|C.Mascolo}@cs.ucl.ac.uk

Abstract

Middleware solutions for wired distributed systems cannot be used in a mobile setting, as mobile applications impose new requirements that run counter to the principle of transparency on which current middleware systems have been built. We propose the use of reflection capabilities and meta-data to pave the way for a new generation of middleware platforms designed to support mobility.

1. The Rationale

The increasing popularity of wireless devices, such as mobile phones, personal digital assistants and the like, is enabling new classes of applications that present challenging problems to application designers. These devices face temporary loss of network connectivity when they move; they discover other hosts in an ad-hoc manner; they are likely to have scarce resources, such as low battery power, slow CPU speed and small amounts of memory; and they are required to react to frequent and unannounced changes in the environment, e.g. variable network bandwidth.

Middleware technologies [2] have been designed and successfully used to support the development of stationary distributed systems built with fixed networks. Their success has been mainly due to their ability of making distribution *transparent* to both users and software engineers, so that systems appear as single integrated computing facilities.

However, completely hiding the implementation details from the application becomes both more difficult and makes little sense in a mobile setting. Mobile systems need to quickly detect and adapt to drastic changes happening in the environment. A new form of *awareness* is needed, as opposed to transparency, to allow application designers to *inspect* the execution context and *adapt* the behaviour of middleware accordingly.

2 Research Directions

We believe that reflection and metadata can be successfully exploited to develop middleware targeted to mobile settings. Through metadata we obtain separation of concerns, that is, we distinguish what the middleware does from how the middleware does it. Reflection is the means that we provide to applications in order to inspect and adapt middleware metadata, that is, influence the way middleware behaves, according to the current context of execution.

We have developed XMIDDLE [3], a middleware for mobile computing that focuses on synchronization of replicated XML documents. In order to enable application-driven conflict detection and resolution, XMIDDLE supports the specification of conflict resolution policies through meta-data definition using XML Schema.

The following step has been the definition of a global model for the design of mobile middleware systems, based on the principles mentioned above. In [1], we have discussed a reflective conceptual model and a reflective architecture of middleware systems targeted to support mobile applications that call for context-awareness, where by context we do not mean only location but everything in the physical environment that can influence the behaviour of the application, such as memory and battery power.

References

[1] L. Capra, W. Emmerich, and C. Mascolo. Reflective Middleware Solutions for Context-Aware Applications. Technical Report RN/01/12, UCL-CS, 2001. Submitted for Publication.

[2] W. Emmerich. Software Engineering and Middleware: A Roadmap. In *The Future of Software Engineering - 22nd Int. Conf. on Software Engineering (ICSE2000)*, pages 117–129. ACM Press, May 2000.

[3] C. Mascolo, L. Capra, and W. Emmerich. XMIDDLE: A Middleware for Ad-hoc Networking. 2001. Submitted for Publication.

Position Summary: Separating Mobility from Mobile Agents

Kåre J. Lauvset , Kjetil Jacobsen and Dag Johansen
Dept. of Computer Science, University of Tromsø, Tromsø, Norway

Keith Marzullo
Dept. of Computer Science, University of California San Diego, La Jolla, USA.

Mobile agents, like *processes*, are separate units of concurrent execution. They differ in how they view the processor upon which they run. For processes, the processor is abstracted away: each process can consider itself to be running on an independent virtual machine. For mobile agents, the processor upon which they run is not abstracted away: it is a first-class entity that is under program control. A mobile agent can move from one processor to another in order to profit from the details - such as fast access to local data, use of computational resources and I/O devices, and so on - of the new processor.

The reasons for using mobile agents are well-known: moving computation to data to avoid transferring large amounts of data; supporting disconnected operation by, for example, moving a computation to a network that has better connectivity; supporting autonomous distributed computation by, for example, deploying a personalized filter near a real-time data source. Many mobile agent systems have been constructed and are in the public domain. But, despite these well-known advantages and widely available software, mobile agents are not yet being used as a common programming abstraction.

We have been working since 1993, under the name of TACOMA, on operating system support and application of mobile agents. We have addressed issues including fault-tolerance, security, efficiency, and runtime structures and services. We have built a series of mobile agent middleware systems and evaluated them by building realistic and deployed applications. We have found that mobile agents are especially useful for large-scale systems configuration and deployment, system and service extensibility, and distributed application self-management.

The programming model TACOMA supports has changed over these years to reflect our experience with writing real applications. Like other mobile agent systems, TACOMA started with a programming model that resembled the characterization given above of mobile agents being processes with explicit control over where they execute. We call this the *traditional model* of mobile agents. Using the traditional model leads to several problems including: the complexity of code that contains an explicit and dynamic trajectory; the overhead of implicit state capture; and the temptation to support only a single programming language (which is typically Java, whose use presents yet other problems).

This work was supported by NSF (Norway) grants No. 112578/431 and 126107/431 (Norges Forskningsråd, DITS program).

More fundamentally, we have found that mobile agents are best thought of as one of several tools used together to build distributed applications. A distributed application has, in addition to its *function*, nonfunctional aspects such as its deployment, monitoring, adaptation to a changing runtime environment, and termination. We call this the *factored* model of distributed applications. This model separates the functional aspect of the application from its *mobility* and *management* aspects. Mobile agent platforms can be used to implement the mobility aspect, and the mobile agents themselves to implement the management aspects.

Legacy and COTS software constitute a significant portion of the function of many real-world distributed applications. The mobility aspect provides the mechanisms and structures necessary for deploying the function and for its adaptive reconfiguration. The management aspect manages both function and mobility at a higher level. More specifically, it implements polices for *when*, *where*, and *how* to execute the function. Examples of management policies of applications include fault-tolerance, server cloning to accommodate increased demand, and invoking security countermeasures in response to intrusion detection alarms.

We have redesigned TACOMA to only directly provide the mobility aspect of distributed applications. This version, called νTOS, provides less than full-fledged mobile agent systems and more than remote execution facilities such as `ssh`, `rsh` and `rexec`. It is rather small: it consists of approximately 90 lines of Python code. Despite its diminutive size, it can be used to implement itinerant mobile agents that move over encrypted network channels.

We have used νTOS to construct some simple but realistic distributed applications such as a parallel image renderer based on COTS components. We are now building a personal computational grid called *Open Grid*. Doing so is making concrete issues of deployment, security, fault-tolerance, and adaptation.

Further details on the νTOS project can be found at **http://tacoma.cs.uit.no**.

Position Summary: The Importance of Good Plumbing

Reconsidering Infrastructure in Distributed Systems

Andrew Warfield and Norm Hutchinson
University of British Columbia
{andy,norm}@cs.ubc.ca

The fundamental abstractions and mechanisms used to deliver communications within distributed systems have remained essentially unchanged for a considerable time. With very few exceptions, operating systems implement a simple, socket-based approach to communications in which the host's involvement in a particular communication stream ends at the network interface. TCP/IP provides an environment in which the notion of a data stream does not actually exist within the network, but rather is an abstraction made by 'connected' endpoints.

Above the network, significant developments have been made to advance the state of distributed systems technology. Many sizeable middleware infrastructures have been developed and are actively being used in the construction of commercial distributed applications. Despite the benefits provided by these packages, they remain dependent on an insufficient infrastructure, which may be considered according to three fundamental flaws: 1. TCP/IP simply does not provide adequate network functionality for distributed systems. The shortcomings of the protocol provide a list of ongoing research problems including mobility, quality of service, and group collaboration. 2. The primary OS abstraction for a stream, the socket, is inflexible and represents a poor coupling between the network and the OS. 3. Remote procedure calls, which are the *de facto* approach to distributed invocation almost universally attempts to hide the network, obscuring failures (and features) from overlying applications.

Within the network TCP/IP also proves problematic. Considerable research efforts exist in the ongoing attempts to carry a legacy protocol well beyond the scope of its initial design. Traffic management, congestion control, and resource reservation all remain largely unsolved problems due to the difficulties of managing data streams within the network.

The existing infrastructure successfully provides a servicable network. We feel that the ongoing functionality of the existing system explains the thrust of research towards addressing individual deficiencies rather than addressing the system as a whole. However, there is an opportunity to realize substantial benefits through the development on an abstraction that is understood and supported by both the OS and the network routers. Our work to date has been in the design and development of a stream-centric model for communications which we have called the *flow* [1]. A flow is a uniquely named, message-based, multicast communications stream. Flows are named by FlowIDs which represent a collection of resources used to provide a communications stream in the same manner that process IDs represent resources associated with a computational task. By providing distinct names for these multicast streams, services become decoupled from network endpoints. This single property provides sweeping benefits for mobility, fault tolerance, and resource location.

In addition to naming, we have implemented three properties of flows which we feel are beneficial to distributed applications. First, flow messages are *banded*. Each flow has a label space of 128 bands within which messages may be sent. This allows an external separation of concerns for messages within a stream, and also provides an effective means to tie administrative and fault messages to a stream from points within the network. Second, flows support a notion of *locality*. Locality acts as an extension of TTL that allows message transmission to be scoped according to criteria such as geographic area, available bandwidth, or latency. Finally, flow messages are delivered from the network to client-defined queues. These queues allow local delivery options, such as drop strategy and message ordering, to be defined and implemented at the application. Queues may be attached to various bands of a flow, providing a great degree of flexibility in how and where the message stream is used.

We have finished an initial implementation of flows as a network middleware and have become convinced that they are an interesting and useful communications abstraction. We are currently extending our definition to allow the recursive embedding of flows, providing a hierarchy of streams. We feel that this property will be very beneficial both in terms of traffic management and software design. Additionally, we are investigating methods of typing individual data streams in order that the format of their content may be advertised to devices along the transmission path. Our investigation of these two properties coincides with efforts to implement flows efficiently at the network layer.

References

[1] Flows project web page. www.cs.ubc.ca/spider/andy/flows/.

We would like to acknowledge Alexander Fraser and Glenford Mapp at AT&T research for their support and ideas, in particular that of recursive flows.

Position Summary - Towards Global Storage Management and Data Placement

Alistair Veitch, Erik Riedel, Simon Towers and John Wilkes
Hewlett Packard Laboratories
{aveitch, riedel, stowers, wilkes}@hpl.hp.com

As users and companies increasingly depend on shared, networked information services, we continue to see growth in data centers and service providers. This happens as services and servers are consolidated (for ease of management and reduced duplication), while also being distributed (for fault-tolerance and to accommodate the global reach of customers). Since access to data is the lifeblood of any organization, a global storage system is a core element in such an infrastructure. Based on success in automatically managing local storage, we believe that the key attribute of such a system is the ability to flexibly adapt to a variety of application semantics and requirements as they arise and as they change over time. Our work has shown that it is possible to automatically design and configure a storage system of one or more disk arrays to meet a set of application requirements and to dynamically reconfigure as needs change, all without human intervention. Work on global data placement expands the scope of this system to a world of distributed data centers.

Data location

Ensuring that data is available in the right location is a key challenge as data and applications go global. Due to speed of light and congestion, network performance will always be a bottleneck. The system will have to transparantly migrate data to have the data that each application needs co-located with the servers that are currently operating on it. Whether data follows a particular user as they travel around the globe; supports a global design team in its daily work; or handles customer data or inventory for a global corporation, the individual data "shadows" of all types of users and applications must be supported efficiently. Such a system can be viewed as a network of "cache" devices – each data center provides a pool of storage that at any one time is caching a subset of the global store. The key problem is deciding when to move data from one to another, when to keep multiple copies, and how many copies to keep – automating data placement such that load is balanced both within and across data centers.

