D-Stampede: Distributed Programming System for Ubiquitous Computing *

Sameer Adhikari, Arnab Paul, Umakishore Ramachandran
College Of Computing, Georgia Institute of Technology
801 Atlantic Drive, NW, Atlanta, GA 30332-0280, USA
sameera, arnab, rama@cc.gatech.edu

Abstract

We focus on an important problem in the space of ubiquitous computing, namely, programming support for the distributed heterogeneous computing elements that make up this environment. We address the interactive, dynamic, and stream-oriented nature of this application class and develop appropriate computational abstractions in the D-Stampede *distributed programming system. The key features of D-Stampede include indexing data streams temporally, correlating different data streams temporally, performing automatic distributed garbage collection of unnecessary stream data, supporting high performance by exploiting hardware parallelism where available, supporting platform and language heterogeneity, and dealing with application level dynamism. We discuss the features of D-Stampede, the programming ease it affords, and its performance.*

1 Introduction

Complex ubiquitous computing applications require the acquisition, processing, synthesis, and correlation (often *temporally*) of streaming data such as video and audio. Such applications usually span a multitude of devices that are *physically distributed* and *heterogeneous*. Different kinds of sensors and their data aggregators collect raw data and perhaps do limited processing. However, extraction of higher order information content from such raw data requires significantly more processing power. For example, microphones may collect audio data, but voice recognition requires more processing. Thus there is a continuum of computation and communication resources as depicted in Figure 1. Also such applications are highly dynamic. The unique characteristics of this emerging class of distributed applications calls for novel solutions. In particular, there is a need for a distributed programming system for them.

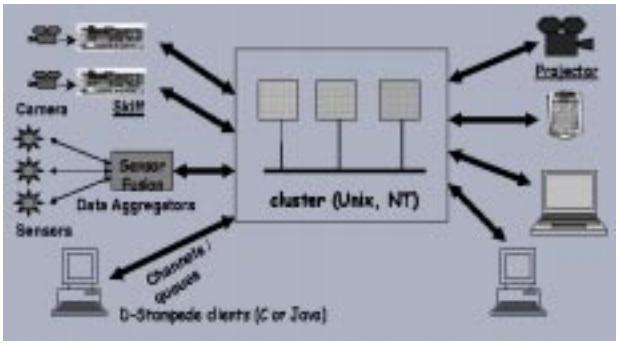

Figure 1. Octopus Hardware Model

We have developed a novel distributed programming system, called *D-Stampede*, that enables the coupling of network of spatially distributed sensors and data aggregators to a cluster. It has five main features. It is *based on an "Octopus" architectural model* (Figure 1) in which a cluster ("body") supports a network of sensors and data aggregators ("tentacles"). It *supports a heterogeneous hardware* continuum encompassing networked sensors, data aggregators connected to sensors, and backend computational clusters. It *supports a heterogeneous programming environment* wherein parts of the same application can be programmed in C and Java. All the parts have access to the same set of abstractions. It supports the *sharing of streaming data across multiple nodes and address spaces*. All system aspects, from sharing primitives to an automatic garbage collection facility, are governed by a common trait of this application domain i.e., time-sequenced data. It is *designed for high performance*, and supports both task and data parallelism in the applications.

D-Stampede system is operational and in use for developing prototype ubiquitous computing applications with other researchers at Georgia Tech. We have studied the performance of the D-Stampede system at two levels. One, the overhead of using D-Stampede's high-level API compared to the basic message transports (UDP, TCP). Two, application level scalability of a workload (akin to distributed

*To Appear in 22^{nd} ICDCS, July 2002. The work has been funded in part by an NSF ITR grant CCR-01-21638, NSF grant CCR-99-72216, Compaq Cambridge Research Lab, the Yamacraw project of the State of Georgia, and the Georgia Tech Broadband Institute. The equipment used in the experimental studies is funded in part by an NSF Research Infrastructure award EIA-99-72872, and Intel Corp.

telepresence) implemented on top of D-Stampede.

