The Active Streams Approach To Adaptive Distributed Applications And Services

A Thesis
Presented to
The Academic Faculty

by

Fabián E. Bustamante

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
November, 2001

The Active Streams Approach To Adaptive Distributed Applications And Services

Approved:

Dr. Karsten Schwan
(College of Computing), Chairman

Dr. Mustaque Ahamad
(College of Computing)

Dr. Greg Eisenhauer
(College of Computing)

Dr. Calton Pu
(College of Computing)

Dr. Kishore Ramachandran
(College of Computing)

Dr. Peter Steenkiste
(School of Computer Science, Carnegie Mellon University)

Date Approved

Acknowledgments

The cover page of this dissertation claims this to be the work of just one person. Nothing could be further from the truth. Neither this nor anything else I have achieved is due solely to me.

I would first like to thank my family, whose love, sacrifice and support brought me here, and my fiancée Jeanine, for the love and friendship that took me through. This work, as everything else, is dedicated to them.

I am in eternal debt with my advisor Karsten Schwan. Since the Fall of 1995, when he got me the job that made this possible, and throughout the following years in graduate school, Karsten has acted not only as an advisor but also as a mentor. He has been an unstoppable source of ideas, and has provided me with endless support and encouragement through the classical ups and downs of a Ph.D. life.

I would like to acknowledge the members of my thesis committee. Mustaque Ahamad and Kishore Ramachandran have supported me and patiently endured me through my growing pains. Calton Pu has provided a fresh new look into my work and challenged me just as I was starting to get too comfortable. Greg Eisenhauer has been a friend and a great research collaborator providing the firm grounding in reality that I have, many times, needed. I would finally like to thank Peter Steenkiste for gracefully accepting to be part of this committee and providing me with many insightful comments on this work and its presentation.

The College of Computing at Georgia Tech, and in particular the Systems Research Group, has been a fantastic place to work and has provided me with many friends who have made the process enjoyable. I am especially indebted to Patrick Widener, Michael Covington, Ada Gavrilovska, Maria Hybinette, Vijaykumar Krishnaswamy, Vivekand Velanky, Vernard Martin, and Beth Plale.

My good friends – Thorsten Hertel, Sumeer Bhola, Aparna Pappu, Leonardo Heuchert, Richard West, Robin Kravets, and Rob Kooper – are responsible for whatever degree of sanity that may remain in me.

A mis padres, Ernesto y Sonia,
mis hermanos, Alejandro y Andrés,
y a vos, mi amiga y compañera, Jeanine.

Contents

List of Tables

List of Figures

Summary

The widespread deployment of inexpensive communication technologies, computational resources in the networking infrastructure, and network-capable end devices offers a rich design space for novel distributed applications and services. Exploration of this space has given rise, for instance, to the notions of grid and peer-to-peer computing. Both technologies promise to change the way we think about and use computing, by harvesting geographically distributed resources in order to create a universal source of pervasive computing power that will support new classes of applications.

Despite the growing interest in these new environments and the increasing availability of the necessary hardware and network infrastructure, few actual applications are readily available and/or widely deployed. Such scarcity results from a number of technical challenges that must be addressed before the full potential of these technologies can be realized. Most of these applications, as well as the services they utilize, are expected to handle dynamically varying demand 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 resources' availability and demand. The Active Streams approach, advocated in this dissertation, aims to facilitate the task of building new distributed systems with these characteristics. To this end, the approach considers the contents of the information flowing across the application and its services, it adopts a component-based model to application/service programming, and it provides for dynamic adaptation at multiple levels and points in the underlying platform. In addition, due to the complexity of building such systems, it tries to ease the programmer's task by facilitating the needed infrastructure for resource monitoring, self-monitoring and adaptation. This dissertation explores the *Active Streams* approach and its supporting framework in the context of these new distributed applications and services.

Chapter 1: Introduction

The widespread deployment of inexpensive communication technologies, computational resources in the networking infrastructure, and network-capable end devices offers a rich design space for novel distributed applications and services. Exploration of this space has given rise, for instance, to the notions of grid [93, 92] and peer-to-peer [131, 100, 26] computing. Both technologies promise to change the way we think about and use computing, by harvesting geographically distributed resources in order to create a universal source of pervasive computing power that will support new classes of applications.

Despite the growing interest in these new environments and the increased availability of the necessary hardware and network infrastructure, few actual applications are readily available and/or widely deployed. Such scarcity results from a number of technical challenges that must be addressed before the full potential of these technologies can be realized. This dissertation explores a novel middleware approach and its supporting framework for building distributed applications in an attempt to meet some of these difficult challenges. Specifically, we address the inherent heterogeneous nature of these environments and the dynamically varying demand and availability of their resources.

1.1 Motivation

Improvements in communication technologies are leading many to consider more decentralized approaches to the problem of computing power. While the power of distributed computing and the opportunity presented by idle computer systems have been recognized for some time and have spawned a significant amount of research on its detection and utilization, true wide-area distributed computing had to wait for the spread of the Internet and the arrival of more powerful end devices. The past decade has witnessed an explosive growth of the Internet, with a number of hosts growing from just over a million on January 1993 to an estimated 100 million in January 2001 [25]. In parallel with this development, the raw power of individual computers, following Moore's Law [88], has kept doubling every 18 months.

Grid and peer-to-peer computing are two well-known attempts to let users "tap processing power off the Internet as easily as electrical power can be drawn from the electricity grid" [111]. Despite the great promise of these emerging technologies, relatively few general applications exist. Their realization is complicated by the characteristics of the target environments, including their heterogeneous nature as well as the dynamic variation on demands and availability of their resources.

Heterogeneity is an intrinsic property of the Internet, both across the network infrastructure as well as within end systems. Figure 1 shows the high variance in client and network capability

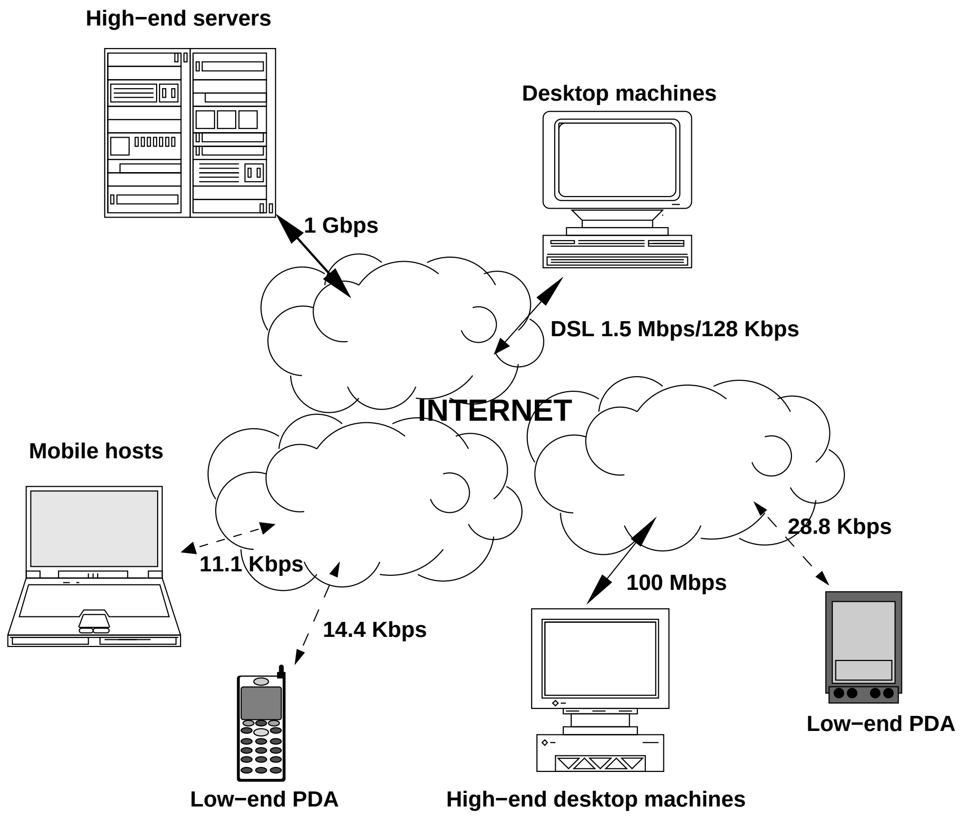


Figure 1: End-systems and network infrastructure heterogeneity.

today. While some host may be attached through a high-speed Gigabit Ethernet network, others are connected by 10Mbps Ethernet links, 1.3Mbps Asynchronous DSL connections, slow dialup or wireless links (see Table 2). In end-systems, this heterogeneity manifests itself in a variety of ways, from processing power to memory capacity and screen resolution (Table 1).

Compounded with the challenge posed by such heterogeneity, these applications must also deal with their run-time varying demands on resources in environments where availability cannot be guaranteed ‘a priori’. Dynamic variations in resource usage are typically due to applications’ data dependencies and/or users’ dynamic behaviors, and run-time variation in resource availability is a consequence of failures, resource additions or removals, and most importantly, contention for shared resources. Figure 2 presents some measurements that illustrate these widely dynamic variations on resource availability. We measured, over a 24-hour period, the round-trip times between three different hosts of a cluster located at Georgia Tech and the web servers of three universities in Argentina, Italy and Hong Kong. Each point in Figure 2-a is the average value of 10 `pings` at 5 seconds intervals using the default ICMP ECHO_REQUEST packet size of 64 bytes. The ranges of the error bars are computed based on the *mean deviations*. The second figure reports the variations in load, also over a 24-hour period, for three other hosts as indicated by the average number of jobs in their run queues over the last 15 minutes.

System Characteristic	PDA	Laptop	Midrange PC	Workstation
Memory Capacity	2MB	128MB	512MB	4GB
Number of CPUs	1	1	1	4
CPU Speed	16MHz	1GHz	2GHz	450 MHz
Screen Resolution	160x160	1024x768	1280x1040	1280x1024

Table 1: Example of end-host heterogeneity.

Network	Bandwidth	Latency
Gigabit	1Gbps	0.5ms
Local Ethernet	10-100Mbps	1ms
DSL	1083(down)/199(up) Kbps	100ms
Wireless	14.4-56Kbps	200-400ms

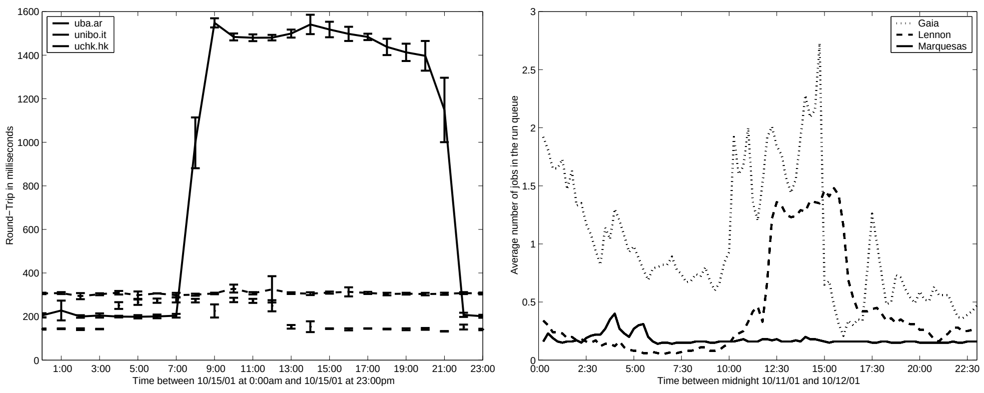
Table 2: Example of network heterogeneity.

To deal with these challenges: the Internet's intrinsic heterogeneity as well as the run-time changes on resource availability and demand, we believe that new services need to be customizable, applications need to be dynamically extensible, and both applications and services should be resource-aware and include some level of *introspection* [123].

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 deciding on such adaptations is complicated, it should ease the programmer's task by providing the needed infrastructure for resource monitoring, self-monitoring and adaptation.

This dissertation proposes *Active Streams*, a middleware approach and its associated framework for building distributed applications and services that exhibit these characteristics. Active Streams permits users to dynamically attach location-independent functional units, called *streamlets*, to the data streams flowing between applications and their services. Application evolution and various degrees of adaptation are possible through the attachment/detachment of streamlets, the re-deployment of streamlets over the available resources on the datapath, and the on-line tuning of the streamlets' behaviors through dynamic updates of their parameters. The associated framework provides an execution environment for these functional units and a pull-based mechanism for distributing them; it includes a sub-system for resource and self-monitoring and a directory service to "hold" everything together.

This chapter presents a high-level synopsis of the dissertation. The next section contains a basic overview of the key thesis components. This is followed by highlights of the main contribution of the thesis, and a brief summary of background and related research. The chapter closes with a description of the overall organization for the rest of the dissertation.



(a) Round-trip to different hosts in Argentina (uba.ar), Italy (unibo.it) and Hong Kong (uchhk.hk). Ranges of error bars are computed based on the *mean deviations*.

(b) Average load in three different hosts as indicated by the average number of jobs in their run queue over the last 15 minutes.

Figure 2: Dynamic variations in resource availability.

1.2 The Active Streams Approach

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. These data streams are made *active* by attaching application- or service-specific location-independent functional units, called *streamlets*.

Streamlets are self-contained units that operate on records arriving on their incoming streams and generate records placed onto their outgoing streams. Streamlets can be obtained from a number of locations; they can be downloaded from clients, provided by servers, or retrieved from a streamlet repository.

A critical issue for large-scale system design and evolution is the choice of an architectural style that permits the integration of separately-developed components into large systems. Active Streams adopts an event-based or implicit invocation architectural style for system composition [51, 52]; with this approach a component can invoke another one without needing to know its name by simply announcing the occurences of certain "events" in which the other component has registered interest.

Application evolution and/or a relatively coarse form of adaptation are obtained by the attachment/detachment of streamlets that operate on and change a data stream's properties. Finer grain adaptations are possible through the tuning of an individual streamlet's behavior via remotely updated parameters, and by the re-deploying of streamlets to best leverage the dynamically-changing 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 computational units make themselves available by running as Active Streams Nodes

(ASNs), where each ASN provides a well-defined environment for streamlet execution.

The deployment and redeployment of possibly independently developed streamlets requires the coordination and interaction of multiple producers and consumers that may be geographically (and organizationally) dispersed. Active Streams relies on a Streamlet Repository Service to provide the basic functionality needed for these tasks.

Active Streams applications rely on a push-based customizable service for resource and self-monitoring (ARMS). Through ARMS, applications can collect a selected subset of the data made available by distributed monitors. These monitoring 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 as well as in the types of applications targeted by the approach makes the passive client interfaces of classical directory services inappropriate. Thus, the Active Streams framework includes a *proactive* directory service (PDS) with a publish/subscribe interface through which clients can register for notification of changes to objects currently of interest to them. The levels of detail and granularity of these notifications can be dynamically tuned by the clients.

These four components: the Active Streams Nodes, the Streamlet Repository Service, the Active Resource Monitoring Services, and our Proactive Directory Service, constitute the core of the Active Streams framework.

1.3 Contributions

This dissertation offers a novel approach, together with the design and implementation of its supporting framework, for building distributed applications and services that can effectively operate in heterogeneous and highly dynamic environments. Specifically:

We present *Active Streams*, a new approach to building adaptive distributed systems. This approach supports the dynamic customization of services, run-time extensibility of applications, and the dynamic adaptation of applications and services to environmental changes. It adopts a component-based model to system programming with an implicit invocation mode of integration, centered around two simple abstractions – *streams* and *streamlets*. Streams are sequences of typed, self-describing, application-specific data units connecting parts of and applications and services. These streams are made active by attaching streamlets, application- or service-specific location-independent functional units.

We describe the design and implementation of the *Active Streams Framework*, an architecture for building adaptive and extensible distributed systems following this approach. The framework supports dynamic system adaptation at multiple levels and points in the underlying platform; it provides a pull-based service for code distribution with security considerations; it facilitates the needed infrastructure for resource monitoring, self-monitoring and adaptation, thus allowing the designer to focus on the application-specific logic; and it includes a directory service with an extended proactive interface more suited to the dynamism of the targeted environments.

Finally, to demonstrate the utility and flexibility of the Active Streams approach and to partially illustrate the use of its supporting framework, we experiment with two different types of applications. We present our experiences with the implementation of their prototypes and summarize what we have learned from them.

The common thread connecting our approach as well as most of the components of its supporting framework is the idea of *activity*. In Computer Science, a system entity is commonly referred to as *active* when some sort of processing has been attached to it, and this processing is to be implicitly invoked upon certain pre-stated conditions.

In the Active Streams approach, activity is used to ease the development and evolution of dynamically adaptive application and services. In the context of ARMS, activity is intended to improve an application's reactiveness to changes in itself or its environment. Finally, in the Proactive Directory Service, activity is utilized as a way for clients to regain control over notification, something they implicitly relinquished by making use of its proactive interface.

1.4 Related Work

This section contextualizes the research described in this dissertation by presenting an overview of related work. Our goals are twofold: we seek to point out the limitations of current state-of-the-art approaches and present previous works that have contributed to the definition of ours. A more complete discussion appears in Chapter 7.

Many of the ideas underlying the Active Streams approach have originated in early work done by our group concerning on-line steering and visualization of high-performance scientific applications [70, 121, 135, 118, 61] and resource-aware computing [71, 78]. Active Streams is itself part of Infosphere [97], a much broader research effort aimed at creating the virtual spaces of interaction for the post-PC era of computing.

The idea of "activity" has been widely used in systems over the last ten years, in projects ranging from message-based communication for parallel computers [137] to wide-area services location [133]. To our knowledge, ours is the first approach to make consistent use of this concept throughout its model and supporting framework.

Distributed adaptation can be application-transparent or application-aware and can occur at the system or application levels. Protocol Boosters and Transformer Tunnels support transparent network adaptation at the protocol level through the insertion of functional units in the datapath for the incremental construction of protocols [43], or the creation of tunnels in order to deal with problematic links [126]. Such systems can cope with localized changes in the underlying network but cannot adapt to behaviors that differ widely from the norm.

More general than the previous approaches and also complementary to our work, Active Networks [130] provide an infrastructure that allows application code to be attached to individual packets or deployed over the network routers. Although this approach provides a very general adaptation mechanism, its deployment requires significant changes to the existing network infrastructure.

Proxy-based solutions [152, 48, 10] have demonstrated the potential benefits of using the processing power available on the datapath, as they depart slightly from the basic client/server model by introducing a third entity, the proxy server. New environments provide additional processing units in the datapath, a potentially greater number of idle hosts and a longer, more complex network connecting clients and servers. These characteristics indicate the need for more a general, multi-point approach to adaptation. Although multiple proxies could be distributed over the datapath, the paradigm provides no assistance in making them cooperate.

Badrinath et al. [9] present a conceptual framework for adaptive software systems that synthesize the commonality of various projects the authors have been involved with. Our Active Streams model has much in common with the proposed framework as both advocate dynamic adaptation over the datapath to changing environmental conditions and application requirements. In contrast to Active Streams, their model associates application specific adapters with their equivalent of our Active Streams Nodes (Adaptation Agencies) instead of with the actual data streams. We have opted to associate streamlets with streams, as streamlets are location-independent and their mapping onto nodes is determined at run-time in response to changing environmental conditions.

Conductor [150] provides an application-transparent adaptation framework that allows multiple adaptation modules to be spread along the datapath between application and services. Although its transparency insures backward compatibility, it also limits its flexibility. In contrast to this, Active Services [7] allows client applications to explicitly start one or more services on their behalf that can transform the data they receive from end services but it does not provide for dynamic adaptation to changing environment characteristics or application requirements.

The goal of adaptive distributed approaches is to provide good end-to-end services, where the end points are located in applications. Without considering the applications' and their users' needs, no adaptive solution at the network level alone can solve the entire problem. Ninja [56], CANS [49] and Active Frames [84] are three projects that, as Active Streams, take an application-level approach to adaptation.

CANS [49] is an application-level framework for injecting application-level functionality into the datapath. The CANS infrastructure is closely related to Active Streams as both support the dynamic composition of application functionality over datapaths as well as their run-time adaptation to changing environmental conditions. CANS proposes an interesting extended-type-based composition to automate component selection based on link characteristics. Despite the high-level similarities, both approaches differ in aspects that include: the Active Streams focus on wide-area, heterogeneous, and highly-dynamic environments; its adoption of event-based techniques for component integration; and its target on high-performance applications.

Closely related to our work is the Active Frames approach, as proposed by Lopez and O'Hallaron [84], for building heavy-weight services such as scientific visualizations. In Active Frames, each frame includes both the data and code necessary for processing it. Active Streams, in contrast, adopts a demand pull-based approach to code distribution. Our reasoning is that (1) for lightweight frames, the overhead of sending code with each frame may not be acceptable, and (2) in our experience with scientific visualization and ubiquitous applications, a common set of functionality is typically applied across multiple frames, permitting the amortization of its deployment costs. Finally, (3) it is often desirable to have third parties be involved in defining and associating certain functionality with current information flows. The Active Streams approach to code distribution permits this.

Distributed applications executing in non-dedicated environments must be able to adapt to variations in resource availability. A number of research efforts have proposed resource-aware distributed computing and investigated adaptation models [124] and the infrastructure support needed by such an approach. Bolliger et al. [14] present a framework-based approach to developing network-aware applications, concentrating on network monitoring and the mapping between application-level and network-centric quality metrics.

The systems discussed above have demonstrated the value of customizing service functionality, dynamically extending clients, and adapting applications and services to dynamically changing environments. Active Streams builds upon them, providing a comprehensive approach to constructing adaptive distributed applications and services that exhibits these characteristics.

1.5 Organization

This section previews the remaining chapters and describes the overall organization of the dissertation. Chapter 2 presents an overview of the Active Streams approach as well as a description of its model's key components. In Chapter 3 we demonstrate the use of Active Streams through two sample applications: an active storage utility for scientific collaboration and an active video-stream system. The Active Streams supporting framework, its design and details about its current implementation, are presented in Chapter 4 and Chapter 5. Chapter 6 presents our evaluation results. Chapter 7 discusses a wide variety of related research in greater detailed than the presented in this chapter. Finally, Chapter 8 summarizes the conclusion of this dissertation and highlights opportunities for additional research.

Chapter 2: Active Streams

This chapter presents the Active Streams middleware approach to building adaptive distributed application and services. The approach facilitates the construction of new distributed systems by considering the contents of the information flowing across the application and its services, by adopting a component-based model to system programming, and by enabling the adaptive deployment of system components within a distributed system.

Central to our approach is the abstract concept of *active streams*. Streams are sequences of self-describing application data units flowing through the network, between applications' components and services. These streams are made active by the attachment of functional units that operate and modify the streams' contents. As is the case with the Active Network Initiative [130] this computational model supports the concept of plumbing components within a programmable infrastructure, but restricts the programmability to application-level services, and does it in a location independent way that enables adaptation through re-deployment of the functional units. In the following sections we provide a brief overview of the Active Streams approach and discuss various aspects of it in more detail.

2.1 Introduction

We model distributed systems as 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 typed, self-describing application-specific data units, such as movie frames, complex data structures containing chemical concentration levels in an atmospheric model, or simple reports on stock values. Stream types are described by C-style structures made up of atomic (including integers, floats, and null-terminated strings) and previously defined structured data types.

Data streams are made *active* by the attachment of application- or service-specific functional units, called *streamlets* (Figure 3). Streamlets are self-contained, location-independent functions that each perform a particular activity. They operate on records arriving on their incoming streams and generate records placed onto their outgoing streams. Active Streams adopts an event-based or implicit invocation architectural style for system composition [51, 52]; with this approach a component can invoke another one without needing to know its name by simply announcing the occurences of certain "events" in which the other component has registered interest. 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

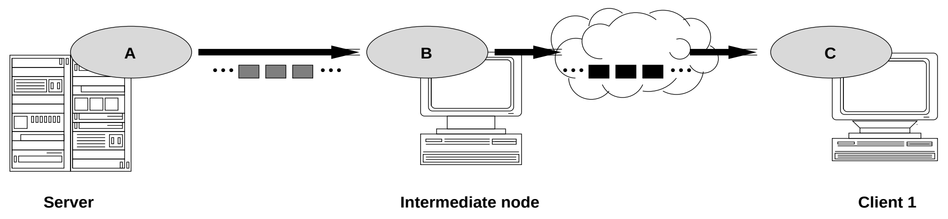


Figure 3: Active Stream.

well-defined environment for streamlet execution. ASNs configure themselves into overlay networks over which Active Streams are deployed. Clients explicitly select the ASN overlay network they wish to use and connect to it through a close (in network sense) ASN.

While the network has traditionally been seen as transporting data between end points, Active Streams extends this notion to enable applications and end services to dynamically inject application-specific components into it. An active stream can thus be seen as a description of a processing sequence used in a connection between end-points.

2.2 Streamlets and Stream-Based Programming

Active Streams adopts a dataflow-based programming model much like those used in signal processing [82, 53], parallel databases [34, 55, 147], and more recently, cluster file I/O [8]. A system is expressed as a directed graph of large-grain nodes, each a collection of sequential work that execute from start to finish without synchronization, and edges that depict the asynchronous flow of data from one node to the next. The topology of the graph defines the flow of data units from sources to sinks, implicitly defining the application's software architecture.

Large-grain dataflow provides a natural description for many data processing applications, with each node representing a function to be performed on an infinite stream of data that flows on the arcs of the graph. The streams of data can be generated by sensors sampling the environment at periodic rates, by trackers of application-level state transitions in legacy software, or by monitors reporting researchers of important changes to scientific models. The dataflow methodology facilitates the understanding of the processing performed by depicting the structure of the algorithm.

Applications and services are extended or customized, following a component-based approach, through the attachment of a set of one or more functional units or streamlets. The idea is to create systems that are easier to understand and reason about than those that explicitly send/receive messages [142]. Streamlets are self-contained code units that can perform a particular activity. They are the basic unit of composition in Active Streams. Streamlets operate on typed records arriving on their incoming streams and generate typed records that placed onto their outgoing streams. The queues associated with incoming and outgoing streams are provided by the streamlet's execution environment. These queues decouple the streamlets' execution by introducing explicit control boundaries and providing rate-matching between them. Streams in Active Streams are

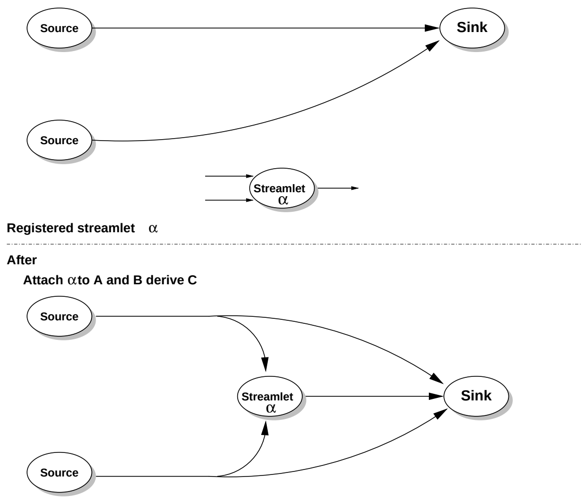


Figure 4: Attachment of streamlets.

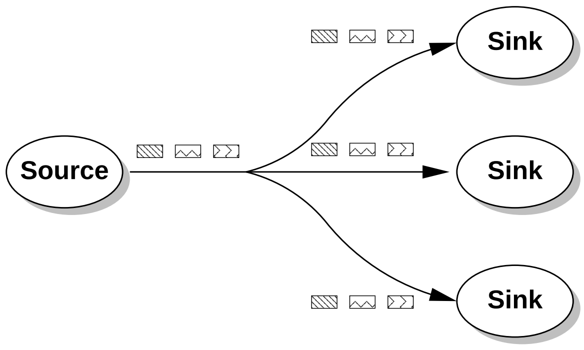


Figure 5: Multiple sinks per stream.

similar to channels in a Kahn process network [74], but differ from these in the semantics of stream reads. While a stream's data units are read exactly once *per* sink, as with Kahn's channels, each of the possible multiple sinks attached to a single stream will have its "own" copy of the stream. This is intended to simplify the coordination process required for the insertion of a streamlet into an ongoing stream, since the original streams can remain active as long as needed (Figure 4).