Data replication and consistency

For many applications, the most efficient solution will be to have multiple replicas of the same data. Along with the core requirement of availability in the event of local failures, the makes it necessary to store the same data in multiple global locations and keep it consistent. The ability to adapt consistency levels within the storage system to the varying requirements of individual applications is a key enabler for global data placement. Ideally this would be done transparently, without changes to existing application code, and much of the necessary information and flexibility is available even with storage interfaces designed for local resources.

Mechanisms for maintaining consistency across global sites range from expensive pessimistic approaches with multiple round trips of locking overhead, to low latency optimisitic approaches that allow occasional inconsistencies or require rollback. To evaluate the cost of such schemes, we analyzed traces for a number of applications, including email, software development, databases, and web browsing. At the storage level - after cache accesses have been eliminated - the results do not seem very promising. A high fraction of requests are updates, the ratio of metadata to data is high, and a high fraction of requests are synchronous. However, considering individual applications in isolation, these metrics vary widely, making adaptive consistency that uses different mechanisms as appropriate attractive.[1] We have also begun to quantify how much sharing takes place, and see the fraction of "hard" sharing in a large store is promisingly low.[2]

Security

Data must be secure, especially in a system where facilities are shared amongst many different organizations. This requires strong authentication, authorization and encryption mechanisms, none of which are necessary in the context of local storage systems. Initial analysis shows that large stores quickly encompass large numbers of objects to be protected and principals requiring authentication, posing scalability problems. However, if we consider the number of objects that an individual user handles - their "shadow" on the entire store - and the commonality among access patterns to these objects, the scope of security quickly becomes more tractable. The question then becomes which levels of abstraction to provide for different classes of users and data.

System management and control

Our local management system can determine appropriate placement in the local case with local information. We anticipate a hierarchy in the global setting, with some optimization best done within the data center, and a more global view controlling movement across centers, all informed by the "shadow" that supports a particular coherent data set or user. Such a system must operate at a range of time scales and granularities, and will critically depend on the ability to accurately and efficiently model and predict the behavior of all the components within and the links across the system.

1. Technical memo *HPL-SSP-2001-1*, HP Labs, March 2001.
2. "When local becomes global" *20th IPCCC*, April 2001.

Position Summary: Towards Zero-Code Service Composition

Emre Kıcıman, Laurence Melloul, Armando Fox
{emrek, melloul, fox}@cs.stanford.edu
Stanford University

Zero-Code Composition For many years, people have been trying to develop systems from modular, reusable components[2]. The ideal is *zero-code composition*: building applications out of components without writing any new code. By investigating zero-code composition, our goal is to make composition easy enough to be of practical use to systems researchers and developers. We are focusing on identifying and removing systemic impediments to composition, and on exploiting composition to achieve system-wide properties, such as performance, scalability, and reliability.

Impediments to Composition Today, even when components are designed to be reused, software developers have difficulties composing them into larger systems. We believe the problem lies with the methods and fundamental abstractions used to package and compose components. For example, abstractions such as function calls work well when building small systems, however, they actually enforce properties on components that significantly impede composition and reuse generally. These impediments can be classified into two categories:

- Control flow impediments relate to the ordering of execution of components [1]. For example, two components cannot be used together when they make different assumptions about the sequencing of computation and passing of control between them.
- Interface impediments occur when components contain statically bound information about other components' interfaces, such as method names, data types and orderings, and communication protocols. However, this information will be invalid in different contexts, and will prevent the component from being reused in an arbitrary composition.

A Data Flow Composition Model To avoid these impediments, we advocate that compositions be built of autonomous *services* connected together in a data flow network. Autonomous services avoid control model mismatches by keeping their own locus of control. Interface impediments are avoided by allowing services to only name their own input and output ports. The data flow model is defined by the data dependencies between services, and provide an explicit description of the composition. A generic run-time system handles passing data from one component's output port to another's input port according to the data flow description of the composition.

Explicitly exposing the structure of applications enables systematic inspection, manipulation, and augmentation of applications. We can inspect the data flow composition for bottlenecks in performance, and strategically move, replicate or replace parts of a composition which are performing poorly. For example, one simplistic strategy is to dynamically place caches around strings of expensive services in a composition to improve performance. We can similarly manipulate a composition to increase its fault-tolerance, scalability and reliability.

Current Status We have implemented a prototype composition architecture [3], and are beginning to implement dynamic manipulations of compositions, and explore the relationships between these manipulations, system-wide properties and various service attributes such as determinism or idempotency.

References

[1] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of International Conference on Software Engineering '95*, Seattle, April 1995.

[2] P. W. Gio Wiederhold and S. Ceri. Towards megaprogramming: A paradigm for component-based programming. *Communications of the ACM*, (11):89–99, 1992.

[3] E. Kıcıman and A. Fox. Using dynamic mediation to integrate cots entities in a ubiquitous computing environment. In *Handheld and Ubiquitous Computing (HUC 2000), Second International Symposium*, Sept. 2000.

Position Summary: Aspect-Oriented System Structure

Yvonne Coady, Gregor Kiczales, Michael Feeley,
Norman Hutchinson, Joon Suan Ong and Stephan Gudmundson
University of British Columbia

Operating system structure is important – it leads to understandable, maintainable, 'pluggable' code. But despite our best efforts, some system elements have been difficult to structure. We propose a new analysis of this problem, and a new technology that can structure these elements.

Primary functionality in system code has a well defined structure as layered abstractions. Other key elements naturally defy these structural boundaries – we say that they *crosscut* the layered structure. For example, prefetching for mapped files involves coordinated activity at three levels: predicting the pattern of access and allocating pages in the VM layer, determining the contiguity of blocks in the disk layer, and reconciling the costs of retrieval in the file system layer. Because of its inherent crosscutting structure, the implementation of prefetching is scattered through the primary functionality in each of the layers involved (Figure 1a).

In FreeBSD v3.3, prefetching for mapped files is approximately 265 lines of code, grouped into 10 different clusters, scattered over 5 functions from VM and FFS alone. Dynamic context, such as flagging VM-based requests, is passed as parameters from high level functions down through lower ones. Portions of prefetching code violate layering by accessing high level abstractions from lower level functions, such as freeing and page-flipping VM pages from within FFS. In this form, there is no structure to the implementation of prefetching – it is hard to understand, hard to maintain, and certainly hard to unplug.

Aspect-oriented programming (AOP) [3, 2] uses linguistic mechanisms to support the separation of crosscutting elements, or *aspects* of the system, from primary functionality. Aspects declare code to execute *before*, *after* or wrapped *around* existing primary function calls, within the execution flow of other function calls, and with access to specific parameters of those calls. AOP improves the comprehensibility of crosscutting elements in two ways: it allows small fragments of code that would otherwise be spread across functions from disparate parts of the system to be localized; and it makes the localized code more coherent, because interaction with primary functionality is declared explicitly and within shared context.

We have developed a proof-of-concept AOP implementation of prefetching in FreeBSD [1]. In our implementation, we have been able to modularize prefetching. The

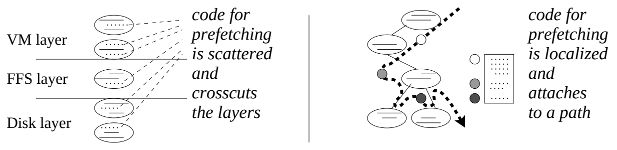

VM layer
FFS layer
Disk layer
code for prefetching is scattered and crosscuts the layers
code for prefetching is localized and attaches to a path
(a) Original structure (b) Aspect−oriented structure

Figure 1: Prefetching and primary functionality.

internal structure of the prefetching code and its interaction with the overall VM and FFS activity are explicitly defined as a sequence of activities that occur at well defined points *along a page-fault path*, rather than being broken into layers (Figure 1b). The AOP implementation is designed to allow us to see precisely how low-level prefetching code acts in service of high-level prefetching code. Primary page fault handling functionality no longer includes prefetching code, nor does it explicitly invoke prefetching functionality.

In the AOP implementation, one aspect captures how prefetching plays out over page-fault handling for sequentially accessed mapped files: first the page map is locked and pages are pre-allocated according to a prediction, then these and possibly other pages are synchronously brought into the file buffer cache and page-flipped where appropriate, and finally further pages may be asynchronously prefetched into the cache. We can clearly see how this differs from the prefetching aspect for the non-sequential case, where pages may be de-allocated if it is not cost-effective to retrieve them, the file buffer cache is not involved, and further asynchronous prefetching is not applied. Structured this way, prefetching gains context, is more tractable to work with, and is even unpluggable.

We believe that other key elements of operating systems are crosscutting and that their unstructured implementation is excessively complex. We are currently developing AspectC, and plan to use it to further explore the structure of elements such as paging in layered system architectures, consistency in client-server architectures, and scheduling in event-based architectures.

References

[1] AspectC. www.cs.ubc.ca/labs/spl/aspects/aspectc.html.

[2] AspectJ. www.aspectj.org.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

Position Summary. An Streaming Interface for Real-Time Interprocess Communication

Jork Löser Hermann Härtig Lars Reuther
Dresden University of Technology, Germany

Abstract

Timely transfer of long, continuous data streams and handling data omission are stringent requirements of multimedia applications. To cope with these requirements, we extend well known mechanisms for inter-address-space data transmission, such as zero-copy and fast IPC, by the notion of time. Therefore, we add a time track to data streams and add mechanisms to limit the validity of data. For cases of overload we add notification and revocation techniques.

Inadequacies of current IPC mechanisms Efficient interprocess communication schemes for long data transfers in non real-time applications [2, 5, 6] address the problem of copy avoidance by using shared memory. But they do not cover issues of time, such as data loss due to CPU shortage.

In hard real-time systems data loss does not occur, because the entire system is designed for the worst case regarding resource needs during execution. However, this results in poor resource utilization and is therefore not practical. Designing the system for the average case improves the overall resource utilization [4, 1], at the cost of quality. Resource shortages during execution can happen and lead to data loss then. To cope with this, data loss should be expressed at the communication layer.

Tolerating occasional resource shortages allows multiple applications to share resources, e.g. memory pools for communication buffers. This in turn requires retracting these resources in overload situations and hence must be supported.

The DROPS Streaming Interface The DROPS Streaming Interface (DSI) is our approach to a real-time interprocess communication subsystem. It defines a user-level timed packet-oriented zero-copy transport protocol between real-time components. The data flows are represented by streams with assigned traffic specifications.

For actual data transfer, DSI uses a consumer-producer scheme on a ring buffer containing packet descriptors. The packet descriptors provide an indirection for data access and allow a flexible use of the shared data buffers.

The specifications of streams in DSI base on jitter-constrained periodic streams (JCS) [3]. JCS allow to estimate the resources needed for a given stream, e.g. buffer capacity. On stream creation, DSI uses these estimates when establishing the shared data buffers. If the communication peers behave conforming to their specification, no buffer shortage and no data loss occurs.

