Implementing Infopipes: The SIP/XIP Experiment

Calton Pu[1], Galen Swint[1], Charles Consel[2], Younggyun Koh[1], Ling Liu[1], Koichi Moriyama[3], Jonathan Walpole[4], Wenchang Yan[1]

Abstract

We describe an implementation of the Infopipe abstraction for information flow applications. We have implemented software tools that translate the SIP/XIP variant of Infopipe specification into executable code. These tools are evaluated through the rewriting of two realistic applications using Infopipes: a multimedia streaming program and a web source combination application. Measurements show that Infopipe-generated code has the same execution overhead as the manually written original version. Source code of Infopipe version is reduced by 36% to 85% compared to the original.

1 Introduction

One of the fundamental functions of operating systems (OS) is to provide a higher level of programming abstraction on top of hardware to application programmers. More generally, an important aspect of OS research is to create and provide increasingly higher levels of programming abstraction on top of existing abstractions. Remote Procedure Call (RPC) [1] is a successful example of such abstraction creation on top of messages, particularly for programmers of distributed client/server applications.

We have proposed the Infopipe concept [16, 10, 9, 2] as a high level abstraction to support information-flow applications. Unlike RPC, which has clearly defined procedural semantics, Infopipe can have several flavors, depending on the kind of application for which it is being specialized. Examples include data streaming and filtering [10] and multimedia streaming [2]. In this paper, we describe the SIP/XIP variant of Infopipe currently under development at Georgia Tech, the software tools that implement SIP/XIP, and experiments that evaluate the concept as well as software tools.

The main contributions of this paper are the implementation of software tools for SIP/XIP Infopipes and an experimental evaluation. From a top-down perspective, our software tools consist primarily of a series of translators that successively creates appropriate abstract machine code from the previous higher level abstraction. Our experiments show that the execution overhead of SIP/XIP-generated code is minimal compared to a hand-written version (on the order of a few percent), but the gains in code simplicity are substantial (code size reduction between 36% and 85% of representative applications).

The rest of the paper is organized as follows. Section 2 summarizes the Infopipe abstraction. Section 3 outlines the implementation strategy. Section 4 describes the experimental evaluation results. Section 5 summarizes related work and Section 6 concludes the paper.

2 The Infopipe Abstraction

2.1 Background and Motivation

Remote procedure call (RPC) is a well-established mechanism for constructing distributed systems and applications, and a considerable amount of distrib-

[1] Center for Experimental Research in Computer Systems (CERCS), Georgia Institute of Technology, Atlanta, Georgia. {firstname.lastname}@cc.gatech.edu

[2] INRIA/LaBRI/ENSEIRB, Bordeaux, France. {consel@labri.fr}

[3] Sony Corporation, Tokyo, Japan. This author's work was done during an extended visit to Georgia Tech.

[4] OGI School of Science & Engineering, OHSU, Portland, Oregon. {walpole@cse.ogi.edu}

uted systems research has centered on it. RPC is based on the procedure call abstraction which raises the level of abstraction for distributed systems programming beyond raw message passing and naturally supports a request-response style of interaction that is common in many applications. The widespread use and acceptance of RPC has led to the development of higher-level architectural models for distributed system construction. For example, it is a cornerstone for models such as client/server, DCOM, and CORBA. The client/server model is widely considered to be a good choice for building practical distributed applications, particularly those using computation or backend database services.

On the other hand, while these models have proven successful in the construction of many distributed systems, RPC and message passing libraries offer limited support for information-driven applications. One example is bulk data transfers [6]. Another example is when information flows are subject to real-world timing constraints certain elements of distribution transparency – an often-cited advantage of RPC – can cause more problems than they solve. For example, restrictions on the available bandwidth or latency over a network link between two components of a media-streaming application are a serious concern and should not be hidden by the programming abstraction. Similarly, the reliability and security-related characteristics of a connection may be significant to applications that are streaming critical or sensitive information.

Several important emerging classes of distributed applications are inherently information-driven. Instead of occasionally dispatching remote computations or using remote services, such information-driven systems tend to transfer and process streams of information continuously (e.g., Continual Queries [11, 12]). Member of this class range from applications that primarily transfer information over the wires such as digital libraries, teleconferencing and video on demand, to applications that require information-intensive processing and manipulation, such as distributed multimedia, Web search and cache engines. Other applications such as electronic commerce combine heavy-duty information processing (e.g., during the discovery and shopping phase, querying a large amount of data from a variety of data sources [18]) with occasional remote computation (e.g., buying and updating credit card accounts as well as inventory databases).

We argue that an appropriate programming paradigm for information-driven applications should embrace information flow as a core abstraction and offer the following advantages over RPC. First, data parallelism among flows should be naturally supported. Second, the specification and preservation of QoS properties should be included. And third, the implementation should scale with the increasing size, complexity and heterogeneity of information-driven applications. We emphasize that such a new abstraction offers an alternative that complements RPC, *not* to replace it. In client/server applications, RPC is clearly the natural solution.

2.2 The Infopipe Abstraction

We have proposed the Infopipe concept [16, 10, 9, 2] as an abstraction for capturing and reasoning about information flow in information-driven applications. Intuitively, an Infopipe is the information dual of an RPC. Like RPCs, Infopipes raise the level of abstraction for distributed systems programming and offer certain kinds of distribution transparency. Beyond RPCs, Infopipe is specified by the syntax, semantics, and quality of service (QoS) properties. Examples of QoS properties include the quality, consistency, reliability, security and timeliness of the information flowing through Infopipes. In this paper, we only include enough description of Infopipes to make this paper self-contained. Many important Infopipe features such as QoS properties and restructuring of Infopipe (topics of active research) are beyond the scope of this paper.