Streamlets are registered with the runtime system with an associated set of conditions for their activation. Such conditions are currently stated as the minimum set of data units (threshold) that must be available per incoming stream before streamlet execution. A streamlet is eligible for execution whenever all of its input streams are over threshold. More elaborate triggers are possible, given the information available to the execution environment, but this is left as a topic for future exploration.

An example of a streamlet is given in Figure 6. This streamlet takes its inputs from a single stream of PPM frames and produces an output stream of PGM frames. Each data unit placed into

```
{
  int col, row, indx, outIndx, tmp;

  output.ppm1 = input.ppm1;
  output.ppm2 = '5';
  output.width = input.width;
  output.height = input.height;

  indx = 0;
  outIndx = 0;
  for (row = 0; row < input.height; row = row + 1) {
    for ( col = 0; col < input.width; col = col + 1) {
      tmp = (0.299 * input.buff[indx]  + 0.587 * input.buff[indx+1] +
             0.114 * input.buff[indx+2]) + 0.5;
      output.buff[outIndx] = tmp;
      indx = indx + 3;
      outIndx = outIndx + 1;
    }
  }
  output.size = output.width * output.height;
}
```

Figure 6: A streamlet that converts PPM to PGM frames.

the output stream is the "black and white" rendering of the, in order, corresponding incoming data unit. The quantization formula used is $0.299r + 0.587g + 0.114b$.

Streamlets can be obtained from a number of locations; they can be downloaded from clients, servers, or retrieved from a streamlet repository.

Summarizing some characteristics of streamlets:

1. They consume and produce typed records.

2. They are passive, moving data from input streams to output streams in a demand-driven fashion. They are activated once the required input records are available.

3. They consume and produce data at the granularity of an integral number of application-specific units.

4. Streamlets can store small amounts of state in static variables that persist for the streamlet's lifetime in a node. This soft-state can be reconstructed simply by restarting the streamlets. Given a semantically equivalent sequence of input records, a streamlet always produces a semantically equivalent sequence of output records.

5. Each streamlet has a unique ID, a cryptographic hash (e.g. SHA-1 [132] or MD5 [115]) of the streamlet's code.

The process of attachment of streamlets to streams defines a directed acyclic graph that serves as a set of constraints to the dynamic mapping algorithm of functional components to computational devices.

2.2.1 Error Handling

An important issue with Active Streams is error handling. The addition of distributed programming at multiple dynamically chosen points complicates exceptional conditions and error detection and handling. We adopt the use of *exceptions* to deal with this.

To *raise* an exception is to signal an exceptional condition. To *catch* it is to handle it – to take whatever actions are necessary to recover from it. Raised exceptions are published on a special exception channel associated with each stream. Applications can subscribe to this exception channel and define functions to catch raised exceptions.

2.3 Implicit Invocation

Traditionally, in a system in which components' interfaces provide a collection of procedures or functions, components interact with each other by explicitly invoking those routines. Recently there has been considerable interest in event-based or implicit invocation as an alternative integration techniques for system software [103, 54, 38, 142].

The idea behind implicit invocation is that instead of invoking a procedure direcly, a component can announce one or more events. Other components in the system can register interest in an event and associate a procedure with this. This procedure will then be "implicitly" invoked by the event announcement.

Among other importan benefits, implicit invocation provides strong suport for software reuse and system evolution. Any component can be introduced into a system simply by registering it for the appropriate events, and components can be replaced by newer ones without affecting the interfaces of related components in the system.

This architectural style of integration is naturally supported by publish/subscribe systems, where a number of systems components transmit and/or receive pieces of information, called *messages* (*events notifications* or simple *notifications*), in response to *events* occurrences. A component that detects events and sends messages is commonly refer to as a *publisher*, *source*, or *informer*; while one that receives messages is refer to as a *subscriber*, *sink*, or *listener*. As a module can, at any time, invoke and be invoked by other modules, a component can act as both a publisher and a subscriber.

2.4 Service and Service Customization

In general, a *service* defines a collection of operations a server can perform on behalf of its clients. The task of building and maintaining services is challenging in part because of the need to satisfy dynamically varying clients requests, on widely heterogeneous environments, while simultaneously preserving the services' essential properties of scalability, availability and fault-tolerance.

A well-known issue with today's services is what is sometimes called "the client-level server" problem [18]. Traditionally, systems are vertically integrated in an attempt to provide entire solutions to clients' needs. Vertically integrated systems, however, often make it difficult if not impossible for a user to obtain exactly the service she requires (such as lower resolution on a video stream, for displaying in her PDA, or information on stocks trading outside a certain range). Faced with the inflexibility of such services, users end up constructing application-specific servers that act as clients of a default server.

Active Streams supports the notion of “value-added” services, more specifically, *in-session* distributed service extensions [54], by which we mean run-time no persistant extensions. The functionality of services in Active Streams is reduced to a minimal required set of appropriate abstractions, flexible enough to allow the construction, through extensions, of arbitrarily customized versions of them. Applications can build their customized services, extending them at the granularity of individual procedures or modules, through the attachment of streamlets that service programmers or other users make available.

2.5 Adaptation in Active Streams

Active Streams supports three forms of adaptation to deal with dynamic changes in the environment and/or the behavior of the application itself (see Figure 7): (1) active stream configuration, where the composition of an active stream is modified through the attachment/detachment of streamlets, (2) active stream re-deployment, where the streamlets composing a given active stream are re-deployed over the available computational units on the datapath, and (3) streamlet parameterization, where the functionality of a streamlet attached to an active stream can be fine-tuned by the run-time update of a selected set of parameters.

2.5.1 Active Stream Configuration

Coarse-grain adaptation can be obtained through the attachment/detachment of streamlets that operate on and change a data stream. As in many extensible systems, an event-based invocation mechanism is used [127, 103]. With this mechanism, extensions are designed as events handlers, where any component of the system may raise an event upon the occurrence of some system activity, at which time the corresponding extension is processed. The binding between an event and its handler can be dynamically altered. Therefore, streamlets can be seen as event handlers invoked by to notifications issued over their incoming streams.

2.5.2 Active Stream Re-Deployment

Adaptation can take place through the dynamic re-deployment of streamlets over the computational units available on the datapath. Mappings are re-evaluated when interesting changes on resource demand or availability occur. Since what’s “interesting” typically depends on the particular application at hand, Active Streams resource monitoring and its adaptation policy are themselves customizable.

2.5.3 Streamlet Parameterization

At a finer grain, the functionality of streamlets can be modified through parameterization, by remotely updating the contents of blocks of data associated with streamlets in a push-type operation. As an example consider the streamlet in Figure 8.

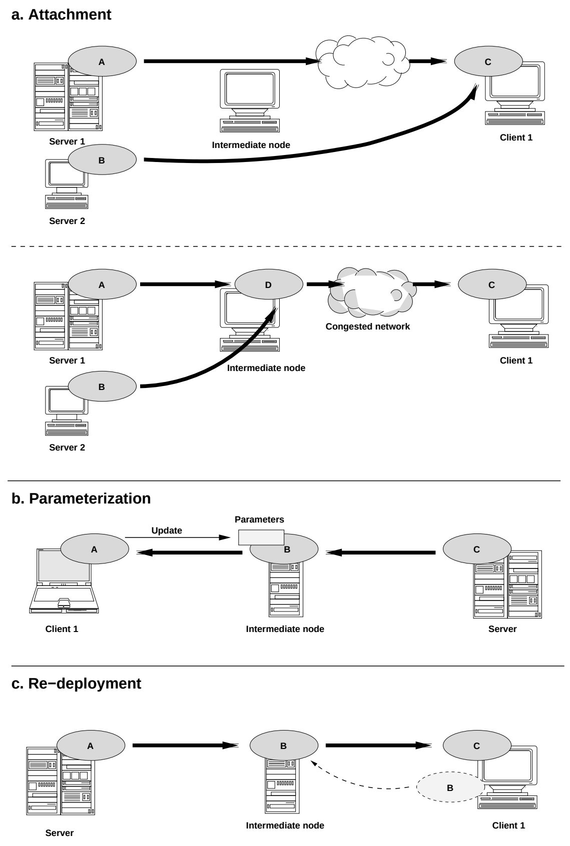


Figure 7: Types of adaptation in Active Streams.

This streamlet would filter out any record with 'range' outside $(LOWEND, HIGHEND)$. If a client wanted to change this range it could potentially detach the streamlet and attach a new one with the desired range. A more natural solution is to associate with the streamlet a parameter block that will hold the high and low bounds of the range. The range can then be dynamically updated as needed. Figure 9 shows the new parameterized streamlet.

Another example of a parameterized streamlet is given in Figure 10. This streamlet takes its input from two streams of PPM frames and produces an output stream of PPM frames. Each data unit placed in the output stream is the mix of two incoming data units using a fade factor provided as a parameter. The fade factor parameter may be in the range from 0.0 (only the first data units)

```
{
  if ((input.range > LOWEND) || (input.range < HIGHEND)) {
     return 1; /* submit record into output stream */
  }
  return 0; /* do not submit record */
}
```

Figure 8: A streamlet that passes only records with 'range' in a specified interval.

```
{
  if ((input.range > param.range_lowend) ||
      (input.range < param.range_highend)) {
     return 1; /* submit record into output stream */
  }
  return 0; /* do not submit record */
}
```

Figure 9: A simple example of streamlet parameterization.

```
{
  int col, row, indx; long longfactor;
  output.ppm1 = inputQueue[0].ppm1;
  output.ppm2 = inputQueue[0].ppm2;
  output.width = inputQueue[0].width;
  output.height = inputQueue[0].height;
  output.maxval = 255;
  longfactor = param.fade * 65536;
  indx = 0;
  for (row = 0; row < inputQueue[0].height; ++row) {
    for (col = 0; col < inputQueue[0].width; ++col) {
      output.buff[indx] = inputQueue[0].buff[indx] +
        (((inputQueue[1].buff[indx] - inputQueue[0].buff[indx])
          * longfactor) >> 16);
      output.buff[indx+1] = inputQueue[0].buff[indx+1] +
        (((inputQueue[1].buff[indx+1] - inputQueue[0].buff[indx+1])
          * longfactor) >> 16);
      output.buff[indx+2] = inputQueue[0].buff[indx+2] +
        (((inputQueue[1].buff[indx+2] - inputQueue[0].buff[indx+2])
          * longfactor) >> 16);
      indx += 3;
    }
  }
}
```

Figure 10: A streamlet that mixes two image frames with a specified ‘fade’ factor.

to 1.0 (only the second data units). Anything in between gains a smooth blend between the images in the two data units.

Chapter 3: Example Applications

This chapter introduces two example applications with the intention of further motivating the Active Streams approach and illustrating its use. We review the application of our ideas in the context of collaborative research environments in Section 3.1, and sensor-rich spaces in Section 3.2, discussing qualitatively how the application of our approach in those cases impacts end-to-end application performance. The choice of applications is intended to demonstrate the flexibility of the approach to accommodate a wide variety of data streams, client devices and network characteristics; as well as different classes of applications, from high-performance scientific data processing to sensor-based ones.

3.1 Collaborative Research Environments

The evolution of Internet Grid-style computing over the last decade has changed the way we tackle complex problems. Computational science has evolved from single independent investigators to encompass distributed, collaborative, multidisciplinary groups and massive computational science datasets. Collaborative research environments are a concrete example of this. Projects like our group's Distributed Laboratories [50], NSF's NEESgrid [93], and those part of NPACI's Alpha [92], aim to create virtual spaces where geographically dispersed communities of scientists can interact to collaboratively solve problems by manipulating local and remote computational tools, analyzing shared data, and discuss their results.

Among the tools needed for such collaboration are those used for the analysis and processing of various real-time or archived data streams. The volume of these computational science datasets has been one of the major limiting factors for this style of research. A dataset of coarse-grained satellite data (with 4.4 km pixels), covering the whole earth surface and captured over a relatively short period of time (10 days) is about 4.1GB; a finer grained version (1.1 km per pixel) contains about 65 GB of sensor data. In medical imaging, the size of a single digitized composite slide image at high power from a light microscope is over 7GB (uncompressed), and a single large hospital can process thousands of slides per day [3].

The sample application used in our research is a global atmospheric climate model [76] developed by our group in collaboration with scientists from the School of Earth and Atmospheric Sciences at Georgia Tech. The model simulates the flow and interaction of various chemical species through the stratosphere.

Experimental scientists using this model would like to explore the data generated and compare it with archived outputs. As the simulation evolves, their foci of interest may change or they may want to share their finding and discuss their results with other researchers in other parts of the country. Given the long duration of most experimental runs in this field, researchers would like to do this not only from within their labs but also from desktops, laptop computers, or even from small portable devices during their commute time.

During its execution the model produces a number of streams composed of application-level units such as the concentration levels of certain chemicals in the atmosphere. These output streams can be grouped by chemical components, atmospheric level, or geographic location, and can be made available through a directory service to simplify their exploration. Since the number and characteristics of these streams can vary over time, a proactive directory services allows researchers to find out about these changes as they occur.

As researchers change their foci of interest and/or their execution environments, a number of application-level customizations can be used to dynamically adapt to variations in resource demand and availability. These application-level customizations can be applied, in the form of streamlets, to the data streams flowing between the model and the researchers' visualization interfaces. Table 3 provides some examples and a brief description of streamlets we have implemented for experimentation with this application.

Table 3: Atmospheric model streamlets examples

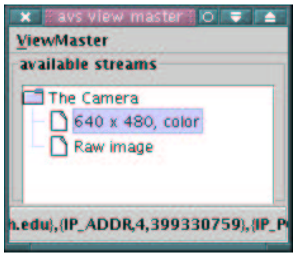
Name	Description	Parameters
spec2grid, grid2spec	conversions from/to spectral/grid representation	-
runavg	summarize record based on short running average	vector length
levelout	filter out records from a given range of atmospheric levels	levels
gridred	focus on a 3D area of interest	3D coordinates
smoother	filters out no significant updates	significance factor

It is to be expected that most researchers working on the same, or a similar application, would require a common enough set of functionality to make code reuse beneficial. The deployment and redeployment of independently developed streamlets is facilitated by a demand pull-based repository service, a component of our framework.

3.2 Sensor-Rich Spaces

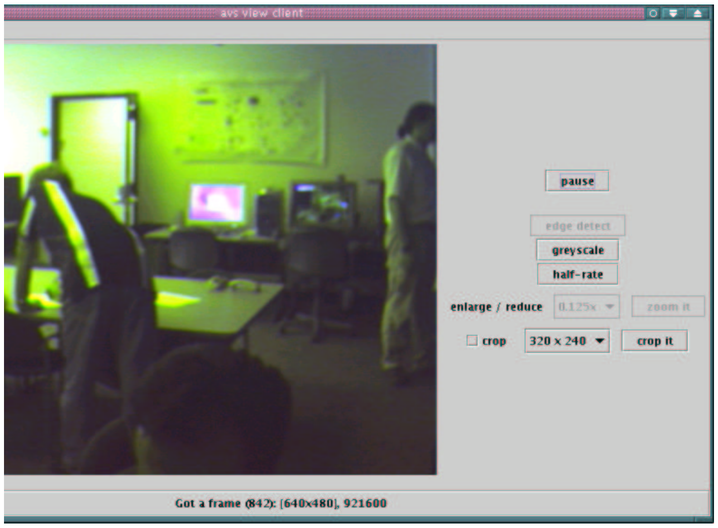
The advances in embedded processors, low cost sensor technologies, and wireless communication are fragmenting the computing infrastructure and resulting in environments characterized by small, low-cost computer devices that are virtually everywhere [22]. Samples of these types of artifacts are already available as handheld computers, Internet-ready cellular phones, and wireless networks. As they grow in number, such densely populated sensor-rich environments will become commonplace.

Sensor-rich environments will collect large amounts of information that must be combined and processed before being delivered to interested end-users. An infrastructure is needed that can facilitate this, and as part of our research we have built an application to evaluate the use of Active



n.edu),(IP_ADDR,4,399330759),(IP_P

(a) Available video streams.



Got a frame (842): [640x480], 921600

(b) A viewer displays a video stream and gives access to possible adaptations.

Figure 11: Command Center Interface.

Streams in this context.

In our application scenario, sources are cameras attached to unmanned vehicles and streaming video images over wireless links to a set of one or more command centers. In this application, streams are sequences of raw portable anymap frames (raw ppm or pgm) captured by a video device. Multiple video streams originate at, potentially, multiple vehicles can be simultaneously available at any given time. As the number of available video streams may change over time, so can the users' foci of interest. The limited bandwidth availability, property of this environment, requires that trade-off be made between image quality and frame rate across multiple video streams. The use of application-level stream customization to enable this type of adaptation is our motivation in this context.

Our experimental setup includes a number of portable computers with video-devices attached to them and connected over wireless links to a set of client computers acting as our command centers. Our sources are USB cameras that we access through *VDev*, a library we have built for this purpose. The command center interface is built in Java and connected to our infrastructure making use of the Java Native Interface.

We use our VDev library for accessing video devices in Linux. VDev allows users to fetch and control the image streams captured by USB video cameras connected to a Linux PC box. Through the VDev client interface, users can also configure common parameters such as: frame rate, image size, brightness, contrast, hue, and color.

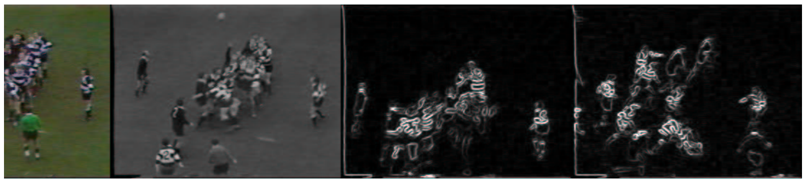


Figure 12: A number of adaptations are applied over a sequence of frames composing a video stream.

There are a variety of application-level customizations that can be used to deal with the restricted bandwidth availability in the context of this application. Ideally, applications (or the end-users themselves) should be able to dynamically select and apply certain transformation in order to reduce resource consumption or better suit their clients' environments (Figure 12). We have implemented an interesting set of possible customizations for experimentation, and Table 4 provides a brief description of their functionality. Streamlets attached to ongoing streams operate on them and modify their characteristics (by converting an image from color to gray, reducing the required bandwidth to a third; by cropping the interesting part of it, resulting in additional savings; etc). To make this available to end-users we have adopted an extensible-service paradigm for interaction. Each attachment of a streamlet to an existing stream creates an alternative service, an additional (virtual) camera, which other users can utilize. We achieve this by using PDS to update the listing of cameras seen by all clients (real or virtual) as soon as they become available or disappear. Without a proactive interface a researcher will be restricted to poll the directory service in the hope that somebody, at some point, will create the stream she wants to use; thus generating unnecessary network traffic and processing loads at both client and service.

Applications such as our sensor-based example or collaborative research environments like the one described in Section 3.1 execute in mostly no-dedicated and highly heterogeneous environments. Since the availability of resources in such environments often determines overall application performance and, thus, users' experiences, adaptivity is necessary for efficient and more predictable executions. Adaptive applications require prompt information about their own behavior and the status of the execution environment. The Active Streams framework provides this needed functionality through ARMS, a push-based customizable service for resource and self-monitoring (ARMS).

Table 4: AMS Streamlet examples

Name	Description	Parameters
concatenate	concatenate two frames	concatenation order
crop	crop a portable bitmap	area coordinates
edge	edge-detect a portable graymap	-
enlarge	enlarge a frame N times	enlargement factor
reduce	reduce a frame N times	reduction factor
grey	converts a portable pixmap frame to gray	-
mix	blends two frames	fade factor
half-rate	drops every-other frame	-

Through ARMS, applications can collect a selected subset of the data made available by distributed monitors. These monitoring streams can be integrated to produce application-specific views of system state and decide on possible adaptations.

3.3 Summary

This chapter has demonstrated the flexibility of the Active Streams approach to accommodate a wide variety of data streams, client devices and network characteristics; as well as different classes of applications. We have shown how the different component of our framework facilitate the construction of dynamic customizable services, run-time extensible applications, and applications and services that are able to dynamically adapt to changes in themselves and their environments.

Chapter 4: Active Streams Framework

This chapter presents the Active Streams supporting framework for building distributed applications and services. Successfully enabling dynamic distributed adaptation requires several important services. Our current Active Streams framework is comprised of four core components: the Active Streams Node, the Streamlet Repository Service, the Active Resource Monitoring Service and our Proactive Directory service. We describe each of them after giving a bird's-eye view of the entire framework.

4.1 Overview

Active Streams are realized by mapping streamlets and their associated streams onto the resources of the underlying distributed platform, seen as a collection of loosely coupled, interconnected computational units.

Computational units make themselves available by running as Active Streams Nodes (ASNs), where each ASN provides a well-defined environment for streamlet execution. Similar to a virtual machine, an ASN acts as a manager of the streamlets' required resources and provide uniform interfaces for the general administrative tasks needed to support their execution. In contrast to classical virtual machines, ASNs do not interpret streamlets' code but rely on dynamic code generation for their efficient execution.

Since Active Streams customized services and applications are built by assembling possible independently developed components, their deployment and re-deployment processes requires a mechanism that support their storage and distribution. The Active Streams Framework includes the Streamlet Repository Service (SRS), a scalable distribution service for safe management, transport, and storage of streamlets.

Support for distributed adaptation requires ways in which an application can be notified of any important changes in the environment. The Active Streams framework provides a push-based/active customizable resource monitoring service (ARMS). Through ARMS, an application inherits the mechanisms needed for providing introspection, thus allowing the application designer to focus on the application-specific logic. Moreover, an application using ARMS can easily redefine its monitoring views by adapting/replacing filters used for monitoring while the system is running, perhaps in response to workload and/or environment changes.

As is common in distributed systems, a directory service provides the "glue" for the Active Streams framework. However, the number and the dynamic nature of most relevant objects in Active Streams environments makes the passive client interfaces of classical directory services inappropriate.

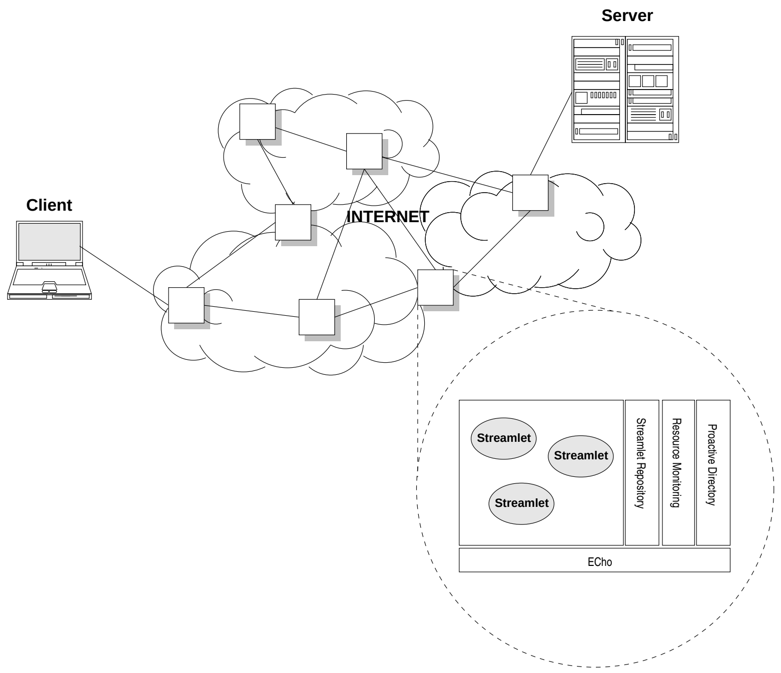


Figure 13: Active Streams Framework.

Thus, the Active Streams framework includes a *proactive* directory service (PDS) that supports a customizable interface through which clients can register for notification about changes to objects. The level of detail and granularity of these notifications can be dynamically tuned by clients through filter functions instantiated at the server or at object owners.

These four components: the Active Streams Node, the Streamlet Repository Service, the Active Resource Monitoring Service and our Proactive Directory service, comprise the core of the Active Stream framework (Figure 13).

4.2 Active Streams Nodes

Computational units make themselves available by running as Active Streams Nodes (ASNs), where each ASN provides a well-defined environment for streamlet execution. Much like a virtual machine, an ASN decouples the application- and streamlet-specific functionality from the general administrative tasks needed to support streamlet execution, such as scheduling, monitoring and reconfiguration. An ASN also provides the basic resources needed by streamlets and enforces constraints on how those resources can be used.

An ASN could be assigned to serve a particular application or a set of them. By adopting the first approach, the costs of inter-streamlet and streamlet-ASN communication could be minimized and many protection issues could be avoided. On the other hand, assigning an ASN to a set of applications would allow us to do cross-application adaptations. Since the trade-offs presented by both options make the selection of the "right" approach far from clear, we have opted for a flexible solution that will allow us to further investigate this topic. In our current implementation, ASNs register themselves with a set of other ASNs, with ASNs in a give set comprising an overlay network structured as a fully-connected graph. When an ASN is started at a particular host, it is given the names of the overlay network(s) it is supposed to join.

4.3 Streamlets

Since the environments we are targeting are heterogeneous, the architecture and/or the operating systems of two nodes serving as ASNs may differ. While the Active Streams middleware provides for interoperability, there is still the streamlet code to consider. Truly location independent code must run on the hetereogenous collection of machines that are common in today's computational environments.

In Active Streams we have opted to support a restricted programming language to insure that streamlets can be dynamically deployed and efficiently executed across heterogeneous environments. In addition, this approach, combined with the use of cryptography, also allows us to guarantee a streamlet's safe execution. Streamlets can be created using ECL, a subset of a general procedural language, and dynamic code generation is used to create a native version of the given streamlet on the target host. ECL's dynamic code generation capabilities are based on Icode, an internal interface developed at MIT as part of the 'C project[108]. Icode is itself based on Vcode[40], also developed at MIT by Dawson Engler. Vcode supports dynamic code generation for MIPS, Alpha and Sparc processors, and has been extended to support MIPS n32 and 64-bit ABIs, Sparc 64-bit ABI, and x86 processors [38]. ECL, although extensible, is currently a subset of C, supporting the C operators, `for` loops, `if` statements and `return` statements.

For experimentation, however, and in order to consider potential useful extentions to ECL, we have also enabled the use of general C-based shared-object modules.

4.4 Streamlet Repository Service

Instantiating (and later adapting) a given active stream requires the deployment of its associated streamlets within an ASN's overlay network. Since active streams are composed of possibly independently developed components, such deployment and re-deployment involves multiple component producers and consumers that may be geographically and organizationally dispersed.

Because traditional configuration management systems have focused on the development activity of source code control, they lack support for deployment tasks other than configuration and installation. The Active Streams Framework includes a repository services that offers the basic functionality needed for these tasks. In this section we describe such services, begining with a discussion of its design goals.

4.4.1 Streamlet Distribution

Code distribution in Active Streams must support the dynamic extensibility and adaptability of the approach while also providing the performance and security properties of more static approaches that can resort to ready-loaded nodes and/or static linking. Performance is important because, assuming an efficient streamlet execution environment, distribution will be a determinant factor on the overall run-time overhead of our approach. Security is a natural concern with any approach that involves some form of mobile code.

Note that these requirements are not exclusive to Active Streams. As more systems are built using distributed component technology, including proxy-based systems [48], distributed objects [95, 128], and active networks [130], the "missing component" problem will become increasingly common.

A good scheme for streamlet storage and distribution must be efficient, flexible, and adaptable. It must scale to wide-area settings, minimize (re-) deployment time, and make it difficult to compromise the system. The Streamlet Repository Service for management, transport, and storage of streamlets meets these goals.

While many alternative designs are possible, we have opted for a scheme where streamlets are stored in distributed locations (which can be located through a directory service such as PDS) upon registration, and delivered to interesting nodes upon request. Each requesting node maintains a *streamlet cache*, keyed by streamlet ID, that contains all streamlets loaded into the node and could potentially be pre-loaded to reduce setup cost.