In cases of resource shortages, the communication peers cannot always meet their traffic specification. To cope with this at the sender, DSI adds timestamp information to the transferred data packets. To cope with resource shortage at the receiver, DSI limits the validity of data by time. This means, the data packets produced at the sender will expire after a certain time, even they were was not consumed by the receiver. For both techniques, DSI uses virtual time, which is assigned to and stored together with each data packet. The virtual time corresponds to the position of the data in the entire stream. The mapping of virtual time to real-time is the responsibility of the communication partners.

A problem arises when the expiration of data must be enforced. It must not impose any blocking, but the sender must know for sure, that the receiver will not continue to access old data anymore. Thus sending a message to the receiver and waiting for an answer is not an option. To enforce the expiration of data DSI uses virtual memory techniques. For this, the sender can request retraction of shared memory pages from the receiver. When the receiver noticed the retraction, it requests re-establishing the memory mapping. This allows an immediate notification without blocking.

References

[1] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1173, Sept. 1990.

[2] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 189–202, Asheville, NC, Dec. 1993.

[3] C.-J. Hamann. On the quantitative specification of jitter constrained periodic streams. In *MASCOTS*, Haifa, Israel, Jan. 1997.

[4] H. Härtig, L. Reuther, J. Wolter, M. Borriss, and T. Paul. Cooperating resource managers. In *Fifth IEEE Real-Time Technology and Applications Symposium (RTAS)*, Vancouver, Canada, June 1999.

[5] F. W. Miller, P. Keleher, and S. K. Tripathi. General data streaming. In *19th IEEE Real-Time Systems Sysmposium (RTSS)*, Madrid, Spain, Dec. 1998.

[6] V. S. Pai, P. Druschel, and W. Zwanenpoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, Feb. 2000.

Position summary: *eOS* – the dawn of the resource economy

John Wilkes, Patrick Goldsack, G. (John) Janakiraman,
Lance Russell, Sharad Singhal, and Andrew Thomas

Hewlett-Packard Laboratories, Palo Alto, CA 94304
{john_wilkes, patrick_goldsack, john_janakiraman, lance_russell, sharad_singhal, andrew_thomas}@hpl.hp.com

We believe that achieving the benefits of a resource economy, which supports the execution of services wherever and whenever is most convenient, cost-effective, and trustworthy, represents the next big computer systems research opportunity. That is, the emphasis in the operating research community should move away from extracting a few more percentage points of speed from individual computing resources, and focus instead on how to size, provision, and manage those resources to serve the needs of a rapidly diversifying set of services. HP Laboratories are embarking on a major endeavor to pursue this, and are actively seeking research partners to collaborate with us.

The vision

Scene: a Corporate IT director's office, the day before a company board meeting. The COO, Chris, knocks on the door, and comes in without waiting.

Chris: Jean: what's all this about us getting billed for computing resources in Singapore? How am I going to explain that? We don't have a facility there! What's going on here?

Jean: Calm down! It's ok—really.

Chris: Not good enough. You know I have to give a bullet-proof answer tomorrow … so you don't have much time.

Jean: Ok, ok. Do you remember the end of the last month? We had the R&D guys needing to do their protein shape calculations to meet an FDA deadline …

Chris: Yes—but weren't they using some cheap compute cycles in Prague that you'd found for them?

Jean: … and then the marketing team wanted a new computer-generated-graphics commercial in time for Comdex that included film of our Malaysian manufacturing plant as a backdrop …

Chris: Yes—but you said that wouldn't be a problem …

Jean: … and the financial results that you are holding, by the look of it, needed some decision analysis that we don't usually do, in collaboration with our Japanese partners …

Chris: Yes—but you told me …

Jean: Please—if I could finish?

Chris: Sorry. It's been a bit hectic today.

Jean: No way could the Prague facility keep both the chemists and the video team happy: both the computation needs and storage space requirements were way over the top. And the Malaysian plant has nothing, really, so it looked like it would cost us a fortune. … All because those geeks couldn't get their timing right.

Chris: [Sigh.] I must have told them a dozen times …

Jean: But then our eOS system discovered that we are co-buying storage space for the Malaysian lot-failure analysis data with our Hong Kong subsidiaries in Singapore; and it checked out the effects of migrating the data back to Prague and the computations to Singapore.

Chris: But wouldn't that have been a huge hassle to get right? Moving all that data?

Jean: Not at all—I didn't even find out until 2 days after it had happened!

Chris: What do you mean? You let it make a decision like that?

Jean: Sure! It even reported that the average response time for our OLTP jobs were 30% better than usual—they usually get hammered by the decision support people at the end of the month. Probably because of the time-zone effects. If you look, I think you'll find we even saved money - we used to have a team of people doing this stuff, trying to keep one step ahead of the next wave of demands. They could never keep up, so the customers were always unhappy—and they were people who we couldn't really afford to have spending their time on that when there were more important things they could do for us, like rolling out new services.

Chris: But what about the users while things were being changed?

Jean: They hadn't even noticed! My biggest headache is our accounting systems: they make the resource location visible at your level—but nobody else cares.

Chris: You didn't even have to come up with this solution yourself?

Jean: Nope. I didn't do a thing. ***eOS did it all!***

What we're up to

The proliferation of computers and the Internet into all aspects of commerce and society is well under way. Many of the fundamental technical issues to do with the components of modern computing systems are either solved, or well in hand. It is our position that the next wave of innovation—and hence research opportunities—lies in the field of aggregating pools of computing resources in support of the explosion in scale, complexity, and diversity of computing services. The eOS program at HP Labs is aimed at removing the technical barriers to this happening.

eOS is not a single artifact: it is better thought of as a set of related research activities that are striving towards achieving the vision described above over the next few years. It is akin to other research efforts (e.g., Oceano, Grid, OceanStore) in large scale systems in its scope. The research focus in eOS is to discover methods to abstract and virtualize computing and storage resources and make them available upon demand at a global scale.

A fuller version of this paper is obtainable from the URL http://www.hpl.hp.com/personal/John_Wilkes/papers

Position Summary: Transport Layer Support for Highly-Available Network Services

Florin Sultan, Kiran Srinivasan, Liviu Iftode
Department of Computer Science
Rutgers University, Piscataway, NJ 08854-8019
{*sultan, kiran, iftode*}@*cs.rutgers.edu*

We advocate a transport layer protocol for highly-available network services by means of transparent migration of the server endpoint of a live connection between cooperating servers that provide the same service. The current connection-oriented transport layer protocol of the Internet (TCP) reacts to what it perceives as lost or delayed segments only by retransmitting to the same remote endpoint of the connection. TCP provides no means to alleviate a performance degradation (low throughput, many retransmissions etc.) caused by adverse factors like server overload or failure, or network congestion on a given path. At the same time, TCP creates an implicit association between the server contacted by a client and the service it provides. This is overly constraining for today's Internet service models, where the end user of a service is concerned more with the quality of the service rather than with the exact identity of the server.

We propose a transport protocol that *(i)* offers a better alternative than the simple retransmission to the same server, which may be suffering from overload or a DoS attack, may be down, or may not be easily reachable due to congestion, and *(ii)* decouples a given service from the unique/fixed identity of its provider. Our protocol can be viewed as an extension to the existing TCP, and compatible with it. To start a service session, the client establishes a TCP connection with a preferred server, which supplies the addresses of its cooperating servers, along with authentication information. At any point during the lifetime of the session, the server endpoint of the connection may migrate between the cooperating servers, transparent to the client application. The current and the new server hosts must *cooperate* by transferring supporting state in order to accommodate the migrating connection.

We assume that the state of the server application can be logically split among the connections being serviced, so that there exists a well-defined state associated with each service session. Transfer of this state ensures that a new server can resume service to the client in the presence of other concurrent service sessions. In addition to the associated application-level state, transfer of in-kernel TCP connection state reconciles the TCP layer of the new server with that of the client.

Our proposed solution provides a minimal interface to the OS for exporting/importing a per-connection application *state snapshot* by a server. The origin server executes the export operation in order to *(i)* define an execution restart point for the stateful service on the connection in case of its migration, and *(ii)* synchronize the service state (reached as a sequence of reads/writes on the connection) with the in-kernel TCP state. The new server executes the import operation to reinstate the connection at the restart point, and resumes service on it, transfering data without altering the TCP exactly-once semantics.

We intend to integrate this mechanism in a general migration architecture in which the *client* side TCP initiates connection migration, in response to various *triggers* that can reside either at the client or at the server(s). Triggers are events like a degradation in perceived traffic quality (on the client side), failure, DoS attack, a load balancing decision etc. (on the server side).

The features of our protocol are: *(i)* It is general and flexible, in that it does not rely on knowledge about a given server application or application-level protocol. *(ii)* It allows fine-grained migration of live individual connections, unlike a heavy-weight process migration scheme. *(iii)* It is symmetric with respect to and decoupled from any migration policy.

We have implemented an operational prototype of our protocol in FreeBSD. We are currently building several applications that can take advantage of the protocol, including a transactional database application with migration support.

Issues that we plan to address in the future are: explore and evaluate various migration trigger policies, evaluate the two options for connection state transfer (eager vs. lazy), develop support to implement fine-grained fault tolerance, and explore the performance tradeoffs of our scheme. More details can be found at our site: *http://discolab.rutgers.edu/projects/mtcp.htm*.

MANSION: A Room-based Multi-Agent Middleware

Guido J. van't Noordende, Frances M.T. Brazier, Andrew S. Tanenbaum, Maarten R. van Steen
Division of Mathematics and Computer Science, Faculty of Sciences
Vrije Universiteit, Amsterdam, The Netherlands
{guido,frances,ast,steen}@cs.vu.nl

Keywords: Middleware, Multi-Agent Systems, Distributed Systems, Mobile Agents, Virtual Environments

Abstract

In this paper we present work in progress on a worldwide, scalable multi-agent system, based on a paradigm of hyperlinked rooms. The framework offers facilities for managing distribution, security and mobility aspects for both active elements (agents) and passive elements (objects) in the system. Our framework offers separation of logical concepts from physical representation and a security architecture.

1 The Mansion Paradigm

Our framework consists of a world (or possibly multiple disjoint worlds), each containing a set of hyperlinked rooms. Each room contains agents and objects. At any instant, an agent is in one room, but agents can move from room to room and they can take objects with them.

In essence, a room forms a shared data-space for agents with regard to visibility. Agents can interact only with objects in the same room, but can send messages to agents anywhere in the world. However, normally an agent will do most of its business with other agents in the same room.

Entities in a room can be agents, objects, or hyperlinks. Each agent is a (possibly multithreaded) process running on one host. No part of the internal process state of an agent can be accessed from the outside by other agents. Objects are strictly passive: they consist of data and code hidden by an interface. Hyperlinks determine how rooms are connected.

Every world also has an *attic*. The attic contains global services and is directly accessible to agents in any room. Through the attic, an agent can obtain world-scoped information, for example, the topology (hyperlink layout) of a world, directory services, or a bulletin board service (e.g., for publishing agent information.)

An agent enters a world by entering a room. Once in an entry room, an agent may move to any other room to which that room is hyperlinked. Directly moving to internal rooms (behind an entry room) is not allowed; agents can only follow hyperlinks. Except for following hyperlinks, a mobile agent may also move to a different host. However, our framework also allows for remote access to rooms, so that immobile (static) agents may also use our system.