Projects such as MIT Oxygen [8], Berkeley Endeavour [9], and OGI/Georgia Tech Infosphere [10] share our high level objectives, namely, to support ubiquitous computing. We differ in specific research goals from them. Oxygen's focus is to develop the fundamental technology for a ubiquitous computing fabric, and address the issues of networking them, and developing adaptive applications on top of them. Endeavour's focus is to develop scalable services such as file systems on planetary scales on top of ubiquitous infrastructure with varied network connectivity. Infosphere's focus is to devise middleware that will help the end user combat with the explosive growth of information on the Internet and allow the information infrastructure to scale as the World Wide Web grows. Our research goal is a seamless distributed programming system, which is complementary to these projects. Messaging layers such as MPI [1] and PVM [2] and middleware such as CORBA [3] and RMI [4] provide the basic transport and remote procedure call mechanisms needed in such distributed applications. There have been several language proposals for parallel and distributed computing such as Linda [5], Orca [6], and Cid [7]. Such languages provide fairly generic programming capabilities for data sharing and synchronization and do not offer any specific help for the characteristics found in ubiquitous computing applications.

We first discuss the requirements of ubiquitous computing applications in Section 2. We present the architecture and implementation of D-Stampede in Section 3. Programming experience with D-Stampede is presented in Section 4, performance of D-Stampede in section 5, and conclusions in section 6.

2 Requirements

We enumerate the programming requirements for ubiquitous computing applications:

Stream Handling: Consider gesture and speech inputs for a ubiquitous application. A gesture is a stream of images, and speech of audio samples. As ubiquitous computing becomes more prevalent we expect there to be a preponderance of stream data. Thus efficient support for streams is needed.

Temporal Indexing and Correlation: Datasets from different sources may be combined, correlating them temporally. For example, stereo vision would combine images captured from two cameras. Other analyzers may work multi-modally, *e.g.*, by combining vision, audio, etc. This suggests that the programming system offer some support for indexing data by time.

Distributed Heterogeneous Components: The very nature of the applications suggests distributed and heterogeneous components. The heterogeneity may be both at the level of hardware and software that runs on the components.

Real-time Guarantees: Due to the interactive nature, certain events in the application may need soft real-time guarantees. For example, in telepresence timely visual and audio feedback to the participants is essential.

Dynamic Start/Stop: There should be a natural way for components of the application to join and leave. For example, in a telepresence application participants should be free to come and go as they please.

Plumbing: Ubiquitous computing applications could have components connected in a complex and dynamic fashion. The programming system should allow for intuitive, efficient, and flexible ways for the dynamic generation of complex graphs of computational modules.

3 The D-Stampede Programming System

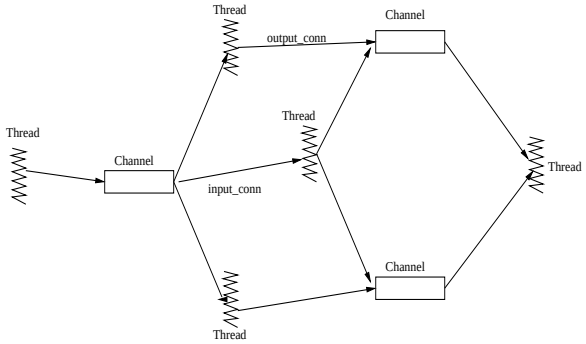

Figure 2. Computational Model: Dynamic Thread Channel Graph

The computational model supported by D-Stampede is shown by the thread-channel graph in Figure 2. The model captures a huge class of distributed ubiquitous computing applications. The threads map to computing devices scattered in the Octopus hardware model (Figure 1). Channels serve as the application level conduits for time-sequenced stream data among the threads.

3.1 Architecture

The D-Stampede architecture has the following main components: *Channels*, *Queues*, *Threads*, *Garbage Collection*, *Handler Functions*, *Real-time Guarantees*, *Nameserver*, and support for *Heterogeneity*.

Threads, Channels, and Queues: D-Stampede provides a uniform set of abstractions across the entire hardware continuum: threads, channels, and queues. Stampede threads can be created in different protection domains (address spaces) [11]. Channels (queues) are system-wide unique names, and serve as containers for *time-sequenced* data. They facilitate inter-thread communication and synchronization regardless of the physical location of the threads and channels (queues). A thread (dynamically) connects to a channel (queue) for *input* and/or *output*. Once

connected, a thread can do I/O (get/put items) on the channel (queue). The items represent some application-defined stream data (e.g. video frames). The timestamps associated with an item in a channel (queue) is user defined. The collection of time-sequenced data in the channels and queues is referred to as *space-time memory* [12]. Conceptually a D-Stampede computation with threads, channels, and queues is akin to a distributed set of processes connected by sockets. The power of D-Stampede is the ability to reason about program behavior based on time, and temporal correlation among data generated by different sources.