The Active Streams' Streamlet repository service provides for:

- insertion,
- retrieval and
- removal of streamlets with given set properties, as well as
- queries on streamlets available with a given set of properties.

4.4.2 Streamlet Description

Each streamlet is described by a set of properties (Table 5) that include most of those advocated by the Open Software Description Format (OSD), an application of XML to support automated software distribution environments proposed by Marimba Incorporated and Microsoft Corporation [66]. We have included an attribute-value list to cope with less common or unforeseen attributes.

Table 5: Streamlet description fields in the repository

Field	Example	Note
Name	CropImage-1	Name and version
UID	MD5 string	Cryptographic hash
OSRelease	SunOS-5.7	`uname -rs`
Processor	SPARC	`uname -p`
Content	CSObj	Lang. and format
Author	mafalda@quino.net	Author's email
Description	Reduce image	Descriptive comment
Attribute list	$<< a1 = v1 >< a2 = v2 >>$	List of attr-value pairs

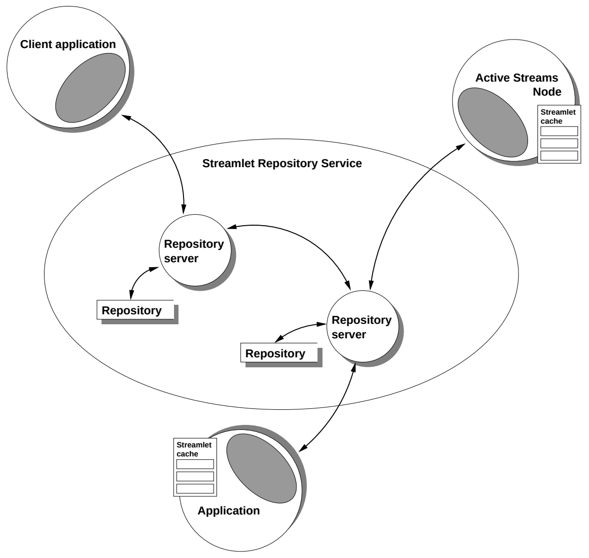


Figure 14: Streamlet repository service.

4.4.3 SRS Architecture

The Streamlet Repository Service (see Figure 14) is composed of a number of repository servers, and a client library that can be linked to any application, making it thus capable of fetching components from these servers. The client-side library is sufficiently small to be used in end-devices with limited resources such as PDAs.

Each SRS client maintains a streamlet cache, keyed by streamlet ID, that contains all information on the streamlet as well as the streamlet's code.

There are at least two ways by which a client application can find a streamlet repository server. The client could be configured with a list of unicast repository server addresses or could come with an anycast address [105]. This anycast address or the list of unicast addresses can be requested through a directory services such as PDS.

4.4.4 Safety Considerations

SRS clients fetch streamlets for executing them in their hosts. Without the proper precautions, a node may import and run unsafe code, thereby opening the host to abuse through misuse of its resources. Solutions to the problem of unsafe code fall into three categories [64]: user control, implicit trust, and verified access. User control is the simplest to implement and the one currently used in mobile code systems such as Java. With this alternative the user controls the resources that the imported code component may access, either by setting up access control lists for the resources, or by responding to dialog boxes that describe the type of resources the component requires and allows the user to decide whether or not to grant the request. There are several problems with this approach: first it may require excessive user interaction to control the resources available to the component; second, any non-trivial component will need to access resources in a potentially unsafe manner. Asking the user to make an intelligent decision about each possible required access seems unreasonable.

At the other extreme from user control is implicit trust. In this solution a component is considered safe if it has originated at a trusted entity (verified through the use of digital signatures, for example). Since it is infeasible for the user to carefully examine the source code to check its correctness and automatic method will work the only feasible way of running a component's code is to implicitly trust it, and therefore the person or entity that created it. Although trusting a component based only on its source may seem incautious, it is not different from what is actually done with shrinkwrapped software.

Although the use of digital signatures permits a client application to verify the identity of the producer of an imported code, it does not guarantee that this code is safe to execute. Thus it may be necessary to resort to some form of safety checks in order to verify a component's integrity before executing it.

An interesting alternative is the use of a third party to do the verification of imported code (Hartman et al. [64] compare this model to the use of the Underwriters Laboratory [69] safety certification on manufacured products). With an equivalent assurance, a client application can trust the imported module and run it without re-verification.

We address security in SRS through policy and cryptology. Authors and streamlets have unique IDs. We use authors' email addresses as their IDs. Each streamlet has a unique identifier which consists of a cryptographic hash of the streamlet code. This scheme eliminates the need for a centralized streamlet naming system; guarantees that the streamlets attached to streams are those specified by the user, assuming that the configuration commmand has not been modified on the wire and that the ASN has not been compromised; and eliminates code versioning problems. In addition, each configuration command carries a pre-computed hash key that can be used to rapidly look up the streamlets' code in a cash. Since each ASN can reliably check the source of a streamlet and the identity of its developer, it's able to decide, based on that information, whether or not it should accept and execute the streamlet's code.

4.5 Active Monitoring of Resources

Many distributed applications, such as those targeted by Active Streams, are expected to handle dynamically varying demand 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. Dynamic variations in resource usage are typically due to applications' data dependencies and/or users' dynamic behaviors, while the run-time variation in resource availability is a consequence of failures, resource additions or removals, and most importantly, contention for shared resources.

An attractive way of dealing with this variability is making applications *resource-aware* [57], i.e. making them able to periodically adapt to environmental changes by adjusting their resource demands in application-specific ways. The Active Streams framework includes an Active Resource Monitoring Service, ARMS, aimed at facilitating the task of building resource-aware applications.

In this section we present ARMS: its design goals and architecture.

4.5.1 Active Resource Monitoring Service

Central to the process of providing resource-awareness is the collection, aggregation, and processing about the execution environment. Although much of the information needed for this end is ready available in commercial middleware, the ways of obtaining it are not well documented and are different for each piece of information and/or platform. Additionally, applications have no option but to poll the system for this information without any guarantee that it will be of any use and with the hope that their polling frequency will match that of significant changes to the environment.

ARMS supports a customizable push-based interface to resource monitoring intended to solve some of these problems. ARMS client applications and/or the adaptive runtime system can select a subset of the data made available by distributed monitors, and integrate that data to produce application-specific monitoring metrics and decide on possible adaptations. Different objects (including devices) are the sources of monitoring data. ARMS' clients express interest in different monitored objects by selectively registering themselves with the streams associated with those objects.

Applications are not generally interested in all raw data reported by the different monitoring objects at potentially high volumes. Instead, each application needs a subset of the available data, at different times, and commonly would like to integrate different data to produce application-specific monitoring information (that will match application-specific adaptation options). An application using ARMS can easily redefine its monitoring views by adapting/replacing the filters used for monitoring while the system is running, as its workload and environment change. In this manner, applications have an additional level of adaptation that would not be possible were the monitoring and adaptation mechanisms coded statically for each of them.

In addition, as is the case with most monitoring systems for distributed environments [35, 149], ARMS must provide adequate predictive accuracy; should be easily portable as to guarantee ubuiquity; must be mostly resilient to failures, so common in wide-area environments; and should impose as little additional load as possible on the monitored resources.

4.5.2 ARMS Architecture

Through ARMS an application inherits a significant amount of the mechanisms needed for providing introspection [123], allowing the application designer to focus on the application-specific logic of monitoring and adaptation. An overview of the ARMS architecture is presented in Figures 16 and 15.

Hosts of interest to applications must become ARMS nodes. Different objects (including devices) at ARMS nodes are the sources of monitoring data. ARMS nodes connect to other nodes in specific sets or ARMS networks. ARMS clients connect to an ARMS network through an ARMS node that could reside on its own or another host. ARMS clients express their interests in different monitored objects by requesting state reports on such objects or selectively registering themselves with the streams associated with those same objects. Application request for continuous reports follow a

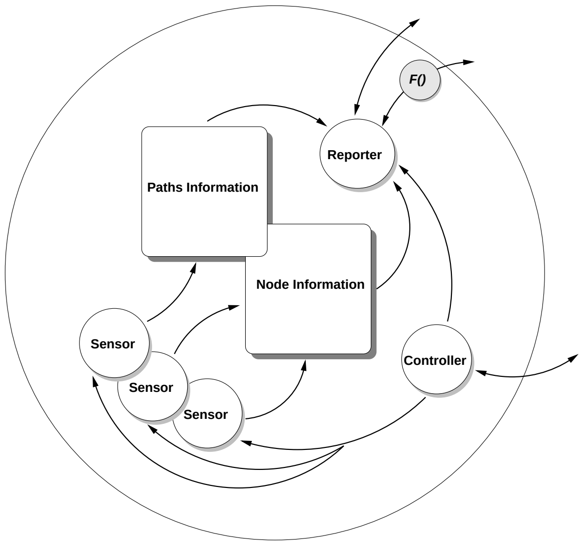


Figure 15: ARMS Node.

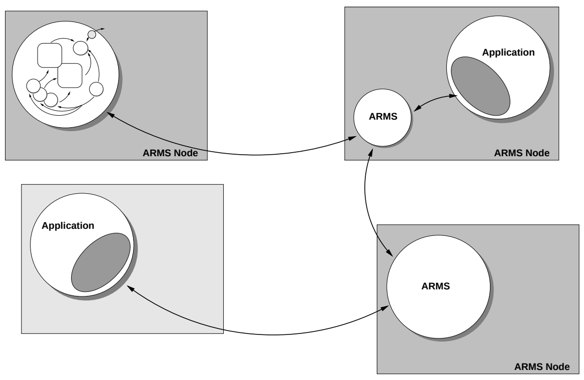


Figure 16: ARMS network.

lease model. Requests are to last for a user-specified (but limited) time span after which they will be canceled. Previously issued requests can be re-issued, before lease expiration, by using the request number. Correspondingly, a non-expired request can be canceled using this same number.

ARMS clients can select a subset of the data made available by different sensors, and integrate that data to produce application-specific monitoring metrics and decide on possible adaptations. In addition, filters in ARMS can be instantiated directly into the source of monitoring data streams with the corresponding savings in bandwidth, and in sink or source processing.

4.5.3 Sensors and Forecasting

ARMS obtains monitoring information from sensors. In the case of resources, sensors report the observed performance that the resource is able to deliver at the time of measurement. Measurements are taken at the application level, since the intention is to forecast actual application performance, and this is done using standard operating systems calls (such as vmstat and uptime) to facilitate portability.

Currently, ARMS nodes report information on memory availability, CPU load, disk free space, up-time, and number of users currently logged on, as well as host name, IP, number of CPUs, and configure triplet (as reported by GNU config.guess). Status reports of network paths between two ARMS nodes include latency and bandwith (see Table 7). ARMS uses a combination of passive and active sensors as needed [148]. Passive sensors exercise an external system utility, such as `uptime`, and scan the utility's output to obtain the required information. Active sensors, on the other hand, must conduct a performance experiment, such as timing a message exchange between two hosts, to measure the availability of the monitored resources.

ARMS sensors maintain histories of measurements. In order to forecast performance availability, ARMS includes a library with a number of predictive algorithms including mean-based methods such as running, trimmed, and sliding window averages as well as minimum, maximum, median and support to build autoregressive models.

4.6 Extending Directory Service

A very important component of most distributed systems infrastructures is a directory services that provides information about different objects in the environment, including services, resources, applications, and people, to other applications and their users. Well-known examples of directory services includes the Metacomputing Directory Service (MDS) [44] for Globus-based environments and the Intentional Naming System (INS) [2] for applications developed in the Oxygen [86] pervasive computing project. Directory services in both types of environments must support sophisticated object descriptions and query patterns, operate in highly dynamic environments, and scale to an increasingly large number of objects and users.

Recent research has addressed the limitations of current sevices, such as DNS and LDAP, originally designed for fairly static environments where updates are rare. Various research projects have focused on different aspects including their object descriptions and query languages and/or their scalability. However, most directory services still offer traditional, exclusively 'inactive' interfaces, through which clients interested in the values of certain objects' attributes must explicitly request such information from the server. When object attributes are frequently updated, those clients who need up-to-date information have no alternative but to query servers at rates that (at least) match the rates at which changes occur.

We argue that an *exclusively* inactive interface to a directory service for these new environments can hinder service scalability and indirectly restrict the behavior of potential applications [122]. In response, we extend the interfaces of directory services with a customizable *proactive* mode by which clients can express their interests in, and be notified of, changes in the environment.

The Proactive Directory Service (PDS) is an efficient and scalable information repository with an interface that includes a proactive push-based access mode. Through this interface, PDS clients can learn of objects (or types of objects) being inserted/removed from particular contexts (such as addition or removal of services or devices) and/or about changes to pre-existent objects.

4.6.1 The Proactive Directory Service

A potential disadvantage of a proactive approach is the loss of control from the client's perspective since, after having registered its interest on changes to an object, it is then at the "mercy" of the object's owner. PDS clients can regain control by dynamically customizing these notifications through filter functions instantiated at the server (or the object's owner) and by tuning these filters' functionalities via remote updates of some of their parameters. For expressiveness, PDS supports an attribute-value-based description of objects, similar in spirit to that used by INS.

Proactivity is a well-established system design technique. Physical devices such as buses and disks use interrupts as a proactive means of informing operating system kernels that a state change has occurred, allowing kernels to avoid repeated polling. Write-through caches are 'proactive', in that they ensure cache consistency without resulting in cache faults for applications that use such data. The publish/subscribe paradigm of distributed system design, with its roots in non-distributed reactive programming models, is another instance of proactivity, because peers in a publish/subscribe or event-passing system proactively notify interested parties of state changes. Finally, the active database [145, 89] approach and, in the world-wide web, Continual queries [83] propose proactivity with motivations similar to those behind PDS.

The use of proactivity in directory services has some precedents in DNS NOTIFY [136] and Ninja's Secure Directory Service (SDS) [29]. Until recently, changes to a DNS zone (DNS zones are contiguous subdivisions of a namespace) were propagated among the interested (replicated) name servers through a pulling mechanism initiated by the replicas. In order to reduce the load imposed on the master name servers, longer refresh times of the zone's data were normally adopted, but that benefit would come at the cost of long intervals of incoherence among authority servers whenever the zone is updated. In response, RFC 1996 proposed a mechanism for prompt notification of zone changes (DNS NOTIFY) through a proactive approach. Master servers can now inform slave servers when the zone has changed through an interrupt "which it is hoped will reduce propagation delay while not unduly increasing the masters' load". In SDS, services in the environment announce (broadcast) themselves periodically on a well-known multicast channel. Clients can thus eavesdrop on that same channel to hear what services are running and become aware of changes.

4.6.2 Adding Proactivity to a Directory Service

Our use of proactivity in directory services is straightforward. With each object managed by the directory service we associate a publish/subscribe channel for change notification and allow clients to subscribe to it. Changes are reported to interested parties over notification channels in the form of events. Examples of types of events include the creation or removal of an entry or changes to (the attributes of) an existing entry in the directory.

The primary advantage to clients of pull-based interfaces is control. Pull-based interfaces allow clients to manage when/if messages are sent and to anticipate replies (since the fact that a reply is impending and the type of information the reply carries are both known). Proactivity allows clients to trade control for performance, as message traffic is only generated when updates occur. As long as updates occur infrequently, this lack of control is not significant. However, a client that registers interest in an object that begins changing with unanticipated frequency soon finds itself swamped with update messages. These update messages may not even be needed when they arrive, or may only be needed depending on other application-specific factors; proactivity in this case does more harm than good.

At first glance, providing a filter at the client to discard unwanted or unneeded updates might seem enough. Although this does allow the application to ignore updates, the update messages are still sent across the network, increasing the load on the server, the network, and the client. Providing a single interface at the server to control proactive traffic is also insufficient, as different clients interested in changes may have different criteria for discarding update messages.

A better approach allows client-specific *customization* of the update channel. To customize a channel, a client provides a specification (in the form of a function) of what events it will be interested in. The server then uses these specifications, on a per-client basis, to determine whether or not to send the update event.

Preliminary evaluations comparing PDS with off-the-shelf implementations of DNS and LDAP confirm the performance advantages derived from proactivity, for clients and servers (see Section 6.4). For instance, we demonstrate experimentally that, contrary to common wisdom, the customization of notification through filter functions executed by the directory server need not translate into an overloaded server or result in excessive loads imposed on objects' owners. Instead, it can improve performance, as the additional filter code executed by the server or by owners is outweighed by gains in performance due to the elimination of unnecessary message communications (i.e., executions of protocol stacks).

4.6.3 PDS Concepts and Abstractions

In PDS, related information is organized into well-defined collections called *entities*. Each entity represents an instance of an actual type of object in the environment and has an associated set of properties, or *attributes*, with particular values.

Entities may be bound to names in different *contexts* and each context contains a list of name-to-entity bindings. Contexts themselves may be bound to names in other contexts, building an arbitrary directed naming graph.

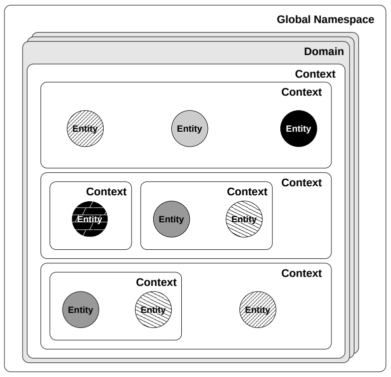


Figure 17: Domains, contexts and entities in PDS.

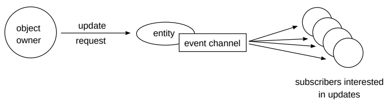


Figure 18: Achieving proactivity: when the owner of an object updates any of the object's attributes, notifications are sent to all interested clients.

Since a single secular namespace does not scale well, PDS divides the global name space into subspaces and assigns them to *domains*. Each domain implements its own namespace roughly as a tree, a single 'root' context and a context for each vertex.

Each element in PDS, be it a domain, a context, or an entity, has associated with it an event channel. The PDS interface allows clients to subscribe to the event channel for a particular object. With registration PDS clients provide a function that serves as an event handler and is invoked by the communications system when new events arrive. Each of the state-changing operations implemented by PDS submits an event to the notification channel associated with the appropriate object. When the owner of an object changes the value of some of the object's attributes (such as when 'Woodstock' is added to the 'Peanuts' cluster or 'Rerun' gets more memory), a change event is submitted to the channel belonging to the entity representing it.

PDS clients are able to customize notification channels by supplying filter functions, written in a portable subset of C, which are then dynamically compiled and installed at the server. Inside a filter function, it is possible to examine the notification data to determine whether or not the notification should be sent. For example, a notification corresponding to an update in CPU utilization in a given host may only be useful to an application if the new value is outside a certain range (see Figure 19). Notice that this customization is on a per-client basis; the actions and interests of one client do not affect updates received by another.

```
{
    if ((input.cpuUsage < 0.1) || (input.cpuUsage > 0.6)) {
        return 1; /* submit event into notification channel */
    }
    return 0; /* do not submit event into notification channel */
}
```

Figure 19: A specialization filter that passes only CPU-usage updates outside a pre-defined range.

4.6.4 PDS Architecture

The PDS architecture includes three main components: PDS clients, servers and object owners. PDS clients want to discover available objects in the environment and become aware of any change to them that could affect their functionality and/or performance. Object owners make their objects available by publicizing them through the directory services. Servers act as mediators between clients and objects's owners.

Owners' changes to their objects are published in the objects' associated notification channels. PDS clients can register interest in those changes by subscribing to the corresponding channels. Subscribed clients can then customize these notifications by supplying filter functions which are then dynamically compiled and installed at the server.

Filter functions are expressed in ECL, a subset of a general procedural language, and dynamic code generation is used to create a native version of F on the server. For ECL's details, the reader is directed to Section 5.3.

Chapter 5: Implementation

This chapter describes the prototype implementation of the Active Streams Framework (a description of its architecture was presented in Chapter 4). The framework, illustrated in Figure 20, is comprised of four core components: (1) the Active Streams Node provides the environment for streamlet execution; (2) the Streamlet Repository Service offers a pull-based service for code distribution; (3) the Active Resource Monitoring Service provides the needed infrastructure for resource monitoring, self-monitoring and adaptation; and last (4) the Proactive Directory Service acts as the framework's information repository providing an extended proactive interface more suited to the dynamism of the targeted environments.

The performance and flexibility requirements of this infrastructure are satisfied by the use of the *Portable Binary I/O* library (PBIO), an implementation of the *Native Data Representation* wire-format. PBIO provides the low-level support for typed, self-describing communication units required by the Active Streams approach. The framework relies on the ECho publish/subscribe communication infrastructure [37] for data and control transport and as the basis for the Active Streams implicit invocation mode of integration.

After presenting the implementations of the PBIO and ECho libraries as well as the current realizations of streamlets, detailed descriptions of each component's implementation are provided.

5.1 Native Data Representation and PBIO

This dissertation proposes a component-based approach to application/service programming for highly heterogeneous and dynamic environments. This and related projects on tool- and component-based approaches have increased the need for flexible and high performance communication systems. High-performance computing applications are being integrated with a variety of software tools to allow on-line remote data visualization [106], enable real-time interaction with remote sensors and instruments, or provide novel environments for human collaboration [102]. There has been a growing interest among high-performance researchers in component-based approaches, in an attempt to facilitate software evolution and promote software reuse [15, 104, 65]. When trying to reap the well-known benefits of these approaches, the question of what communications infrastructure should be used to link the various components arises.

In this context, *flexibility* and *high-performance* seem to be incompatible goals. Traditional HPC-style communications systems like MPI offer the required high performance, but rely on the assumption that communicating parties have *a priori* agreements on the basic contents of the messages being exchanged. This assumption severely restricts flexibility and makes application maintenance

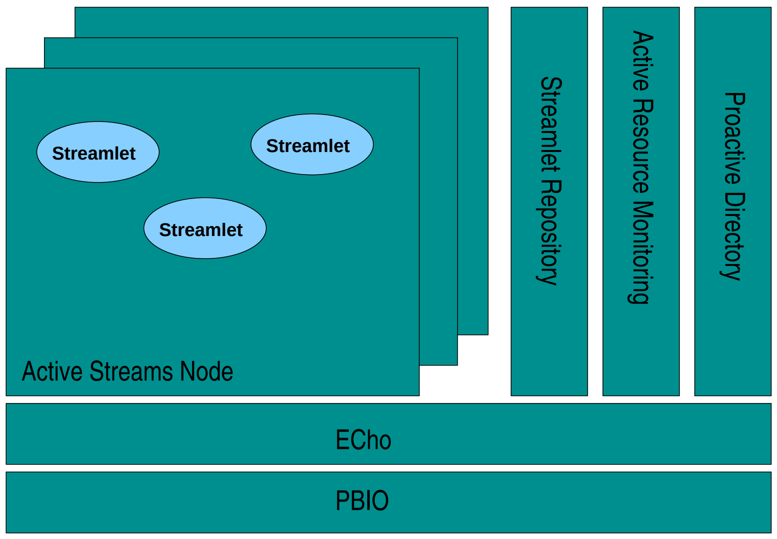


Figure 20: Active Streams Framework

and evolution increasingly onerous. The need for flexibility has led designers to adopt techniques such as the use of serialized objects as messages (Java's RMI [146]) or the use of meta-data representations like XML [138]. Both alternatives, however, have high marshalling and communications costs in comparison to the more traditional approaches [16, 153].

We observe that the flexibility and baseline performance of a data exchange system are strongly determined by its *wire format*, or how it represents data for transmission in a heterogeneous environment. Upon examining the flexibility and performance implications of using a number of different wire formats, we propose an alternative approach that we call *Native Data Representation*.

In the following subsections we describe the NDR approach, and provide some details on its implementation in the *Portable Binary I/O* library. PBIO provides the low-level flexible and high-performance communication required by Active Streams.

5.1.1 Native Data Representation as a Wire Format

The idea behind Native Data Representation (NDR) is quite simple. We avoid the use of a common wire format by adopting a "receiver makes it right" approach, where the sender transmits the data in its own native data format and it is up to the receiver to do any necessary conversion. Any translation on the receiver's side is performed by custom routines created through dynamic code generation (DCG). By eliminating the common wire format, the up and down translations required by approaches like XDR are potentially avoided. Furthermore, when sender and receiver use the same native data representation, such as in exchanges between homogeneous architectures, this approach allows received data to be used directly from the message buffer eliminating high copy overheads [116, 141]. When sender's and receiver's formats differ, NDR's DCG-based conversions have efficiency similar to that of systems that rely on *a priori* agreements to make use of compile- or link-time stub generation. However, because NDR's conversion routines are dynamically generated at data-exchange initialization, our approach offers considerably greater flexibility. The meta-data required to implement this approach and the runtime flexibility afforded by DCG together allow us to offer XML or object-system levels of plug-and-play communication without compromising performance.

5.1.2 Marshalling and Unmarshalling

Minimizing the costs of conversions to and from wire formats is a known problem in network communication [6]. Traditional marshalling/unmarshalling can be a significant overhead [32, 117], and tools like the Universal Stub Compiler (USC) [99] attempt to optimize marshalling with compile-time solutions. Although optimization considerations similar to those addressed by USC apply in our case, the dynamic form of the marshalling problem in PBIO, where the layout and even the complete field contents of the incoming record are unknown until run-time, rules out such static solutions.

Marshalling. Because PBIO's approach to marshalling involves sending data largely as it appears in memory on the sender's side, marshalling is computationally inexpensive. Messages are prefixed with a small (32-128 bits) *format token* that identifies the format of the message. If the format contains variable length elements (strings or dynamically sized arrays), a 32-bit length element is also added at the head of the message. Message components that do not have string or dynamic subfields (such as the entire message of Figure 21) are not subject to any processing during marshalling. They are already in 'wire format'. However, components with those elements contain pointers by definition. The PBIO marshalling process copies those components to temporary memory (to avoid destroying the original) and converts the pointers into offsets into the message. The end result of PBIO's marshalling is a vector of buffers which together constitute an encoded message. Those buffers can be written on the wire directly by PBIO or transmitted via another mechanism to their destination.

Unmarshalling. It is clear that the NDR approach greatly reduces sender side processing and increases flexibility since it allows the receiver to make run-time decisions about the use and processing of incoming messages without any previous knowledge of their formats. These benefits, however, come at the cost of potentially complex format conversions on the receiving end. Since the format of incoming records is principally defined by the native formats of the writers and PBIO has no *a priori* knowledge of the native formats of the communicating parties, the precise nature of this format conversion must be determined at run-time. Receiver-side unmarshalling thus essentially requires conversions of the various incoming 'wire' formats to the desired 'native' formats, which may require byte-order changes (byte-swapping), movement of data from one offset to another, or even a change in the basic size of the data type (for example, from a 4-byte integer to an 8-byte integer). While such conversion overheads can be nil for some homogeneous data exchanges, they can be considerably high (66%) for heterogeneous exchanges.

PBIO's approach to unmarshalling is based on dynamic code generation. Essentially, for each

```
typedef struct small_record
{
  int ivalue;
  double dvalue;
  int iarray[5];
} small_record, *small_record_ptr;
IOField small_record_fld[] =
{
  {"ivalue", "integer", sizeof(int),
   IOOffset(small_record_ptr,ivalue)},
  {"dvalue", "float", sizeof(double),
   IOOffset(small_record_ptr,dvalue)},
  {"iarray", "integer[5]", sizeof(int),
   IOOffset(small_record_ptr,iarray)},
  {NULL, NULL, 0, 0}
};
```

Figure 21: An example of message format declaration. `IOOffset()` is a simple macro that calculates the offset of a field from the beginning of the record.

incoming wire format, PBIO creates a specialized native subroutine that converts incoming records into the receiver's format. These native conversion subroutines are cached and reused based upon the format token of the incoming record. Reuse allows the costs of code generation to be amortized over many conversions. The run-time generation of conversion subroutines is essentially a more dynamic approach to the problems addressed by tools like USC[99]. Systems that can rely upon prior agreement between all communicating parties have no need for the extra dynamism we offer. However, more flexible communication semantics are required for today's component- and plug-and-play systems. In those situations, our DCG approach is a vital feature in order not to sacrifice performance in account of the needed flexibility.