A simple Infopipe has two ends – a consumer (input) end and a producer (output) end – and implements a unidirectional information flow from a single producer to a single consumer. The processing, buffering, and filtering of information happen in the middle of the Infopipe, between the two ends. As mentioned before, an Infopipe links information producers to consumers. The information producer exports an explicitly defined information flow, which goes to the input end of the Infopipe. After appropriate transportation, storage, and processing,

the information flows through the output end to the information consumer.

Infopipe is a language and system independent mechanism to process information in a distributed system. This is done on purpose since one of the main reasons for RPC's success among practical imperative programming languages is their universal adoption of the procedure call abstraction. As a consequence, stub generators are able to hide the tedious details of marshalling and unmarshalling parameters for all practical languages. There are two additional sources of problems in the implementation of stub generators: (1) the heterogeneity of operating systems and hardware, and (2) the translation between the language level procedure call abstraction and the underlying system level message-based implementation. The eventual definition of an Interface Description Language (IDL) solved both problems, by encapsulating the translation functions in a portable IDL compiler.

Our approach to making Infopipes language and system independent parallels that used in RPC. We define a generic interface for Infopipe manipulation, and use the equivalent of IDL and stub generators to hide the technical difficulties of marshalling and unmarshalling data and manipulating system-specific mechanisms for QoS property enforcement. By adopting this approach we shield the application developer from the complexity of heterogeneous operating systems and hardware and the translation from language-level abstractions to underlying message-based implementations.

2.3 Infopipe Specification Language

The specification of Infopipe is divided into three components: syntax, semantics, and QoS properties. The software that wraps the first two components corresponds directly to RPC stub generators, since an Infopipe Specification Language compiler can generate the plumbing code so Infopipe programmers don't have to write code to manipulate the explicit representation and description of an Infopipe.

Between its consumer and producer ends, an Infopipe is a one-way mapping that transforms information units from its input domain to the output range. Probably it is not surprising to the reader that there are many concrete examples of existing information flow software. A familiar example is the Unix filter programs. Combining filters we have a Unix pipeline, which is a precursor of the Infopipe programming style. Another concrete example of information flow manipulation language is SQL in relational databases.

In this paper, we use the SIP (for Specifying InfoPipes) variant of Infopipe Specification Languages. SIP is a domain-specific language being developed at Georgia Institute of Technology to support information flow applications. SIP is a generic Infopipe specification language that supports a number of communications abstract machines, including the ECho publish/subscribe messaging middleware and the common TCP socket/RPC invocations. Since our focus is on the implementation and evaluation, we omit the language definition and include examples in the Appendix as illustration. From the system point of view, SIP is similar to other domain-specific languages such as Devil [14] for writing device drivers. SIP encapsulates domain knowledge (in this case, distributed information flow applications and communications mechanisms) so the applications written in SIP can be more concise and portable.

Composition of Infopipes is an active area of research and space constraints limit the number of experiments in this paper. In Section 4.4, we outline an experiment with a simple serial composition of Infopipes in an application that combines information from several web sources. This small experiment only illustrates the potential interesting problems in the area of Infopipe composition.

3 Implementation Outline

3.1 Implementation Strategy

Our design of software tools to translate SIP into executable code consists of two steps. First, SIP is translated into an intermediate representation, called XML Infopipe Specification Language (XIP). Then, XIP is translated into executable code using one of the communications abstraction machines. There are three main reasons for this intermediate representation and translation steps.

First, we are planning for several variants of Infopipe Specification Language, of which SIP is just one instance. This is an area of active research,

particularly from the domain specific language point of view. Each variant may also evolve over time, as new functionality is added. Instead of trying to create and maintain different software tools for each variant of Infopipe Specification Language, we decided to create a standard extensible intermediate representation based on XML (XIP). This way, the second step (the actual code generation) can be developed in parallel to the design and evolution of the variants of Infopipe Specification Languages.

Second, we are planning the generation of code for several communications abstract machines. The experiments described below use a publish/subscribe event messaging mechanism called ECho. A standard format such as XIP simplifies the addition of new abstract machines for the code generator. We also have implemented a prototype version that translates XIP into RPC and sockets, which have lower overhead for message exchanges.

Third, we will be attaching a variety of metadata to the data stream being carried by Infopipes. This metadata includes data provenance annotations (e.g., when and where the information was generated, and how it was processed) and other data processing instructions (e.g., filtering algorithms that understand the semantics of this particular data stream). Further discussion of the metadata issue is beyond the scope of this paper, but it is an important reason for the XIP standard format.

Currently, the first step of SIP translation (into XIP) is done by hand. This is primarily due to the fast evolution of SIP. The second step (from XIP into executable) is described in the following section.

3.2 Code Generation Process

We skip the details of XIP in this paper, since it is an intermediate representation invisible from the programmer's point of view. Furthermore, XIP is used only during code generation and therefore contributes little to the run-time overhead, the other major concern of this paper. At the risk of oversimplification, XIP can be described as a union of all variants of Infopipe Specification Languages. By union we mean combined functionality from these variants, not syntax. We chose XML due to its extensibility, capable of handling all the three aspects of an Infopipe (syntax, semantics, and QoS). Even though XML was originally designed as a data interchange format, not an intermediate representation, it has worked very well so far.