Garbage Collection: API calls in D-Stampede facilitate a given thread to indicate that an item (or a set of items) in a channel or queue is garbage so far as it is concerned. Using this per-thread knowledge, D-Stampede automatically performs distributed garbage collection [13] of timestamps (i.e. items with such timestamps) that are of no interest to any thread in the D-Stampede computation.

Handler Functions: D-Stampede allows association of *handler functions* with channels (queues) for applying a user-defined function on an item in a channel (queue). The handler functions are handy in various situations. To transport a complex data structure on channels, these functions can define the serialization (deserialization) used by D-Stampede. Similarly D-Stampede can invoke a handler registered by a thread, when an item becomes garbage.

Real-time Guarantees: The timestamp associated with an item is an indexing system for data items. For pacing a thread relative to real time, D-Stampede provides an API borrowed from the Beehive system [14]. Essentially, a thread can declare real time interval at which it will resynchronize with real time, along with a tolerance and an exception handler. As the thread executes, after each "tick", it performs a D-Stampede call attempting to synchronize with real time. If it is early, the thread is blocked until that synchrony is achieved. It if is late by more than the specified tolerance, D-Stampede calls the thread's registered exception handler which can attempt recovery in an application specific manner.

Name Server: D-Stampede infrastructure includes a name server. Application threads can register (de-register) information (channels, queues, their intended use, etc.) with the name server. Any thread in the application anywhere in the entire Octopus model can query this name server to determine resources of interest that it may want to connect to in the computation model of Figure 2. This facilitates the dynamic start/stop feature referred in Section 2.

Heterogeneity: We have architected the D-Stampede system as "client" libraries on the end devices (Octopus tentacles) and a "server" library on the cluster (Octopus head). This organization has no bearing on the generality of the computational abstractions provided by D-Stampede. The API calls of D-Stampede are available to a thread regardless of where it is executing. D-Stampede accommodates component heterogeneity in two ways. Firstly, the server library has been ported to multiple platforms (Alpha-Tru64, x86-Linux, x86-Solaris, and x86-NT). Secondly, D-Stampede supports *language* heterogeneity on the clients, which can be programmed in either C or Java. An application can have some parts written in C and some parts in Java, sharing the same abstractions.

Detailed discussion of the D-Stampede API is beyond the scope of this paper [1]. D-Stampede provides API for thread and channel creation, thread-channel connection setup, and the actual I/O on the connections. The system takes care of the buffer management, synchronization, and garbage collection, allowing the programmer to concentrate on the application specifics.

3.2 Implementation

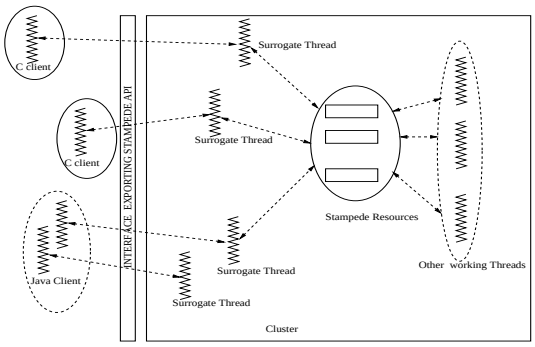

Figure 3. D-Stampede Implementation

D-Stampede is implemented as a runtime library that has two parts, a *server* written in C running on a cluster, and *clients* (in C or Java) that can run anywhere in the distributed system. Figure 3 shows the organization. Though the programming model is uniform and does not distinguish between client and server, this dichotomy exists for both historical reason (Stampede was originally developed as a cluster computing system) and a reflection of the Octopus analogy (Figure 1). The distributed end points (clients) are usually connected to sensors and serve as the capture and access points, while heavy duty computation (e.g. tracking) is performed on the cluster (server).

Client Library The D-Stampede APIs are exported to the distributed end points in a manner analogous to RPC [15]. Client libraries are available for both C and Java. The Java client library encapsulates the D-Stampede APIs as objects. The application level D-Stampede programs running on the end devices (i.e. clients) can be written in C or Java and they can coexist as parts of a single application. A TCP/IP socket is used as the transport for client-server library communication. The Java client library uses our own

[1] D-Stampede API header file may be found at http://www.cc.gatech.edu/ rama/stampede/api_h.txt

data representation to perform the marshalling and unmarshalling of the arguments, while the C client library uses XDR [16].

Server Library The server library is implemented on top of a message-passing substrate called CLF. CLF provides reliable, ordered point-to-point packet transport between the D-Stampede address spaces within the cluster, with the illusion of an infinite packet queue. It exploits shared memory within an SMP. Between cluster nodes it can use Memory Channel [17], Myrinet [18]. If nothing is available it can use UDP over a LAN.