5.1.3 Dealing with Formats

PBIO's implementation of NDR separates the detailed format descriptions from the actual messages exchanged. Format descriptions are registered with a format service and messages are prefixed with a small *format token* that identifies them. Record format descriptions in PBIO include the names, types, sizes and positions of the fields in the messages exchanged. Figure 21 shows a C language declaration that builds a format description for use in PBIO. Because the size and byte offset of each field may change depending upon the machine architecture and compiler in use, those values are captured using the C `sizeof()` built-in and the PBIO `IOOffset()` macro.

The format description in Figure 21 may look somewhat obscure, but it could easily be generated from the C typedef. In fact, both the typedef and the PBIO field list declaration can be generated from a higher level specification, such as a CORBA IDL struct declaration or even an XML schema (see Figure 22).

Different forms of specification are appropriate for different applications. The form of Figure 21 is easiest for integrating PBIO-based messaging into an existing C application, but forms such as those in Figure 22 may be more convenient for new applications. Regardless of the form of the specification, the capabilities of PBIO are defined by the types of messages that can be represented and marshalled.

```
interface small {
   struct small_record {
      long ivalue;
      double dvalue;
      long< 5 > iarray;
   };
}
```

(a) CORBA IDL specification

```
<schema>
   <element name="ivalue" type="integer"/>
   <element name="dvalue" type="double"/>
   <element name="iarray" type="integer"
             minOccurs=5 maxOccurs=5/>
</schema>
```

(b) XML Schema specification

Figure 22: Alternative message structure definitions.

PBIO types are C-style structures whose fields may be atomic data types, substructures of those types, null-terminated strings, and statically- or dynamically-sized arrays of these elements. In the case of dynamically-sized arrays, the array is represented by a pointer to a variable-sized memory block whose length is given by an integer-typed element in the record.[1]

Dynamic Formats. Because PBIO formats roughly correspond to a description of a C-style structures, the formats used by individual applications tend to be relatively static (as are those structures), and the field lists of locally-used records are known at compile time. However unlike many marshalling and communication mechanisms, PBIO does not depend in any way upon compile-time stub generation or any other compile-time techniques for its efficiency or normal operation. Field lists of the form of Figure 21 supplied at run-time are all that PBIO requires to build formats for marshalling and unmarshalling. These highly dynamic capabilities of PBIO are useful in creating plug-and-play components that operate upon data that may not be specified until run-time. These more highly dynamic features of PBIO are also exploited by an XML interface that 'dehydrates' XML into a PBIO message for transmission and 're-hydrates' it at the receiver based on a run-time specified schema described in detailed in [144].

Format Description Representation and Size. One factor affecting the cost of dealing with formats is the actual size of the format information to be exchanged. Unlike the records they describe, PBIO format information is represented on the wire by a well-defined structure that includes some general format information of fixed size (≈ 16 bytes), the format name, and information for each of the fields in the format's field list. The information for each field consists of a fixed size portion (currently 12 bytes) and a variable size portion (the field name and base type). A general expression for the approximate wire size of format information is:

$$\texttt{size} \approx 16 + strlen(format_{name}) + \sum_{f \epsilon Fields} (12 + strlen(f_{name}) + strlen(f_{type}))$$

The first two bytes of the format information give its overall length and are always in network byte order. One of the next bytes specifies the byte order of the remaining information in the format.

[1]PBIO does not attempt to represent recursively defined pointer-based data structures such as trees or linked lists.

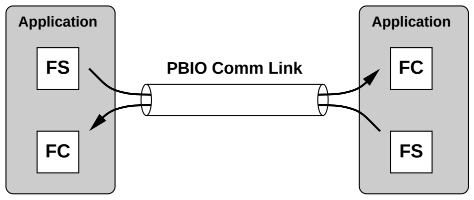


Figure 23: Single PBIO connection showing per-connection format servers (FS) and caches (FC).

PBIO format operations that involve the transfer of format descriptions always use this wire format for their exchanges. It is important to note that such operations are associated only with one-time events, such as a new format description registration or the first occurrence of a message of a particular format.

Format Servers and Format Caches. The *format service* in PBIO is provided by *format servers* which issue format tokens when formats are registered with them. For identical formats (same fields, field layout, format name, and machine representation characteristics), a format server issues identical format tokens. Format tokens can be presented to format servers in order to retrieve complete format information. *Format caches* are repositories of format information, indexed by format tokens, and exist on both the encoding and decoding clients to optimize communication with format servers.

The details of format communication in PBIO depend to some extent upon the circumstances of its use. Two principal modes are:

- **Connected PBIO:** where PBIO performs marshalling/unmarshalling and directly controls transmission on the network, and
- **Non-connected PBIO:** where PBIO performs marshalling/unmarshalling, but is not in direct control of network transmission.

The first case is the simplest one because PBIO can ensure that the format information for a given record is sent across the wire before the first record of that format. In this case, format information is issued (by the sender) and cached (by the receiver) on a per-connection basis. Because formats are always interpreted in the context of a particular connection, format tokens in this mode are simple 32-bit integers where the token with value i is the i th format transmitted on that connection. Since the sender always transmits format information first, essentially pre-loading the receiver's format cache, there are no requests for it. This situation is depicted in Figure 23.

The case of non-connected PBIO is more interesting. Here a message (along with its format token) is sent by non-PBIO means to a third party. Because PBIO does not control transmission, it cannot pre-load the format cache of the receiver as in the former case. Instead, the third-party receiver must be able to use the format token to retrieve the full format information. This is essentially a naming problem, and there are a number of possible implementation options. In the

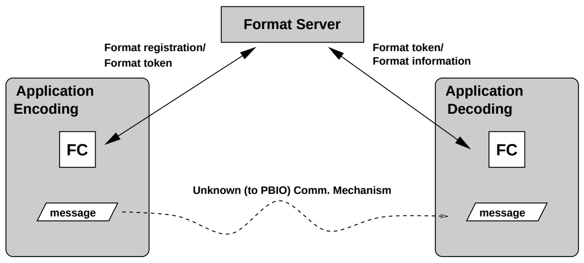


Figure 24: Simple format service arrangement in non-connected PBIO. A single PBIO format server is shared by all communicating applications. Each application has its own format cache (FC) to minimize communication with the server.

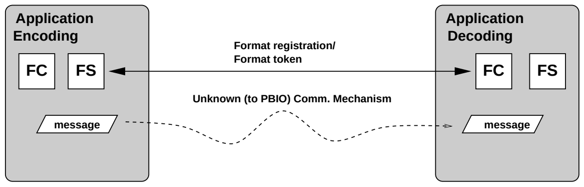


Figure 25: The “self-service” format arrangement in non-connected PBIO. Each application acts as a format server (FS) for its own formats and maintains a cache (FC) for formats registered elsewhere.

simplest one, depicted in Figure 24, a format server, located at a well-known address, serves all communicating parties.

Any format token can be presented to the server to retrieve full format information. At the other end of the spectrum, each communicating client can act as the format server for its own formats, as shown in Figure 25.

The self-server arrangement is similar to the connected PBIO arrangement of Figure 23, except that there is one server and one cache per process instead of one per connection. Because the communication mechanism is unknown to PBIO and because format tokens are only meaningful when presented to the issuing server, the format token must contain enough information for the client to identify the correct server. Communication with the issuing format server is generally *not* via the channels that propagate the message, though PBIO can be made to use one-to-one third-party communication links for format communication through special callbacks.

These two schemes, and variants between the extremes, have different performance characteristics with respect to communication startup costs. For example, the single format server approach maximizes the benefits of caching because identical formats always have identical format tokens, to the benefit of long-running clients. However, format registration for a new client always requires a communication with the central format server. In the scheme where each client is its own format server, format registration requires no network communication and is therefore quite cheap. However, caches will be less well utilized because two clients of the same machine architecture and transmitting the same records will have non-identical format tokens.

5.2 Publish/Subscribe Systems

Recently there has been considerable interest in event-based or implicit invocation as an alternative integration techniques for system software [103, 54, 38, 142]. This architectural style of integration is naturally supported by publish/subscribe systems. Publish/subscribe systems have become an important component of many distributed applications and services as they are also well-suited to the reactive nature of applications such as collaborative environment, mobile [140] and pervasive computing [60].

Under the publish-subscribe paradigm a number of system components transmit and/or receive pieces of information, called *messages*, in response to *events* occurrences. Receivers can subscribe to messages published by producers upon the occurrence of some event. A given component can, at any time, act as both a publisher and a subscriber. Events can be occurrences such as the modification of a component state, a change in the availability of resource, or just the sending of an application message.

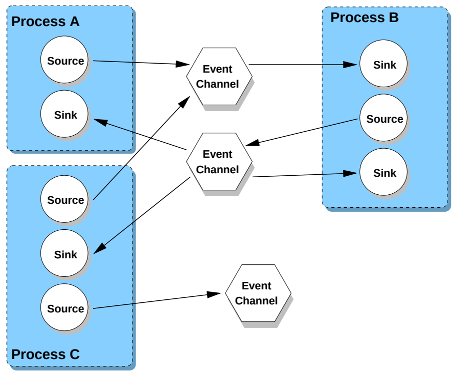
ECho is a high-performance publish/subscribe communication infrastructure developed at Georgia Tech [37]. Several attributes distinguish ECho from previous work and make it a good fit for the roles of data and control transport, and as the basis for the style of component integration used in Active Streams. First, ECho transports distributed data with performance similar to that achieved by communication systems typically used for high-performance applications (like MPI [45]). This level of performance is required if the communication mechanism is to support the normally large data flows that are part of applications such as distributed collaboration. Second, ECho efficiently supports communication across heterogeneous machines, a capability partly derived from its ability to recognize and provide runtime translation for user-defined message formats. Finally, it provides for efficient dynamic type extension and reflection essential to support system evolution. The following subsections describe ECho, its functionality and some details on its implementation.

5.2.1 ECho Model and Functionality

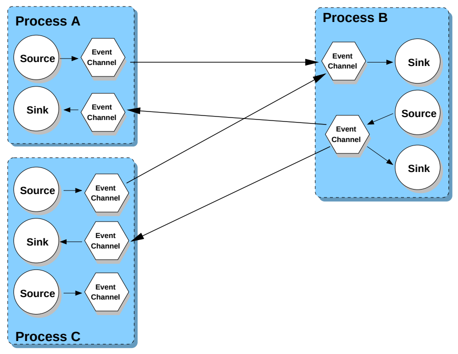
At a high-level, ECho implements a model similar to that proposed by the CORBA Event Services specifications [58] and shares many ideas with other related research efforts such as IBM's Gryphon [125], Siena [20] and Elvin [119].

ECho supports the publish/subscribe paradigm using a simple subscription mechanism commonly known as *channel*. Channels serve as the rendezvous point for publishers and subscribers. Components notify the occurrences of events by posting notifications to one or more channels. Every notification posted is delivered by the underlying mechanism to all the interested parties that have subscribed to the channel. Channels are essentially entities through which the extent of event notification propagation is controlled. iBus [85] and CORBA Event Service [58] adopt a similar model.

ECho's principal contribution to specializing data flows is the concept and realization of *derived event channels*. Receivers in publish/subscribe systems have normally a way of specifying the messages they are interested on. In the context of ECho's model, one way to approach this problem would be to create a new event channel and interpose an event filter as shown in Figure 27.



(a) Abstract view of Event Channels.



(b) ECho Realization of Event Channels.

Figure 26: Using Event Channels for Communication.

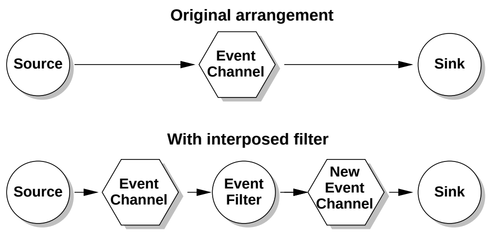
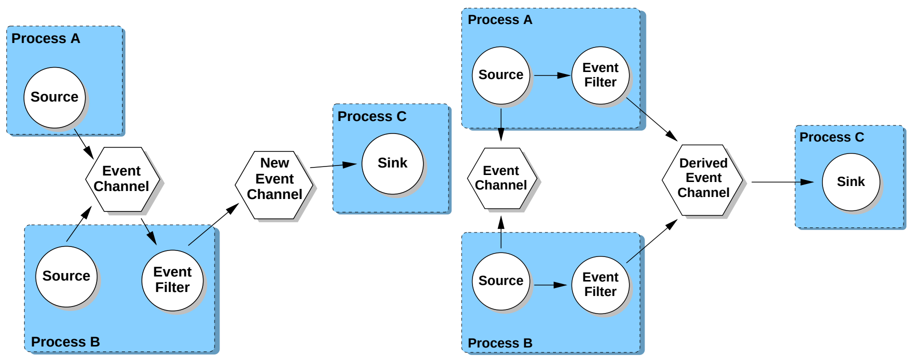


Figure 27: Source and sink with interposed event filter.



(a) Filter with more than one source

(b) A derived event channel and function F moved to event sources

Figure 28: Channel configurations.

The event filter could be located on the same node as the event source and be perceived as a normal event sink by the original event channel and a normal source by the new, or filtered, event channel. Although this solution would not disturb the normal function of the original event channel, it fails if there were more than one event source associated with the original event channel as there would still be raw, unfiltered events traveling over the network from Process A to Process B (Figure 28-(a)).

The normal semantics of event delivery schemes do not offer an appropriate solution to the event filtering problem.

Since the normal semantics of event delivery schemes do not offer an appropriate solution, ECho extends it with the concept of a *derived* event channel. Rather than explicitly creating new event channels with intervening filter objects, subscribers that wish to receive filtered event data can create a new channel whose contents are derived from the contents of an existing channel through an subscriber-supplied derivation function. The event channel implementation will move the derivation function to all event sources in the original channel, execute it locally whenever events are submitted, and transmit any resulting event via the derived channel. This approach has the advantage of eliminating unwanted event traffic and the associated waste of computational and network resources. In fact, if the derived event channel has sinks that are local to any of the sources in the original traffic, network traffic between those elements is avoided entirely. Figure 28-(b) shows the logical configuration of a derived event channel.

While the concept of derived event channels bears some similarity to prior work on content-based filtering (as in Siena [19] and Elvin [119]) and pattern-based filter/transformation (as in Gryphon [125]), ECho allows more general computations over event data and accomplishes those computations efficiently. This efficiency is based on semantics that don't require centralized event distribution and the use of dynamic code generation to create native filter/transformation functions. Our work is complementary with that of the Gryphon project as we are not looking at the optimal mapping of an information flow graph onto a network of brokers but rather concern ourself with the most efficient execution of such computations. The Java-based approach of DACE [42] offers broad generality in content-based subscriptions, but lacks the transformation capacity of derived event channels and offers significantly lower throughput and high latency than ECho.

5.2.2 Efficient Notification Propagation

ECho event channels, unlike many CORBA event implementations and other event services such as Elvin [119], are not centralized in any way. ECho channels are light-weight virtual entities. Figure 26-(a) depicts a set of processes communicating using event channels. The event channels are shown as existing in the space between processes, but in practice they are distributed entities, with bookkeeping data residing in each process where they are referenced as depicted in Figure 26-(b). Channels are *created* once by some process, and *opened* anywhere else they are used. The process which creates the event channel is distinguished, in that it is the contact point for other processes wishing to use the channel. The channel ID, which must be used to open the channel, contains the contact information for the creating process (as well as information identifying the specific channel). However, event notification distribution is not centralized and there are no distinguished processes during notification propagation. Event messages are always sent directly from an event source to all sinks and network traffic for individual channels is multiplexed over shared communications links.

ECho is implemented on top of the Communication Manager (CM) and PBIO, packages developed at Georgia Tech to simplify connection management and heterogeneous binary data transfer. As such, it inherits from these packages portability to different network transport layers and threads packages. CM and PBIO operate across the various versions of Unix and Windows NT, have been used over the TCP/IP, UDP, and ATM communication protocols and across both standard and specialized network links like ScramNet [27].

In addition to offering interprocess event message delivery, ECho also provides mechanisms for associating threads with event handlers allowing a form of intra-process communication. Local and remote sinks may both appear on a channel, allowing inter- and intra-process communication to be freely mixed in a manner that is transparent to the event sender. When sources and sinks are within the same address space, an event message is delivered by directly placing the message into the appropriate shared-memory dispatch queue. While this intra-process delivery can be valuable, this paper concentrates on the aspects of ECho relating to remote delivery of event messages.

5.2.3 Event Notification Types and Typed Channels

One of the differentiating characteristics of ECho is its support for efficient transmission and handling of fully typed events. Some event delivery systems leave event data marshalling to the application. ECho allows types to be associated with event channels, sinks and sources and will automatically handle heterogeneous data transfer issues. Building this functionality into ECho using PBIO allows for efficient layering that nearly eliminates data copies during marshalling and unmarshalling. As others have noted [81], careful layering to minimize data copies is critical to delivering full network bandwidth to higher levels of software abstraction. The layering with PBIO is a key feature of ECho that makes it suitable for applications that demand high performance for large amounts of data.

Base Type Handling and Optimization. Functionally, ECho event types are most similar to user defined types in MPI. The main differences are in expressive power and implementation. Like MPI's user defined types, ECho event types describe C-style structures made up of atomic data types. Both systems support nested structures and statically-sized arrays. ECho's type systems extends this to support null-terminated strings and dynamically sized arrays.[2]

While fully declaring message types to the underlying communication system gives the system the opportunity to optimize their transport, MPI implementations typically do not exploit this opportunity and often transport user defined types even more slowly than messages directly marshalled by the application. In contrast, ECho achieve a significant performance advantage by adopting NDR for wire format (Section 5.1).

Type Extension. ECho supports the robust evolution of sets of programs communicating with events, by allowing variation in the data types associated with a single channel. In particular, an event source may submit an event whose type is a superset of the event type associated with its channel. Conversely, an event sink may have a type that is a subset of the event type associated with its channel. Essentially this allows a new field to be added to an event at the source without invalidating existing event receivers. This functionality is extremely valuable for system evolution since it may avoid the need for simultaneous upgrades upon minor changes on types. ECho even allows type variation in intra-process communication, imposing no conversions when source and sink use identical types but performing the necessary transformations when source and sink types differ in content or layout.

The type variation allowed in ECho differs from that supported by message passing systems and intra-address space event systems. For example, the Spin event system [103] supports only statically typed events. Similarly, MPI's user defined type interfaces do not offer any mechanisms through which a program can interpret a message without *a priori* knowledge of its contents. Additionally, MPI performs strict type matching on message sends and receives, specifically prohibiting the type variation that ECho allows.

In terms of the flexibility offered to applications, ECho's features most closely resemble those of systems that support the marshalling of objects as messages. In these systems, subclassing and type extension provide support for robust system evolution that is substantively similar to that provided by ECho's type variation. However, object-based marshalling often suffers from prohibitively poor performance. ECho's strength is that it maintains the application integration advantages of object-based systems while significantly outperforming them.

5.3 Streamlets

A critical issue in the implementation of streamlets and filters for ECho's derived event channels is the nature of the function and its specification. Since the function is specified by the client but must be evaluated at the (possibly remote) source, a simple function pointer is obviously insufficient. There are several possible approaches to this problem, including:

[2] In the case of dynamically sized arrays, the array size is given by an integer-typed field in the record.

- severely restricting the function specification language, perhaps to a set of relational, equality and logical operators;
- using a generic, language such as C, but relying on pre-generated shared object files; or
- using interpreted code, like Tcl/Tk [101] or Java [77].

Having a relatively restricted filter language is the approach chosen in the CORBA Notification Services [59] and in Siena [19]. While this facilitates efficient interpretation, the restricted language may not be able to express the full range of conditions useful to an application, thus limiting its applicability.

Once agreed that a more general programming language is a better alternative, a remainig question is its translation in heterogeneous environments. One might consider supplying these functions in the form of a shared object file that could be dynamically linked into the process of the ASNs or an event source. Using shared objects allows these functions to be more general, but requires the client to supply them in a native object file for each possible destination. This is relatively easy in a homogeneous system, but becomes increasingly difficult as heterogeneity is introduced.

In order to avoid problems with heterogeneity one might supply such functions in an interpreted language such as a Tcl or Java. This would allow general functions and alleviate the difficulties with heterogeneity, but it would impact efficiency. Because of our focus on high performance computing and since most streamlets and filter we have found are quite simple, we have chosen a different approach that maintains high efficiency at some price in flexibility. We express functions in ECL, a subset of C, and resort to dynamic code generation to create efficient native versions of such functions on the target host. ECL may be extended as future needs warrant, but currently it is a subset of C, supporting the C operators, `for` loops, `if` statements and `return` statements. For experimentation, however, and in order to consider potential useful extensions to ECL, we have also enabled the use of general C-based shared-object modules.

ECL's dynamic code generation capabilities are based on Icode, an internal interface developed at MIT as part of the 'C project [108]. Icode is itself based on Vcode [40], also developed at MIT by Dawson Engler. Vcode supports dynamic code generation for MIPS, Alpha and Sparc processors. We have extended it to support MIPS n32 and 64-bit ABIs, Sparc 64-bit ABI, and x86 processors[3]. Vcode offers a virtual RISC instruction set for dynamic code generation. The Icode layer adds register allocation and assignment. ECL consists primarily of a lexer, parser, semanticizer and code generator.

Figure 29 shows a streamlet example extracted from our atmospheric model application. The streamlet computes (and passes) the average velocity of wind in a given area.

5.3.1 Streamlet Parameterization and Static Variables

Streamlets and filters for ECho's derived channel expressed in ECL have the ability to label variables as "static", so that their values persist across multiple invocations of the function. This allows the

[3]Integer x86 support was developed at MIT. We extended Vcode to support the x86 floating point instruction set (only when used with Icode).

```
{
   int i, j;
   double sum = 0.0;
   for (i = 0; i < 37; i = i + 1) {
      for (j = 0; j < 253; j = j + 1) {
         sum = sum + input.wind_velocity[j][i];
      }
   }
   output.average_velocity = sum / (37 * 253);
}
```

Figure 29: A streamlet that computes (and passes) the average of an input array.

implementation of functionality that depends on some amount of persistent state, such as a moving average computation.

Another useful feature is the ability to *parameterize* a derivation function. Parameterization allows a parameter block to be associated with a function. This block is read-only to the function, but can be updated remotely in a push-type operation. This is useful for fine-grain adaptations such as "tuning" the range of a functions that filters out records outside a given range (Figure 30).

5.4 Active Streams Node

Computational units make themselves available by running as Active Streams Nodes (ASNs), where each ASN provides a well-defined environment for streamlet execution. ASNs decouple the application- and streamlet-specific functionality from the general administrative tasks needed to support streamlet execution, such as scheduling, monitoring and reconfiguration. An ASN also provides the basic resources needed by streamlets and enforces constraints on how those resources can be used and includes an interface that allows the control of its execution.The existing implementation of ASN is built on top of ECho and is designed to be portable and efficient, and eventually safe and secure.

ASNs register themselves with a set of other ASNs, with ASNs in a give set comprising an overlay network structured as a fully-connected graph. When an ASN is started at a particular host, it is given the names of the overlay network(s) it is supposed to join (as well as other configuration parameters such as maximum memory available and directory service contact point).

An ASN could be assigned to serve a particular application or a set of them. By adopting the first approach, the costs of inter-streamlet and streamlet-ASN communication could be minimized (directly placing a message into a memory location shared by colocated sources and sinks), in

```
{
    if ((input.trade_price < param.range_low_bound) ||
        (input.trade_price > param.range_high_bound)) {
        return 1; /* submit event into derived channel */
    }
    return 0; /* do not submit event into derived channel */
}
```
Figure 30: A filter function with range specified as a parameter.

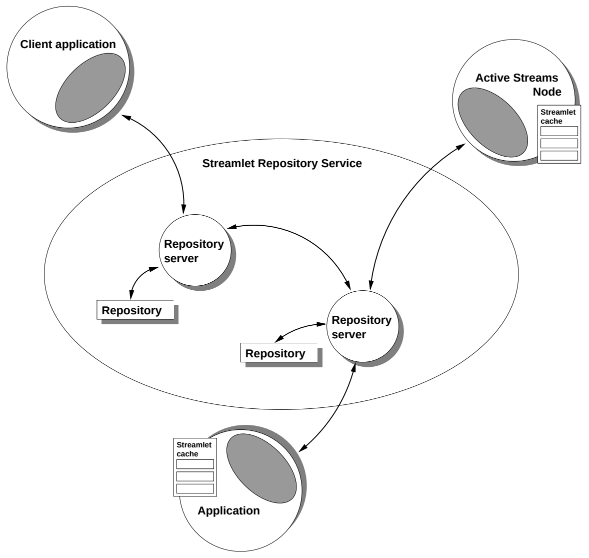


Figure 31: Streamlet repository service.

addition many protection issues could be avoided in this manner. On the other hand, assigning an ASN to a set of applications would allow us to do cross-application adaptations. Since the trade-offs presented by both options make the selection of the "right" approach not obvious, we have opted for a flexible solution that will allows to further investigate this topic.

ASNs provide the thread on which streamlets execute. Streamlets are registered through the runtime system with an associated set of conditions for their activation. Such conditions are stated as the minimum set of data units (threshold) that must be available per incoming stream before execution. A streamlet is eligible for execution whenever all of its input streams are over threshold. Once triggered, a streamlet executes to completion. In addition, ASNs provide the queues associated with a streamlet's incoming and outgoing streams. These queues decouple the streamlets execution by introducing explicit control boundaries and provide rate-matching between them.

5.5 Streamlet Repository Service

The Streamlet Repository Service (see Figure 31) is composed of a number of repository servers, and a client library that can be linked to any application, making it thus capable of fetching components from these servers. The client-side library requires about 20 KBytes of memory, which makes it sufficiently small to be used in end-devices with limited resources such as PDAs.

The SRS client API supports the insertion, search, retrieval and removal of streamlets from a given repository based on the streamlet characteristics and/or ID.

Each SRS client maintains a streamlet cache, keyed by streamlet ID, that contains all information on the streamlet as well as the streamlet's code. The SRS cache API allows the creation and destruction of caches, as well as the insertion, lookup and removal of streamlets from a given cache, based on the streamlet ID.

There are at least a couple of ways by which a client application can find a streamlet repository server. The client could be configured with a list of unicast repository server addresses or it could come with an anycast address [105]. This anycast address or the list of unicast addresses can be requested through a directory service like PDS.

5.5.1 Streamlets Descriptions

Streamlets are described by a set of properties (see Table 6 for a detailed description of each) that includes most of those advocated by the Open Software Description Format (OSD), an application of XML to support automated software distribution environments proposed by Marimba Incorporated and Microsoft Corporation [66]. We have included an attribute-value list to cope with less common or unforeseen attributes.

Table 6: Streamlet description fields in the repository

Field	Type	Example	Note
Name	string	CropImage-1	Name and version
UID	string	MD5 string	Cryptographic hash
OSRelease	enum	SunOS-5.7	`uname -rs`
Processor	enum	SPARC	`uname -p`
Content	enum	CSObj	Lang. and format
Author	string	mafalda@quino.net	Author's email
Description	string	Reduce image	Descriptive comment
Attribute list	a-v list	$<< a1 = v1 >< a2 = v2 >>$	List of attr-value pairs

5.5.2 Safety Considerations

SRS clients fetch streamlets for executing them in their hosts. Without the proper precautions, a node may import and run unsafe code, thereby opening the host to abuse through misuse of its resources.

We address security in SRS through policy and cryptology. Authors and streamlets have unique IDs. We use authors' email addresses as their IDs [4]. Each streamlet has a unique identifier consisting of a cryptographic hash of the streamlet code. Our current implementation uses MD5 [115] to this end. This scheme eliminates the need for a centralized streamlet naming system; guarantees that the streamlets attached to streams are those specified by the user, assuming that the configuration command has not been modified on the wire and that the ASN has not been compromised; and eliminates code versioning problems. In addition, each configuration command carries a pre-computed hash key that can be used to rapidly look up the streamlets' code in a cash. Since each ASN can reliably check the source of a streamlet and the identity of its developer, it's able to decide, based on that information, whether or not it should accept and execute the streamlet's code.