All mechansims for moving to rooms, obtaining (binary) interfaces to objects or for inter-agent communication, as well as security mechanisms are hidden inside the middleware layer of our system. Agents can in principle be written in any programming language. A world designer should provide support for this language in the middleware.

2 Examples

As an example of the Mansion paradigm, consider a world designed for buying and selling raw materials for industry. An entry room is set up where interested parties can obtain information about the products for sale. Hyperlinks from this room lead to rooms for specific products, such as ore, water, and electricity.

Agents for users that want to buy or sell certain products can be launched into the system and go to an appropriate room where they can meet other agents that offer or want products.

An offer may be negotiated, after which an agent can either return to its owner with the current offer, or communicate with other agents to try to negotiate a package deal (e.g., optimizing for the cheapest combination of ore, water, and electricity). Some global information such as up-to-date currency exchange rates, freight rates, etc., may be available to all agents through the attic.

In short, the Mansion paradigm replaces the World Wide Web paradigm of a collection of hyperlinked documents that users can inspect with that of a collection of hyperlinked rooms in which agents can meet to do business.

Active Streams: An approach to adaptive distributed systems

Fabián E. Bustamante, Greg Eisenhauer, Patrick Widener, Karsten Schwan, and Calton Pu[*]
College of Computing, Georgia Institute of Technology
{fabianb, eisen, pmw, schwan, calton}@cc.gatech.edu

An increasing number of distributed applications aim to provide services to users by interacting with a correspondingly growing set of data-intensive network services. Such applications, as well as the services they utilize, are generally expected to handle dynamically varying demands on resources and to run in large, heterogeneous, and dynamic environments, where the availability of resources cannot be guaranteed *a priori* — all of this while providing acceptable levels of performance.

To support such requirements, we believe that new services need to be customizable, applications need to be dynamically extensible, and both applications and services need to be able to adapt to variations in resource availability and demand. A comprehensive approach to building new distributed applications can facilitate this by considering the contents of the information flowing across the application and its services and by adopting a component-based model to application/service programming. It should provide for dynamic adaptation at multiple levels and points in the underlying platform; and, since the mapping of components to resources in dynamic environment is too complicated, it should relieve programmers of this task. We propose *Active Streams* [1], a middleware approach and its associated framework for building distributed applications and services that exhibit these characteristics.

With Active Streams, distributed systems are modeled as being composed of *applications*, *services*, and *data streams*. Services define collections of operations that servers can perform on behalf of their clients. Data streams are sequences of self-describing application data units flowing between applications' components and services. They are made *active* by attaching application- or service-specific location-independent functional units, called *streamlets*. Streamlets can be obtained from a number of locations; they can be downloaded from clients or retrieved from a streamlet repository. The Active Streams C-based framework employs dynamic code generation in order to insure that streamlets can be dynamically deployed and efficiently executed across heterogeneous environments. Application evolution and/or a relatively coarse form of adaptation is obtained by the attachment/detachment of streamlets that operate on and change data streams' properties. Finer grain adaptation involves tuning an individual streamlet's behavior through parameters remotely updated via a push-type operation, and by re-deploying streamlets to best leverage the available resources over the datapath.

Active Streams are realized by mapping streamlets and streams onto the resources of the underlying distributed platform, seen as a collection of loosely coupled, interconnected computational units. These units make themselves available by running as Active Streams Nodes (ASNs), where each ASN provides a well-defined environment for streamlet execution. Active Streams applications rely on a push-based customizable resource monitoring service (ARMS) to collect resource information and trigger adaptation. Through ARMS, applications can select a subset of the data made available by distributed monitors. These data streams can be integrated to produce application-specific views of system state and decide on possible adaptations.

As is common in distributed systems, a directory service provides the "glue" that holds the Active Streams framework together. The dynamic nature of most relevant objects in Active Streams makes the passive client interfaces of classical directory services inappropriate. Thus, the Active Streams framework includes a *proactive* directory service with a publish/subscribe interface through which clients can register for notification on changes to objects currently of interest to them. The levels of detail and granularity of these notifications can be dynamically tuned by the clients.

The implementation of Active Streams is mostly complete, and we plan on making it available by December 2001.

References

[1] F. E. Bustamante and K. Schwan. Active Streams: An approach to adaptive distributed systems. Tech. report, College of Computing, Georgia Institute of Technology, Atlanta, GA, June 1999.

[*] Active Streams is part of the Infosphere Project, partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs, by NSF/CISE under the CCR and ANIR programs, and by Intel.

Position Summary. Smart Messages: A System Architecture for Large Networks of Embedded Systems

Phillip Stanley-Marbell, Cristian Borcea, Kiran Nagaraja, Liviu Iftode
Department of Computer Science
Rutgers University
Piscataway, NJ 08854
{*narteh@ece, borcea@cs, knagaraj@cs, iftode@cs*}*.rutgers.edu*

We propose a system architecture and a computing model, based on *Smart Messages (SMs)* , for computation and communication in large networks of embedded systems. In this model, communication is realized by sending SMs in the network. These messages are comprised of code, which is executed at each hop in the path of the message, and data which the message carries in the network. The execution at each hop determines the next hop in the message's path – SMs are responsible for their own routing.

The nodes that support the execution of SMs are termed *Cooperative Nodes (CNs)*. The primary logical components of these nodes are a virtual machine that provides a hardware abstraction layer for executing SMs, and a *Tag Space* that provides a structured memory region consisting of tags persistent across the execution of SMs. Tags are used to store data that can be used for content-based addressing, routing, data sharing, or synchronization between SMs. An SM consists of code and data components. Upon admission at a CN, a task is created out of these components and executed on the virtual machine.

Figure 1 illustrates a network consisting of three types of nodes, represented with squares, circles and triangles. The nodes represented by squares are nodes of interest to an SM which is launched from the circular node in the lower left of Figure 1. The goal of the application implemented by this SM is to visit the five square nodes and to propagate a local data item of each node to the next one visited in order. The SM may use other nodes in the network, the circular and triangular nodes, as intermediates hops as it navigates through the network.

Admission at a CN is restricted based on tag availability, and resource demands of an SM. Tags can be used for synchronization between SMs executing on a CN : an SM may be de-scheduled on a read of a tag, pending a write on that tag by another SM, or the expiration of the tag in question. CNs employ a simple scheduling policy to accept and run multiple SMs. Once executing, an SM may create,

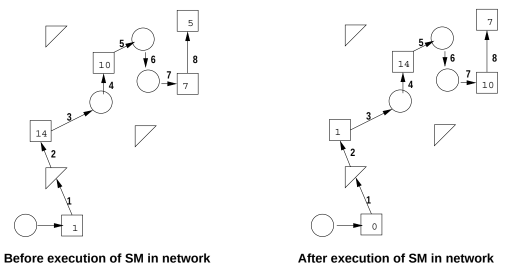

5
10
7
14
1
1
2
3
4
5
6
7
8
Before execution of SM in network
7
14
10
1
0
1
2
3
4
5
6
7
8
After execution of SM in network

Figure 1. Smart Message Model Example

delete, read or write tags, either on the CN or in its own data component, subject to access restrictions and tag lifetimes. An SM may also create and send new SMs, building them out of its constituent code and data components, or it may migrate itself to another CN.

The Smart Message architecture is meant to provide a pervasive computing infrastructure for networks of embedded systems, such as sensor networks and computational fabrics – woven textiles with embedded computing elements. These networks will be inherently heterogeneous in their hardware architectures and inter-networking technologies, since each node will typically be specialized for performing a specific function, hence the need for a hardware abstraction layer such as a virtual machine, and will be volatile, due to node mobility and node failure. Applications utilizing these networks to perform a global task must be willing to accept partial results, or executions that satisfy a specific *Quality of Result (QoR)*.

Issues to be addressed in the architecture include: evaluating tradeoffs between flexibility and overhead of migration, defining a QoR for a partially successful execution, and CN security. A prototype implementation using Bluetooth technology for networking, and Sun Microsystem's KVM for the virtual machine is under development. More information can be found at:
`http://discolab.rutgers.edu/projects/sm.htm`.

Position Summary: Supporting Hot-Swappable Components for System Software

Kevin Hui [†] Jonathan Appavoo [†] Robert Wisniewski [‡]
Marc Auslander [‡] David Edelsohn [‡] Ben Gamsa [§]
Orran Krieger [‡] Bryan Rosenburg [‡] Michael Stumm [§]

A hot-swappable component is one that can be replaced with a new or different implementation while the system is running and actively using the component. For example, a component of a TCP/IP protocol stack, when hot-swappable, can be replaced—perhaps to handle new denial-of-service attacks or improve performance—without disturbing existing network connections. The capability to swap components offers a number of potential advantages such as: online upgrades for high availability systems, improved performance due to dynamic adaptability and simplified software structures by allowing distinct policy and implementation options to be implemented in separate components (rather than as a single monolithic component) and dynamically swapped as needed.

In order to hot-swap a component, it is necessary to (i) instantiate a replacement component, (ii) establish a quiescent state in which the component is temporarily idle, (iii) transfer state from the old component to the new component, (iv) swap the new component for the old, and (v) deallocate the old component. In doing so, three fundamental problems need to be addressed:

- The first, and most challenging problem, is to establish a quiescent state when it is safe to transfer state and swap components. The swap can only be done when the component state is not currently being accessed by any thread in the system. Perhaps the most straightforward way to achieve a quiescent state would be to require all clients of the component to acquire a reader-writer lock in read mode before any call to the component. Acquiring this external lock in write mode would thus establish that the component is safe for swapping. However, this would add overhead in the common case, and cause locality problems in the case of multiprocessors.
- The second problem is transferring state from the old component to the new one, both safely and efficiently. Although the state could be converted to some canonical, serialized form, one would like to preserve as much context as possible during the switch, and handle the transfer efficiently in the face of components with potentially megabytes of state accessed across dozens of processors.
- The final problem is swapping all of the references held by client components so that the references now refer to the new one. In a system built around a single, fully typed language, like Java, this could be done using the same infrastructure as used by garbage collection systems. However, this would be prohibitively expensive for a single component switch, and would be overly restrictive in terms of systems language choice.

[†]University of Toronto, Dept of Computer Science
[‡]IBM T. J. Watson Research Center
[§]University of Toronto, Dept of Electrical and Computer Engineering

We have designed and implemented a mechanism for supporting hot-swappable components that avoids the problems alluded to above. More specifically, our design has the following characteristics:

- zero performance overhead for components that will not be swapped
- zero impact on performance when a component is not being swapped
- complete transparency to client components
- minimal code impact on components that wish to be swappable
- zero impact on other components and the system as a whole during the swapping operation
- good performance and scalability; that is, the swapping operation itself should incur low overhead and scale well on multiprocessor systems.

Our mechanism has been implemented in the context of the K42 operating system (www.research.ibm.com/K42), in which components in the operating system and in applications that run on K42 have been made hot-swappable. Our design and implementation, preliminary performance numbers with respect to swapping overhead, and some of the performance benefits such a facility can provide are presented in www.research.ibm.com/K42/full-hotos-01.ps.