The translation of XIP into executable code is accomplished through a series of transformations on the XIP specification of Infopipe. For convenience, we call these internal representations XIP+*k*, where *k* is the number of stage in the series. The input files for the XIP translator are the XIP specification of Infopipe and the abstract machine description (executable code templates) file.

- The main transformation from XIP to XIP+1 is the explicit naming of all inputs and outputs, by using the information in the XIP file and the abstraction machine description.
- The transformation from XIP+1 to XIP+2 is the flattening of composite Infopipes into elementary Infopipes (with one input and one output) plus the syntactic data types, data filters, and aspect [7] (e.g., end-to-end latency management) templates.
- From XIP+2 to XIP+3, the aspects doing work are filled in, while the unnecessary aspects are removed.
- From XIP+3 to XIP+4, the aspects are woven together and the templates are used to generate executable code from XIP+4 and the abstraction machine description file.

In the current implementation, we generate code for two concrete communications abstract machines: (1) the ECho publish/subscribe messaging facility, and (2) the popular Unix sockets interface. Also, the translation process from XIP to XIP+4 is in main memory for performance reasons. The transformation algorithms are designed so each stage can write XIP+*k* to disk to accommodate arbitrarily large XIP descriptions.

4 Experimental Evaluation

4.1 The Statistical Treatment

Many system components are involved in the measurement of software systems such as ours, with variations being introduced by the hardware (e.g., cache misses), OS kernel variations (e.g., schedul-

scheduling and memory management decisions), and network (e.g., very short temporary interferences with other nodes). This is particularly the situation with I/O operations such as Infopipes. Some operations (e.g., Infopipe initialization) cannot be repeated many times in a warmed cache to reduce variance, since their normal mode of operations is an execution without warmed cache.

Therefore, we took some care in our evaluation to clarify the interpretation of measured results. We are using a simple statistical treatment called two-sample t-test, where the mean of two sets of measurement results are compared. We assume two independent sets of random samples, each consisting of independent and identically distributed random variables. Our null hypothesis is that the means from the two samples are equal, i.e., the difference between the two sets of measurements is statistically not meaningful. To decide whether to accept or reject the null hypothesis, we put the *t-statistic* (derived from the two samples) into a student-t distribution and adopt 95% confidence interval in the test. For most of the experiments, the sample size was 100 (the same experiment was run 100 times).

4.2 Microbenchmarks

The first set of experiments consists of microbenchmarks to evaluate the overhead of Infopipe basic functions. The hardware used in the experiments is a pair of Dell dual-CPU workstations with Pentium III (800MHz, 512MB, 256KB L1 cache) running Linux 2.4.9-smp. The two machines are connected through a lightly loaded 100Mb Ethernet and sharing the same file system.

The first microbenchmark measures the overhead of transmitting one single integer, repeated 100,000 times. (As mentioned above, each test is repeated 100 times and the mean of the result compared.) This can be seen as the worst case scenario that maximizes the transmission overhead. For the results below, we see that obviously sockets carry lower overhead than ECho.

Single Integer	Mean Time	Std. Dev.
ECho/Infopipe	2.0 sec	0.015 sec
TCP socket	0.12 sec	0.003 sec

The second microbenchmark measures the overhead of transmitting 1000 integers, repeated 10,000 times. (As mentioned above, each test is repeated 100 times and the mean of the result compared.) This can be seen as the normal case for bulk transmissions. For the results below, we have a t-statistic of 146.6, so even though the ECho version is only slightly slower than sockets (about 2% difference), the difference is statistically significant at 95% confidence interval. Intuitively, the small difference is significant because the measurements have been very precisely reproducible (with standard deviations that are one order of magnitude smaller than the difference in response time.

1000 Integers	Mean Time	Std. Dev.
ECho/Infopipe	3.46 sec	0.004 sec
TCP sockets	3.39 sec	0.003 sec

In these microbenchmarks, an obvious experiment would be the comparison between the Infopipe-generated code using ECho and manually written ECho code, or a similar comparison using TCP sockets. Since the code and the measured results are the same, we omit them here. See the next Section for similar results.

4.3 Data Streaming Experiment

Our first system level experiment is an evaluation of Infopipes for a multimedia streaming application. This application is representative of many distributed information flow applications where bulk data streaming happens. Our application has real-time requirements (unprocessed bits drop on the floor) that are implemented by quality of service (QoS) support. Although QoS is an integral part of Infopipe research, it is a complex topic. We will report on Infopipe support for QoS in a paper dedicated to that topic. In this paper, we focus on the effectiveness of Infopipe as a high level abstraction for information flow applications.

The evaluation consists of two parts. The first part is a comparison of measured overhead of two versions of the application: the *original* version was hand-written and the *Infopipe* version is the same application written using SIP/XIP Infopipes. This is a refinement of the microbenchmarks in Section 4.1, and shows the effectiveness of our implementa-

tion in a realistic scenario. The second part is a comparison of the source code length between the original version and the Infopipe version. This is an evaluation of the effectiveness of the Infopipe abstraction for the programming of information flow applications.

The multimedia streaming application is a medium-sized demonstration program being developed for DARPA's PCES program. The program includes contributions from several universities and is integrated by BBN. The current version of the program (successfully integrated and demonstrated in April 2002) gathers video input from several sources, processes them, and redistributes the video streams to several destinations including video displays and automatic target recognition programs. Although the program contains significant technical innovation such as quality of service control, in this experiment we focus on the effect of Infopipe abstraction in terms of performance overhead and code size.

The experiment consists of taking the original application code and rewriting it using Infopipes for information flow processing. Both the original version and the Infopipe version use the same publish/subscribe communications middleware called ECho [5]. The video streams are 320X240 pixels, 24-bit color depth raw images in the Unix Portable Pixmap (PPM) format.