[4]Although in our currently implementation authors' identities are not certified, we have taken measures to add public-key certificates following a protocol similar to X.509 [21]

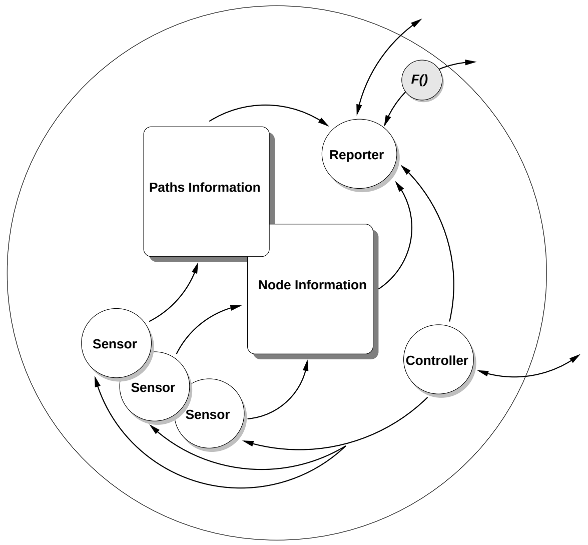


Figure 32: Structure of an ARMS Node.

5.6 Active Resource Monitoring Service

Through ARMS an application inherits a significant amount of the mechanisms needed for providing introspection [123], allowing the application designer to focus on the application-specific logic of monitoring and adaptation. An overview of the ARMS architecture is presented in Chapter 4, Figures 16 and 15.

Hosts of interest to applications must become ARMS nodes. Different objects (including devices) at ARMS nodes are the sources of monitoring data. ARMS nodes connect to other nodes in specific sets or ARMS networks. ARMS clients connect to an ARMS network through an ARMS node that could reside on its own or another host. ARMS clients express their interests in different monitored objects by requesting state reports on such objects or selectively registering themselves with the streams associated with those same objects. Application request for continuous reports follow a *lease* model. Requests are to last for a user-specified (but limited) time span after which they will be canceled. Previously issued requests can be re-issued, before lease expiration, by using the request number. Correspondingly, a non-expired request can be canceled using this same number.

An ARMS node obtains monitoring information from sensors. In the case of resources, sensors report the observed performance that the resource is able to deliver at the time of measurement. Measurements are taken at the application level, since the intention is to forecast actual application performance, and this is done using standard operating systems calls (such as vmstat and uptime) to facilitate portability. ARMS uses a combination of passive and active sensors as needed [148]. Passive sensors exercise an external system utility, such as `uptime`, and scan the utility's output to obtain the required information. Active sensors, on the other hand, must conduct a performance experiment, such as timing a message exchange between two hosts, to measure the availability of the monitored resources.

Different passive and active sensors collect information on node and path characteristics such as available memory and CPU in a node, or perceived latency in a path. Table 7 describes the information currently collected by ARMS nodes. The reporters provides the collected information to subscribed or requesting clients.

Status reports of network paths between two ARMS nodes include latency and bandwidth. Latency is measured sending an arbitrarily small message between two nodes and using one-half of the round-trip time as an approximation. Bandwidth, or better, effective bandwidth is obtained by sending an arbitrarily big sized message (and waiting for the reply) and applying the following formula:

$$Effective - Bandwidth = \frac{D}{T_{transfer} - RTT}$$

where D is the data size transfer, $T_{transfer}$ is the data transfer time and RTT is the predicted round trip time.

For CPU availability ARMS uses the same passive approach as NWS [148]; the percentage of CPU availability is computed as:

$$Available\_CPU = T_{idle} + T_{user}/RP + (T_{user} * T_{system}/RP)$$

where T_{idle} is the percentage of time the CPU is idle, T_{user} is the percentage of time CPU is executing user code, T_{system} is the percentage of time the CPU is executing system code, and RP is the number of runnable processes. The resulting value can be used to compute the CPU slowdown a process will experience due to contention. The rationale for this formula is that a new job (running with standard priority) should be entitled to all the idle time, and a fair share of the available user time.

ARMS sensors maintain histories of measurements. In order to forecast performance availability, ARMS includes a library with a number of predictive algorithms including mean-based methods such as running, trimmed, and sliding window averages as well as minimum, maximum, median and support to build autoregressive models.

Table 7: ARMS node and path status report information

Resource	Description
hostname	Host name
ipAddress	IPv4/IPv6 address
numProc	Number of processors
configGuess	Configuration triplets from GNU autotools
freeMem	History of available memory
cpuAvailable	History of CPU availability
machineUp	How long the system has been running
numUsers	Number of users currently logged on
diskFree	History of disk free space (in KB)
latency	Path latency history between two nodes
bandwidth	Path bandwidth history between two nodes

```
<<device = camera>
    <<type = USB>
     <name = /dev/video0>
     <image
       <max-resolution = 640x480>
       <min-resolution = 160-120>
       <format = RGB>>
     <fps = 20>>>
```

Figure 33: Attribute-value representation of a camera.

Clients can select a subset of the available data and integrate it to produce application-specific monitoring metrics and decide on possible adaptations. Clients can easily redefine their monitoring views by adapting/replacing the filters used for monitoring while the system is running, as their workload and environment change. In this manner, applications have an additional level of adaptation that would not be possible were the monitoring and adaptation mechanisms coded statically for each of them. In addition, most of these filters can be instantiated directly in the source of monitoring data streams with the corresponding savings in bandwidth, and in sink and source processing.

5.7 Proactive Directory Service

The Proactive Directory Service (PDS) is an efficient and scalable information repository with an interface that includes a proactive push-based access mode. Through this interface, PDS clients can learn of objects (or types of objects) being inserted/removed from particular contexts (such as addition or removal of services or devices) and/or about changes to pre-existent objects. PDS is implemented in C/C++ making use of ECho and PBIO.

In PDS, related information is organized into well-defined collections called *entities*. Each entity represents an instance of an actual type of object in the environment and has an associated set of properties, or *attributes*, with particular values. An attribute can be seen as a category in which the entity can be classified and a value, as the entity's classification in that category. Attribute-value pairs can be organized in a hierarchical manner indicating some kind of *dependency*. As an example consider a USB camera with devices name `/dev/video0`, that generates RGB images at 20 fps with a maximum resolution of 640x480 and a minimum of 160x120, this can be represented as in Figure 33.

Entities may be bound to names in different *contexts* and each context contains a list of name-to-entity bindings. Contexts themselves may be bound to names in other contexts, building an arbitrary directed naming graph. Since a single secular namespace does not scale well, PDS divides the global name space into subspaces and assigns them to *domains*. Each domain implements its own namespace roughly as a tree, a single *root* context and a context for each vertex.

Each element in PDS, be it a domain, a context, or an entity, has associated with it an ECho event channel. The PDS interface allows clients to subscribe to the event channel for a particular object. With registration PDS clients provide a function that serves as an event handler and is invoked by the communications system when new events arrive. Each of the state-changing operations implemented by PDS submits an event to the notification channel associated with the appropriate object. When the owner of an object changes the value of some of the object's attributes (such as when 'Woodstock' is added to the 'Peanuts' cluster or 'Rerun' gets more memory), a change event is submitted to the channel belonging to the entity representing it.

PDS clients are able to customize notification channels by supplying filter functions, written in ECL, which are then dynamically compiled and installed at the server. To this end, it makes use of ECho's derived event channels capability. Inside a filter function, it is possible to examine the notification data to determine whether or not the notification should be sent. For example, a notification corresponding to an update in CPU utilization in a given host may only be useful to an application if the new value is outside a certain range. Notice that this customization is on a per-client basis; the actions and interests of one client do not affect updates received by another.

Chapter 6: Evaluation

This chapter presents evaluation results that quantify the costs and benefits of the Active Streams approach as reflected by the current implementation of its associated framework. We progress in a bottom-up fashion, starting with an analysis of the performance and flexibility of the *Portable Binary I/O* library (PBIO), our implementation of the *Native Data Representation* wire-format.

We proceed, in Section 6.2, to show the results from a detailed performance evaluation of ECho. Given that ECho is the publish/subscribe communication infrastructure used by Active Streams for data and control transport, and as the basis for component integration, these results are a determining factor in the cost/benefit analysis of the overall approach.

In Section 6.3 we present experimental results that demonstrate the costs and benefits of Active Streams for end-user applications. We discuss results illustrating the basic overhead of our approach, the benefits of stream specialization, and the need for multiple points of adaptation over the datapath.

One of the innovative components of the Active Streams Framework is our Proactive Directory Service. This chapter also includes evaluation results that demonstrate the costs and benefits of this new proactive approach.

We close the chapter with a summary of our evaluation results.

6.1 PBIO Implementation of NDR

This section compares the performance and flexibility of PBIO with that of systems like MPI, CORBA, and XML-based ones. None of systems compared share PBIO's stated goals of supporting both flexible and efficient communication. MPI is chosen to represent traditional HPC communication middleware that depends upon *a priori* knowledge and sacrifices flexibility for efficiency. XML-based mechanisms are examined because they emphasize plug-and-play flexibility in communication without considering performance. CORBA is chosen as a relatively efficient representative of the object-based approach that is becoming popular in distributed high-performance computing. Some object-based notions of communication, such as exchanging marshalled objects, can potentially offer communication flexibility that is as good or better than PBIO's. However, current implementations of object marshalling are too inefficient for serious consideration in high performance communication. Because of this, our measurements of CORBA in this context are only for one-way invocations where the data is carried as a single CORBA `struct` parameter. In this style of communication, CORBA offers little flexibility (no reflection or subclassing are possible).

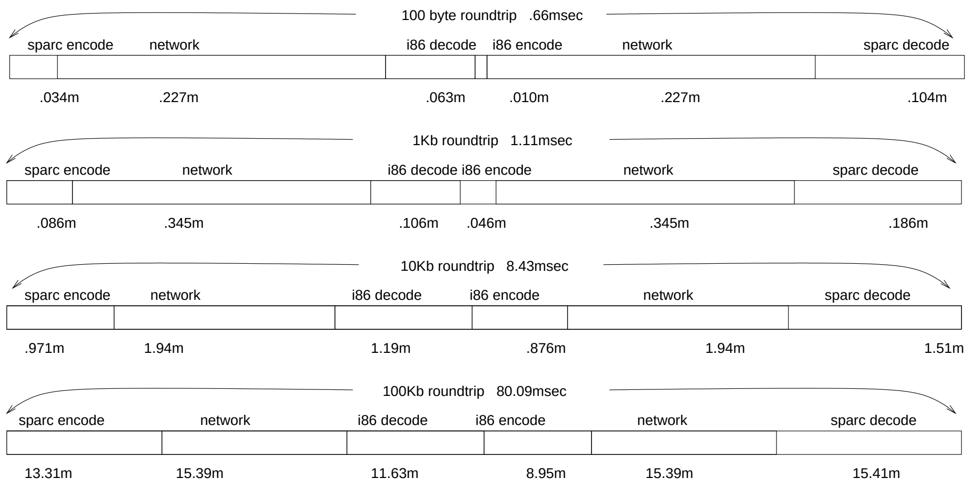


Figure 34: Cost breakdown for message exchange.

6.1.1 Analysis of Costs in Heterogeneous Data Exchange

Before analyzing the various packages in detail, it is useful to examine the costs in an exchange of binary data in a heterogeneous environment. As a baseline for this discussion, we use the MPICH [79] implementation of MPI. Figure 34 represents a breakdown of the costs of an MPI message round-trip between a x86-based PC and a Sun Sparc connected by 100 Mbps Ethernet.[1]

The time components labeled "Encode" represent the time span between the point at which the application invokes `MPI_send()` and its eventual call to write data on a socket. The "Decode" component is the time span between the `recv()` call returning and the point at which the data is in a form usable by the application. This delineation allows us to focus on the encode/decode costs involved in binary data exchange. That these costs are significant is clear from the figure, where they typically represent 66% of the total cost of the exchange.

Figure 34 shows the cost breakdown for messages of a selection of sizes (using examples drawn from a mechanical engineering application), but in practice, message times depend upon many variables. Some of these variables, such as basic operating system characteristics that affect raw end-to-end TCP/IP performance, are beyond the control of the application or the communication middleware. Different encoding strategies in use by the communication middleware may change the number of raw bytes transmitted over the network; much of the time those differences are negligible, but where they are not, they can have a significant impact upon the relative costs of a message exchange.

The next subsections will examine the relative costs of PBIO, MPI, CORBA, and an XML-based system in exchanging the same sets of messages. We first compare the sending- and receiving-side communication costs for all of the alternatives evaluated. Then we discuss our use of dynamic-code-generation to reduce unmarshalling costs and evaluate the resulting performance improvements. We

[1]The Sun machine is an Ultra 30 with a 247 MHz cpu running Solaris 7. The x86 machine is a 450 MHz Pentium II, also running Solaris 7.

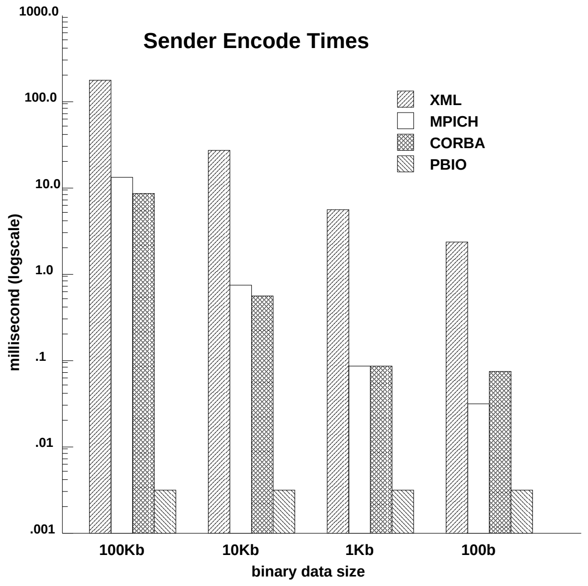


Figure 35: Sending-Side encoding times.

conclude the section with an analysis of the performance effects of flexibility in PBIO.

6.1.2 Sending-Side Costs

We start by comparing the sending-side data encoding times on the Sun Ultra-30 Sparc for an XML-based implementation[2], MPICH, CORBA, and PBIO. Figure 35 shows the different encoding times in milliseconds. An examination of this plot yields two conclusions:

- **XML wire formats are inappropriate.** The figure shows dramatic differences in the amount of encoding necessary for the transmission of data (which is assumed to exist in binary format prior to transmission). The XML costs represent the processing necessary to convert the data from binary to string form and to copy the element begin/end blocks into the output string. The result is an encoding time that is at least an order of magnitude higher than other systems. Just one end of the encoding time for XML is several times as expensive as the entire MPI round-trip message exchange (as shown in Figure 34). Further, the message represented in the ASCII-based XML format is significantly larger than in the binary-based representations, translating into a significantly larger network transmission time, increased latency and substantially decrease in possible message rates.
- **The NDR-approach significantly improves sending-side costs.** As is mentioned in Section 5.1, we transmit data in the native format of the sender. As a result, no copies or data conversions are necessary to prepare simple structure data for transmission. So, while MPICH's costs to prepare for transmission on the Sparc vary from 34μsec for the 100 byte

[2]A variety of implementations of XML, including both XML generators and parsers, are available. We have used the fastest known to us at this time, Expat [23].

record up to 13 msec for the 100Kb record and CORBA costs are comparable, PBIO's costs are a flat 3 μsec.

6.1.3 Receiving-Side Costs

In analyzing the receiving-side cost of communication, we identify several components that contribute to it, including: (1) byte-order conversion, (2) data movement costs, and (3) control costs.

Byte order conversion costs are to some extent unavoidable. If the communicating machines use different byte orders, the translation must be performed somewhere, regardless of the capabilities of the communications package.

Data movement costs are harder to quantify. If byte-swapping is necessary, data movement can be performed as part of the process, without incurring significant additional costs. Otherwise, clever design of the communication middleware can often avoid copying data. However, packages that specify an intermediate wire format for transmitted data have a harder time being clever in this area. One of the basic difficulties is that the native format for mixed-datatype structures on most architectures has gaps, unused areas between fields, inserted by the compiler to satisfy data alignment requirements. To avoid making assumptions about the alignment requirements of the machines they run on, most packages use wire formats that are fully packed and have no gaps. This mismatch *forces* a data copy operation in situations where a clever communications system might otherwise have avoided it.

Control costs represent the overhead of iterating through the fields in the record and deciding what to do next. Packages that require the application to marshall and unmarshall their own data have the advantage that this process occurs in specially-written compiler-optimized code, minimizing control costs. Systems such as CORBA, where the marshalling code can generally be pre-generated and compiled based upon static stubs, have a similar advantage. However, to keep that code simple and portable, such systems uniformly rely on communicating in a pre-defined wire format, therefore incurring the data movement costs described in the previous paragraph.

Packages that marshall data themselves typically use an alternative approach to control, where the marshalling process is controlled by what amounts to a table-driven interpreter. This interpreter marshalls or unmarshalls application defined data, making data movement and conversion decisions based upon a description of the structure provided by the application and its knowledge of the format of the incoming record. This approach to data conversion gives the package significant flexibility in reacting to changes in the incoming data and was our initial choice when implementing NDR.

XML necessarily takes a different approach to receiving-side decoding. Because the "wire format" is an annotated continuous string, XML is parsed at the receiving end. The Expat XML parser [23] calls handler routines for every data element in the XML stream. That handler can interpret the element name, convert the data value from a string to the appropriate binary type and store it in the appropriate place. This flexibility makes XML extremely robust to changes in the incoming record. The parser we have employed is quite fast, but XML still pays a relatively heavy penalty for requiring string-to-binary conversion on the receiving side.

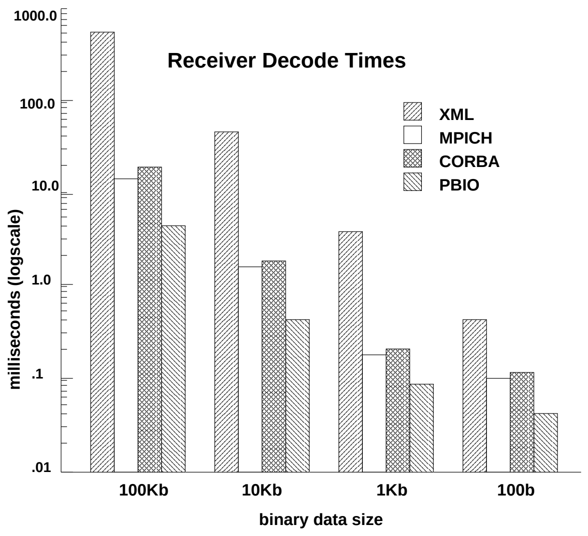


Figure 36: Receiving-Side decode times.

6.1.3.1 Comparing Receiving-Side Costs

Figure 36 shows a comparison of receiver-side processing costs on the Sparc for interpreted converters used by XML, MPICH (via the `MPI_Unpack()` call), CORBA, and PBIO. XML receiver conversions are clearly expensive, typically between one and two orders of decimal magnitude more costly than our NDR-based converter for this heterogeneous exchange. On an exchange between homogeneous architectures, PBIO, CORBA and MPI would have substantially lower costs, while XML's costs would remain unchanged. Our NDR-based converter is highly optimized and performs considerably better than MPI, in part because MPICH uses a separate buffer for the unpacked message rather than reusing the receive buffer (as we do). However, NDR's receiving-side conversion costs still contribute roughly 20% of the cost of an end-to-end message exchange. While a portion of this conversion overhead must be attributed to the raw number of operations involved in performing the data conversion, we believe that a significant fraction of this overhead is due to what is, essentially, an interpreter-based approach.

6.1.3.2 Optimizing Receiving-Side Costs in PBIO

Our decision to transmit data in the sender's native format results in the wire format being unknown to the receiver until run-time. PBIO's implementation of NDR makes use of dynamic code generation to create a customized conversion subroutine for every incoming record type. These routines are generated by the receiver on the fly, as soon as the wire format is known. PBIO dynamic code generation is performed using a package for dynamic code generation developed in Georgia Tech [39], that provides a virtual RISC instruction set. Early versions of PBIO used the MIT Vcode system [41].

The instruction set provided by DRISC is relatively generic, and most instruction generation calls produce only one or two native machine instructions. Native machine instructions are generated

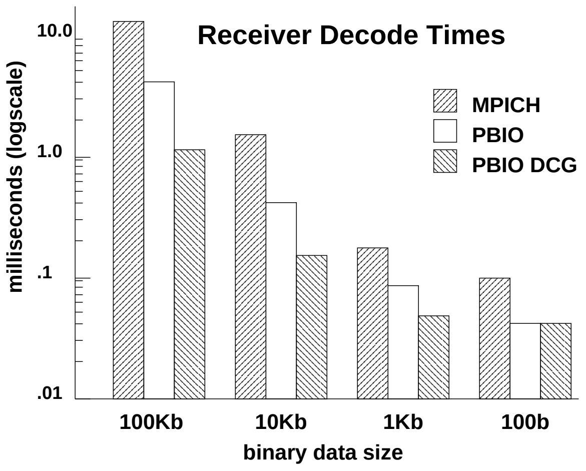


Figure 37: Receiving-Side costs for interpreted conversions in MPI and PBIO and DCG conversions in PBIO.

directly into a memory buffer and can be executed without reference to an external compiler or linker.

Employing DCG for conversions means that PBIO must bear the cost of generating the code as well as executing it. Because the format information in PBIO is transmitted only once on each connection and typically used to process multiple messages, conversion routine generation is not normally a significant overhead. The proportional overhead encountered varies significantly depending upon the internal structure of the record. To understand this variability, consider the conversion of a record that contains large internal arrays. The conversion code for this case will consists of a few `for` loops that process large amounts of data. In comparison, a record of similar size consisting solely of independent fields of atomic data types requires custom code for each field.

The benefits derived from the use of DCG are apparent from the execution times for these dynamically generated conversion routines, which are shown in Figure 37 (we have chosen to leave the XML conversion times off of this figure to keep the scale to a manageable size). From these measurements, it is clear that the dynamically generated conversion routine operates significantly faster than its interpreted version. This improvement removes conversion as a major cost in communication, bringing it down to near the level of a copy operation, and it is the key to PBIO's ability to efficiently perform many of its functions.

The cost savings achieved by PBIO and described in this section are directly reflected in the time required for an end-to-end message exchange. Figure 38 shows a comparison of PBIO and MPICH message exchange times for mixed-field structures of various sizes. The performance differences are substantial, particularly for large message sizes where PBIO can accomplish a round-trip in 45% of the time required by MPICH. The performance gains are due to: (1) virtually eliminating the sending-side encoding cost by transmitting in the sender's native format, and (2) using dynamic code generation to customize a conversion routine on the receiving side.

Once again, Figure 38 does not include XML times to keep the figure to a reasonable scale.

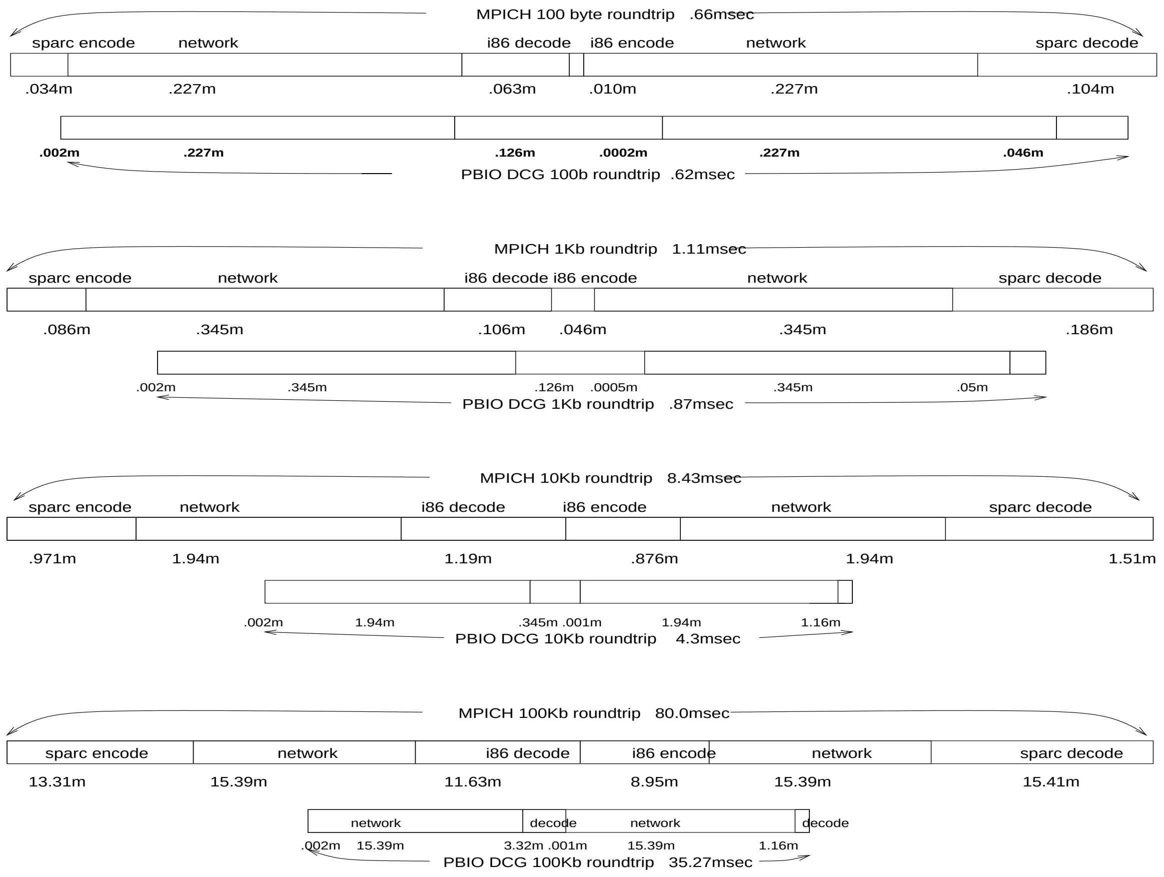


Figure 38: Cost comparison for PBIO and MPICH message exchange.

Instead, Table 8 summarizes the relative costs of the round-trip exchange with XML, MPICH, CORBA, and PBIO.

6.1.4 High Performance and Application Evolution

The principal difference between PBIO and most other messaging middleware is that PBIO messages carry format meta-information, somewhat like an XML-style description of the message content. This meta-information can be a useful tool in building and deploying enterprise-level distributed systems and is key in enabling the Active Streams approach. This meta-information is what allows generic components to operate upon data about which they have no *a priori* knowledge, and makes possible the evolution and extension of the basic message formats used by an application without requiring simultaneous upgrades to all application components. In other words, PBIO offers limited support for *reflection* and *type extension*, features commonly associated with object systems.

PBIO supports reflection by allowing message formats to be inspected before the message is received. Its support of type extension derives from doing field matching between incoming and expected records by name. Because of this, new fields can be added to messages without disruption

Original	Round-trip time			
Data Size	XML	MPICH	CORBA	NDR
100Kb	1200ms	80ms	67.47ms	35ms
10Kb	149ms	8.4ms	8.83ms	4.3ms
1Kb	24ms	1.1ms	1.01ms	0.87ms
100b	9ms	.66ms	0.6ms	0.62ms

Table 8: Cost comparison for round-trip message exchange for XML, MPICH, CORBA, and NDR.

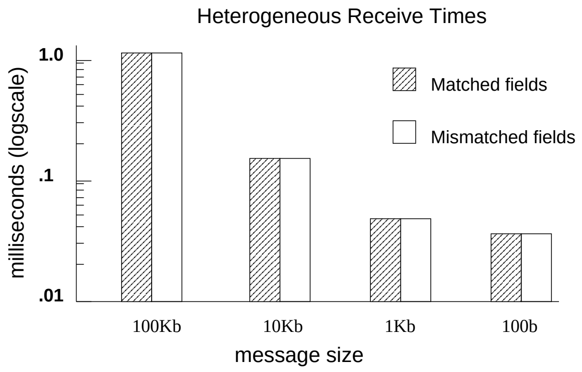


Figure 39: Receiving-Side decoding costs with and without an unexpected field: Heterogeneous case.

since application components that don't expect the new fields will simply ignore them.