Applying the VVM Kernel to Flexible Web Caches

Ian Piumarta, Frederic Ogel, Carine Baillarguet, Bertil Folliot

email: {ian.piumarta, frederic.ogel, carine.baillarguet}@inria.fr, bertil.folliot@lip6.fr

1 Introduction

The VVM (virtual virtual machine)[1] is a systematic approach to adaptability and reconfigurability for portable, object-oriented applications based on bytecoded languages such as Java and Smalltalk [FP+00].

The main objectives of the VVM are (i) to allow adaptation of language *and* system according to a particular application domain; (ii) to provide extensibility by allowing a "live" execution environment to evolve according to new protocols or language standards; and (iii) to provide a common substrate on which to achieve true interoperability between different languages [FPR98,Fol00].

On the way to implement a VVM we already implemented VVM1 (and it's application to active networks [KF00]) and VVM2 (and it's application to flexible web cache and distributed observation). The VVM2 is a highly-flexible language kernel which consists of a minimal, complete programming language in which the most important goal is to maximise the amount of reflective access and intercession that are possible—at the lowest possible "software level".

2 Our architecture

The VVM2 contains a *dynamic compiler* front-end/back-end, which converts input into optimized native code and an object-oriented environment (with automatic, transparent memory management) used internally by the VVM2 (this work is under consideration for publication).

3 Example application: flexible web caches

Flexibility in web caches come from the ability to configure a large number of parameters[2] that influence the behaviour of the cache (protocols, cache size, and so on). What's more, some of these parameters cannot be determined before deploying the cache, like: user behaviour, change of protocol or the "hot-spots-of-the-week" [Sel96]. However, reconfiguring current web caches involves halting the cache to install the new policy and then restarting it, therefore providing only "cold" flexibility. Our flexible cache architecture is built directly over the VVM2 and so provides "warm" replacement of policies, without compromising the ease of writing new protocols found in existing web caches. Other advantages include the ability to tune the web cache on-line, to add arbitrary new functionality (observation protocols, performance evaluation, protocol tracing, debugging, and so on) at any time, and to remove them when they are no longer needed.

Our approach supports both initial configuration, based on simulation, *and* dynamic adaptation of the configuration in response to observed changes in real traffic as they happen.

This approach is highly reflexive because the dynamic management of the cache is expressed in the *same* language that is used to *implement* the cache. The resulting cache, called C/NN[3], can be modified at any time: new functionality and policies can be introduced and activated during execution. It is therefore possible to *dynamically* define reconfiguration policies to process adaptations, while preserving the cache contents and delaying request for a few μs.

In order to evaluate the flexibility of our cache we made both quantitative and qualitative measurements[4]. Timing the principal operations in C/NN was trivial because of the use of the highly-reflexive VVM2 at the lowest level. We were able to "wrap" timers around the functions without even stopping the cache. Results are very promising : handling a hit (the main bottleneck for the cache itself) takes less than 200μs, switching from one policy to another takes less than 50μs, at least defining a new policy and re-evaluating 5,000 documents takes a couple of tens of ms. It seems clear that dynamic flexibility does not penalise the performance of the cache. It is also important to consider the ease of use of reconfiguration in our cache : typical replacement and reconfiguratoin functions are short and quickly written (a few minutes for a system administrator).

4 Conclusions

This paper presented and evaluated shortly, due to lack of space, the benefits of using a highly-flexible language kernel, the VVM2, to solve a specific computer science problem : flexible web caching. The resulting web cache, C/NN, demonstrates that reconfigurability can be simple, dynamic *and* have good performance.

We finished to incorporate Pandora[PM00a] into VVM2. Pandora is a system for dynamic evaluation of the performance of web cache configurations: this opens the way for "self-adapting" web caches, were the policies are constantly re-evaluated and modified as *and* when needed.

References

[Fol00] B. Folliot, *The Virtual Virtual Machine Project*, Invited talk at the SBAC'2000, Brasil, October 2000.

[FPR98] B. Folliot, I. Piumarta and F. Ricardi, *A Dynamically Configurable, Multi-Language Execution Platform* SIGOPS European Workshop 1998.

[FP+00] B. Folliot, I. Piumarta, L. Seinturier, C. Baillarguet and C. Khoury, *Highly Configurable Operating Systems: The VVM Approach*, In ECOOP'2000 Workshop on Object Orientation and Operating Systems, Cannes, France, June 2000.

[KF00] C. Khoury and B. Folliot, *Environnement de programmation actif pour la mobilit*, Proceedings of Jeunes Chercheurs en Systemes, GDR ARP et ASF, Besanon, France, June 2000.

[PM00a] S. Patarin and M. Makpangou, *Pandora: a Flexible Network Monitoring Platform* Proceedings of the USENIX 2000 Annual Technical Conference, San Diego, June 2000.

[Sel96] Margo Seltzer, *The World Wide Web: Issues and Challenges* , Presented at IBM Almaden, July 1996.

[ZMF+98] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd and V. Jacobson, *Adaptive Web Caching: towards a new global caching architecture*, Computer Networks and ISDN Systems, 30(22-23):2169-2177, November 1998

[1]VVM is both a concept, an implementation and the project's name.

[2]See the configuration file for Squid...

[3]The *Cache with No Name*.

[4]On a G3 233MHz, running LinuxPPC 2000

Position summary: Hinting for goodness' sake

David Petrou, Dushyanth Narayanan, Gregory R. Ganger, Garth A. Gibson, and Elizabeth Shriver[†]
Carnegie Mellon University and [†]Bell Labs, Lucent Technologies

Modern operating systems and adaptive applications offer an overwhelming number of parameters affecting application latency, throughput, image resolution, audio quality, and so on. We are designing a system to automatically tune resource allocation and application parameters at runtime, with the aim of maximizing user happiness or *goodness*.

Consider a 3-D graphics application that operates at variable resolution, trading output fidelity for processor time. Simultaneously, a data mining application adapts to network and processor load by migrating computation between the client and storage node. We must allocate resources between these applications and select their adaptive parameters to meet the user's overall goals. Since the user lacks the time and expertise to translate his preferences into parameter values, we would like the system to do this.

Existing systems lack the right abstractions for applications to expose information for automated parameter tuning. *Goodness hints* are the solution to this problem. Applications use these hints to tell the operating system how resource allocations will affect their goodness (utility). E.g., a video player might have no goodness below some allocation threshold and maximum goodness above another. Goodness hints are used by the operating system to make resource allocation decisions and by applications to tune their adaptive parameters. Our contribution is a decomposition of goodness hints into manageable and independent pieces and a methodology to automatically generate them.

One half of a goodness hint is a *quality-goodness mapping* which tells us how application qualities translate into user happiness. Qualities are measures of performance (latency, throughput) or of fidelity (resolution, accuracy). We hope to leverage user studies from the human-computer interaction community to generate these mappings. The system will also use user feedback to dynamically tailor the mappings to specific users.

A *resource-quality mapping* forms the other half of a goodness hint; our current research focusses on this half. This mapping describes the relationship between an application's resource allocation and its qualities. To do this, we first map adaptive parameters to resource usage by monitoring the application, logging its resource usage for various parameter values, and using machine learning to find the relationship between parameter values and resource usage. We create this mapping offline with controlled experiments to explore the parameter space, and update it online based on dynamic behavior.

Given the resource usage and allocation of an application, we predict its performance using simple models. E.g., a processor-bound computation requiring 1×10^6 cycles and allocated 2×10^6 cycles will have a latency of 0.5 sec. More complex applications will use multiple resources, perhaps concurrently. We will use machine learning techniques to specialize our models to particular applications.

Finally, given some resource allocation, an application must pick adaptive parameter values that maximize its goodness. An *optimizer* searches the parameter space to find the optimal values. By embedding the optimizer in the goodness hint, the operating system is also made aware of what the application will choose. The operating system itself uses a similar optimizer to find the resource allocation that will maximize goodness across applications.

We are building a prototype to validate these concepts. Currently, the prototype supports two resources: processor and network. To map adaptive parameters to resource usage we use linear least squares regression. To search through the space of application parameters and resource allocations, we use a stochastic version of Powell's conjugate direction-set method. We have two very different applications: a 3-D graphics radiosity application [Narayanan, et al., WMCSA 2000], and an Abacus data mining application [Amiri, et al., USENIX 2000].

Our initial results are encouraging. Our system generates accurate resource-quality mappings for both applications. (The quality-goodness half was constructed by hand.) In simulation, our resource allocator is always able to maximize overall goodness, which is a weighted sum of application goodnesses. However, the overhead of the search algorithm is prohibitive, and we are investigating alternatives.

This work raises several research questions: How can we talk about resource usage and allocation in a platform independent way? What is the best way to combine individual application goodnesses into user happiness? What kind of online feedback can we expect from a typical user, and how can we use it to dynamically refine goodness hints?

Position Summary: Censorship Resistant Publishing Through Document Entanglements

Marc Waldman and David Mazières
Computer Science Department, NYU
{waldman,dm}@cs.nyu.edu

Today, most documents available over the the Internet are easy to censor. Each document can usually be traced back to a specific host or even the individual responsible for publishing the document. Someone wishing to censor a document can use the courts, threats, or some other means to force the host administrator or author to delete a particular file. Certainly, there are some high profile documents that are widely mirrored across the internet, however this is not an option for most published documents.

Clearly, a censorship resistant system must replicate a published document across many hosts. However, no standard naming convention exists that allows one to easily specify several hosts via a single name. Even if such a naming convention existed it merely makes the censor's job somewhat harder — the censor still knows exactly which hosts contain the content and therefore which hosts to attack.

Currently, there is little incentive or justification for a server administrator to store documents that he is being pressured into deleting. We propose a system, named Tangler, that we believe provides some incentive to retain such documents and solves the document naming problem.

Tangler is a censorship resistant distributed file system that employs a unique document storage mechanism. A group of documents, called a collection, can be published together under a single name. This collection is named by a public key, K. Only the individual possessing K's corresponding private key can publish a collection under the name K. By naming the published collections in a host and content independent manner we allow the publisher to securly update the collection at some point in the future. This naming convention also allows a collection to include pointers, called soft links, to other collections. These collections may be owned and updated by other individuals.

In order to publish a collection, C, one runs a program that fetches random blocks of previously published collections and $entangles$ these blocks with those of C. Once $entangled$, collection C is dependent on the randomly chosen blocks. Without these blocks, collection C cannot be reassembled. Therefore the publisher has some incentive to retain these blocks, some of which belong to other collections. Notice that this implies that each block may belong to several collections and that each block can be used to reassemble more than one collection.

Tangler's local caching policy causes the replication of these dependent blocks which, at some point in the future, may be reinjected into the distributed file system. This caching policy and reinjection mechanism helps make the published collection difficult to censor.

Tangler consists of a dynamic group of file servers that can publish documents to a distributed file system. Each file server donates local disk space to the system. Servers can join or leave the system at will. The participating servers collectively form a MIX based network that is used for untraceable communication among the servers.

Our block $entanglement$ algorithm is based on Shamir's secret sharing scheme. In this scheme a secret, s, can be split into n pieces, called shares, such that any $k \leq n$ of them can be combined to form s. In our case s is the collection block we wish to $entangle$.