The measurements were conducted on a Dell laptop (700 MHz Pentium III, 256 MB memory) running Linux 2.4.2. The following table shows the measured overhead of ECho channel initialization time for both versions. We ran the program 100 times with a cold start initialization (new process). The statistical tests show a significant difference for the initialization time (t-statistic = -8.96). The small difference is due to minor differences in the code generated.

Initialization	Mean Time	Std. Dev.
Original	26.0 ms	0.4 ms
Infopipe	28.2 ms	2.4 ms

We also measured the time it takes to transfer a frame (the steady state). The table below shows the measured overhead as mean over 100 runs. The statistical analysis shows no significant difference for the steady state performance of the two versions (t-statistic = -1.85).

Frame Trans.	Mean Tune	Std. Dev.
Original	320.4 ms	4.1 ms
Infopipe	319.4 ms	3.5 ms

For the quantitative source code evaluation, we restricted our attention to the 1182 lines (not including blanks and comments) in 5 source files that refer to video streams, at both sender and receiver. The application consists of approximately 15,000 lines of code, using many significant and relatively large middleware packages such as ECho (publish/subscribe messaging middleware). From these files, 441 lines are closely related to ECho. The application was rewritten using Infopipes (SIP) and hand-translated into XIP. The source code for this experiment is included in Appendix 7.1.

While the code savings are potentially better with SIP, a domain-specific language designed to support information flow, we decided to compare primarily with XIP. Although XIP is more verbose, it is also more "general-purpose" in its coverage of many flavors of Infopipe specification languages. Consequently, it is more directly comparable with the original hand-written code. This comparison also becomes independent of specific Infopipe Specification Language syntax. Comparing the XIP version to the original version, 12 lines were added, 171 lines were removed, and 37 lines were changed. The following table summarizes the change process from the original version to the Infopipe version. The result is the elimination of about 36% of the original source code related to information flow (in lines of code – loc).

Infopipe-Related	Code Added	Code Removed	Code Modified
441 loc	12 loc	171 loc	37 loc

4.4 Web Source Composition Experiment

Our second system level experiment is an evaluation of Infopipes for a web information processing application. It takes an address, fetches a map for

that address, and filters the map for display on a personal digital assistant (PDA) with limited resolution, capability (e.g., grayscale only), and network bandwidth. This is an application that could be written using an "agent" style of programming. The control passes from site to site, gathering information or processing and filtering the information. Eventually it produces a useful result.

Instead of using a control-driven model such as agents, we model the application as an information flow, which is implemented using Infopipes. Although we no longer have "agents" visiting different sites, the information flow goes through the appropriate sites and the information is augmented, processed, filtered, and transformed along the way. The result is useful information at the end of the composite Infopipe.

The concrete implementation of the application has four main components. At the beginning is a GUI with a wrapper to translate its output into an XML format. The GUI collects the user input (address) and through the wrapper sends it to the first stage of information pipeline, GetMapURL, which sends the address to MapQuest for translation. MapQuest sends back the URL of a color map. The URL is passed to the second stage of information pipeline, GetMapImage, which fetches the map (also from MapQuest in this particular case). Once GetMapImage receives the map, it passes the data to the third stage of the information pipeline, ImageConverter, which filters the image to an appropriate grayscale image of appropriate resolution for the PDA. At the end, ImageConverter sends the results back to the GUI running on the PDA, which then displays the grayscale image.

We also divided this experiment into two parts. Since there is no sustained data transfer, the first part (execution overhead) was done on the latency of application execution. We used the same desktops described in Section 4.2 with the same configuration (single machine). The GUI was run as a PalmOS application on the PalmOS Emulator 3.0a7.

For this kind of applications, the latency measurements are usually dominated by network access times. In addition, since there are external accesses (e.g., twice to MapQuest.com), it is difficult to reproduce measured results. Despite the large variances, the Mean measured latencies (over 10 executions) of the two versions show no statistically significant difference.

Latency	Mean	Std. Dev.
Hand-written	6.18 sec	0.12 sec
Infopipe	6.22 sec	0.15 sec

The second part of the experiment, quantitative code comparison, showed a more dramatic code reduction. This is due to the repeated I/O management code in each stage of information pipeline plus data, socket, and XML handling code, when XML data streams must be parsed and interpreted. By generating these "bureaucratic" code segments automatically, the Infopipe version is able to reduces the code count to about 15% of the hand-written code size.

Web Compos.	Hand-written	Infopipe
GetMapURL	95 loc	19 loc
GetMapImage	104 loc	28 loc
ImageConverter	121 loc	43 loc
House-keeping	507 loc	26 loc
Total	827 loc	116 loc

5 Related Work

Remote Procedure Call (RPC) [1] is the basic abstraction for client/server software. By raising the level of abstraction, RPC facilitated the programming of distributed client/server applications. For example, RPC automates the marshalling and unmarshalling of procedure parameters, a tedious and maintenance-heavy process. Despite its usefulness, RPC provides limited support for information flow applications such as data streaming, digital libraries, and electronic commerce. To remedy these problems, extensions of RPC such as Remote Pipes [6] were proposed to support bulk data transfers and sending of incremental results.

Instead of trying to extend further RPC-style abstractions, which provide convenient building blocks for the programming of *distributed computations*, Infopipes can be seen as a complementary

abstraction to RPC and its derivatives. For distributed data streaming, for example, Infopipes provide good abstractions for distributed information flow with "local" computations (as filters within Infopipes).

