Supporting Causal Multicast in Distributed Operating Systems: An Experiment in Architectural Approaches

Roger M. Ellingson
B.S.
University of North Dakota

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

January 1992

The thesis "Supporting Causal Multicast in Distributed Operating Sytems: An Experiment in Architectural Approaches" by Roger M. Ellingson has been examined and approved by the following Examination Committee:

Jonathan Walpole
Assistant Professor
Thesis Research Adviser

Steve Otto
Assistant Professor

Robert G. Babb II
Associate Professor

Acknowledgements

I wish to thank the faculty of the Oregon Graduate Institute for their assistance in the completion of this work. In particular, thank you, Professor Walpole, for leadership, insight, patience, and flexibility. Thanks for taking an interest in me and directing, challenging and mentoring my growth; I appreciate the effort and am very pleased with the results. Thank you, Professors Otto and Babb for dedicated thesis committee work. I appreciate your time, efforts, comments and suggestions. Professor Kieburtz, thank you for helping me get started at OGI and for assembling the best CS/E department in the Northwest!

Thanks to Ken Birman, Cornell University, for his help gathering and analyzing ISIS performance figures. Ken's ISIS work and related research have played a significant role in my grounding in this area. He and his group must be complemented on their work.

The OGI staff deserves credit for their work in support of this institution. Thanks to the department computer staff for keeping both hardware and software in top-notch condition. Special thanks to John Pochmara for his basically volunteer, cheerful system support and crucial bug fixes. Thanks to the department administration staff for freeing me of almost all worldly details, allowing me to concentrate on the actual work to be done. Also, thank you for all the parties and special activities. The library staff deserve a big thank you for creating and maintaining the best research library in the Northwest. In particular, thank you, Mary Vatne, for help, encouragement, and concern. Thanks goes to the Office of the Registrar, in particular Betty Shannon, for help, delicious treats, and your consistently cheerful attitude. I thank the facilities staff for their careful attention to our comfort and for a clean, well-maintained work environment. Thank you, OGI Student Council for the great picnics and outdoor cooking.

Thank you, fellow members of the extended Distributed Systems Research Group for the caring and feeding of me. Thank you, fellow students at OGI, for making my studies at OGI a most stimulating and enjoyable experience. I must single out Jon Inouye for individual kudos. Jon's knowledge, precision, sensitivity to others, and *ability to share* must be acknowledged.

Thank you, Dr.Chedley Arouri, Intel Corp, and Tim Bennington-Davis and Mary Richardson, Tektronix, Inc., for both personal and professional encouragement and support.

Thanks to all my friends and Care Group members for their support. I would say their prayers of concern and encouragement have been generously answered.

I sincerely thank my mother and father, and wife's mother and father for providing the foundations and on-going support for endeavors such as this work. Theirs and their childrens' support, prayers, and encouragement are deeply appreciated and have been a big help.

The best has been saved for last. I thank my wonderful wife, Marie, for her steadfast love and commitment. Marie has consistently demonstrated an understanding, unselfish, patient, and peaceful attitude about this whole affair. Most important, Marie has kept me correctly focused—professionally and spiritually.

Dedication

What shall I render to the Lord
For all His benefits toward me?
Psalm 116:12

Contents

Acknowledgements iii

Dedication v

Abstract xii

1 Introduction 1

1.1 Distributed Programming Problems 3
1.2 Ordered Multicast Communication 5
1.3 Distributed Operating System Support for Causal Multicast Communication 6
1.4 Thesis Outline 8

2 Distributed Operating System and Application Abstractions for Communication 9

2.1 Distributed Operating System Communication Abstractions 10
2.1.1 Port and Message Abstractions 10
2.1.2 Communication Group Abstractions 11
2.2 High-Level Ordered Multicast Communication Abstractions 13
2.2.1 Multicast Communication 13
2.2.2 Multicast Group Addressing 14
2.2.3 Ordered Multicast Protocols 14
2.2.4 ISIS Toolkit 17
2.3 Communication Abstraction Summary 19

3 Salon: System for Architectural Experimentation in Causal Multicast Support 21

3.1 Salon Library 22
3.1.1 Salon Causally Ordered Communication Protocol 22
3.1.2 Salon Communication Group Abstraction 25
3.1.3 Salon Multicast Abstraction 26

3.1.4 Salon Library Design 26
3.1.5 Visualization Tools for Debugging and Demonstration 28
3.2 Causal Multicast Experimental Architectures 29
3.2.1 User-Level Salon Implementation 30
3.2.2 Server-Level Salon Implementation 30
3.2.3 Split-Level Salon Implementation 32
3.3 Summary of Experimental Setup 33