There is a *listener* thread on the server that listens for new end devices joining a D-Stampede computation. Upon joining, a specific *surrogate* thread (see Figure 3) is created on the cluster for the new end device. All subsequent D-Stampede calls from this end device are carried out by this specific surrogate thread. State information pertaining to an end device is maintained by the associated surrogate thread. The surrogate thread ceases to exist when the end device goes away, mirroring the joining and leaving of an end device.

Supporting Heterogeneity Java clients can run on any end device that has a JVM. The server library (which is in C) and the C client library have been ported to the following platforms: DEC Alpha-Tru64 Unix, x86-Linux, x86-Solaris, and x86-NT. The implementation supports combinations of 32-bit and 64-bit platforms for the end devices and the cluster. Any combination of end devices and a cluster platform can host a single D-Stampede application.

Supporting Handler Functions for End Devices As we mentioned earlier (Section 3.1), D-Stampede allows user-defined actions to be performed on an item in a channel or queue via the handler function. Garbage collection is a good example for use of this mechanism. Upon the D-Stampede runtime determining that an item in a channel is garbage, it calls the user-registered notification handler. While this is easily implemented for cluster threads in D-Stampede, end devices need special handling. When requested by an end device to install a handler, its surrogate installs a generic handler function on the cluster. When D-Stampede invokes this generic handler, the handler collects the information on behalf of the end device, and communicates it to the end device at an opportune time.

4 Programming using D-Stampede

Consider a video conferencing application. This application involves combining streams of audio and video data from multiple participants, and sending the composite streams back to the participants. For simplicity we only consider video streams. An implementation of this application using D-Stampede has the following components:

Server program on the cluster that has a set of channels (one per end device) for storing the video streams; a mixer thread that takes corresponding timestamped frames from these channels to create a composite; and a channel for placing the composite video output of the mixer thread.

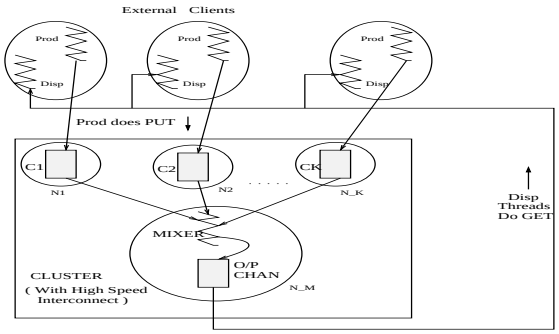

Figure 4. Video Conferencing Application

Camera and display as end devices for each participant. A client program on each end device is comprised of a producer thread that `puts' its timestamped video stream on its assigned channel; and a display thread that `gets' the composite video and displays it.

Figure 4 shows the structure of this application. When the server program starts the following happens:

The server creates multiple address spaces $N_1, N_2 \ldots N_k$ in the cluster.
The server spawns a listener thread in each address space.
The server creates address space N_M, where a mixer thread and an output channel C_o are created.
The id of channel C_j created by j-th client $(1 \leq j \leq m)$ in N_i $(1 \leq i \leq k)$ is passed to N_M.
The id of channel C_o in N_M is passed to each client via the nameserver.

The mixer thread does the following:

Create input connections to channels C_j.
Create output connection to the channel C_o.
Get matching timestamped images from each C_j.
Create the composite item (image).
Put composite item on channel C_o.

The j-th client program does the following:

The client communicates with a listener thread in one of the address spaces N_i $(1 \leq i \leq k)$.
The client program creates a channel C_j in address space N_i.
The client program starts a producer thread which creates an output connection to channel C_j and puts images in C_j.
Client program starts a display thread, which creates an input connection to channel C_o and gets composite images from C_o.

As can be seen from the above example, the high level abstraction provided by D-Stampede ease the task of developing such an application.

5 Performance

We have experimentally evaluated the performance of the D-Stampede system. We have measured the performance of the system at two levels. First, at the micro-level

to determine the latency of D-Stampede operations. Second, at the level of a video-conferencing application (Section 4). The hardware setup for the experiments is as follows: A cluster of 17 eight-way SMPs interconnected by Gigabit Ethernet. Each processor is a 550MHz Pentium III Xeon. Each cluster node runs RedHat Linux 7.1 and has 4GB RAM and 18GB SCSI disk. Even in experiments that involve the C and Java client libraries, a cluster node acts as an end device. This setup ensures that the measurements reflect the impact of D-Stampede runtime and are not perturbed by the end device hardware capabilities.