6 Conclusion

In this paper, we briefly motivate and summarize the concept of Infopipe [16, 10, 9, 2] to support distributed information flow applications. Then, we describe the implementation of the SIP variant of Infopipe Specification Languages. The implementation translates SIP into an XML-based intermediate representation called XIP, which is then stepwise transformed into executable code.

We used one set of microbenchmarks and two realistic applications for an experimental evaluation of the SIP/XIP software tools. The measurement results show that the run-time overhead of generated Infopipe is comparable to the manually written code. For example, statistically there is no difference for steady state data transfers, with only 7% additional overhead for Infopipe initialization.

The evaluation of the source code for these experiments shows a significant reduction in number of lines of code. The Infopipe code is 36% smaller than the original code for the multimedia streaming application and reduced to only 15% of the original code for the web source combination application. The declarative nature of SIP/XIP also makes it easier to write and maintain the Infopipe version of these applications (see Appendix Section 7 for a direct comparison).

These experiments show the promise of the Infopipe approach and the advantages of our SIP/XIP implementation strategy. We are moving forward with the addition of QoS support and the application of program specialization techniques [13, 15, 17] to improve the performance of generated code.

Funding Acknowledgements

This work was done as part of the Infosphere project, funded by DARPA through the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs. The research was also partially funded by NSF's CISE directorate, through the ANIR and CCR divisions. In addition, the research was partially funded by Intel.

References

1. A. Birrell and B. Nelson, "Implementing Remote Procedure Calls", in *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, Pages 39-59. Also appeared in Proceedings of SOSP'83.
2. A. Black, J. Huang, R. Koster, J. Walpole, and C. Pu, "Infopipes: an Abstraction for Multimedia Streaming", in ACM Multimedia Systems Journal. To appear in 2002.
3. G. Box, S. Hunter, and W. Hunter, "Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building", John Wiley and Sons, 1978.
4. D. Buttler, L. Liu, and C. Pu, ``A Fully Automated Object Extraction System for the World Wide Web", to appear in the *Proceedings of the 2001 International Conference on Distributed Computing Systems (ICDCS'01)*, May 2001, Phoenix, Arizona.
5. G. Eisenhauer, F. Bustamante and K. Schwan, "Event Services in High Performance Systems", *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, Vol. 4, Num. 3, July 2001, pp 243-252
6. D. Gifford and N. Glasser, "Remote Pipes and Procedures for Efficient Distributed Communication", in *ACM Transactions on Computer Systems*, Vol. 6, No. 3, August 1988, Pages 258-283.
7. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, "Aspect-Oriented Programming". *Springer-Verlag LNCS* Vol. 1241. June 1997. Also in the *Proceedings of ECOOP, Finland, 1997.*
8. R. Koster, A. Black, J. Huang, J. Walpole and C. Pu, "Thread Transparency in Information Flow Middleware". In the *Proceedings of Middleware 2001 - IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, November, 2001.
9. R. Koster, A. Black, J. Huang, J. Walpole and C. Pu, "Infopipes for Composing Distributed Information Flows". In the *Proceedings of the*

ACM Multimedia Workshop on Multimedia Middleware, Ottawa, Canada, October 2001.
10. L. Liu, C. Pu, K. Schwan and J. Walpole, "InfoFilter: Supporting Quality of Service for Fresh Information Delivery", *New Generation Computing Journal* (Ohmsha, Ltd. and Springer-Verlag), Special issue on Advanced Multimedia Content Processing, Vol. 18, No. 4, August 2000.
11. L. Liu, C. Pu, W. Tang, and W. Han, "Conquer: A Continual Query System for Update Monitoring in the WWW", *International Journal of Computer Systems, Science and Engineering*. To appear in the Special issue on Web Semantics, 1999.
12. L. Liu, C. Pu, and W. Tang, "Continual Queries for Internet Scale Event-Driven Information Delivery", *IEEE Transactions on Knowledge and Data Engineering*, Special issue on Web Technologies, Vol. 11, No. 4, July/August 1999.
13. D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet, "Specialization Tools and Techniques for Systematic Optimization of System Software", *ACM Transactions on Computer Systems*, Vol. 19, No. 2, May 2001, pp 217-251.
14. F. Merillon, L. Reveillere, C. Consel, R. Marlet, G. Muller, "Devil: An IDL for Hardware Programming", in *Proceedings of the 2000 Conference on Operating System Design and Implementation (OSDI),* pp 17-30, San Diego, October 2000.
15. G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu and A. Goel, "Fast, Optimized Sun RPC using Automatic Program Specialization", *Proceedings of the 1998 International Conference on Distributed Computing Systems*, Amsterdam, May 1998.
16. C. Pu, K. Schwan, and J. Walpole, "Infosphere Project: System Support for Information Flow Applications", in *ACM SIGMOD Record*, Volume 30, Number 1, pp 25-34, (March 2001).
17. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole and K. Zhang, "Optimistic Incremental Specialization: Streamlining a Commercial Operating System", *Proceedings of the Fifteenth Symposium on Operating Systems Principles (SOSP'95),* Colorado, December 1995.
18. D. Steere, A. Baptista, D. McNamee, C. Pu, and J. Walpole, "Research Challenges in Environmental Observation and Forecasting Systems", *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom'00)*, Boston, August 2000.
19. D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A Feedback-Driven Proportion Allocator for Real-Rate Scheduling", *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI'99),* New Orleans, February 1999.