4 Results & Analysis **35**
4.1 Results 36
4.1.1 Experimental Synchronous Performance 41
4.1.2 Experimental Asynchronous Performance 44
4.2 Performance Analysis 47
4.2.1 Performance Analysis of User-Level Salon 47
4.2.2 Performance Analysis of Server-Level Salon 54
4.2.3 Performance Analysis of Split-Level Salon 56
4.3 Predicted Implementation Performance 57
4.3.1 Projected Causal Multicast Application Library Performance . . . 58
4.3.2 Projected Port Group Multicast with Causal Protocol Performance 58
4.3.3 Projected Port Group Multicast with User-Level Causal Protocol Performance 60
4.4 Summary 62

5 Conclusions **64**
5.1 Limitations of Our Work 66
5.2 Suggestions for Future Work 66

Bibliography **68**

A Event Plot Server **73**

B Perturber Server **75**

C Salon Protocol Visual Debugging Environment **80**
C.1 User-Level Salon Development 81
C.2 Server-Level Salon Development 81

D ISIS Toolkit Performance 87
D.1 Results 88

E Data Significance 91
E.1 Statistical Significance of Results 92

List of Tables

4.1 Synchronous user-level salon fundamental performance. 42
4.2 Synchronous server-level salon fundamental performance. 42
4.3 Synchronous split-level salon fundamental performance. 42
4.4 Asynchronous user-level salon fundamental performance. 45
4.5 Asynchronous server-level salon fundamental performance. 45
4.6 Asynchronous split-level salon fundamental performance. 45
4.7 Synchronous Mach MSD2.6—no Salon header. 48
4.8 Asynchronous Mach MSD2.6—no Salon header. 48
4.9 Synchronous Mach MSD2.6—Salon header. 51
4.10 Asynchronous Mach MSD2.6—Salon header. 51
4.11 Synchronous Mach MSD2.6—Salon header, no application data copying. . 54
4.12 Asynchronous Mach MSD2.6—Salon header, no application data copying. 55
4.13 Estimated synchronous causal multicast port group performance. 60
4.14 Estimated synchronous port group multicast with user-level protocol performance. 61

D.1 Synchronous ISIS v2.1 performance. 88
D.2 Asynchronous ISIS v2.1 performance 89

E.1 Pairwise t-test probability of equality between mean architecture performance. 93

List of Figures

1.1 Typical distributed operating system architecture and interface levels. . . 2
1.2 Event diagram illustrating message delivery ordering problem. 4
1.3 Multicast group communication and addressing. 7

2.1 Overlapping multicast group communication ordering problem. 18
2.2 ISIS causal multicast communication architecture. 19

3.1 Global view of vector time. 24
3.2 Experimental architectures investigated. 30
3.3 User-level Salon architecture. 31
3.4 Server-level Salon architecture. 32
3.5 Split-level Salon architecture. 33

4.1 Synchronous user-level salon scaling performance. 43
4.2 Synchronous server-level salon scaling performance. 43
4.3 Synchronous split-level salon scaling performance. 43
4.4 Asynchronous user-level salon scaling performance. 46
4.5 Asynchronous server-level salon scaling performance. 46
4.6 Asynchronous split-level salon scaling performance. 46
4.7 Hypothetical kernel-level causal communication support. 59
4.8 Hypothetical multicast port group/user protocol support. 61

A.1 Salon plot event server display. 74

B.1 Perturber and plot servers with no message perturbation. 77
B.2 Perturber and plot servers with message delayed at perturber. 78
B.3 Multicast followed by delayed message release. 79

C.1 User-level Salon distributed application without causally ordered protocol. 83
C.2 User-level Salon distributed application with Salon protocol enabled. . . . 84
C.3 Server-level Salon distributed application with Salon protocol disabled. . . 85

C.4 Server-level Salon distributed application with Salon protocol enabled. . . 86

D.1 Synchronous ISIS v2.1 scaling performance. 89
D.2 Asynchronous ISIS v2.1 scaling performance. 90

Abstract

Supporting Causal Multicast in
Distributed Operating Systems:
An Experiment in Architectural Approaches

Roger M. Ellingson, M.S.
Oregon Graduate Institute of Science & Technology, 1992

Supervising Professor: Jonathan Walpole

Most current operating system research is focused on message-passing based operating systems. Such operating systems define the simple low-level abstractions of messages and ports upon which almost all higher level system services are built. Several of these operating systems have extended the point-to-point communication naturally supported by ports and messages with multicast group communication abstractions.