The $entangle$ algorithm takes three parameters, a collection block b and two blocks from previously publish collections. Call these two blocks p_1 and p_2 respectively. These two blocks will become $entangled$ with b. Block b will therefore depend on p_1 and p_2. Each block has the same format; it essentially consists of an x and y value. The points implied by p_1, p_2 and $(0, b)$ uniquely define a quadratic equation. This quadratic is then evaluated at two random x values. This produces two new blocks which we will call q_1 and q_2. Blocks p_1, p_2, q_1 and q_2 are cached and copied to the distributed file system. Notice that we have not cached or copied block b. Block b can be reconstructed from any three of the four stored blocks, namely p_1, p_2, q_1 or q_2.

The $entanglement$ process has defined a (3,4) threshold secret sharing scheme where 3 of any 4 shares can recover the secret (block b). Our publish algorithm is essentially Shamir's secret sharing scheme with a slight twist. Rather than randomly selecting the coefficiants of a quadratic equation we create the quadratic equation from blocks of published collections and the block we wish to publish.

Position Summary: Architectures For Adaptation Systems

Eyal de Lara[†], Dan S. Wallach[‡], and Willy Zwaenepoel[‡]
Departments of Electrical and Computer Engineering[†] and *Computer Science*[‡]
Rice University
{delara,dwallach,willy}@cs.rice.edu

1 Introduction

Modern systems need support for adaptation, typically responding to changes in system resources such as available network bandwidth. If an adaptation system is implemented strictly at the system layer, data adaptations can be added within the network or file system. This makes the adaptation system portable across applications, but sacrifices opportunities to change an application's behavior. It's not possible, for example, to first return a low-quality version of an image and later upgrade it should excess network capacity be available. On the flip side, the adaptation logic could be built into each and every application, with the system providing information to the applications in order to help them adapt their behavior. This becomes impractical because many applications will never be written to perform adaptation, and an application writer may not be able to foresee all possible adaptations that may be desirable.

We argue that adaptation systems should be centralized, where they can make global observations about system usage and resource availability. We further argue that applications should *not* be written to perform adaptation. Instead, applications should support an interface where the adaptation system can dynamically modify an application's behavior as it runs.

How would such an interface work? Largely, we would like applications to make visible their *document object model* (DOM) – the hierarchy of documents, containing pages or slides, containing images or text, etc. Likewise, we would like a standard way to know what portions of a document are on the user's screen. Finally, it's quite helpful when the file formats are standardized, such that the system can see and manipulate the components within a file.

In order to support adaptation while documents are being edited, we would like a standard way to learn which components are "dirty" and to compute diffs between those dirty components and their original contents. Likewise, it would be helpful for applications to support conflict detection and resolution between components.

2 Experience

We observe that many "component-based" applications already support interfaces for external programs to manipulate their components as the application is running. Taking advantage of this, we developed a system called Puppeteer [1], as it "pulls the strings" of an application.

Puppeteer currently supports Microsoft Word, PowerPoint, and Internet Explorer, as well as their StarOffice equivalents. In terms of implementation complexity, Puppeteer has roughly 8000 lines of Java code shared across applications. The Internet Explorer drivers are 2700 lines and the PowerPoint drivers are 1800 lines of code.

Our current system supports adaptation for read-only files. We achieve significant improvements in user-perceived latency at a modest cost in system overhead.

3 Future Work

Building on the base Puppeteer system, we are working on a number of extensions. We are investigating a "thin client" version of Puppeteer to minimize the client memory footprint – an important consideration on PDAs. We are designing a special-purpose language to express adaptation policies at a high-level. We are investigating alternative network transmission protocols and hierarchical scheduling of network transmissions to better reflect the priorities of the adaptation policy. We are also working on extensions to Puppeteer to support writes, dealing with issues like cache coherence and conflict resolution. So far, the Puppeteer architecture has proven flexible enough to accommodate such a wide variety of extensions without sacrificing its portability or core architecture.

References

[1] E. de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, Mar. 2001.

Position Summary: Anypoint Communication Protocol

Ken Yocum, Jeff Chase, and Amin Vahdat
Department of Computer Science, Duke University
{grant,chase,vahdat}@cs.duke.edu

It is increasingly common to use redirecting intermediary switches to *virtualize* network service protocols. Request redirection enables an intermediary to represent a dynamic set of servers as a unified service to the client. Services virtualized in this way include HTTP (using L4-L7 switches), NFS, and block storage protocols.

Virtualization using intermediaries is a powerful technique for building scalable cluster-based services while insulating clients from the details of server structure. However, intermediaries are controversial and difficult to implement in part because transport protocols are not designed to support them. For example, intermediaries compromise the end-to-end guarantees of point-to-point transports such as TCP. Current service intermediaries are constrained to either route requests at a connection granularity (L4-L7 switches for HTTP), use weak transports such as UDP, or terminate connections at the intermediary. These limitations compromise performance and generality. In particular, independent routing of requests is necessary for any content-based routing policy, but we know of no efficient intermediary scheme that supports independent routing for multiple requests arriving on the same transport connection. The challenges are increasingly evident as designers attempt to build intermediaries for commercially important protocols such as HTTP 1.1 and iSCSI.

These difficulties motivate consideration of new transport protocols with more decentralized notions of what constitutes a connection "endpoint". We are developing such a transport called the Anypoint Communication Protocol (ACP). ACP clients establish connections to abstract services, represented at the network edge by Anypoint intermediaries. The intermediary is an intelligent network switch that acts as an extension of the service; it encapsulates a service-specific policy for distributing requests among servers in the *active set* for each service. The switch routes incoming requests on each ACP connection to any active server at the discretion of the service routing policy, hence the name "Anypoint".

The ACP transport is similar to SCTP and TCP in that it provides reliable, sequenced delivery with congestion control. However, ACP defines some protocol properties as *end-to-edge* rather than *end-to-end*. A critical respect in which ACP is end-to-edge is that it does not define the delivery order for requests routed to different servers, or for responses returned from different servers. Ordering constraints and server coordination are the responsibility of the service protocol and its routing policy.

An Anypoint intermediary orchestrates the movement of requests and responses at the transport layer. To this end, ACP frames service protocol requests and responses at the transport layer in a manner similar to SCTP. Transport-level framing allows an Anypoint switch to identify frames from the network stream in a general way. The switch applies the service-specific routing policy to each inbound frame, and merges outbound frames into a single stream to the client.

While ACP is fundamentally similar to other reliable Internet transports, a central design challenge is that ACP connection endpoint state and functions are distributed between the intermediary and the end server nodes. ACP is designed to enable fast, space-efficient protocol intermediaries with minimal buffering. Acknowledgment generation, buffering of unacknowledged frames, and retransmission are the responsibility of the end nodes, thus reliable delivery is guaranteed end-to-end rather than end-to-edge. The Anypoint switch maintains a mapping between sequence number spaces seen by the client and end server nodes for each connection, for a bounded number of unacknowledged frames. The switch also coordinates congestion state across the active set of participants in each ACP connection. The congestion scheme assumes that the bottleneck transit link is between the switch and the client, or that the ACP stream may be throttled to the bandwidth to the slowest end server selected by the routing policy.

The Anypoint abstraction and ACP protocol enable virtualization using intermediaries for a general class of wide-area network services based on request/response communication over persistent transport connections. Potential applications include scalable IP-based network storage protocols and next-generation Web services.

Position Summary: Secure OS Extensibility Needn't Cost an Arm and a Leg

Antony Edwards and Gernot Heiser

University of NSW, Sydney 2052, Australia
{antonye,gernot}@cse.unsw.edu.au

Abstract

This position paper makes the claim that secure extensibility of operating systems is not only desirable but also achievable. We claim that OS extensibility should be done at user-level to avoid the security problems inherent in other approaches. We furthermore claim (backed up by some initial results) that user-level extensibility is possible at a performance that is similar to in-kernel extensions. Finally, user-level extensions allow the use of modern software engineering techniques.

Extensibility is a way to build operating systems that are highly adaptable to specific application domains. This allows, for example, the use of subsystems that are highly tuned to a particular usage patterns, and thus should be able to outperform more generic systems.

In the past, user-level extensibility in systems like Mach and Chorus has lead to poor performance. This has triggered approaches like loadable kernel modules in Linux, which require complete trust in extensions, or secure extensible systems like Spin or Vino, which use trusted compilers or in-kernel protection domains to achieve security. We believe that secure extensibility is possible, with good performance, at user level.

We think that extensibility will only work if they are secure, minimal restrictions are imposed, performance is not degraded, and modern software engineering techniques are supported.

We have developed an extension system based on *components* [2] for our Mungi single-address-space operating system. The component model provides interfaces based on CORBA, and supports modularisation and reuse to make is suitable for building large systems. It supports dynamic binding of extensions, and independent customisation (different users can invoke different, even mutually incompatible extensions).

The single address space helps to achieve performance goals, as it minimises the payload sizes and the amount of marshaling required for component invocations (data is usually passed by reference). In combination with an appropriate protection model, it also makes it easy to expose system resources, to make them accessible to extensions.

The security of the extension model is ensured by a protection system that combines discretionary access control (via password capabilities), with mandatory access control. The former supports *least privilege* while the latter is used to enforce system-wide security policies. These security policies are defined by user-level security objects that are themselves extensions. Both aspects of the protection model are used to restrict the data the extensions can access, as well as who can access the extensions. Mandatory security supports the confinement of extensions, to prevent them from leaking data, even between different clients invoking the same extension.

Mungi	Spin	Vino	COM	omniORB	ORBacus
100	101	885	1993	768	9319

The table compares invocation costs (microseconds) various extensible architectures. These are to be taken with a grain of salt, as they have been measured on different hardware and normalised according to SPECint-95 ratings. However, these results clearly show that Mungi's performance is superior to existing component architectures, and at least equivalent to existing extensible operating systems. This is being achieved while providing full protection, and without relying on type-safe languages.

For more information see [1].

References

[1] A. Edwards and G. Heiser. A component architecture for system extensibility. Technical Report UNSW-CSE-TR-0103, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Mar 2001. URL ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/0103.pdf.

[2] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, Essex, England, 1997.

Position Summary: The Lana Approach to Wireless Computing

Chrislain Razafimahefa Ciarán Bryce Michel Pawlak[*]
Object Systems Group – University of Geneva
Switzerland

Wireless communication is revolutionizing computer systems. On the one hand, long-distance technologies like GSM, UMTS, or satellite facilitate access to existing information services. It is expected that wireless Internet access in Europe will exceed fixed line access by the year 2005 [4]. Further, short-distance wireless (SDW), e.g., Bluetooth or wireless LAN, permit ad hoc or *spontaneous* systems composed of people carrying PDAs. These networks can offer new kinds of services, e.g., micro-payments, localization of offers at a market place. The hardware infrastructure for these systems exists, and it is expected that there might be 700 billion Bluetooth enabled devices by the year 2004 [4].

There are three major issues that SDW application programmers have to be aware of: *disconnected operation, security* and *coordination*. The composition of SDW networks can be very dynamic, so disconnections have to be planned for. This requires mechanisms that enable a node application to continue running despite changes in its network configuration. Coordination covers activities such as service announcement and lookup in a network. Thus, when a node joins a network, its programs can locate other programs and services. Security is required so that sensitive exchanges between two devices are not attacked.