7 Appendices

In the source code attached below, we use the normal font (between sizes 11 and 9) to show areas of interest. When code is irrelevant for our comparison purposes, we reduce it to an illustrative size (sizes 4 or smaller) to show that the code is there, but it is not part of our comparison.

7.1 SIP Example – Multi-UAV Application

The first step in integrating an infopipe into an application is to write the SIP specification for it. This involves creating declarations for data types, filters, and pipes. Data types are built out of primitive types which roughly mirror types available in C. Eventual plans are to mirror the types available as part of the SOAP specification.

Our contribution to the Multi-UAV demo involves generating the communication code from the sending process to the player, which displays the images as a movie. In this case, we have two infopipes which are composed together. The first infopipe makes the data available on the network, and the second infopipe delivers the data to the player. The data type for the exchange is also specified in SIP. The application relies on several filters to process the information before transmission to reduce network load. These can be defined by name and referenced in the specification of an infopipe.

The SIP version of the Multi-UAV demo:

```
//320x240 color image
add type Raw_data_2C
( tag:integer,
  ppm1:byte,
  ppm2:byte,
  size:integer,
  width:integer,
  height:integer,
  buffer:array[230400] of byte);


//320x240 grey scale image
add type Raw_data_2G
( tag:integer,
  ppm1:byte,
  ppm2:byte,
  size:integer,
  width:integer,
  height:integer,
  buffer:array[76800] of byte);


add filter Fdata2G
    input Raw_data
    output Raw_data1G
    source "../strlt/greyImage.ecl";


add ipipe Source
    input none //no input since we are
               //source - a half-pipe
               //really
    output Raw_data; //output data
                     //type defined
                     //above


compose Player2GPipe
        input Source //get our input
                     //from the pipe
                     //"Source"
        output none  // make this op-
                     // tional if none
        withfilter Fdata2G;
```

From the SIP code, XIP code is produced and in turn executable code is generated from the XIP code. Generation of XIP is a straightforward conversion from SIP. The XML form adds no new information to the specification.

The XIP code of the Multi-UAV demo:

```
<InfopipeSpec name="Player2G">
<dataDef name="Raw_data_2C" >
  <arg type="integer" name="tag"/>
  <arg type="char" name="ppm1"/>
  <arg type="char" name="ppm2"/>
  <arg type="integer" name="size"/>
  <arg type="integer" name="width"/>
  <arg type="integer" name="height"/>
  <arg type="integer" name="maxval"/>
  <arg type="char" size="size"
                   name="buff"/>
</dataDef>

<dataDef name="Raw_data_2G" >
  <arg type="integer" name="tag"/>
  <arg type="char" name="ppm1"/>
  <arg type="char" name="ppm2"/>
  <arg type="integer" name="size"/>
  <arg type="integer" name="width"/>
  <arg type="integer" name="height"/>
  <arg type="integer" name="maxval"/>
  <arg type="char" size="size"
                   name="buff"/>
</dataDef>

<pipe name="Source"
      inType="CAPPED"
      outType="Raw_data_2C">
</pipe>
<pipe name="Player2CPipe"
      inType="CAPPED" outType="CAPPED">
  <connections>
    <join pipe="Source" />
    <filter name="FData1G"
       location="greyImage.ecl"/>
  </connections>
</pipe>
</InfopipeSpec>
```