5.1 Micro Measurements

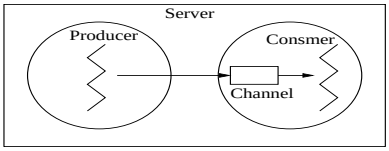

Figure 5. Experiment 1

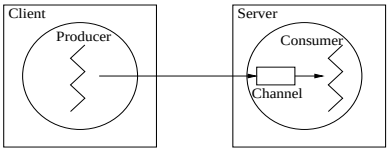

Figure 6. Experiment 2, Configuration 1

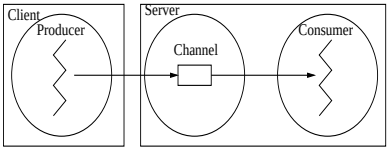

Figure 7. Experiment 2, Configuration 2

The micro-level measurements compare an end-to-end data exchange between two D-Stampede threads against a similar data exchange using the underlying messaging layer. The intent is to show that the overhead incurred for the high-level data abstractions in D-Stampede is minimal compared to raw messaging. For raw messaging, this data exchange amounts to a send-receive combination. A put-get combination is the logical equivalent for D-Stampede. We conduct three experiments, each involving a pair of producer-consumer threads. In each experiment we compare different configurations of producer-consumer pair performing a data exchange in D-Stampede against a commensurate send-receive based data exchange.

In the graphs that correspond to these experiments (Figures 9, 10, and 11), the message size (in bytes) is plotted along the X-axis, and the latency (in microseconds) is plotted along the Y-axis. The readings shown are for data sizes ranging from 1000 to 60000 bytes, in steps of 1000 bytes. We restricted our readings to 60000 bytes because

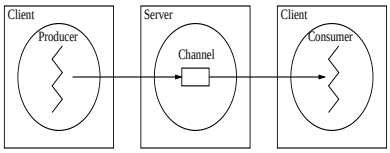

Figure 8. Experiment 2, Configuration 3

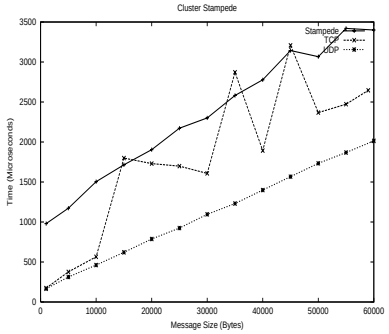

Figure 9. Intra-Cluster (Experiment 1)

UDP, which is one of the raw messaging layers we compare against, does not allow messages greater than 64 KB.

Experiment 1: In this experiment (pictorially shown in Figure 5) the producer and consumer threads are on different nodes *within* the cluster. The channel used for communication is located in the consumer's address space. The producer puts items on the channel. The consumer gets items from the channel. We ensure that the put and get do not overlap. Latency is measured as the sum of the put and get operations.

This D-Stampede configuration is compared against two other alternatives. One uses UDP for communication and the other TCP/IP between the producer-consumer pair. Within the cluster, D-Stampede uses CLF, a reliable packet transport layer built on top of UDP. CLF does not have the connection management overheads of TCP/IP, and hence the choice of the two alternative scenarios being compared against D-Stampede. To ensure that the send and the receive do not overlap for the two alternatives, we do the following: The producer sends a message and the consumer receives it. The consumer sends a message and the producer receives it. The exchange latency is assumed to be half the time taken

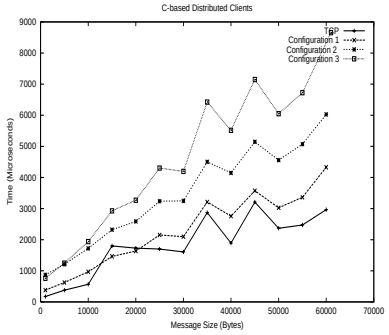

Figure 10. C Clients (Experiment 2)

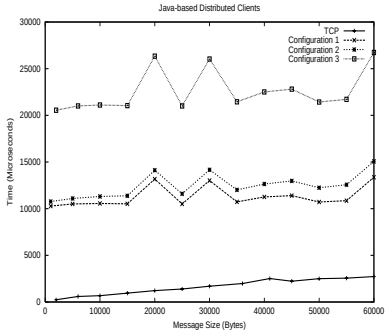

Figure 11. Java Clients (Experiment 3)

for this cycle. The programs for this experiment are all written in C. The performance results are shown in Figure 9.