Most systems that support reflection and type extension in messaging, such as those that use XML as a wire format or marshall objects as messages, suffer prohibitively poor performance compared to systems such as MPI which have no such support. Therefore, it is interesting to examine the effect of exploiting these features upon PBIO performance. In particular, we evaluated the performance effects of type extension by introducing an unexpected field into the incoming message and measuring the change in receiving-side processing.

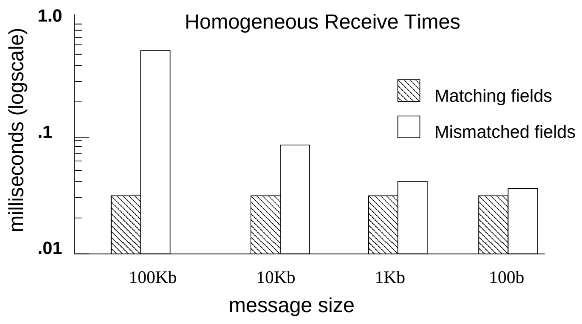


Figure 40: Receiving-Side decoding costs with and without an unexpected field: Homogeneous case.

Figures 39 and 40 present receiving-side processing costs for an exchange of data with an unexpected field. These figures show values measured on the Sparc side of heterogeneous and homogeneous exchanges, respectively, using PBIO's dynamic code generation facilities to create conversion routines. Figure 39 clearly indicates that the extra field has no effect upon the receiving-side performance. Transmitting would have added slightly to the network transmission time, but otherwise the support of type extension adds no cost to this exchange.

Figure 40 shows the effect of the presence of an unexpected field in the homogeneous case. Here, the overhead is potentially significant because a homogeneous exchange would normally not impose any conversion cost in PBIO. The presence of the unexpected field creates a layout mismatch between the wire and native record formats that requires the relocation of fields by the conversion routine. As the figure shows, the resulting overhead is no negligible, but it is never as high as in the heterogeneous case. For smaller record sizes, most of the cost of receiving data is actually caused by the overhead of the kernel `select()` call. The difference between the overheads for matching and extra field cases is roughly comparable to the cost of `memcpy()` operation for the same amount of data.

As noted earlier in Section 6.1.3, XML is extremely robust with respect to changes in the format of the incoming record. It transparently handles precisely the same types of change in the incoming record as PBIO. Thus, new fields can be added or existing fields reordered without worry that the changes will invalidate existing receivers. Unlike PBIO, XML's behavior does not change substantially when such mismatches are present. Instead, XML's receiving-side decoding costs remain essentially the same, as shown in Figure 36. However, these costs are several orders of magnitude higher than those in PBIO.

It is worth noticing that the PBIO results shown in Figures 39 and 40 are actually based upon a worst-case assumption, where an unexpected field appears before all expected ones in the record, causing field offset mismatches in all expected fields. In general, the overhead imposed by a mismatch varies proportionally with the extent of this. An evolving application might exploit this feature of PBIO by adding additional fields at the end of existing record formats. This would minimize the overhead caused to application components which have not been updated.

6.2 ECho

Active Streams relies on ECho, our publish/subscribe communication infrastructure, for data and control transport and as the basis for its implicit invocation mode for component integration. ECho is a core element of our Active Streams Framework as most of the framework's components make use of it. ASN communication and control, resource monitoring reports in ARMS, proactivity in PDS, and code distribution in SRS are all built around ECho channels. This section summarizes the results of our evaluation of ECho.

6.2.1 Comparing ECho Performance

Figures 41 and 42 represent the basic performance characteristics of a variety of communication infrastructures that might be used for event-based communication in high performance applications.

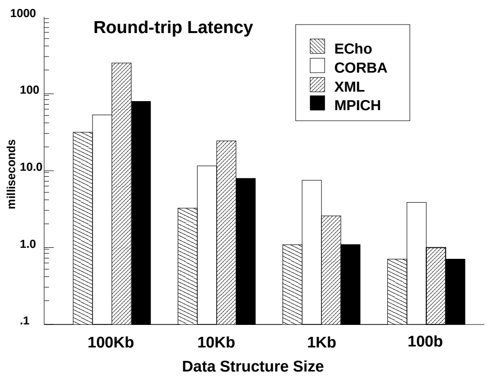


Figure 41: A comparison of latency in basic data exchange in event infrastructures

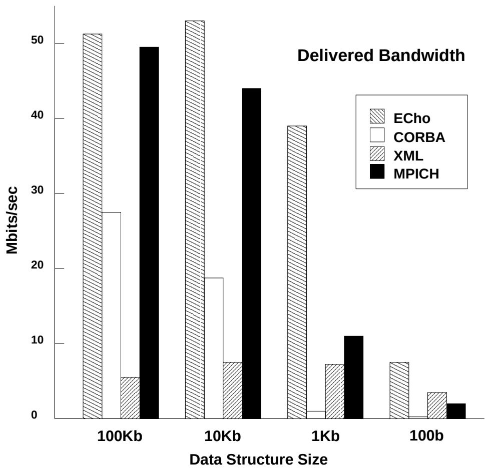


Figure 42: A comparison of delivered bandwidth in event infrastructures

	ECho	CORBA (ORBacus)	MPICH	XML
Total Round-Trip	30.6	53.0	80.1	1249
Sparc Encode	0.037	0.74	13.3	176
Network Transfer	13.9	13.9	13.9	182
x86 Decode	1.6	1.6	11.6	276
x86 Encode	0.015	0.64	8.9	124
Network Transfer	13.9	13.9	13.9	182
Sparc Decode	1.2	0.58	15.4	486

Table 9: Cost breakdown for heterogeneous 100Kb event exchange (times in msecs).

The values are of basic event latency and bandwidth in an environment consisting of a x86-based PC and a Sun Sparc connected by 100 Mbps Ethernet.[3] Note the use of a logarithmic vertical scale in Figure 41. This is useful in presenting latencies for a range of message sizes on the same graph, but it tends to minimize the substantial performance advantage that ECho demonstrates as compared to the other infrastructures.

The infrastructures compared don't all share the same characteristics and features, a fact that accounts for some of their performance differences. ECho's strength is that it provides the important features of these systems while maintaining the performance achieved by traditional high-performance systems like MPICH.

In particular, ECho provides for event type discovery and dynamic type extension in a manner similar to that of XML, or that which can be achieved by serializing objects as events (as in Java RMI). CORBA is also gaining acceptance as distributed systems middleware and its Event Services provide similar features.

6.2.2 Breakdown of Costs

Table 9 shows a breakdown of costs involved in the roundtrip event latency measures of Figure 41. We present round-trip times because they naturally show all the combinations of send/recv on two different architectures in a heterogeneous system. The time components labeled "Encode" represent the span of time between an application submitting data for transmission and the point at which the infrastructure invokes the underlying network `send()` operation. The "Network Transfer" times are the one-way times to transmit the encoded data from sending to receiving machines. The "Decode" times are the time between the end of the `recv()` operation and the point at which the data is presented to the application in a usable form. This breakdown is useful for understanding the different costs of the communication and in particular, how they might change with different networks or processors.

We have excluded Java RMI from the breakdown in Table 9 because it performs its network `send()` operations incrementally during the marshalling process. This allows Java to pipeline the encode and network send operations making a simple cost breakdown impossible. However, as a result of this design decision, Java RMI requires tens of thousands of kernel calls to send a 100Kb

[3]The Sun machine is an Ultra 30 with a 247 MHz cpu running Solaris 7. The x86 machine is a 450 MHz Pentium II, also running Solaris 7.

	ORBacus		ECho	
Data size	Send side overhead	Receive side overhead	Send side overhead	Receive side overhead
100Kb	0.74	0.40	0.037	0.034
10Kb	0.22	0.046	0.037	0.034
1Kb	0.19	0.016	0.037	0.034
100b	0.17	0.010	0.037	0.034

Table 10: Cost breakdown for homogeneous event exchange (times in msecs).

message, seriously impacting performance.

Additionally, while the round-trip times listed in Table 9 are near the sum of the encode/xmit/decode times, this is not true for the CORBA numbers. This is because implementations of the CORBA typed event channel service typically rely on CORBA's dynamic invocation interface to operate. In the ORBs we have examined, DII does not function for intra-address-space invocations. The result of this is that the CORBA typed event channel must reside in a different address space than either the event source or event sink, adding an extra hop to every event delivery. This could be considered an implementation artifact that might be handled differently in future CORBA event implementations.

6.3 Effects of Stream Specialization

In this section we present experimental results that demonstrate the benefits of Active Streams for end applications, in particular, the end effects of stream specialization through streamlets.

All experiments use a cluster of Sun Sparc Ultra 30's (247 MHz CPU and 128 MB) running Solaris 7 and connected by switched 100Mbps Ethernet. Sample data streams are derived from the messaging requirements of an actual application similar to the one introduced in Section 3.1. The message type of size 100KB is a non-homogeneous structure taken from a mechanical engineering simulation of the effects of micro-structural properties on solid-body behavior. The smaller message types (10KB, 1KB, and 100B) are representative subsets of the mixed-type message.

6.3.1 Basic Overheads

The effectiveness of our approach would depend on the cost/benefit tradeoffs of streamlet execution and their effects on stream characteristics. Understanding the benefits of stream specialization is straightforward if we abstract issues such as source, sink, and network contention. For instance, if one half of the data streamed destined to a sink is unwanted, stream specialization would allow it to receive twice the amount of *useful* information with the same bandwidth.

With gains in reduced network usage, however, come the additional costs of streamlet instantiation and execution. Consider the streamlet presented in Figure 43 that filters out records outside a given range. To instantiate this streamlet, our dynamic code generation facility requires 4 milliseconds on a Sun Sparc Ultra 30 and it executes in about 165 nanoseconds. Given our communication

```
{
  if ((input.range > LOWEND) || (input.range < HIGHEND)) {
     return 1; /* submit record into output stream */
  }
  return 0; /* do not submit record */
}
```

Figure 43: A streamlet that passes only records with 'range' in a specified interval.

```
{
  int i;
  int j;
  double sum = 0.0;

  for (i = 0; i < MAXI; i = i + 1) {
    for (j = 0; j < MAXJ; j = j + 1) {
      sum = sum = input.array[i][j];
    }
  }
  output.avg_array = sum / (MAXI * MAXJ);
  return 1; /* submit record */
}
```

Figure 44: A streamlet that computes the average of an input array and passes it to its output.

infrastructure's 7.5Mbps transfer rate for small (100 byte) messages that implies that the stream's source is spending about 107 microseconds to send each message. In other words, in this case, it is 'cheaper' to execute the filter than it is to send an unneeded message. Consequently, contrary to intuition, the load at a stream source may be decreased, not increased, with the execution of a streamlet that reduces the bandwidth requirement of a given stream. In comparison, the same filter function implemented in Java (JDK 1.2.2), the most likely alternative representation for streamlets, requires 1.8 microseconds for execution with Just-In-Time compilation enabled (or 3.7 microseconds otherwise).

The second streamlet example (Figure 44), taken from a scientific visualization application, computes the average of an input array; a typical task when downsampling a data stream for visual rendering on variable-quality displays. This function is somewhat more complex than the first example. It requires 5.5 msecs for code generation and 1.28 msecs to execute. Furthermore, unlike the first example, this streamlet does not reject messages, but transforms them reducing their sizes. As a result, resource savings are due to reductions in required network processing at the server and in the reduction of required network bandwidth, important considerations for remote data visualization [70]. Again for comparison, the same function implemented in Java requires 13.33 msecs for execution with Just-In-Time compilation enabled, (75.43 msecs otherwise). This indicates that Java-based specialization may have such high run-times costs that server extension would no longer be a win-win situation, *i.e.*, the increase in server computation cost might not be recouped in reduced network costs.

6.3.2 Overhead of Streamlet Execution

As evident from the previous section, the overhead introduced by our approach much depends on the functionality of the streamlet(s) utilized. In order to measure basic overheads, we have implemented

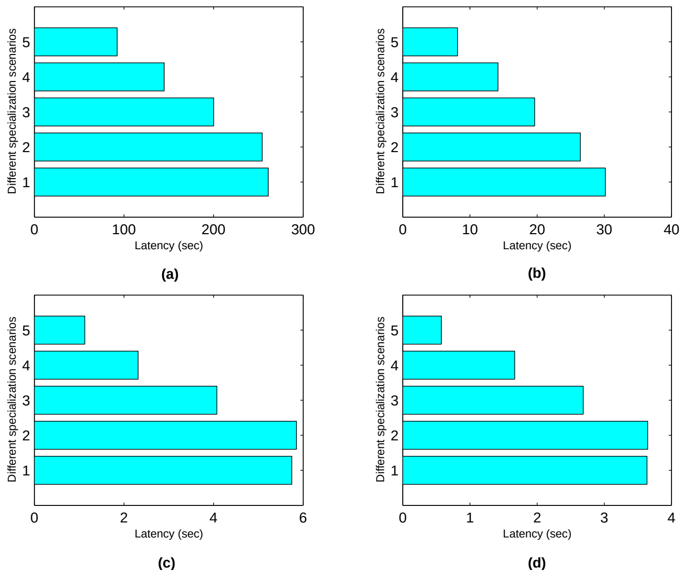


Figure 45: End-to-end latency for 20,000 messages with different degrees of specialization at the source.

an *identity* streamlet that leaves the stream to which it is attached un-modified. Figures 45 and 47, in their specialization scenario #2, show this overhead in terms of end-to-end latency and source CPU utilization. From these figures, by comparing the scenario without stream specialization to that one where specialization is done with the *identity* streamlet, it is clear that the basic streamlet mechanism we have implemented does not add significant overheads to any pipeline-structured, data streaming applications built with Active Streams.

6.3.3 Benefits of Stream Specialization

One of the intended uses of streamlet attachment is stream specialization. The primary benefit of stream specialization is the possible reduction/elimination of wasteful message traffic (moving streamlets 'up' the stream) and the subsequent savings in resource consumption. In order to gauge the potential benefits of stream specialization, we have measured the effects of different types of streamlets: a filter-type streamlet that discards a given portion of the stream content, and a transformation streamlet that converts data units from its incoming stream into a different, smaller type for its outgoing one (functionally similar to the examples in Figures 43 and 44 or a downsampling filter for a Hydrology simulation reported in [90]).

We next apply these streamlets to our sample data streams and measure the effects on latency and CPU utilization of moving streamlets 'up' the stream and toward the source. Figure 45 shows the benefits in end-to-end latency as perceived by a client for different message sizes and degrees of specialization at the source. Each sub-figure corresponds to a different message size: (a) 100K, (b) 10K, (c) 1K, and (d) 100B. The y-axis represents the different specialization scenarios: (1) without specialization; (2) with an 'Identity' streamlet, which leaves the stream unchanged; and (3), (4), and (5) with a streamlet that "filters out" 30, 60, and 90%, respectively, of the stream's contents.

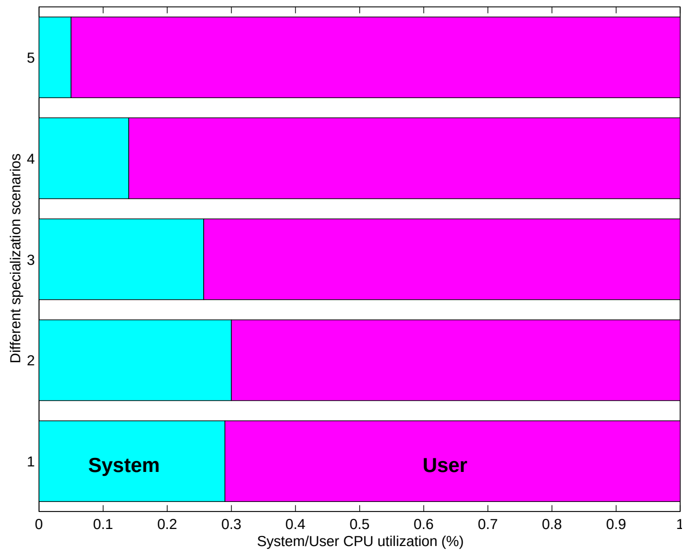


Figure 46: Percentage of Source CPU utilization shared between User and System.

Compare the end-to-end latency of the first scenario (no specialization) to that of the third, fourth, and fifth ones, where 90, 60 and 30% of the stream contents are “filtered out”, improving latency by 3-6X.

Being able to place a specialization streamlet anywhere on the datapath, one would like to place them right at the streams’ sources, the data servers, in order to harness all of the potential savings in network bandwidth. Such savings, however, come with additional costs in the form of generation and execution of streamlets incurred at the servers. Interestingly, and counter to common wisdom, performance may be improved even when ‘giving servers extra work’ by moving streamlets onto them. For a filter-type streamlet, for instance, this is easy to understand, as the additional code executed by the server eliminates unnecessary protocol stack execution. Figure 46 clearly demonstrates this effect. The figure shows the percentage of Source CPU utilization shared between User and System for different specialization scenarios, when transmitting 20,000 10K-messages. As with Figure 45, the y-axis represents the different specialization scenarios at the source: (1) without specialization; (2) with an ’Identity’ streamlet; and (3), (4), and (5) with a streamlet that “filters out” 30, 60, and 90%, respectively, of the stream’s contents. Notice how an increase in the ‘user’ share of server CPU utilization results in a reduction of the ‘system’ share (and network blocking time), as messages are being “filtered out”.

Figure 47 shows the effects on CPU load at the server for our four example data streams. Each sub-figure corresponds to a different message size: (a) 100K, (b) 10K, (c) 1K, and (d) 100B. The y-axis represents the different specialization scenarios: (1) without specialization; (2) with an ’Identity’ streamlet; and (3), (4), and (5) with a streamlet that “filters out” 30, 60, and 90%, respectively, of the stream’s contents. The percentage of CPU utilization is computed with respect to the execution time of the first scenario (i.e. no-specialization), since the actual share of CPU utilization remains constant or increases as the total time needed to transmit a fixed number of messages decreases.

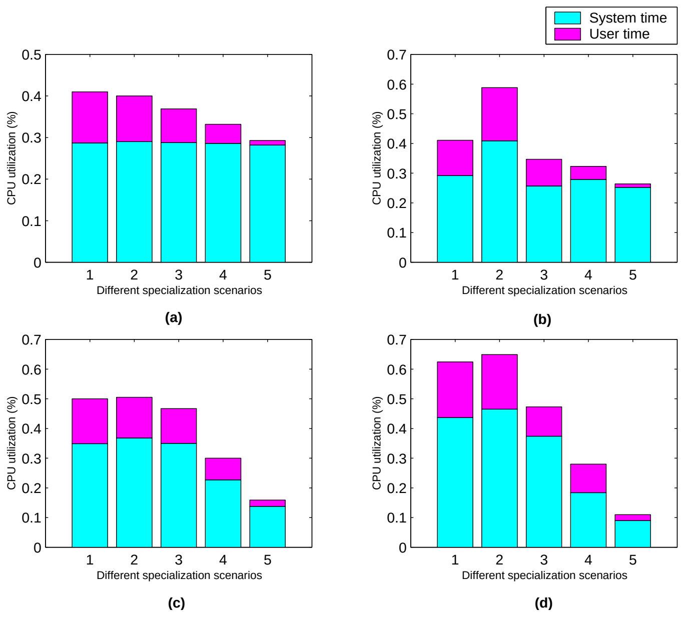


Figure 47: Percentage of Source CPU utilization consumed to transmit 20,000 messages with different degrees of specialization at the source.

6.3.4 Active Streams Nodes Across the Datapath

Active Streams Nodes (ASN) are instantiated across the datapath to host the application- or service-specific streamlets attached to data streams. Other projects have already demonstrated the potential benefits of placing computation over the datapath, including proxy-based approaches [152, 48], and protocol-oriented approaches [109, 126, 43], and more general infrastructures such as Conductor [150] and Active Networks [130].

A clear need for ASN across the datapath appears when trying to customize data streams via "upstream" migration of streamlets. The benefits of "upstream" migration start to decrease when these actions unduly increase the computational load at the stream (intermediate) source. More computationally expensive streamlets and a larger number of streamlets run by any give host would eventually slow down its execution, thereby decreasing the performance experienced by its sinks. In trying to understand this problem, we configured an experiment in which an increasing percentage of sinks (clients) customize their streams at the source. Figure 48 contrasts server CPU utilization with the end-to-end latency experienced by the clients when transmitting 20,000 1K-messages to 30 clients, with a varying number of them specializing the stream at the source. The specialization transforms the stream from 1K- to 100B-messages. The y-axis represents the percentage of clients applying this transformation at the stream's source. The dashed line shows the end-to-end latency perceived by non-specializing clients, while the dotted line represents the one experienced by specializing clients. α signals the point from which there are no additional benefits to be gained from specialization, and β indicates the point of diminishing returns. The percentage of CPU utilization is computed with respect to the execution time of the first scenario (i.e. no-specialization) since the actual share of CPU utilization remains constant or increases as the total time needed to transmit a fixed number

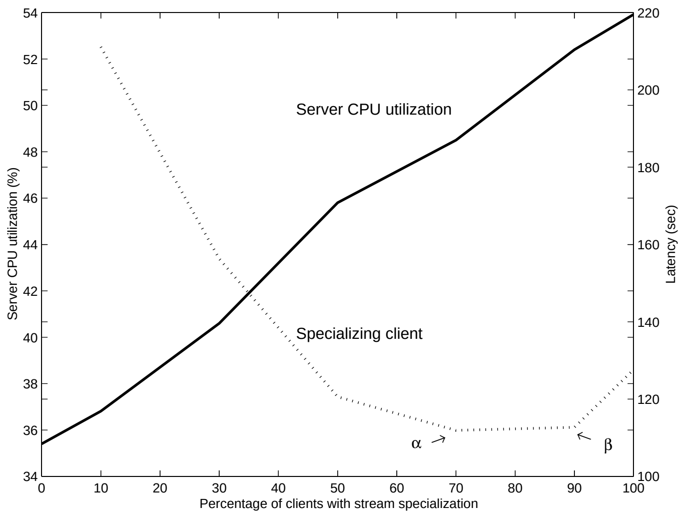


Figure 48: Percentage of Source CPU utilization contrasted with end-to-end latency.

of messages decreases.

ASNs ‘down’ the path could be used to host streamlets once the computational devices ‘up’ the path start getting overloaded. Naturally the additional hops, although still allowing us to collect some of the benefits of specialization, such as reductions in CPU utilization, it may translate in an increasing end-to-end latency. Note that this will not hold if the available bandwidth through that last hop were significantly restricted, as is the case in mobile and sensor rich environments, and in many research collaboration scenarios.

6.4 Costs and Benefits of a Proactive Interface to Directory Services

The Proactive Directory Service (PDS) is an efficient and scalable information repository and a core component of our Active Streams Framework. The main innovative aspect of PDS is the inclusion of a customizable proactive access mode as part of its client interface. Through this interface, PDS clients can learn of objects (or types of objects) being inserted/removed from particular contexts (such as addition or removal of services or devices) and/or about changes to pre-existent objects.

This section presents evaluation results that confirm the expected performance advantages of PDS' proactive approach. These experiments [4] contrast the scalability of PDS (featuring pull- and push-based interfaces) with that of off-the-shelf implementations of DNS (BIND DNS 8.2.2-7 [24]) and LDAP (OpenLDAP 1.2.11 [47]) (both of which use a strictly pull-based approach). BIND DNS is used because it is a highly optimized directory system; OpenLDAP is an LDAP implementation that provides functionality and extensibility similar to PDS and form the basis for some metacomputing directory services [44]. Before presenting the results of our experimental comparison, we compare

[4]Each host used in our test has 4 Intel Pentium Pro processors with 512MB of RAM, runs RedHat Linux 6.2, and is connected with the other hosts by a 100Mbps Fast Ethernet.

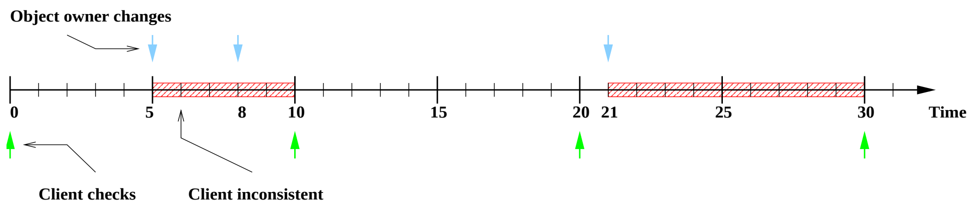


Figure 49: Timeline of client's periods of inconsistency.

both approaches analytically.

6.4.1 Analytical Comparison

The loads imposed by pull-based and push-based directory interfaces can be compared through a simple model in which a directory server manages a number of entities on behalf of their owners. We assume that different clients want to keep track of different subsets of those entities.

Since clients cache copies of their entities of interest and owners update their entities' attributes, inconsistencies are expected to appear. Assuming a fixed period of time over which the owner of an entity in the directory makes a series of updates, we wish to calculate the number of client requests that would be necessary to maintain different degrees of *consistency* under pure pull- and push-based models. Note that consistency problems exist in any system that uses some form of cache to speed up access. While the consistency problem in directory services is similar to the Web cache consistency problem [17], the consistency requirements of applications in our target environments are significantly more stringent than those of the typical Web user.

Let U be a set of updates, u_i, occurring over a period T at some random instants of time, and let p be the frequency at which a client pulls a non-proactive server. The total time over which the client will have an inconsistent copy can be computed as:

$$I = \sum_{i=1}^{U} d_i = \sum_{i=1}^{U} (p - u_i \bmod p)$$

i.e. the client's degree of consistency, or the percentage of time at which the client will have a consistent copy, is:

$$(T - I)/T * 100\%$$

Note that the degree of consistency depends on the frequency at which the client requests an update from the server and the rate at which the owner of the entity updates its attributes.

For example, if over a 30 hour time period the entity owner changes the object value at hours 5, 8, and 21 while the client polls the server for this object value every 10 hours, the client will have the wrong value $(5 + 9)$ hours over those 30, which corresponds to an accuracy of $(30 - 14)/30 = 0.53$, or 53%. This can be graphically represented as in Figure 49.

As the degree of consistency is determined by the client frequency of pulls and the entity-owner rate of updates, so is the number of messages exchanged between the client and the server (and the

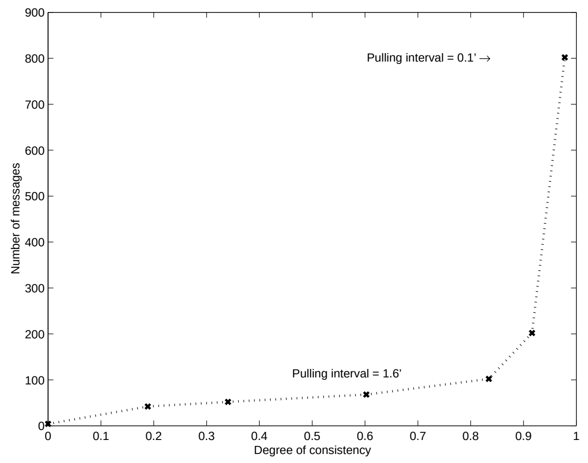


Figure 50: Number of messages required to achieve a given degree of consistency.

```
Length of period: 40 minutes
Number of updates by owner: 15
Times of updates:
 Update 1      2      3      4      5
 Time   2.673  3.678  15.336 15.340 16.64

 Update 6      7      8      9      10
 Time   20.777 21.188 25.265 26.061 26.1568

 Update 11     12     13     14     15
 Time   26.846 28.906 30.257 35.388 37.218

Number of resolver's pull intervals: 7
Pull intervals:
 Period        1   2   3   4   5   6   7
 Pull interval 0.1 0.4 0.8 1.2 1.6 2.0 40.0
```

Figure 51: Owner's updates times and client's pull intervals.

associated load on resources) required to maintain a given degree of consistency. Figure 50 shows the number of messages between the client and the directory server required to maintain a given level of consistency for the updates and pull frequency quoted in Figure 51.