Instead of showing the C code generated from XIP, we include here the original version (also written in C) of the Multi-UAV application program. It is substantially similar to the one generated by XIP (as demonstrated in measured overhead in Section 4.3) and it shows the difference in code quantity and quality as discussed in that section. As can been seen below, there is a lot of code devoted to creating connections and initializing the environment. The areas of large text indicate code that is replaced by the generated code.

```
/* avs_raw.h */
#ifndef RAW_ECHO_INCLUDED
#define RAW_ECHO_INCLUDED

#include <io.h>
#include <common.h>


typedef struct {
  int tag;
  char ppm1;
  char ppm2;
  int size;
  int width;
  int height;
  char *buff;
} Raw_data, *Raw_data_ptr;

extern IOField Raw_data_fld[];

/* 1 - 640 * 480 **********/
```

```
#define AVSIMAGE1C  921600
/* 640 * 480 - color */

typedef struct {
  int tag;
  char ppm1;
  char ppm2;
  int size;
  int width;
  int height;
  int maxval;
  char buff[AVSIMAGE1C];
} Raw_data1C, *Raw_data1C_ptr;

extern IOField Raw_data1C_fld[];

#define AVSIMAGE1G  307200
 /* 640 * 480 - grey  */

typedef struct {
  int tag;
  char ppm1;
  char ppm2;
  int size;
  int width;
  int height;
  int maxval;
  char buff[AVSIMAGE1G];
} Raw_data1G, *Raw_data1G_ptr;

extern IOField Raw_data1G_fld[];
```

```
#define AVSIMAGE1DC    1843200  /* 640 * 480 * 2 - color */

typedef struct {
  int tag;
  char ppm1;
  char ppm2;
  int size;
  int width;
  int height;
  int maxval;
  char buff[AVSIMAGE1DC];
} Raw_data1DC, *Raw_data1DC_ptr;

extern IOField Raw_data1DC_fld[];

#define AVSIMAGE1DG    614400  /* 640 * 480 * 2 - grey  */

typedef struct {
  int tag;
  char ppm1;
  char ppm2;
  int size;
  int width;
  int height;
  int maxval;
  char buff[AVSIMAGE1DG];
} Raw_data1DG, *Raw_data1DG_ptr;

xtern IOField Raw_data1DG_fld[];
```

```
/* Some other data format */
typedef struct {
  int tag;
  char ppm1;
  char ppm2;
  int size;
  int width;
  int height;
  int maxval;
  char buff[AVSIMAGE3G];
} Raw_data3G, *Raw_data3G_ptr;

extern IOField Raw_data3G_fld[];
```

```
#

typedef struct {
  int tag;
  char ppm1;
  char ppm2;
  int size;
  int width;
  int height;
  int maxval;
  char buff[AVSIMAGE3DG];
} Raw_data3DG, *Raw_data3DG_ptr;

extern IOField Raw_data3DG_fld[];
```

```
#define AVSIMAGE4DC    1843200  /* 80 *  60 * 2 - color */

typedef struct {
  int tag;
  char ppm1;
  char ppm2;
  int size;
  int width;
  int height;
  int maxval;
  char buff[AVSIMAGE4DC];
} Raw_data4DC, *Raw_data4DC_ptr;

extern IOField Raw_data4DC_fld[];


#endif /* RAW_ECHO_INCLUDED */
```

```
/*avs_raw.c */
#if HAVE_CONFIG_H
#  include <config.h>
#endif

#include <stdio.h>
#IOField Raw_data1DG_fld[] = {
  {"tag","integer",sizeof(int),IOOffset(Raw_data1DG_ptr,tag)},
  {"ppm1","char",sizeof(char),IOOffset(Raw_data1DG_ptr,ppm1)},
  {"ppm2","char",sizeof(char),IOOffset(Raw_data1DG_ptr,ppm2)},
  {"size","integer",sizeof(int),IOOffset(Raw_data1DG_ptr,size)},
  {"width","integer",sizeof(int),IOOffset(Raw_data1DG_ptr,width)},
  {"height","integer",sizeof(int),IOOffset(Raw_data1DG_ptr,height)},
  {"buff", IOArrayDecl(char,AVSIMAGE1DG), sizeof(char),
   IOOffset(Raw_data1DG_ptr,buff[0])},
  {NULL,NULL},
};


/* 2 - 320 * 240 ***********/

IOField Raw_data2C_fld[] = {
  {"tag","integer",sizeof(int),IOOffset(
    Raw_data2C_ptr,tag)},
  {"ppm1","char",sizeof(char),IOOffset(
    Raw_data2C_ptr,ppm1)},
  {"ppm2","char",sizeof(char),IOOffset(
    Raw_data2C_ptr,ppm2)},

{"size","integer",sizeof(int),IOOffset(
    Raw_data2C_ptr,size)},

{"width","integer",sizeof(int),IOOffset
   (Raw_data2C_ptr,width)},
  {"height","integer",sizeof(int),
    IOOffset(Raw_data2C_ptr,height)},
  {"buff", IOArrayDecl(char,AVSIMAGE2C),
    sizeof(char),
    IOOffset(Raw_data2C_ptr,buff[0])},
  {NULL,NULL},
};

IOField Raw_data2G_fld[] = {
  {"tag","integer",sizeof(int),IOOffset(
    Raw_data2G_ptr,tag)},
  {"ppm1","char",sizeof(char),IOOffset(
    Raw_data2G_ptr,ppm1)},
  {"ppm2","char",sizeof(char),IOOffset(
    Raw_data2G_ptr,ppm2)},

{"size","integer",sizeof(int),IOOffset(
    Raw_data2G_ptr,size)},

{"width","integer",sizeof(int),IOOffset
    (Raw_data2G_ptr,width)},
  {"height","integer",sizeof(int),
    IOOffset(Raw_data2G_ptr,height)},
  {"buff", IOArrayDecl(char,
    AVSIMAGE2G),sizeof(char),
   IOOffset(Raw_data2G_ptr,buff[0])},
  {NULL,NULL},
};
```

```
/* avs_source.c */
/* some global variables, helper
   functions omitted … */
int main(argc, argv)
    int argc;
    char* argv[];
{
 /* … */
 /*Creation of channel and
    registration*/
  gen_pthread_init();
  cm = CManager_create();
  CMfork_comm_thread(cm);
  if (signal(SIGINT, interruptHandler)
             == SIG_ERR)
    Styx_errQuit("Signal error");

  ec = ECho_CM_init(cm);

  chan2C = EChannel_typed_create(ec,
               Raw_data2C_fld, NULL);
  if (chan2C == NULL)
    Styx_errQuit("Failed to create 2C
                  channel.\n");
    sourceHandle2C =
       ECsource_typed_subscribe(chan2C,
                  Raw_data2C_fld, NULL);

    chan = EChannel_typed_create(ec,
                    Raw_data_fld, NULL);
  if (chan == NULL)
    Styx_errQuit("Failed to create
                  channel.\n");
  fprintf(stdout, "Echo channel ID:\n
             %s\n", ECglobal_id(chan));
  sourceHandle =
         ECsource_typed_subscribe(chan,
                    Raw_data_fld, NULL);
  fprintf(stdout, "\nEcho 2C (320x240-
            color) channel ID:\n %s\n",
                  ECglobal_id(chan2C));


    /*
    if (debugging) {
      sprintf(shotsentFile, "shotsent%d.ppm", i+1);
      debuggingfd = open(shotsentFile,
                         O_CREAT|O_WRONLY, S_IRWXU|S_IRGRP|S_IROTH);
      sprintf(header, "%c%c\n%d %d\n%d\n", rawrec2CP->ppm1, rawrec2CP->ppm2,
              rawrec2CP->width, rawrec2CP->height, 255);
      write(debuggingfd, header, sizeof(header));
      if (rawrec2CP->buff != NULL)
        write(debuggingfd, rawrec2CP->buff, rawrec2CP->size);
      close(debuggingfd);
    }
    */


    free(rawrecP->buff);
   }
   else {
    debugrecP->tag = i+1;
    ECsubmit_typed_event(sourceHandle, debugrecP);
   }
   ++NumRecSubmitted;
    /*^^^*/

   if (MeasureMe)
     if (NumRecSubmitted == reported * reportFreq) {
       bwHist = (double*) malloc (((int) (NumRecSubmitted/Freq)) *
                                  sizeof(double));
       sensNet_GetHistory(bwHist, (int) (NumRecSubmitted/Freq), SENSNET_WRITE);
       if (bwHist == NULL) Styx_errQuit("Not enough resources.\n");

       fprintf(stdout, "Bandwidth mean:   \t%.3g Mbps [sdev %2.3g]\r",
               Stats_mean (bwHist, (int) (NumRecSubmitted/Freq)),
               Stats_sdev (bwHist, (int) (NumRecSubmitted/Freq)));
       fflush(stdout);
       free(bwHist);
       ++reported;
     }

  } /* end for */

  dumpStats();

  if (MeasureMe) sensNet_Finish();
  //ECchannel_destroy(chan);
  //CManager_close(cm);

  exit (EXIT_SUCCESS);
} /* main */
```