As can be seen, data exchange using D-Stampede adds an overhead (that ranges from 700 microseconds at 10 KB payload to 1200 microseconds at 60 KB payload) compared to the UDP alternative. The overhead compared to the TCP/IP alternative is much less (starts from around 700 microseconds at 10 KB and falls to 400 microseconds at 60 KB). Overall, the overhead incurred for D-Stampede is fairly low at reasonably high payloads (less than 2X compared to UDP; at best almost the same or better than TCP/IP and at worst within 1.5X compared to TCP/IP).

Experiment 2: This experiment involves the use of the C client library of D-Stampede. The producer thread runs on an end device as a client program. Three configurations used in this experiment, each differing in the location of the consumer thread.

Configuration 1: As shown in Figure 6, the consumer thread is co-located with the channel on a cluster node. This configuration involves one end device to cluster network traversal. The get operation is local to the cluster node. As the client library uses TCP/IP to communicate with the server library, this configuration shows the exact overhead that D-Stampede runtime adds to TCP/IP. For example, for a data size of 55000 bytes, TCP/IP latency is 2500 μs, and D-Stampede latency is 3300 μs.

Configuration 2: As shown in Figure 7, the consumer thread is located on the cluster. The channel is in a different address space from that of the consumer. This configuration involves one end device to cluster network traversal (for the put operation) and one intra-cluster network traversal (for the get operation). Expectedly this configuration has more overhead than the previous one. For 55000 bytes payload, the D-Stampede latency is around 5000 μs.

Configuration 3: As shown in Figure 8, the consumer thread is located on an end device. This configuration involves two end device to cluster network traversals, one for each of the put and get operations. Thus, this configuration has the maximum overhead. For 55000 bytes payload, the D-Stampede latency is around 6100 μs.

Since this experiment involves the client library (which uses TCP/IP for communicating with the server library), we use a TCP/IP based producer-consumer pair written in C for comparison. As before, we ensure that the producer and consumer do not overlap in their communication. The results are shown in Figure 10. As expected, the shape of the D-Stampede curves track the TCP/IP curve for all the configurations. For configuration 1 (with only one end device to cluster network traversal) the D-Stampede runtime overhead over TCP/IP is fairly nominal (at best less than 12%).

Experiment 3: This experiment is the almost the same as the previous one with the only difference that the Java client library is used for D-Stampede, and the comparison TCP/IP program is also in Java. The results for shown in figure 11 For a payload of 55000 bytes, the D-Stampede latency is approximately 11000 μs for configuration 1 (corresponds to Figure 6), approximately 12600 μs for configuration 2 (corresponds to Figure 7), and approximately 21700 μs for configuration 3 (corresponds to Figure 8).

The results of the experiments are summarized below:

Result 1: For D-Stampede, intra-cluster data exchange (Figure 9) performs slightly better than C-based end device to cluster data exchange (Figure 10), which in turn performs better than Java-based end device to cluster data exchange (Figure 11). For example, the latency for a data size of 35000 bytes, is 2580 μs in the first case, 3200 μs in the second and 10700 μs for the third case. Since the hardware is the same for all these data exchanges, the disparity is simply due to the software. As we mentioned earlier, D-Stampede uses CLF for intra-cluster communication, which has less overhead compared to TCP/IP.

Result 2: For TCP/IP based data exchange, the C producer-consumer program results (Figure 10), and Java producer-consumer program results (Figure 11) are similar. However, the D-Stampede data exchange is much better in C compared to Java. In C marshalling and unmarshalling arguments involve mostly pointer manipulation, while in Java they involve construction of objects. Hence the disparity between the two.

5.2 Application Level Measurements

We use the application described in Section 4 for this study. We abstract out the camera and display from the application to study scalability of the D-Stampede system. The producer thread reads a "virtual" camera (a memory buffer) and sends it to the server program continuously. Similarly the display thread simply absorbs the composite output from the mixer without displaying it. This structure allows us to stress the communication infrastructure of D-Stampede without being affected by I/O devices. Sustained *frame rate* is the performance metric of interest in this application.

We have developed three versions of this application.