In addition to low-level operating system research, there has been considerable progress defining and implementing ordered multicast communication abstractions to simplify programming distributed applications. At the application level, multicasting, multicast group addressing and causally ordered message delivery protocol abstractions have been combined to provide a *causal multicast group* abstraction. Distributed system programming experience has shown the causal multicast group abstraction to be an extremely useful fundamental building block when building distributed applications.

This thesis explores the issue of distributed operating system support for distributed

applications through the provision of ordered multicast group communication functionality. We have designed and implemented a causal multicast programming support library based upon functionality extracted from the ISIS toolkit, a popular user-level distributed programming library. Using our library we designed and implemented several distributed operating system experimental architectures to investigate causal multicast support in distributed computing systems. From the results of these experiments we reason about direct kernel support for causal multicast functionality.

Chapter 1

Introduction

The distributed computing systems of interest to us share no memory between computing nodes or sites, nor do they share a global clock. The nodes are connected by a network but communication delays between communication end-points are unbounded. Asynchronous communication between nodes is supported, i.e. senders do not have to wait for receivers to respond to messages.

The widespread availability of networked computing systems combined with the increasing demand for distributed applications has led operating system designers to consider support for distribution a fundamental feature of next generation operating systems. In response to these requirements, the single most prominent trend in operating system design over recent years has been the move towards message-passing micro-kernel operating systems [MVF+90, ABB+86, CZ85, MvRT+90]. Such systems move operating system functionality, such as process and file management, out of the operating system kernel into server processes that communicate with each other through message passing. The underlying micro-kernel provides the functionality necessary to implement such servers and supports the communication between them.

Figure 1 shows the levels of interface typical of micro-kernel operating systems. The system is comprised of low-level kernel and server layers supporting the high-level user or application layer. The micro-kernel interface provides access to the lowest level system services (e.g. process scheduling and communication), while the sub-system interface is comprised of a set of cooperating, trusted servers, typically representing complex operating system abstractions (e.g. UNIX, memory management, network services).

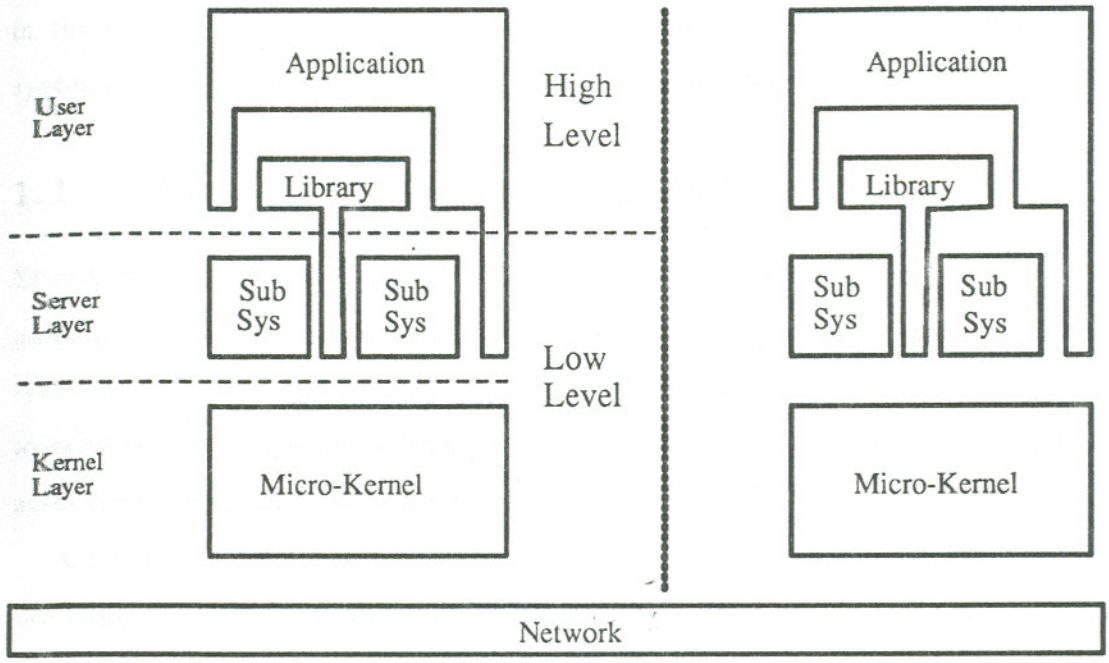


Figure 1.1: Typical distributed operating system architecture and interface levels.

High-level programs implement user-level applications built upon the low-level services.

The micro-kernel is "only" responsible for providing a set of basic, simple, and flexible tools. By providing services as separate processes, accessed by inter-process communication, micro-kernel operating systems offer advantages over monolithic kernel designs: location transparency, extensibility, and ease of debugging. With a micro-kernel, it is possible to develop new services at higher levels, test them, and then possibly incorporate them into the kernel to obtain higher performance[DKSK91].