The goal of the **Lana project** is to develop system support for SDW applications[1]. We chose a *language-based* approach: mechanisms like scoping and typing are used to enforce system properties. The Lana language is strongly influenced by Java [1] – it contains interfaces, packages, single inheritance etc. – though is designed with support for disconnected operation, coordination and security.

Lana supports concurrent *programs*. The language semantics states that all memory locations transitively reachable from a program object must be moved along with the program. Further, the set of programs is organized into a hierarchy. When a program moves between nodes then all of its sibling programs are moved along with it. This feature is used by applications to specify hoarding policies: all related programs and objects are grouped under a common umbrella program which is moved. Method calls between programs are asynchronous; thus, a caller is never blocked awaiting a reply that might never come. Each method call generates a unique *key* object that is used by a program to locate the reply message or exception if ever the program momentarily leaves the network. Return messages – or security or mobility exceptions provoked by the call – have this key value bound to them. Any program that is delegated the key by the caller may therefore service the reply message. Thus a node may leave a network yet safely delegate its pending jobs to other nodes.

Concerning coordination, each environment contains a Linda like message board [3] that is used by devices that meet to exchange an initial set of program or object references. The board is also used to store orphan communication replies (if the caller node disappears during a call).

Concerning security, the language prohibits a program from gaining access to objects stored outside of its scope. Keys are another security mechanism. As seen, keys are used to identity method returns; this also prevents a rogue program from intercepting replies destined at other programs. Entries in a message board are also locked with keys. An entry can only be read if the requesting program furnishes the matching key.

The language approach to wireless is useful because the application programmer has control over the security and hoarding policies. For hoarding, mechanisms implemented in an OS kernel can be inefficient since the lack of application behavior information can lead to the wrong data being hoarded [2]. Security also requires application knowledge so that meaningful security constraints can be enforced.

[*] {razafima,bryce,pawlak}@cui.unige.ch

[1] This work is supported by the Swiss National Science Foundation (FNRS 2100-061405.00).

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, 1998.

[2] B. D. N. et al. Agile application-aware adaptation for mobility. In *ACM SOSP*, pages 276–287, Oct. 1997.

[3] D. Gelernter. Generative Communication in Linda. *ACM Trans. Prog. Lang. Syst.*, 7(1), Jan. 1985.

[4] The-Wireless-World-Research-Forum. The Book Of Visions. RFC draft, The Wireless World Research Forum., Jan. 2001.

Position Summary: A Backup Appliance Composed of High-capacity Disk Drives

Kimberly Keeton and Eric Anderson
Hewlett-Packard Labs Storage Systems Program
{kkeeton,anderse}@hpl.hp.com

Disk drives are now available with capacity and price per capacity comparable to nearline tape systems. Because disks have superior performance, density and maintainability characteristics, it seems likely that they will soon overtake tapes as the backup medium of choice. In this position summary, we outline the potential advantages of a backup system composed of high-capacity disk drives and describe what implications such a system would have for backup software.

In the past, magnetic tape had higher capacity and lower price/GB than magnetic disk; however, technology trends are reversing this relationship. Tape's capacity lead over magnetic disks has shrunk over the last 15 years, and disk capacity is now on par with tape capacity. Furthermore, disk media price is within a factor of 3X of tape media price, and disks cost less per GB than tapes, once the tape drive and the tapes supported by the enclosure are taken into account. Given these trends, it is time to rethink tape's role as the backup medium of choice.

Disks confer tremendous hardware-related benefits for a backup system:

Performance. Disks' 5X faster sequential performance suggests that disks are better for creating and fully restoring backup volumes, allowing easier verification and more efficient data scrubbing. Furthermore, disk bandwidth scales more cheaply, since each disk adds bandwidth, whereas only expensive tape drives add tape bandwidth. Disks' superior random access performance implies that disk-based backup will be better at partial restorations and at satisfying simultaneous restore requests.

Density. Designing the appliance so that only a fraction of the drives are simultaneously powered on reduces power and cooling requirements, allowing denser packing. Back-of-the envelope calculations indicate that a disk-based backup appliance could provide roughly 2X more capacity per unit volume than a tape-based system [1].

Support for legacy devices. Restoring data from tape requires finding a matching tape drive, which can be difficult since tapes come in many formats. Disks include their own read/write heads, eliminating the need to search for a separate drive to retrieve data.

Maintainability. Tape drives need to be periodically cleaned with special cartridges and periodically serviced to ensure that head drift doesn't render a tape unreadable. In contrast, disk drives are enclosed media, which don't require cleaning and don't suffer head drift problems.

Lifetime. Empirical evidence suggests that disks could have a longer shelf life than tapes, implying that disks may ultimately be better archival media. System administration experts advise re-recording tape data every three years. Disks come with warranties for three to five years, and disk experts believe that lifetimes over ten years are possible for backup-optimized disks.

The characteristics of disk-based backup have implications for the creation of backup software:

Design for reliability. Backup software protects data by maintaining a read-only copy that cannot be inadvertently corrupted, and by providing an alternate, simpler software path than a snapshotting file system. Furthermore, we can design the backup system to trade off reliability for performance, by using self- and peer-checking code, storing checksums with each data block and verifying those checksums periodically and when the data is accessed, and pro-actively testing the system.

Design for sharability. A backup system that keeps a fraction of its disks online may be able to approximate the performance of an online snapshot using hierarchical storage management techniques, allowing greater simultaneous sharing, while still maintaining the data protection properties of a backup.

Design for longevity. A final opportunity for backup software is to automatically convert data formats commonly used today into formats that will be easy to read many years in the future, either automatically, or through user control.

Key challenges lie in designing backup software for optimizing reliability and data integrity, scheduling the resources of the backup appliance, and developing APIs for giving users and applications more control over how backups are performed.

[1] K. Keeton and E. Anderson. "A Backup Appliance Composed of High-capacity Disk Drives," HP Laboratories SSP Technical Memo HPL-SSP-2001-3, available from http://www.hpl.hp.com/research/itc/csl/ssp/papers/.

Position Summary: Energy Management for Server Clusters

Jeff Chase and Ron Doyle
Department of Computer Science, Duke University
{chase,doyle}@cs.duke.edu

The Internet service infrastructure is a major energy consumer, and its energy demands are growing rapidly. For example, analysts project that 50 million square feet of data center capacity will come on line for third-party hosting services in the US by 2005. These facilities have typical power densities of 100 watts per square foot for servers, storage, switches, and cooling. These new centers could require 40 TWh *per year* to run 24x7, costing $4B per year at $100 per MWh; price peaks of $500 per MWh are now common on the California spot market. Generating this electricity would release about 25M tons of new CO_2 annually.

The central point of this position paper is that energy should be viewed as an important element of resource management for Web sites, hosting centers, and other Internet server clusters. In particular, we are developing a system to manage server resources so that cluster power demand scales with request throughput. This can yield significant energy savings because server clusters are sized for peak load, while traces show that traffic varies by factors of 3-6 or more through any day or week, with average load often less than 50% of peak. We propose *energy-conscious service provisioning*, in which the system continuously monitors load and adaptively provisions server capacity. This promises both economic and environmental benefits.

Server energy management adds a new dimension to *power-aware resource management* [1], which views power as a first-class OS resource. Previous research on power management (surveyed in [1]) focuses on mobile systems, which are battery-constrained. We apply similar concepts and goals to Internet server clusters. In this context, energy-conscious policies are motivated by cost and the need to tolerate supply disruptions or cooling failures.

Our approach emphasizes energy management in the *network* OS, which configures cluster components and coordinates their interactions. This complements and leverages industry initiatives on power management for servers. Individual nodes export interfaces to monitor status and initiate power transitions; the resource manager uses these mechanisms to estimate global service load and react to observed changes in load, energy supply, or energy cost. For example, under light load it is most efficient to use server power management (e.g., ACPI) to step some servers to low-power states. The servers may be reactivated from the network using Wake-On-LAN, in which network cards listen for special wake packets in their low-power state.

Our premise is that servers are an appropriate granularity for power management in clusters. Although servers consume less energy under light load, all servers we measured draw 60% or more of their peak power even when idle. Simply "hibernating" idle servers provides adequate control over on-power capacity in large clusters, and it is a simple alternative to techniques (e.g., voltage scaling) that reduce server power demand under light load. Since load shifts occur on the scale of hours, power transitions are not frequent enough to increase long-term hardware failure rates.

Dynamic request redirection provides a mechanism to allow changes to the set of active servers. Our system is based on reconfigurable switches that route request traffic toward the active servers and away from inactive servers. This capability extends the redirecting server switches (L4 or L7 switches) used in large-scale Web sites today. It enables the system to concentrate request traffic on a subset of servers running at higher utilizations.

Like other schemes for dynamic power management, energy-conscious service provisioning may trade off service quality for energy savings. Servers handle more requests per unit of energy at higher utilizations, but latency increases as they approach saturation. This fundamental tradeoff leads to several important research challenges. For example, it motivates load estimation and feedback mechanisms to dynamically assess the impact of resource allotments on service quality, and a richer framework for Service Level Agreements (SLAs) to specify tradeoffs of service quality and cost. This would enable data centers to degrade service intelligently when available energy is constrained.

References

[1] A. Vahdat, A. R. Lebeck, and C. S. Ellis. Every joule is precious: The case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.

Position Summary
Bossa: a DSL framework for Application-Specific Scheduling Policies

Luciano Porto Barreto, Gilles Muller
COMPOSE group, `http://www.irisa.fr/compose`
IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France
`{lportoba,muller}@irisa.fr`

Emerging computing models and applications are continuously challenging the operating system scheduler. Multimedia applications require predictable performance and stringent timing guarantees. Embedded systems need to minimize power consumption. Network routers demand isolated execution of active network programs. Meeting all these requirements requires specialized scheduling policies, which traditional scheduling infrastructures are unable to provide.

While it is clear the need for customized scheduling policies, there is a lack of tools that capture the design singularities of schedulers to ease the development process. Moreover, writing schedulers requires deep OS knowledge and involves the development of low-level OS code, which frequently crosscuts multiple kernel mechanisms (*e.g.,* process synchronization, file system and device driver operations, system calls).

We present a framework for easing the development of adaptable process scheduling infrastructures. This framework permits the development and installation of basic scheduling policies, which can be specialized using application-specific policies.

We base our approach on a Domain-Specific Language (DSL) named Bossa. A DSL is a high-level language that provides appropriate abstractions, which captures domain expertise and eases program development. Implementing an OS using a DSL improves OS robustness because code becomes more readable, maintainable and more amenable to verification of properties [2]. Our target is to specialize process schedulers for an application with soft-real time requirements that is able to specify adequate regulation strategies for its CPU requirements.

Our framework architecture relies on two basic components: (i) Virtual Schedulers (VSs) and (ii) Application-Specific Policies (ASPs). We consider a process to be the minimal schedulable entity. Every process in the system is associated with a VS. During initialization, a process registers with a VS either by joining an existing VS or by loading a new VS and joining it afterwards.