The first version uses Unix TCP/IP socket for communication between the client programs and the server program. The mixer (a single thread) obtains images from each client serially, generates the composite, and sends it to the clients serially. We chose TCP/IP sockets as other alternatives (Java RMI [4] or CORBA [3]) have higher overhead. The second version has a single threaded mixer using D-Stampede channels. (i.e. in Figure 4, the mixer in N_M is single threaded). The third version has a multi-threaded mixer using D-Stampede channels (i.e. in Figure 4, the mixer in N_M is multi threaded). There is one thread in the mixer for each client program to get the image from the client and perform a part of the composite image generation. Once the composite is complete, it is placed in the channel by a designated thread, for the client programs to pick up for display.

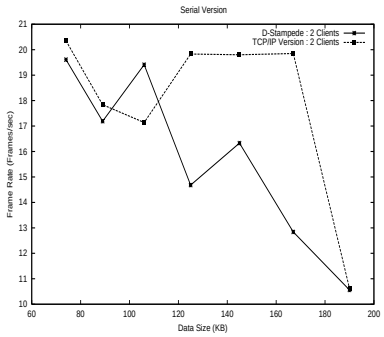

Figure 12. Single Threaded Version

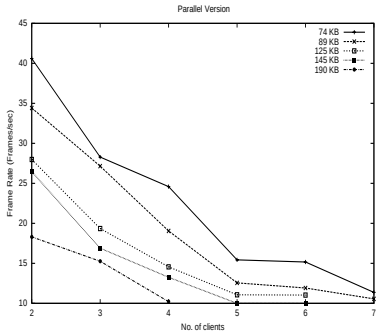

Figure 13. Multi-threaded Version

Data size (KB)	Delivered Bandwidth (MBps)					
	Number of Clients					
	2	3	4	5	6	7
74	11	18	28	28	39	42
89	11	21	26	28	40	46
125	13	20	29	36	48	
145	14	21	29	35	50	
190	13	25	30			

Table 1. Delivered Bandwidth (MBps) as a function of data size and number of clients.

In Figures 12 and 13, we only show readings when the sustained display thread frame rate is higher than 10 frames/sec. We feel that a lesser frame rate than this threshold would be unacceptable for a video conferencing. Figure 12 shows the performance of the first two single threaded versions, with two clients and for image sizes from 74 KB to 190 KB per client. This figure plots the sustained frame rate (on the Y-axis) as a function of the image data size (X-axis). As can be seen, the performance of the socket version and the D-Stampede channel version are comparable. For example, for a data size of 110 KB, they both deliver 18 frames/second. This exercise proved two things. 1) It takes much more effort to write this application using sockets compared to using D-Stampede. 2) The performance of D-Stampede and socket versions are comparable.

In Figure 13, the performance of the multi-threaded version is shown by plotting the sustained frame rate (Y-axis) as a function of the number of participants (X-axis). Each line in the graph is for a different client image size. Therefore, for an instance of the application with K clients, the display thread at each client receives a frame K times bigger than the client image size. The sustained frame rates measured at the different participants varied in a narrow band for each client image size. We chose the slowest display frame rate among the participants as the representative for each client image size.

Comparing Figures 12 and 13, we see that the multithreaded version performs better than the single threaded versions. For two clients with an image size of 74KB, the single threaded version delivers approximately 20 frames/sec. versus approximately 40 frames/second for the multithreaded version. Clearly, thread parallelism in the mixer helps to boost the sustained frame rate seen at the display threads.

Despite the thread parallelism in the mixer, it can be seen from Figure 13 that the sustained frame rate achieved is a function of the number of participating clients and the per client image data size. For an image size of 74 KB, we see a frame rate of around 40 frames/sec for 2 clients, which drops to around 30 frames/sec for 3 clients. Similarly, for 2 clients, with an 89 KB image size, we get approximately 34 frames/sec, which drops to approximately 27 frames/sec for 125 KB image size.

In this version of the application, all the threads of the mixer run in one node (an 8-way SMP) of the cluster. Thus all the client display threads feed out of this one node to get the composite image data from the channel. Since mixing is the most compute intensive operation in this application pipeline, it is highly likely that the requests from the display threads for the composite image are serviced simultaneously. Thus bandwidth probably becomes the limiting factor in application scalability.

For K clients, with a per client image size of S, and a frame rate F, the required bandwidth at this cluster node is

K^2SF (K clients, composite size KS). Table 1 summarizes the actual delivered bandwidth from this cluster node for the various client image sizes and number of clients. This table is derived from the measurements which are plotted in Figure 13. From the table, it can be seen that the sustained frame rate falls below 10 frames/sec when the required bandwidth exceeds 50 MBps, suggesting that this is perhaps the maximum available network bandwidth out of the cluster node. This situation happens with 5 clients when the image size is 190KB, and 7 clients for the other lesser image sizes. We point this out to emphasize that the observed limit to the scalability is the application structure and not any limitation in the D-Stampede implementation.