7.2 *Web Source Combination Application*

In the java version of SIP/XIP Infopipe, all the data flowing through Infopipes are XML-formatted. For example, there is a "mapImage" data type for containing data of a map image. The data format of mapImage exchanged between infopipes will be like this.

```
<infopipeDataFormat version="0.1">
<dataContent type="mapImage">
  <conteType>_contentTypeOfImage_
  </contentType>
  <contentTransferEncoing>
              _EncodingType_
  </contentTransferEncoding>
  <contentBody>_Body_</contentBody>
</dataContent>
</infopipeDataFormat>
```

Each Infopipe parses the XML data, and generates another XML-formatted data stream after processing. Without Infopipes, programmers need to add the parsing and generating code as shown below:

```
public void parseXML(Reader in) throws
    Exception
{
    InputSource inputSource =
                    new InputSource(in);
    DOMParser parser = new DOMParser();
    try {
        parser.parse(inputSource);
       } catch (IOException ioe) {
              ioe.printStackTrace();
              throw new
              Exception(ioe.getMessage());
       } catch (SAXException se) {
            se.printStackTrace();
            throw new
               Exception(se.getMessage());
       }
    Node root =
parser.getDocument().getDocumentElement();
    Node dataNode = null;
    Node currNode = null;
    NodeList nodeList = null;
    try {
       dataNode =
       XPathAPI.selectSingleNode(root,
                              "dataContent");
       if (dataNode == null ||
           !((Element)
   dataNode).getAttribute("type").equals(
"mapImage"))
```

```
        {
         throw new
         Exception(
           "InfopipeDataType_mapImage:
              invalid data");
        }
    currNode = null;
    currNode=XPathAPI.selectSingleNode
      (dataNode,"contentType/text()");
    if (currNode != null) {
      if (currNode.getNodeType() ==
                       Node.TEXT_NODE)
         contentType =
            currNode.getNodeValue();
      } else {
          throw new Exception();
      }
    } else {
         throw new Exception
          ("InfopipeDataType_mapImage:
                contentType is null");
    }
    currNode =
       XPathAPI.selectSingleNode(
         dataNode,
       "contentTransferEncoding/text()"
        );
    if (currNode != null) {
       if (currNode.getNodeType() ==
           Node.TEXT_NODE)
       { contentTransferEncoding =
            currNode.getNodeValue();

       }
       else
          throw new Exception();
       }
    } else {
       throw new Exception (
          "InfopipeDataType_mapImage:
          contentTransferEncoding is
          null");
     }
    currNode =
       XPathAPI.selectSingleNode(
       DataNode, "contentBody/text()");
    if(currNode != null) {
        if (currNode.getNodeType() ==
            Node.TEXT_NODE) {
          contentBody =
             currNode.getNodeValue();
           }
        else {
          throw new Exception();
        }
     }
     else {
          throw new Exception();
     } catch (SAXException se) {
         se.printStackTrace();
         throw new Exception
                (se.getMessage());
     }
}

public String formatToXML() {
    String doc = new String();
    doc = "<infopipeDataFormat
        version=\"0.1\"><dataContent
          type=\"mapImage\">";
    if (contentType == null) {
       doc = doc +
       "<contentType></contentType>";
    }
    else {
       doc = doc + "<contentType>" +
      contentType + "</contentType>";
    }

    if (contentTransferEncoding==null)
       {
        doc = doc +
          "<contentTransferEncoding>
          </contentTransferEncoding>";
       }
    else {
        doc = doc +
          "<contentTransferEncoding>"+
          contentTransferEncoding +
          "</contentTransferEncoding>";
       }
    if (contentBody == null) {
        doc = doc +
          "<contentBody></contentBody>";
    }
    else {
      doc = doc + "<contentBody>" +
           contentBody +
           "</contentBody>";
      }
    doc = doc +  "</dataContent>
                </infopipeDataFormat>";
    return doc;
}
// End of the code
```

Using SIP/XIP, we can replace the above code with only 5 lines:

```
<dataDef name="mapImage">
  <arg type="string"
name="contentType"/>
  <arg type="string"
       name="contentTransferEncoding"/>
  <arg type="string"
name="contentBody"/> </dataDef>
```