In contrast, a proactive interface allows applications to obtain a perfect degree of consistency at a reasonable load on resources, by making the degree of consistency of the client data independent of the frequency with which it is updated. For this example, over 800 messages are required to obtain a perfect degree of consistency without proactivity, while with a proactive interface only 15 are needed.

6.4.2 Performance Comparisons

We now present evaluation results showing the expected performance advantages of PDS' proactive approach. We contrast the scalability of PDS (featuring pull- and push-based interfaces) with that of off-the-shelf implementations of DNS and LDAP.

As already mentioned, a potential disadvantage of a proactive approach is the loss of control from the client's perspective since, after having registered its interest on changes to an object, it is then at the "mercy" of the object's owner. PDS clients can regain control by dynamically customizing these notifications through filter functions instantiated at the server (or the object's owner) and by tuning these filters' functionalities via remote updates of some of their parameters. We also present initial experimental results that show that such customization, depending on the level of filtering, need not translate into additional server processing load but can improve performance.

Intuitively, proactivity provides scalability and high performance by reducing the amount of work done by clients (and correspondingly by servers) in order to become aware of updates. The following metrics capture these facts:

- **Degree of consistency:** the percentage of time that a client has the correct value of a directory entry. While pull-based clients must poll the server with increasing frequency to increase their degree of accuracy, clients of proactive servers are always 100% accurate (because they automatically receive updates when changes are made at the server).
- **Number of client requests required:** the number of client requests that is required for a client to maintain a particular degree of consistency.
- **CPU load on client:** the processing required by the client to send a certain number of requests.
- **CPU load on server:** the processing required at the server to respond to requests by the client.

Proactivity reduces the load on the server by significantly decreasing the number of client requests for updates. Client load is reduced because the server (or the object's owner) is responsible for notifying the client when changes occur to the object. The number of messages in the system is reduced by eliminating client polling for updates, resulting in an optimal message-per-update. These intuitive statements are validated by experimentation described next.

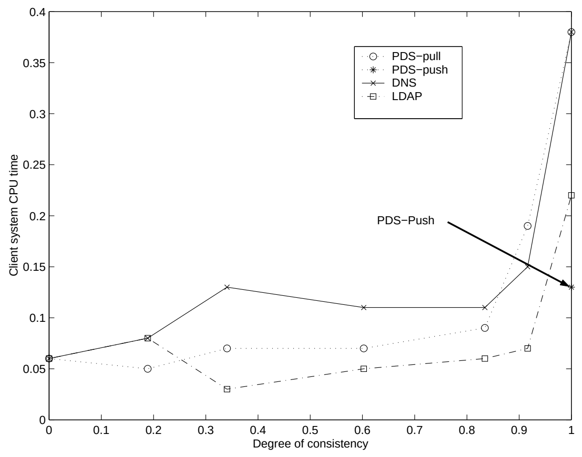


Figure 52: Client load (as measured by system CPU time) needed to reach a given degree of consistency.

We insert a number of updates occurring at randomly generated intervals over a fixed period of time (see Figure 51). Since the actual loads on resources incurred in a particular situation depend on such a wide variety of variables that we cannot characterize the entire performance space, we decided to examine an illustrative example.

Given the sequence of owners' updates quoted in Figure 51 and different degrees of consistency, we measured the load imposed on client and server by using a pull-based, DNS and OpenLDAP, and our push-based interface. For completeness we also show the load imposed by PDS's own pull-based interface. For our experiments all client-side caches were disabled and we ran DNS over TCP (by setting its 'vc' option) for purposes of comparison (OpenLDAP and PDS both use TCP for transport). Figures 52 53 show the client and server loads for all four cases studied.

The benefits of a proactive interface for clients are clear from Figure 52. The figure shows that by using PDS's proactive mode, a perfect degree of consistency can be obtained, at a load on the client that is one-fourth that of DNS and half that of LDAP.

Figure 53 shows the benefits of proactivity to servers. It shows that through proactivity, a perfect degree of consistency can be obtained at a reasonably low server load. The scalability problems of a pull-based interface, as faced by the DNS and LDAP implementations, are clear (our implementation shows a similar, but less severe, trend).

These experiments illustrate, through a simple example, the potential costs and benefits of the proactive approach. In fact, that the load imposed on these servers is not significantly serious. Even at the point of perfect consistency (highest-rate of pull for pull-based clients), this particular experiment imposes a load of only 400 requests/replies over 6 minutes, i.e. about 1.1 messages per second.

In order to avoid the possible drawbacks of a proactive approach (i.e. the client's loss of control), we advocate the dynamic customization of notification channels through filter functions. Initial experiments show that such customization, depending on the level of filtering, does not necessarily

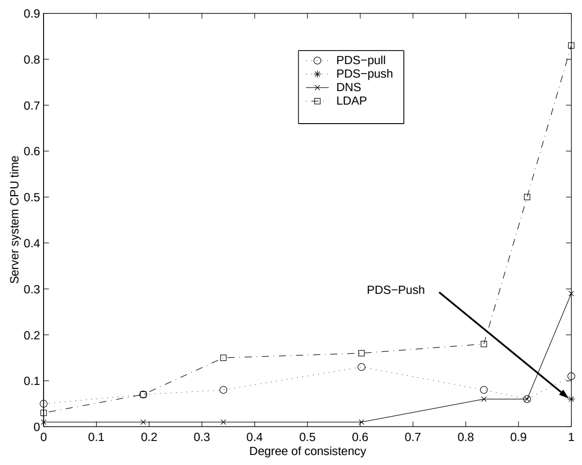


Figure 53: Server load (as measured by system CPU time) needed to reach a given degree of consistency.

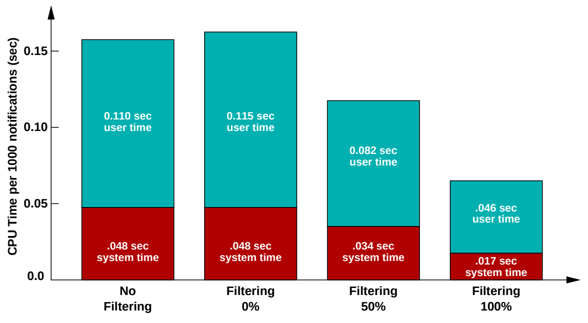


Figure 54: Server CPU utilization under different specialization scenarios.

imply additional processing load but can result in performance improvement. Figure 54 graphically illustrates the costs/benefits of filtering. It shows the CPU utilization of a server sending an update stream to a single client. In this case, the data stream consists of 100 byte notification records generated at a particular rate. The figure shows the server's CPU utilization for four filtering scenarios: (1) no filtering, (2) with filtering but without rejecting any notification, (3) filtering out 50% of the notifications, and (4) filtering out all notifications. Specializing a notification channel obviously adds to the server's computational load in the form of additional overheads when no notifications are rejected. However, the overhead is slight and is quickly recovered if the filter rejects a significant fraction of the notification stream. At higher rejection rates, the use of a filtered stream represents a cost savings to the server, despite the need to execute the filter function for every notification.

The experiments confirm our hypothesis that an inactive model of client-server interaction restricts the scalability of directory services, and they demonstrate the clear benefits of a proactive approach.

6.5 Summary

This dissertation explores Active Streams, a novel middleware approach and its supporting framework for building distributed applications and services for heterogeneous, highly dynamic environments. The reported evaluations quantify the costs and benefits of the approach.

We first present experimental results demonstrating the effectiveness of NDR, and its PBIO implementation, as a general solution for self-describing communication units, also referred to as *wire-formats* below, for heterogeneous distributed systems. The proposed framework relies on the ECho publish/subscribe communication infrastructure [37] for data and control transport and as the basis for its component integration mechanism. Measurements detailing ECho's performance demonstrate that ECho significantly outperforms other systems that provide similar functionality and that it provides throughput and latency comparable to the most efficient middleware communication infrastructures now in common use. In Section 6.3 we present experimental results that demonstrate the costs and benefits of Active Streams for end user applications. We then discuss results that illustrate the basic overheads of our approach. Experiments indicate that the basic streamlet mechanism implemented does not add significant overheads to data streaming, so that the overheads introduced by the use of streamlets primarily depend on the functionality of the specific streamlet(s) utilized by an application. We illustrate the potential benefits of stream specialization and present results indicating that, counter to common wisdom, the benefits of placing streamlets up the stream do not have to come at the cost of additional load to sources. The need for multiple points of adaptation over the datapath, on the basis of performance, was illustrated in Section 6.3. We conclude the chapter with results that demonstrate the basic performance of PDS and the benefits of its proactive approach to providing directory services.

Chapter 7: Related Work

This chapter reviews a wide variety of research efforts that share many of the Active Streams goals and design principles. Related work is examined from different areas including distributed computing, software engineering and networking. We group these related efforts to put our work into perspective. We conclude this chapter with an overall summary of the systems and approaches covered.

7.1 Component-Based Approaches

Various projects have proposed component-based approaches to software development in wide-area distributed computing [13, 65, 15]. Component architectures facilitate the construction of complex applications by allowing the creation of generic reusable components and by easing independent component development. Similar approaches have been proposed by the software engineering community over the last decade [107, 120] and their advantages have been widely recognized in industry, resulting in the development of systems such as Enterprise Java Beans [128], Microsoft's Component Object Model and its distributed extension (DCOM) [36], and the developing specification of the CORBA Component Model (CCM) in OMG's CORBA version 3.0 [96].

Targeted to Grid-based service CCA [15] proposes an approach to building distributed systems that is based on representing services as application-level software components. The WebFlow [65] project aims to provide a Java-based coarse grain packaging model and framework for authoring wide-area distributed applications. Blair et al. [13] propose a reflective-based approach to the design of configurable middleware together with an open and extensible component framework. In contrast with Active Streams, all of these projects follow a coarse-grain object-based approach to composition and have no provision for adaptation to changing environmental conditions or application requirements. It has been argued elsewhere [60] that an object-oriented approach may be ill-suited to wide-area distributed computing as it may complicate application programmability and evolution.

7.2 Publish-subscribe Approaches for Communication and Integration

Event services are an important component of many distributed applications and services. The publish-subscribe paradigm they support is well-suited to the reactive nature of many novel applications, enables the rapid and dynamic integration of legacy software into distributed systems [98], supports software reuse, facilitates software evolution [52, 51], aids in the scalability and fault-tolerance the system and is a good fit for component-based approaches.

Active Streams relies on ECho for communication and component integration. There are several related research efforts concerned with the development of event notification systems and their use for component integration; including IBM's Gryphon [125], Siena [20], Elvin [119], JEDI [28], JECho [153], and the work by Yu et al [151]. ECho is unique in its support of flexible and high performance event-based communication in heterogeneous environments. In addition, ECho enables the dynamic customization of notification channels, on a per-client base, through the instantiation of general filter functions at the channel source.

7.2.1 Wire Formats for Heterogeneous Communication

ECho and, therefore, Active Streams rely on *Native Data Representation* for communication in heterogeneous environments. Traditionally, performance has been the single most important goal of high-performance communication packages such as PVM [129], Nexus [46] and MPI [45]. NDR is a new approach to wire format that has the combined goals of flexibility and high performance.

Most of these packages support message exchanges in which the communicating applications "pack" and "unpack" messages, building and decoding them field by field [129, 46]. By manually building their messages, applications have full control over message contents while ensuring optimized, compiled pack and unpack operations. Relegating these tasks to the communicating applications, however, means that the communicating components must agree on the format of messages.

Other packages, such as MPI, support the creation of user-defined data types for messages and fields and provide some marshalling and unmarshalling support for them. Although this provides some level of flexibility, MPI does not have any mechanisms for run-time discovery of data types for unknown messages, and any variation in message content invalidates communication.

In summary, the operational norm for high-performance communication systems is for all parties to a communication to have an *a priori* agreement on the format of messages exchanged. The consequent need to simultaneously update all system components in order to change message formats is a significant impediment to system integration, deployment and evolution.

Component-based approaches require more flexible communication systems. This need has promoted the use of object-oriented systems and of meta-data representations. Although object technology provides for some amount of plug-and-play interoperability through subclassing and reflection, this typically comes at the price of communication efficiency, and application programmability and evolution [60]. For example, CORBA-based object systems use IIOP [95] as a wire format. IIOP attempts to reduce marshalling overhead by adopting a "reader-makes-right" approach with respect to byte order (the actual byte order used in a message is specified by a header field). This additional flexibility in the wire format allows CORBA to avoid unnecessary byte-swapping in message exchanges between homogeneous systems, but it does not eliminate the need for data copying at both sender and receiver. At issue here is the contiguity of atomic data elements in structured data representations. In IIOP, XDR and other wire formats, atomic data elements are contiguous, without intervening space or padding between elements. In contrast, the native representations of those structures in the actual applications must contain appropriate padding to ensure that the alignment constraints of the architecture are met. On the sending side, the contiguity of the wire format means that data must be copied into a contiguous buffer for transmission. On the receiving side, the contiguity requirement means that data cannot be referenced directly out of the receive buffer, but must be copied to a different location with appropriate alignment for each element. Therefore, the way in which marshalling is abstracted in these systems prevents copies from being eliminated even when analysis might show them unnecessary.

7.3 Active Approaches

The common thread connecting our approach as well as most of the components of its supporting framework is the idea of *activity*. In Computer Science, a system entity is commonly referred to as *active* when some sort of processing has been attached to it, and this processing is to be implicitly invoked upon certain pre-stated conditions.

The idea of "activity" has been widely used in systems over the last ten years. In Active Messages [137], messages are bound to user level processing on the receiving end. This processing is responsible for extracting the message from the network and integrate it into the on-going computation. Active networks [130] extend this idea by attaching processing to the network path [5, 11, 4] or to the messages being forwarded through it [72, 143]. Active Disks [1, 113, 75], on the other hand, attach processing to the I/O streams destined to or originated at disks. Active Services [54, 7] propose the construction of value-added services following an active approach. In the Active Names project [133], wide-area service names have attached processing responsible for locating the service and transporting its response back to the client.

7.4 Adaptation Functionality in the Datapath

Active Streams could be viewed as distributed, adaptive, typed versions of Unix pipes [114]. The original Unix form of interprocess communication, pipes appeared in Unix in 1972 at the suggestion of M.D. McIlroy. Although neither the idea nor the implementation were totally new (the 'communication files' of the Dartmouth Time-Sharing System [31] did nearly the same), they became one of the classical contributions of Unix to the culture of operating systems and command languages. In Active Streams, as in Unix pipes, data streams flow from sources, through a series of modules with standardize input and output, and are ultimately delivered back to their clients. Active Streams extends this concept to distributed environments, allows multiple I/O streams per module (or streamlets), and supports the run-time attachment/detachment of modules for service customization and/or application adaptation.

Other projects have proposed the injection of modules with application functionality into the datapath for adaptation. The introduction of such functionality can be done (only) at end-points or across intermediate nodes in the datapath.

Badrinath et al. [9] present a conceptual framework for adaptive software systems that synthesize the commonality of various of such projects with which the authors have been involved. Our Active

Streams model has much in common with the proposed framework as both advocate dynamic adaptation over the datapath to changing environmental conditions and application requirements. In contrast to Active Stream, their model associates application specific adapters with their equivalent of our Active Streams Nodes (Adaptation Agencies) instead of with the actual data streams. We have opted to associate streamlets with streams as streamlets are location-independent and their mapping onto nodes is determined at run-time in response to variable environmental conditions.

7.4.1 End-Point Adaptation

Rover [73] implements a proxy-based architecture specifically tuned for client-server, mobile applications. The system uses Queued Remote Procedure Calls to overcome periods of dis-connectivity and to better utilize the network link by scheduling transactions intelligently. Rover also makes use of Relocatable Dynamic Objects to offload some resources by, for example, trading upstream compression for network bandwidth savings. Rover has no provision for run-time adaptation to changing environmental conditions.

Odyssey [91] is another application-aware approach to adaptation intended primarily to assist client/server interaction in mobile environments. The Odyssey system consists of a *viceroy*, for resource management; a set of type-specific *wardens* that handle the intercommunication between clients and servers; and applications that negotiate with Odyssey to receive the best level of service availability. Odyssey has no consideration for the dynamic insertion and composition of wardens and provides no support for dynamic composition of adaptation across multiple nodes, making it not flexible enough to cope with the characteristics of our target environments.

The TranSend [48] proxy addresses both network and system heterogeneity by providing an extra level of indirection in the transfer paths between clients and servers. Proxies transform retrieved data, primarily images, to representation that best suit the client connectivity. In the TranSend architecture, clients rendezvous with the system through a front end. The front end contacts the load manager, which deploys transcoders on behalf of the users. Similar to the two previous projects, TranSend provides no support for dynamic composition of adaptation across multiple nodes.

Zenel and Duchamp [152] describe a general design of a proxy-based architecture that includes the notion of “filters” at an intermediate host or proxy server. While the architecture is relatively general, their system does not address issues of multiple coordinated adaptations.

DataCutter [10] is a middleware infrastructure that provides support for processing of large datasets from archival storage systems over wide-area networks. The project’s main focus is on access to archival storage data, including support for indexing and accessing of multidimensional datasets. As with Active Streams, an application processing structure is decomposed into a set of processes, called filters, following a stream-based programming model derived from the research group’s earlier work on active disks [1]. However, the framework is fundamentally proxy-based and does not consider run-time adaptation to variations on environmental conditions.

Proxy-based solutions have demonstrated the potential benefits of using the processing power available on the datapath, as they depart slightly from the basic client/server model by introducing a third entity, the proxy server. New environments provide additional processing units in the datapath, a potentially greater number of idle hosts and a longer, more complex network connecting clients and servers. These characteristics indicate the need for more general multi-point approach to adaptation. Although multiple proxies could be distributed over the datapath, the paradigm provides no assistance in making them cooperate.

7.4.2 Distributed Adaptation

Distributed adaptation can be application-transparent or application-aware and can occur at the network or application levels. Protocol Boosters and Transformer Tunnels support transparent network adaptation at the protocol level through the insertion of functional units in the datapath for the incremental construction of protocols [43], or the creation of tunnels in order to deal with problematic links [126]. Our approach is complementary to these, low-level application-transparent adaptations.

More general than the previous approaches and also complementary to our work, Active Networks [130] provide an infrastructure that allows application code to be attached to individual packets or deployed over the network routers. Active networks are packet-switched networks in which packets can contain code fragments that are executed on the intermediary nodes. The code carried by the packet may extend and modify the network infrastructure. The goal of active network research is to develop mechanism to increase the flexibility and customizability of the network and reduce the difficulty of integrating new technology and standards into a shared network infrastructure. Two commonly distinguished approaches to active networking are: *programmable switches* [5, 11, 4] and *capsules* [143, 94, 72]. The first approach adds functionality to nodes out-of-band from the packets being processed by the node. In the capsule-based approach, capsules contain both code and data as they move through the network and are executed on the nodes they encounter. Although these approaches provide a very general adaptation mechanism, their deployment requires significant changes to the existing network infrastructure.

Conductor [150] provides an application transparent adaptation framework that allows multiple adaptation-modules spread along the datapath between application and services. Conductor proposes the automatic deployment of multiple application-transparent adaptors over the datapath. Although this transparency insures backward compatibility it also limits their flexibility. In contrast to this, Active Services [7] allows client applications to explicitly start one or more services on their behalf that can transform the data they receive from end services.

The goal of adaptive distributed approaches, however, is to provide good end-to-end services, where the end points are located in applications. Without considering the applications' and their users' needs, no adaptive solution at the network level alone can solve the entire problem. Ninja [56], CANS [49] and Active Frames [84] are three projects that, as Active Streams, take an application-level approach to adaptation.

Ninja [56] proposes a data flow model for composing services that is similar to the model underlying Active Streams. However, Ninja is intended to provide robust cluster-based services, and it does not consider dynamic adaptation of data paths or of the paths' components.

CANS [49] is an application-level framework for injecting application-level functionality into the datapath. The CANS infrastructure is closely related to Active Streams as both support the dynamic composition of application functionality over datapaths as well as their run-time adaptation to changing environmental conditions. CANS proposes an interesting extended-type-based composition to automate component selection based on links characteristics. The CANS infrastructure has been implemented on Windows 2000 and uses Java VM as the execution environment at its intermediate hosts. Despite the high-level similarities, both approaches differ in various important aspects including: the Active Streams focus on wide-area, heterogeneous, and highly-dynamic environments; its adoption of event-based techniques for component integration; and its target on high-performance applications.

Focus on interactive heavyweight services such as scientific visualization, Lópes and O'Hallaron [84] propose "Active Frames", an application-level equivalent to the integrated, "capsule" approach to active networks [130]. Active Frames are application-level transfer units that include both application data and a program to be executed over this data. Frames are processed by frame servers installed along the datapath from sources to sinks. Server's functionality can be extended by active frames programs through (Java-based) dynamic class loading or, at initialization time, by the inclusion of application-level libraries. In contrast, Active Streams tries to preserve the semantics of carrying code with every frame while providing the performance achieved when the code is statically loaded in the necessary nodes along the path. If almost each application data frame were to require a new program, a fully dynamic scheme as the one proposed by López and O'Hallaron would provide the necessary flexibility, and the additional overhead would be justified. However, we expect that most frames would require a common enough set of functionality to suggest a demand pull-based approach that would allow the amortization of the deployment cost over a number of frames.

7.5 Resource-Aware Computing

Distributed applications executing in non-dedicated environments must be able to adapt to variation on resource availability. A number of research efforts have proposed resource-aware distributed computing and investigated adaptation models [124] and the right infrastructure support needed by such an approach. Bolliger et al. [14] present a framework-based to developing network-aware applications, concentrating on network monitoring and the mapping between application-level and network-centric quality metrics.

Remos [35] and the Network Weather Service [149] provide the needed infrastructure for resource monitoring. Both approaches includes forecasting services, a facility much needed in distributed dynamic environments. Our work on ARMS is complementary to both, as we propose active interfaces to resource monitoring systems in an attempt to improve application reactivity.

Odyssey [91] seeks to provide a more general approach to resource-aware computing by modifying the interface between applications and the operating system. Their measurement-based approach employs receiver-driven adaptation and concentrates on orchestrating multiple concurrent resource-aware applications on the client. In contrast, our framework uses distributed client- and sender-based adaptation but we have not yet dealt with the complications of multiple concurrent applications.

7.6 Repository Services

ANTS [143] and PAN [94], both capsule-based active network frameworks, provide an automatic demand pull code distribution service to transfer capsules' associated code objects. Closer to our Streamlet Repository Service, Decasper and Plattner [33] propose a design for distributed code caching for active networks that makes use of trusted code servers for distributed active modules.

Software Dock [63] and Tivoli's TME/10 [68] are two architectures that support release management, the process through which software is made available to and obtained by its users. Both include support for configurations, dependencies, installation, and inventory but Tivoli's TME/10 assumes a centralized control with a single site for configuration and releases; while Software Dock is designed as a system of loosely-coupled, cooperating, distributed components, bound together by a wide-area messaging and event system.

7.7 Service Discovery

There are many variants on the common theme of directory services. Classical directory services tend to contain descriptive, attribute-based information on a variety of objects but, generally, without supporting complicated transactions or roll-back schemes as those found in traditional database management systems. However, none support the ability to proactively notify clients of updates as in PDS.

Grapevine [12] and Global Name Service [80] were two pioneering distributed directory services. Other more recent services such as DNS [87] and the X.500 Directory Service [112], provide such services under the assumption of fairly stable mappings between objects' attributes and their values. Such an assumption allows them to make heavy use of caching to improve look-up performance and service scalability. However, they are unsuited for applications in which updates occur with even moderate frequency.

Other more recent research includes the Intentional Naming System (INS) [2] which integrates name resolution and routing. INS allows clients to send messages by describing the attributes of the destination rather than its location on the network. Active Names [133] maps remote service names to chains of mobile programs responsible for locating the remote objects and transporting back the response. Both Active Names and INS concentrate on the problem of efficient and flexible name-to-object resolution, as opposed to PDS' emphasis on providing low-impact client consistency. Each of these objectives is desirable in a wide-area environment, and the concept of proactivity is certainly compatible with either Active Names or INS.

Directory services supporting entries with attributes have made feasible service and resource discovery by processing queries containing a set of desired attributes. Jini [139], the Service Location Protocol (SLP) [134], and the Service Discovery Service [29] all provide attribute-based service discovery for heterogeneous devices. PDS also supports retrieval of entities based on attributes. Jini provides services using Java RMI as a transport mechanism; PDS uses a fast binary transport encoding mechanism that provides superior performance. PDS does not address issues of authentication and secure communication as does SDS. The proactive approach of PDS would be complementary to any of these services.

The use of proactivity in directory services has some precedents in DNS NOTIFY [136], the Ninja's Secure Directory Service (SDS) [29], and Huang and Steenkiste's proposal [67] for an SLP-based wide-area directory service. RFC 1996 proposed a mechanism for prompt notification of zone changes (DNS NOTIFY) through a proactive approach. In SDS, services in the environment announce (broadcast) themselves periodically on a well-known multicast channel. Huang and Steenkiste propose [67] the combined use of pull- and push-based techniques to increase the scalability of service information distribution. In local area networks services use a push-model to announce their services within their domain, while in the wide-area directory agents actively look for such information. All these proposals, however, maintain a passive client interface and, thus, their associated scalability problems.

Our work on PDS has some similarities with research on maintaining cache consistency, although the consistency requirements of distributed applications using PDS may be more stringent than those of the typical Web user. Gwertzman and Seltzer [62] provide a detailed analysis of World Wide Web cache consistency approaches using trace-driven simulation. Cao [17] reports the advantages of proactivity in maintaining cache consistency between Web browsers and servers; consistency is maintained by servers proactively notifying clients (browsers) of changes through invalidation messages. The clients are still responsible for retrieving the updated web page from the server.

Proactivity can be seen as an extension of invalidation. With an invalidation approach, when the client wants to make use of an object after its cached copy has been invalidated, it needs to retrieve a fresh copy. With proactivity, the server pushed the update together with (or instead of) the invalidation. Given the size of most of the objects we expect to keep in the directory, the additional cost of this approach is not significant. Even if the frequency of updates, and so invalidations, were significantly bigger than the frequency of use of the cached object by the users; or if the sizes of the object where to be big enough to tilt the cost-benefit analysis against basic proactivity; dynamically customized proactivity will be a better answer than basic invalidation. Clients can customized proactivity to return only a subset of the object attributes, restrict the update to the modified part or to the difference between the previous and new state, or even retreat to simple invalidation.

7.8 Active Streams and Infosphere

The increasing proliferation of computing into the physical world and the associated, explosive growth in available information have spawn a number of research projects aimed at creating new paradigms for human-computer interaction. Infosphere [97] is one of five projects funded by the DARPA/ITO's Ubiquitous Computing program [30] pursuing this vision. The project, a research collaboration of the Georgia Institute of Technology and the Oregon Graduate Institute, has focused on world-wide, adaptive information flows for composing virtual information spaces for interaction. The Infosphere project addresses a wide variety of issues from quality of service guarantees to specialization and domain specific language techniques [110].

The research presented in this dissertation is part of this broader effort. The Active Streams approach and its supporting framework address a subset of the technical challenges addressed by the Infosphere project, focusing on the inherent heterogeneous nature of these new information-rich environments and the dynamically varying demand and availability of their resources. Other issues addressed by the Infosphere projects include quality of service guarantees, automated generation and composition of data streams, and standing queries for monitoring of information flows.

7.9 Summary of Related Work

The work presented on this dissertation is influenced by research ideas from a number of different related areas. Many of the ideas have originated in early work done by our group concerning on-line steering and visualization of high-performance scientific applications [70, 121, 135, 118, 61] and resource-aware computing [71, 78]. Active Streams is part of the Infosphere Expedition [97], a much broader research effort aimed at creating the virtual spaces of interaction for the post-PC era of computing.

Various projects have proposed component-based approaches to software development in wide-area distributed systems as they facilitate the construction of complex applications by allowing the creation of generic reusable components and by easing independent component development. Active Streams uses the ECho distributed event system for communication and component integration and we briefly surveyed related work on this area.