6 Conclusion

D-Stampede is a programming system that is designed for supporting interactive stream-based distributed ubiquitous computing applications. The key features that D-Stampede provides are indexing and correlating different data streams temporally, performing automatic distributed garbage collection of unnecessary stream data, providing support for parallelism, supporting platform and language heterogeneity, and dealing with the dynamism of such applications. We have illustrated the ease of developing ubiquitous computing applications on D-Stampede. Through micro-level and application-level measurements, we have shown that the programming ease attained with D-Stampede does not adversely affect performance. D-Stampede is available as a runtime library that runs on a variety of platforms. There are a number of system issues that we are currently working on. The first is generalizing the hardware model for which D-Stampede is intended. We would like to extend the D-Stampede to support heterogeneous clusters connected to a plethora of end devices participating in the same application. A second area of future research is dealing with failures, both toward developing a computational model as well as efficient runtime support for the model. Finally an area of ongoing research is the security and privacy issues in ubiquitous computing.

Acknowledgments

A number of people have contributed to D-Stampede. Rishiyur Nikhil, Jim Rehg, Kath Knobe, and Nissim Harel contributed to the space-time memory abstraction. Bert Halstead, Chris Jeorg, Leonidas Kontothanassis, and Jamey Hicks contributed during the early stages of the project. Dave Panariti developed a version of CLF for Alpha Tru64 and Microsoft NT. Members of the "ubiquitous presence" group at Georgia Tech continue to contribute to D-Stampede. Bikash Agarwalla, Matt Wolenetz, Hasnain Mandviwala, Phil Hutto, Durga Devi Mannaru, Namgeun Jeong, Yavor Angelov, and Ansley Post deserve special mention. Russ Keldorph and Anand Lakshminarayanan developed an audio and video meeting application.

References

[1] MPI Forum. *Message-Passing Interface Standard*. Mar 94.

[2] V. S. Sunderam. *PVM: A framework for parallel distributed computing*. Concurrency: Practice and Experience, Vol. 2(4), Dec 90.

[3] Object Management Group. *CORBA* www.corba.org

[4] Sun Microsystems. *Java RMI* www.java.sun.com

[5] S. Ahuja, N. Carriero and D. Gelernter. *Linda and friends*. IEEE Computer, Vol. 19(8), Aug 86.

[6] H.E. Bal, M.F. Kaashoek and A.S. Tanenbaum. *Orca: Language for Parallel Programming of Distributed Systems*. IEEE Trans. on SE, Vol. 18(3), Mar 90.

[7] R. S. Nikhil. *Cid: A parallel shared-memory C for distributed memory machines*. 7th Intl. Workshop on Languages and Compilers for Parallel Computing, Aug. 94.

[8] *Oxygen*. htt//oxygen.lcs.mit.edu

[9] R. H. Katz. *The Endeavour Expedition: Charting the fluid information utility*. htt//endeavour.cs.berkeley.edu

[10] *Infosphere* www.cc.gatech.edu/projects/infosphere

[11] R. S. Nikhil, U. Ramachandran, J. M. Rehg, R. H. Halstead, Jr., C. F. Joerg and L. Kontothanassis. *Stampede: A programming system for emerging scalable interactive multimedia applications*. 11th Intl. Workshop on Languages and Compilers for Parallel Computing, 98.

[12] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg and K. Knobe. *Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications*. 10th ACM SIGPLAN Symposium on PPoPP, May 99.

[13] R. S. Nikhil, and U. Ramachandran. *Garbage Collection of Timestamped Data in Stampede*. PODC, Jul 00.

[14] A. Singla, U. Ramachandran and J. Hodgins, *Temporal Notions of Synchronization and Consistency in Beehive*, 9th Annual ACM SPAA, Jun 97.

[15] A. Birrell and B. Nelson. *Implementing Remote Procedure Calls* ACM TOCS, Vol. 2(1), Feb 84.

[16] R. Srinivasan. *Rfc 1832: Xdr: External data representation standard*. Aug 95.

[17] R. Gillett, M. Collins, and D. Pimm. *Overview of network memory channel for PCI*. The IEEE Spring COMPCON 96, Feb 96.

[18] N. J. Boden, D. Cohen, R. E. Felderman, A.E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. *Myrinet: A Gigabit-per-Second Local Area Network*. IEEE Micro, Feb 95.