A VS is a loadable kernel module that controls the execution of a set of processes. A VS manages processes by selecting a process for execution and determining its CPU quantum. To permit the coexistence of multiple and independent schedulers, VSs can be stacked, forming a tree hierarchy as in [1]. A VS also provides an *interface*. This interface consists of a list of functions and events that can be used in the specification of an ASP. An ASP uses interface functions to exchange data with a VS. These functions provide a local view of the process state stored by a VS.

An ASP is a Bossa program that specializes the behavior of a VS with respect to specific application needs. An ASP is organized into two parts: (i) a list of global variable declarations and (ii) a collection of event handlers. Event handlers are executed either periodically or in response to specific OS events.

To improve robustness, we perform static and dynamic verification on Bossa code. By construction, Bossa programs are not allowed to have unbounded loops and recursive functions. This constraint ensures that event handlers terminate.

We are currently implementing Bossa in Linux. To provide flexibility while retaining acceptable performance, we are implementing a JIT compiler that runs in the kernel. The JIT compiler translates ASP code into machine code, which is then executed in the kernel. We plan to assess our framework by conducting experiments on scheduling infrastructures using real workloads. We will evaluate Bossa by analyzing the behavior of soft real-time applications, such as video players.

References

[1] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *OSDI'96*, pages 91–105, Oct. 1996.

[2] G. Muller, C. Consel, R. Marlet, L. Barreto, F. Mérillon, and L. Réveillère. Towards robust oses for appliances: A new approach based on domain-specific languages. In *ACM SIGOPS European Workshop (EW2000)*, pages 19–24, Sept. 2000.

Position Summary: Authentication Confidences

Gregory R. Ganger
Carnegie Mellon University
ganger@ece.cmu.edu

Abstract

"Over the Internet, no one knows you're a dog," goes the joke. Yet, in most systems, a password submitted over the Internet gives one the same access rights as one typed at the physical console. We promote an alternate approach to authentication, in which a system fuses observations about a user into a probability (an **authentication confidence***) that the user is who they claim to be. Relevant observations include password correctness, physical location, activity patterns, and biometric readings. Authentication confidences refine current yes-or-no authentication decisions, allowing systems to cleanly provide partial access rights to authenticated users whose identities are suspect.*

1 The Case for Authentication Confidences

Access control decisions consist of two main steps: authentication of a principal's digital identity and authorization of the principal's right to perform the desired action. Well-established mechanisms exist for both. Unfortunately, authentication in current computer systems results in a binary yes-or-no decision, building on the faulty assumption that an absolute verification of a principal's identity can be made. In reality, no perfect (and acceptable) mechanism is known for digital verification of a user's identity, and the problem is even more difficult over a network. Despite this, authorization mechanisms accept the yes-or-no decision fully, regardless of how borderline the corresponding authentication. The result is imperfect access control.

Using authentication confidences, the system can remember its confidence in each authenticated principal's identity. Authorization decisions can then explicitly consider both the "authenticated" identity and the system's confidence in that authentication. Explicit use of authentication confidences allows case-by-case decisions to be made for a given principal's access to a set of objects. So, for example, a system administrator might be able to check e-mail when logged in across the network, but not be able to modify sensitive system configurations. This position paper discusses identity indicators, and our full white paper [1] completes the case.

2 Human identification and confidence

Identity verification in most systems accepts any user presenting a predetermined secret (e.g., password) or token (e.g., ID card). The conventional wisdom is that, since they are private, no additional information about the likelihood of true identity is necessary or available. We disagree. For example, a system's confidence in the provided password could certainly depend upon the location of its source. As well, a gap of idle time between when the password was provided and a session's use might indicate that the real user has left their workstation and an intruder has taken the opportunity to gain access.

A controversial emerging authentication mechanism compares measured features of the user to pre-recorded values, allowing access if there is a match. Commonly, physical features (e.g., face shape or fingerprint) are the focus of such schemes, though researchers continue to look for identifying patterns in user activity. Identifying features are boiled down to numerical values called "biometrics" for comparison purposes. Biometric values are inherently varied, both because of changes in the feature itself and because of changes in the measurement environment. For example, facial biometrics can vary during a day due to acne appearance, facial hair growth, facial expressions, and ambient light variations. Similar sets of issues exist for other physical features. Therefore, the decision approach used is to define a "closeness of match" metric and to set some cut-off value — above the cut-off value, the system accepts the identity, and below it, not.

Confidence in identity can be enhanced by combining multiple mechanisms. The simplest approach is to apply the mechanisms independently and then combine their resulting confidences, but more powerful fusing is also possible. For example, merged lip reading and speech processing can be better than either alone. Note that if the outcomes conflict, this will reduce confidence, but will do so appropriately.

References

[1] Gregory R. Ganger. *Authentication Confidences*. CMU-CS-01-123. Technical Report, Carnegie Mellon Univeristy School of Computer Science, April 2001.

Position Summary: Supporting Disconnected Operation in DOORS

Nuno Preguiça, J. Legatheaux Martins, Henrique Domingos, Sérgio Duarte
Departamento de Informática
Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa
Quinta da Torre, 2825-114 Monte da Caparica, Portugal
{nmp, jalm, hj, smd}@di.fct.unl.pt

1. Summary

The increasing popularity of portable computers opens the possibility of collaboration among multiple distributed and disconnected users. In such environments, collaboration is often achieved through the concurrent modification of shared data. DOORS is a distributed object store to support asynchronous collaboration in distributed systems that may contain disconnected computers. In this summary we focus on the mechanisms to support disconnected operation.

The DOORS architecture is composed by servers that replicate objects using an epidemic propagation model. Clients cache key objects to support disconnected operation. Users run applications to read and modify the shared data (independently from other users) – a read any/write any model of data access is used. Modifications are propagated from clients to servers and among servers as sequences of operations – the system is log-based.

Objects are structured according to an object framework that decomposes object operation in several components (figure 1). Each component manages a different aspect of object execution. Each object represents a data-type (e.g. a structured document) and it is composed by a set of sub-objects. Each sub-object represents a subpart of the data-type (e.g. sections).

A new object is created composing the set of sub-objects that store the type-specific data with the adequate implementations of the other components.

The following main characteristics are the base to support disconnected operation in DOORS.

Multiple concurrency control/reconciliation strategies: To support the different requirements posed by multiple data-types we rely on the flexibility provided by the DOORS object framework. The concurrency control component allows the use of different log-based reconciliation strategies. The capsule component allows the definition of different data configurations – e.g. the tentative and committed versions of an object can be easily maintained duplicating the adequate components under the control of the capsule.

Integrated awareness support: The reconciliation among concurrent streams of activity is often performed when users are no longer connected to the system. In DOORS, awareness information may be generated and processed during the reconciliation phase – this approach makes it possible, for example, to provide shared feedback about data evolution and/or to explore off-system communication infrastructures, such as the use of SMS messages.

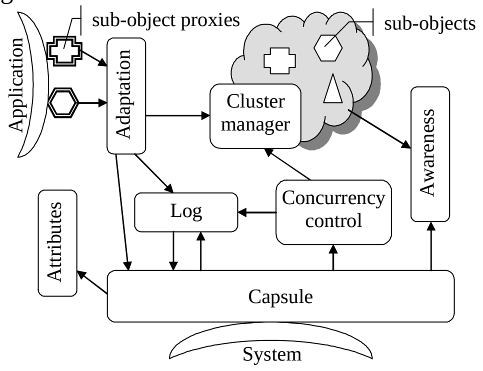


Figure 1. DOORS object framework.

Partial caching: As caching a full data object is sometimes impossible for resource-poor mobile devices, DOORS allows sub-objects to be cached independently.

Blind operation invocation: When some sub-objects are not present in the cache, a disconnected user is still allowed to execute operations on them. A replacement sub-object may be created to present the tentative result of these operations.

Adaptation: The adaptation component allows the use of different strategies to adapt to variable network conditions – as a result, operations may be performed immediately on a server or on the local copy.

The interested reader may find more information on the DOORS system in [1] (including an extended version of this summary). This work was partially supported by FCT, project number 33924/99.

2. References

[1] http://dagora.di.fct.unl.pt

Position Summary. DiPS: A Unifying Approach for Developing System Software

Sam Michiels, Frank Matthijs, Dirk Walravens, Pierre Verbaeten
DistriNet, Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven
Sam.Michiels@cs.kuleuven.ac.be

Abstract

In this position paper we unify three essential features for flexible system software: a component oriented approach, self-adaptation and separation of concerns. We propose DiPS (Distrinet Protocol Stack) [5], a component framework, which offers components, an anonymous interaction model and connectors to handle non-functional aspects such as concurrency. DiPS has effectively been used in industrial protocol stacks [8] and device drivers [6].

Position Statement

This position statement explains why component framework technology is needed for flexible system software (such as device drivers, protocol stacks and object request brokers) and how we are using DiPS (Distrinet Protocol Stack) [5], a component framework, to build complex adaptable system software such as protocol stacks and device drivers.

We state that there are three essential features for flexible system software: a component oriented approach, self-adaptation and separation of concerns. There exist several systems and paradigms that each offer one or two of these features. Recent operating system research [1] [2] shows the advantage of providing reusable system components as basic building blocks. Other operating systems focus on the flexibility of the system that allows to adapt itself to changes in the application set it must support [7]. System software on tomorrow's embedded systems (such as personal digital assistants (PDA's) or cellular phones) must be intelligent enough to adapt the internal structure to new, even non-anticipated services and to integrate the necessary support for it (such as a new communication protocol or a new disk caching strategy), even at run-time. Strict separation of functional and non-functional code has proven to be an essential feature for adaptable, maintainable and reusable software [3] [4]. The DiPS component framework unifies component support, self-adaptability and separation of concerns in one paradigm, which is a strong combination for system software. DiPS is not a complete operating system, but rather a component framework to build system software. We are convinced that other operating system abstractions, such as interrupt handling or memory management, can benefit from the DiPS approach. Our research is also heading in that direction.

References

[1] B. Ford, K. V. Maren, J. Lepreau, and e.a. The flux os toolkit: Reusable components for OS implementation. *In Proceedings of the Sixth IEEE Workshop on Hot Topics in Operating Systems*, May 1997.

[2] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The pebble component-based operating system. *In Proceedings of the USENIX 1999 Annual Technical Conference*, June 1999.

[3] J. Itoh, Y. Yokote, and M. Tokoro. Scone: Using concurrent objects for low-level operating system programming. Technical report, Department of Computer Science, Keio University, 1995.

[4] G. Kiczales. Foil for the workshop on open implementation, Oct. 1994.

[5] F. Matthijs. *Compontent Framework Technology for Protocol Stacks*. PhD thesis, Katholieke Universiteit Leuven, Dec 1999. Available at http://www.cs.kuleuven.ac.be/ samm/netwg/dips/index.html.

[6] S. Michiels, P. Kenens, F. Matthijs, D. Walravens, Y. Berbers, and P. Verbaeten. Component framework support for developing device drivers. *In Proceedings of International Conference on Software, Telecommunications and Computer Networks, vol. 1, FESB, Split, Croatia, pp. 117-126*, June 2000.

[7] M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. *In Proceedings of the Sixth IEEE Workshop on Hot Topics in Operating Systems*, May 1997.

[8] I. Sora and F. Matthijs. Automatic composition of software systems from components with anonymous dependencies. *Submitted to 8th European Software Engineering Conference (ESEC)*, Mar 2001.