ECho and, thus, Active Streams rely on *Native Data Representation*, a new approach to wire-format in heterogeneous communications with the combined goals of flexibility and high performance. We have reviewed alternative solutions and compared them with our NDR approach.

Active Streams shares its goals with many recent efforts that propose the injection of application functionality into the datapath and the use of run-time monitoring and dynamic adaptation to cope with changing environment conditions and application requirements. We have reviewed work in end-point and distributed adaptation, as well as related projects in resource-aware computing.

An important part of this dissertation's contribution is the design and implementation of an architecture for building adaptive and extensible distributed systems following an Active Streams approach. We thus reviewed some of the work that relates to these framework components, including work on resource monitoring, repository, and directory services.

The systems discussed above have demonstrated the value of customizing service functionality, dynamically extending clients, and adapting applications and services to dynamically changing environments. Building on these previous efforts while recognizing their limitations lays the foundation of this work. Active Streams provides an approach to constructing adaptive distributed applications and services that exhibits these characteristics.

Chapter 8: Conclusions

This chapter summarizes key contributions of the Active Streams approach presented in this dissertation. A brief description of the approach main components is included together with a discussion of our main contributions. In closing, future research directions are discussed, and specific opportunities for future exploration are highlighted.

8.1 Summary

This dissertation explores Active Streams, a novel middleware approach and its supporting framework for building distributed applications and services for heterogeneous, highly dynamic environments. Our approach supports the dynamic customization of services, the run-time extensibility of applications, and the dynamic adaptation of applications and services to environmental changes. The approach adopts a component-based model to system programming centered around two simple abstractions - *streams* and *streamlets*. Streams are sequences of typed, self-describing, application-specific data units connecting parts of and applications and services. These streams are made active by attaching streamlets, application- or service-specific location-independent functional units.

We presented the design and implementation of a framework that supports this approach. The Active Streams framework supports dynamic system adaptation at multiple levels and points in the underlying platform; it provides a pull-based service for code distribution with security considerations; it facilitates the needed infrastructure for resource monitoring, self-monitoring and adaptation; and it includes a directory service with an extended proactive interface more suited to the dynamism of the targeted environments.

The Active Streams Framework has been use in a number of class' projects at Georgia Tech. In this dissertation we presented our experiences with the implementation of two applications inteded to demonstrate the utility and flexibility of the Active Streams approach.

Finally, we reported evaluations inteded to quantify the costs and benefits of the approach. We presented experimental results showing the benefits in performance and flexibility of PBIO, and reported results from a detailed performance evaluation of ECho demonstrating how it significantly outperforms other systems that provide similar functionality, offering throughput and latency comparable to the most efficient communication middleware. We presented results that demonstrated the costs and benefits of Active Streams for end-user applications and discussed evaluations that indicated the basic overhead of our approach, the benefits of stream specialization, and the need for multiple points of adaptation over the datapath. Reported experiments have shown that the basic streamlet mechanism implemented does not add significant overheads to data streaming, so that the overheads introduced by the use of streamlets primarily depend on the functionality of the specific streamlet(s) utilized by an application. We illustrated the potential benefits of stream specialization and presented results indicating that, counter to common wisdom, the benefits of placing streamlets up the stream do not have to come at the cost of additional load to sources. The need for multiple points of adaptation over the datapath, on the basis of performance, was shown in Section 6.3. We also presented evaluation results that demonstrate the basic performance of PDS, on of our framework's main components, and the benefits of its proactive approach to providing directory services.

8.2 Future Directions

This thesis opens up several opportunities for future work. Some of the ideas worth exploring directly extend the Active Streams Framework while other touch upon the application of our work to other research problems.

A promising research area is the use of the Active Streams approach to build customizable wide-area data distribution. Through the strategic placement stream of specializations one could potentially increase the ratio of information per data flowing through the environment while minimizing the number of multiple instances of the same data flow over a given data link.

Wide-area resource allocation is another avenue for future research. The use of Active Streams Nodes by multiple applications, scheduling policies for streamlets, heuristics for the automatic placement of streamlets over the Active Streams Node overlay network are just but a few of them.

Finally, various interesting issues remain to be address in the context of safety and security in Active Streams.

"I hope that posterity will judge me kindly, not only as to the things which I have explained, but also to those which I have intentionally omitted so as to leave to others the pleasure of discovery." -Rene Descartes

Bibliography

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 81–91, San Jose, CA, October 1998.

[2] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 186–201, Kiawah Island, NC, December 1999. ACM.

[3] Asmara Afework, Michael D. Beynon, Fabián E. Bustamante, Angelo Demarzo, Renato Ferreira, Robert Miller, Mark Silberman, Joel Saltz, Alan Sussman, and Hubert Tsang. Digital dynamic telepathology - the virtual microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*, 1998. Also appears as Technical Report UMCP-CSD:CS-TR-3892.

[4] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The SwitchWare active network architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):29–36, May/June 1998.

[5] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active bridging. In *Proceedings of the ACM conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM '97)*, pages 101–111, Cannes, France, September 1997.

[6] Guy T. Almes. The impact of language and system on remote procedure call design. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, pages 414–421, May 13-19 1986.

[7] Elan Amir, Steven McCanne, and Randy Katz. An active service framework and its application to real-time multimedia transcoding. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 178–189, Vancouver, BC, Canada, August 1998. ACM.

[8] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, May 1999. ACM Press.

[9] B. Badrinath, Armando Fox, Leonard Kleinrock, Gerald Popek, Peter Reiher, and M. Satyanarayanan. A conceptual framework for network and client adaptation. *Mobile Networks and Applications*, 5(4):221–231, December 2000.

[10] Michael Beynon, Renato Ferreira, Tahsin Kurc, Alan Sussman, and Joel Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the 8th Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pages 119–133, College Park, MD, March 2000.

[11] Samrat Bhattacharjee, Ken Calvert, and Ellen W. Zegura. An architecture for active networking. In *Proceedings of High Performance Networking (HPN'97)*, White Plains, NY, April 1997.

[12] Andrew Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communication of the ACM*, 25(4):260–274, April 1982.

[13] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, 1998.

[14] Jürg Bolliger and Thomas Gross. A framework-based approach to the development of network-aware applications. *ACM Transactions on Software Engineering and Methodology*, 24(5):376–390, May 1998.

[15] Randall Bramley, Kenneth Chiu, Shridhar Diwan, Dennis Gannon, Madhusudhan Govindaraju, Nirmal Mukji, Benjamin Temko, and Madhuri Yechuri. A component based services architecture for building distributed applications. In *Proceedings of the 9th International Symposium on High Performance Distributed Computing (HPDC-9)*, Pittsburgh, PA, August 2000.

[16] Fabián E. Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Efficient wire formats for high performance computing. In *Proceedings of Supercomputing '00 (SC2000)*, November 4-10 2000.

[17] Pei Cao and Chengjie Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4):445–457, April 1998. Published in the 17th IEEE International Conference of Distributed Computing.

[18] Michael J. Carey, David J. DeWitt, Michael J. Franlkin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C.K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD Conference on the Management of Data*, pages 383–394, Minneapolis, MN, May 1994.

[19] Antonio Carzaniga, David S. Rosenblum, and Alex L. Wolf. Challenges for distributed event services: Scalability vs. expressiveness. In *Proceedings of Engineering Distributed Objects (EDO '99), ICSE 99 Workshop*, May 1999.

[20] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evolution of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[21] CCITT. The directory-authentication framework. Recommendation X.509, Consultation Committee, International Telephone and Telegraph, International Telecommunications Union, Geneva, 1989.

[22] Vinton G. Cerf. *Beyond Calculation: The next fifty years of computing*, chapter When They're Everywhere, pages 33–42. Springer-Verlag, New York, NY, 1997.

[23] James Clark. expat - XML parser toolkit. http://www.jclark.com/xml/expat.html.

[24] Internet Software Consortium. Bind domain name service. http://www.isc.org/products/BIND/bind8.html.

[25] Internet Software Consortium. Internet domain survey. http://www.isc.org, January 2001.

[26] Gartner Consulting. The emergence of distributed content management and peer-to-peer content networks. Engagement #010022501, Gartner Group, San Jose, CA, January 2001.

[27] Systran Federal Corporation. Scramnet networks. http://www.systran.com/realtim.htm.

[28] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. Exploting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98),*, Kyoto, Japan, April 1998.

[29] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An architecture for a secure service discovery service. In *Proceedings of ACM/IEEE MOBICOM*, pages 24–35, August 1999.

[30] DARPA ITO. Ubiquitous computing. http://www.darpa.mil/ito/research/uc.

[31] Dartmouth College, Hanover, New Hampshire. *Systems Programmers Manual for the Dartmouth Time Sharing System for the GE 635 Computer*, 1971.

[32] David D.Clark and D.L.Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the symposium on Communications architectures and protocols (SIGCOMM '90)*, pages 200–208, September 26-28 1990.

[33] Dan Decasper and Bernhard Plattner. DAN: distributed code caching for active networks. In *Proceedings of Infocom'98*, San Francisco, CA, March 1998.

[34] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[35] Peter A. Dinda, Thomas Gross, Roger Karrer, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, and Dean Sutherland. The architecture of the Remos systems. In *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10)*, San Francisco, CA, August 2001.

[36] Guy Eddon and Henry Eddon. *Inside Distributed COM.* Microsoft Press, Redmond, WA, 1998.

[37] Greg Eisenhauer, Fabián E. Bustamante, and Karsten Schwan. Event services for high performance computing. In *Proceedings of the 9th International Symposium on High Performance Distributed Computing (HPDC-9)*, Pittsburgh, PA, August 2000. IEEE.

[38] Greg Eisenhauer, Fabián E. Bustamante, and Karsten Schwan. A middleware toolkit for client-initiated service specialization. In *Proceedings of Principles of Distributed Computing (PODC 2000) Middleware Symposium*, Portland, OR, July 2000.

[39] Greg Eisenhauer and Lynn K. Daley. Fast heterogenous binary data interchange. In *Proceedings of the Heterogeneous Computing Workshop (HCW2000)*, May 3-5 2000.

[40] Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[41] Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[42] Patrick Eugster, Rachid Guerraoui, and Joe Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *Proceedings of ECOOP'00*. Springer Verlag (LNCS), June 2000.

[43] David C. Feldmeier, Anthony J. McAuley, Jonathan M. Smith, Deborah S. Bakin, William S. Marcus, and Thomas M. Raleigh. Protocol boosters. *IEEE Journal on Selected Areas in Communications, Special Issue on Protocol Architectures for the 21st Century*, 16(3):437–444, April 1998.

[44] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A directory service for configuring high-performance distributed computation. In *Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC-6)*, pages 365–375, Portland, OR, August 1997. IEEE.

[45] Message Passing Interface (MPI) Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, 1995.

[46] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, pages 70–82, 1996.

[47] OpenLDAP Foundation. The OpenLDAP Project. http://www.openldap.org/.

[48] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambride, MA, October 1996.

[49] Xiaodong Fu, Weisong Shi, Anatoly Akkerman, and Vijay Karamcheti. CANS: Composable, adaptive network services infrastructure. In *3rd USENIX Symposium on Internet Technologies*, San Francisco, CA, March 2001.

[50] Richard M. Fujimoto, Karsten Schwan, Mustaque Ahamad, Scott Hudson, and John Limb. Distributed laboratories: A research proposal. Technical Report GIT-CC-96-13, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, 1996.

[51] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44. Springer-Verlag, LNCS 551, October 1991.

[52] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering, Volume I.* World Scientific Publishing Company, New Jersey, 1993.

[53] Steve Goddard and Kevin Jeffay. Distributed real-time dataflow: An execution paradigm for image processing and anti-submarine warfare applications. In *Proceedings of the 17th IEEE Real-Time Systems Symposium, Work in Progress Abstracts*, pages 55–58, Washington, DC, December 1996.

[54] Ramesh Govindan, Cengiz Alaettinoğlo, and Deborah Estrin. A framework for active distributed services. Tech. Report 98-669, Information Science Institute, University of Southern California, Los Angeles, CA, January 1998.

[55] Goetz Graefe. Volcano, an extensible and parallel query evaluation system. *IEEE Transactions on knowledge and data enginnering*, 6(1):120–135, February 1994.

[56] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The Ninja architecture for robust Internet-scale systems and services. *Special Issue of Computer Networks on Pervasive Computing.*, 2000.

[57] Thomas Gross, Peter Steenkiste, and Jaspal Subhlok. Adaptive distributed applications on heterogeneous networks. In *Proc. 8th Heterogeneous Computing Workshop (HCW '99)*, San Juan, Puerto Rico, April 1999.

[58] Object Management Group. *CORBAservices: Common Object Services Specification*, chapter 4. OMG, 1997.

[59] Object Management Group. Notification service. http://www.omg.org, Document telecom/98-01-01, 1998.

[60] Robert Grumm, Janet Davis, Ben Hendrickson, Eric Lemar, Adam MacBeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. Systems directions for pervasive computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schoss Elmau, Germany, May 2001.

[61] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, and Jeffrey Vetter. Falcon: On-line monitoring and steering of large-scale parallel programs. *Concurrency: Practice and Experience*, 1998.

[62] James Gwertzman and Margo Seltzer. World-wide web cache consistency. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[63] Richard S. Hall, Dennis M. Heimbigner, André van der Hoek, and Alexander L. Wolf. An architecture for post-development configuration management in a wide-area network. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS'97)*, pages 269–278, Baltimore, USA, May 1997.

[64] John Hartman, Udi Manber, Larry Peterson, and Todd A. Proebsting. Liquid software: A new paradigm for networked systems. Tech. Rep. TR 96-11, Department of Computer Science, University of Arizona, Tucson, AZ, June 1996.

[65] Tomasz Haupt, Erol Akarsu, and Geoffrey Fox. Webflow: A framework for web based metacomputing. In *High-Performance Computing and Networking, 7th International Conference (HPCN Europe)*, pages 291–299, Amsterdam, The Netherlands, April 1999. Also published in Lecture Notes in Computer Science, Vol. 1593, Springer, 1999.

[66] Arthur Van Hoff, Hadi Partovi, and Tom Thai. The open software description format (osd). Note, Marimba Incorporated and Microsoft Corporation, August 1997. Latest version http://www.w3.org/TR/NOTE-OSD.

[67] An-Cheng Huang and Peter Steenkiste. A flexible architecture for wide-area service discovery. In *The Third IEEE Conference on Open Architectures and Network Programming (OpenArch-2000)*, Tel-Aviv, Israel, March 2000. Short paper session.

[68] Tivoli Systems Inc. Software distribution. www.tivoli.com, Last access October 2001.

[69] Underwriters Laboratories Inc. Underwriters laboratories. www.ul.com.

[70] Carsten Isert and Karsten Schwan. ACDS: Adapting computational data streams for high performance. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.

[71] Daniela Ivan-Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On adaptive resource allocation for complex, real-time applications. In *IEEE Real-Time Systems Symposium*, San Francisco, CA, December 1997.

[72] Beverly Schwartzand Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, R. Dennis Rockwell, and Craig Partridge. Smart packets: applying active networks to network management. *ACM Transactions on Computer Systems*, 18(1):67–88, February 2000.

[73] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 156–171, Copper Mountain, CO, December 1995. ACM Press.

[74] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Proceedings of The Information Processing Congress (IFIP 74)*, pages 471–475. North-Holland Publishing Co., August 1974.

[75] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Record*, 27(3), September 1998.

[76] Thomas Kindler, Karsten Schwan, Dilma Silva, Mary Trauner, and Fred Alyea. A parallel spectral model for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.

[77] Douglas Kramer. The Java platform: A white paper. *Sun Microsystems Inc*, May 1996.

[78] Robin Kravets, Ken Calvert, and Karsten Schwan. Payoff-based communication adaptation based on network service availability. In *IEEE Multimedia Systems '98 (ICMCS)*, August 1998.

[79] Argonne National Laboratory. MPICH - a portable implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich.

[80] Butler W. Lampson. Designing a global name service. In *Proc. 4th ACM Symposium on Principles of Distributed Computing*, pages 1–10, Calgary, Alta Canada, August 1986. ACM.

[81] Mario Lauria, Scott Pakin, and Andrew A. Chien. Efficient layering for high speed communication: Fast messages 2.x. In *Proceedings of the 7th International Symposium on High Performance Distributed Computing (HPDC-7)*, Chicago, IL, July 1998.

[82] Edward A. Lee. Dataflow process networks. Technical Report UCB/ERL M94/53, University of California, Berkeley, CA, July 1994.

[83] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering, Special issue on Web Technologies*, 11(4):610–628, July/Aug 1999.

[84] Julio López and David O'Hallaron. Evaluation of a resource selection mechanism for complex network services. In *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10)*, pages 171–180, San Francisco, CA, August 2001.

[85] Silvano Maffeis. iBus/MessageBus - the Java intranet software bus. http://www.softwired.ch.

[86] Massachusets Institute Of Technology. MIT project Oxygen. http://www.lcs.mit.edu/.

[87] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 123–133, Stanford, CA, August 1988. ACM.

[88] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[89] Matthew Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Proceedings of the 9th International Conferece on Very Large Data Bases (VLDB93)*, pages 34–42, Florence, Italy, November 1983.

[90] NCSA. Environmental hydrology demo. http://scrap.ssec.wisc.edu/r̃ob/sc98.

[91] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 276–287, Saint Malo, France, October 1997.

[92] NPACI. Alpha projects. http://www.npaci.edu/Alpha, 2001.

[93] NSF:NEES Project. NEESgrid: earthquake engineering virtual collabortory. http://www.neesgrid.org, 2001.

[94] Erik L. Nygren, Stephen J. Garland, and M. Frans Kaashoek. PAN: a high-performance active network node supporting multiple mobile code systems. In *The Second IEEE Conference on Open Architectures and Network Programming (OpenArch-1999)*, pages 100–111, New York, NY, March 1999.

[95] Object Management Group. The common object request broker architecture and CORBA 2.0/IIOP specification. Technical report, OMG, December 1998. http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[96] Object Management Group. Corba components - volume 1. Technical Report orbos/99-07-01, OMG, July 1999. http://www.omg.org/cgi-bin/doc?orbos/99-07-01.

[97] Georgia Insitiue of Technology, Oregon Graduate Institute of Science, and Technology. Infosphere: Smart delivery of fresh information. http://www.cc.gatech.edu/projects/infosphere/.

[98] Van Oleson, Greg Eisenhauer, Calton Pu, Karsten Schwan, Beth Plale, and Dick Amin. Operational information systems - an example from the airline industry. In *First Workshop on Industrial Experiences with System Software*, pages 1–10, San Diego, CA, October 2000.

[99] S. W. O'Malley, T. A. Proebsting, and A. B. Montz. Universal stub compiler. In *Proceedings of the symposium on Communications architectures and protocols (SIGCOMM '94)*, August 1994.

[100] Andy Oram, editor. *Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology.* O'Reilly & Associates, Sebastopol, CA, 2001.

[101] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[102] Carmen M. Pancerella, Larry A. Rahn, and Christine L. Yang. The diesel combustion collaboratory: Combustion researchers collaborating over the Internet. In *Proceedings of Supercomputing '99 (SC1999)*, November 13-19 1999.

[103] Przemyslaw Pardyak and Brian N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 201–212, Seattle, WA, October 1992. USENIX.

[104] Steven Parker and Christopher R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Proc. of Supercomputing 1995 (SC 1995)*, San Diego, CA, December 1995.

[105] Craig Partridge, T. Mendez, and W. Milliken. Host anycasting service. Technical Report Internet RFC 1546, Network Working Group, November 1993.

[106] Bahram Parvin, John Taylor, Ge Cong, Michael O'Keefe, and Mary-Helen Barcellos-Hoff. Deepview: A channel for distributed microscopy and informatics. In *Proceedings of Supercomputing '99 (SC1999)*, November 13-19 1999.

[107] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), October 1992.

[108] Massimiliano Poletto, Dawson Engler, and M. Frans Kaashoek. tcc: A template-based compiler for 'c. In *Proceedings of the First Workshop on Compiler Support for Systems Software (WCSSS)*, February 1996.

[109] Prashant Pradhan, Tzi cker Chiueh, and Anindya Neogi. Aggregate TCP congestion control using multiple network probing. In *Proceedings of the 20th International Conference on Distributed Computer Systems (ICDCS-20)*, Taipei, Taiwan, April 2000.

[110] Calton Pu, Karsten Schwan, and Jonathan Walpole. Infosphere project: System support for information flow applications. *ACM SIGMOD Record*, 30(1), March 2001.

[111] The Economist Technology Quarterly. Computing power on tap. *The Economist*, June 21st 2001.

[112] S. Radicati. X.500 directory services: Technology and deployment. Technical report, International Thomson Computer Press, London, UK, 1994.

[113] Erik Riedel and Garth Gibson. Active disks - remote execution for network-attached storage. Technical Report CMU-CS-97-198, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1997.

[114] Dennis M. Ritchie. The evolution of the Unix time-sharing system. *AT&T Bell Laboratories Technical Journal*, 63(6.2):1577–1593, October 1984.

[115] Ron Rivest. The MD5 message-digest algorithm. Network Working Group Request for Comments Internet RFC 1321, Network Working Group, April 1992.

[116] Marcel-Catalin Rosu, Karsten Schwan, and Richard Fujimoto. Supporting parallel applications on clusters of workstations: The virtual communication machine-based architecture. *Cluster Computing, Special Issue on High Performance Distributed Computing*, 1(1):51–67, January 1998.

[117] M. Schroeder and M. Burrows. Performance of Firefly RPC. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 83–90, December 1989.

[118] Karsten Schwan, Fred Alyea, William Ribarsky, and Mary Trauner. Integrating program steering, visualization, and analysis in parallel spectral models of atmospheric transport. *NASA Science Newsletter: Information Systems*, 1995.

[119] Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the AUUG (Australian users group for Unix and Open Systems) 1997 Conference*, September 1997.

[120] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[121] Dilma Silva, Karsten Schwan, and Greg Eisenhauer. Configurable distributed retrieval of scientific data. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*, pages 120–127, Annapolis, MD, May 1998.

[122] Warren Smith, Abdul Waheed, David Meyers, and Jerry Yan. An evaluation of alternative designs for a grid information service. In *Proceedings of the 9th International Symposium on High Performance Distributed Computing (HPDC-9)*, Pittsburgh, PA, August 2000. IEEE.

[123] Rok Sosic. *The many faces of introspection.* PhD thesis, University of Utah, Salt Lake City, UT, June 1992.

[124] Peter Steenkiste. Adaptation models for network-aware distributed computations. In *3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC'99)*, Orlando, FL, January 1999.

[125] Robert 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 *International Symposium on Software Reliability Engineering '98 Fast Abstract*, 1998.

[126] Pradeep Sudame and B.R. Badrinath. Transformer tunnels: A framework for providing route-specific adaptations. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[127] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transaction on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[128] Sun Microsystems Inc. Enterprise Java Beans technology. http://java.sun.com/products/ejb/.

[129] Validy S. Sunderam, Al Geist, Jack Dongarra, and R. Manchek. The PVM concurrent computing system. *Parallel Computing*, 20(4):531–545, March 1994.

[130] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[131] Peer to Peer Working Group. The working group on peer-to-peer computing. http://www.peer-to-peerwg.org/.

[132] US Department of Commerce/NIST. Fips 180-1 secure hash standard. Technical report, National Technical INformation Service, Springfieldd, VA, April 1995.

[133] Amin Vahdat, Michael Dahlin, Thomas Anderson, and Amit Aggarwal. Active names: flexible location and transport of wide-area resources. In *Proceedings of USENIX Symp. on Internet Technology & Systems*, October 1999.

[134] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol. RFC 2165, Network Working Group, June 1997.

[135] Jeffrey Vetter and Karsten Schwan. High performance computational steering of physical simulations. In *IEEE International Parallel Processing Symposium (IPPS)*, April 1997.

[136] P. Vixie. A mechanism for prompt notification of zone changes (dns notify). RFC 1996, Network Working Group, August 1996.

[137] Thorsten von Eicken, David E. Culler, Seth C. Golsdtein, and Klaus E. Schauser. Active messages: a mechanism for integrated communication and computation. Report UCB/CSD 92/675, Univeristy of California, Berkley, March 1992.

[138] W3C. Extensible markup language (XML). http://w3c.org/XML.

[139] Jim Waldo. Jini architecture overview. Technical report, Sun Microsystems, 1999.

[140] Grirsh Welling and B.R. Badrinath. A framework for environment aware mobile applications. In *Proceedings of the 17th International Conference on Distributed Computer Systems (ICDCS-17)*, pages 384–391, Baltimore, MD, May 1997.

[141] Matt Welsh, Anindya Basu, and Thorsten Von Eicken. Incorporating memory management into user-level network interfaces. In *Proceedings of Hot Interconnects V*, pages 27–36, 1997.

[142] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, Banff, Canada, October 2001. ACM, SIGOPS.

[143] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. ANTS: a toolkit for building and dynamically deploying network protocols. In *The First IEEE Conference on Open Architectures and Network Programming (OpenArch'1998)*, San Francisco, CA, April 1998.

[144] Patrick Widener, Greg Eisenhauer, and Karsten Schwan. Open metadata formats: Efficient xml-based communication for high performance computing. In *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10)*, San Francisco, CA, August 2001.

[145] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems - Triggers and Rules For Advanced Database Processing.* Morgan Kaufman Publishers Inc., San Francisco, CA, September 1994.

[146] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for Java system. In *Proceedings of the USENIX COOTS 1996*, 1996.

[147] Richard Wolniewicz and Goetz Graefe. Algebraic optimization of computation over scientific databases. In *Proceedings of the 19th Very Large Data Base Conference*, Dublin, Ireland, 1993.

[148] Rich Wolski, Neil Spring, and Chris Peterson. Implementing a performance forecasting system for metacomputing: The Network Weather Service. In *Proceedings of Supercomputing '97 (SC1997)*, San Jose, CA, November 1997.

[149] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 1999.

[150] Mark Yarvis, Peter Reiher, and Gerald J. Popek. Conductor: A framework for distributed adaptation. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[151] Haobo Yu, Devorah Estrin, and Ramesh Govindan. A hierarchical proxy architecture for internet-scale event services. In *Proceedings of WETICE'99*, Stanford, CA, June 1999.

[152] Bruce Zenel and Dan Duchamp. A general prupose proxy filtering mechanism applied to the mobile environment. In *Proceedings of Mobile Computing (MOBICOM) 97*, pages 248–259, Budapest, Hungary, September 1997.

[153] Dong Zhou, Karsten Schwan, Greg Eisenhauer, and Yuan Chen. JECho - interactive high performance computing with Java event channels. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 2001)*, April 2001.

Vita

Fábian Ernesto Bustamante was born in Comodoro Rivadavia, Patagonia, Argentina on February 6, 1966. He received the degrees of 'Analista Programador Universitario' (3-year) and 'Licenciatura en Ciencias de la Computacion' (5-year-and-project) from the Universidad Nacional de la Patagonia San Juan Bosco in 1992 and 1993, where he worked until 1994.

After his English studies at the University of Pennsylvania, he initiated his graduate studies in Computer Science at the Georgia Institute of Technology in the fall of 1995. While working as part of the Systems Research Group he received a M.S. degree in Computer Science in 1997.

During the Summer of 1997, he had the opportunity to work on the Virtual Microscope Project as part of the Chaos group at the University of Maryland, College Park and under the direction of Joel Saltz. This project – which aimed to provide a realistic digital emulation of a high-power light microscope and to investigate possible new modes of interaction – went to win the award as the Best Application at the American Medical Informatics Association's 1997 Annual Conference. In the Fall of 1999, he joined Dr. John Wilkes' Storage Systems Program at Hewlett-Packard Labs, and under the direction of Dr. Guillermo Alvarez worked on Pacioli, a set of tools for the construction and validation of analytical performance models originally intended for application to disk array systems.

In November 2001 and under the supervision of Dr. Karsten Schwan, he completed his Ph.D. dissertation entitled "The Active Streams Approach to Adaptive Distributed Applications and Services".