The key primitive abstractions provided by message-passing kernels are ports and messages. Ports are end-points for communication and messages are units of communication. These abstractions naturally support point-to-point communication in which a message is sent to a single destination port. However, several message-passing operating systems have also recognized the need to support group communication, in which a message can be addressed to many destination ports[GLOR91, CZ85].

Group communication can be implicit, based on broadcasting to all ports, or it can be based on explicit multicast, in which case messages are sent to a sub-set of the ports

in the system. Multicast communication can be provided in message-passing operating systems through the abstraction of a port group[MVF+90].

1.1 Distributed Programming Problems

Even though distributed computing systems share no memory and communicate only by passing messages with unbounded delays, distributed application algorithms often need synchronization and coordination between node operations or events. This is difficult to achieve in such systems as there is no common time base and perfectly synchronized node clocks are impossible to achieve.

Central to solutions of the distributed synchronization and coordination problem lies the implementation and maintenance of consistent distributed global state. Each site's awareness of a consistent global system state allows programs to implement distributed programming primitives like barriers, locks, and semaphores which provide support for higher level paradigms such as serializable transactions, replicated data, or distributed shared memory[JB89].

Ordering of communication events has been shown key to determining consistent global state [AAH+85, Ray88, Lam78, Mat89, BSS90, JB89, RST91]. The overall strategy relies on considering sending or receiving each message as an event—if message communication protocols are utilized which "globally" order such events at each site in the system, a serial order on system events can be established.

The *event diagram* shown in Figure 1.2 depicts the message delivery ordering problem. P1-P3 represent processes(a.k.a., sites or nodes) viewed as a sequence of events which are transitions of local state which consume zero time. In Figure 1.2, M3 might be delivered at P3 before M1. This could be a problem if M3 contains M1-dependent information. The motivation for delineating between receive and deliver events in Figure 1.2 will be explained later in the section. At this point consider receive and deliver events to be the same action.

One example of the problem often shows up when reading internet news. There is

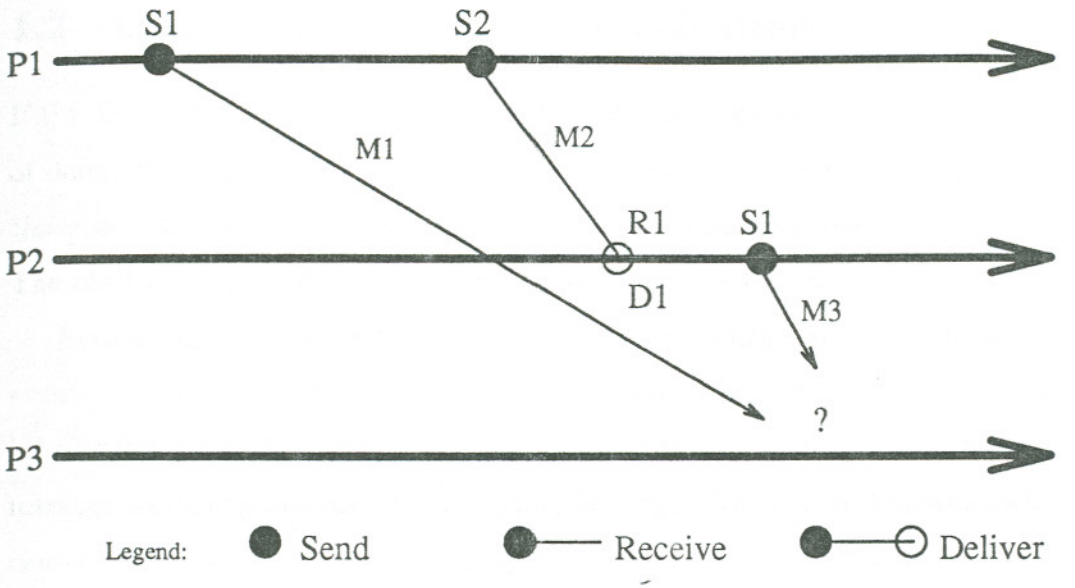


Figure 1.2: Event diagram illustrating message delivery ordering problem.

no guaranteed ordering on the delivery of news articles to a site, therefore it is possible to receive information in an order not corresponding to the original sending order(if one exists). Readers of news might receive a posting that references information contained in a posting not yet delivered to a site. Incorrect delivery order may be caused by effects such as re-routing of normal delivery paths, equipment failure, or message misplacement. The importance of the out-of-order news posting is dependent on the communication context. In certain contexts, pieces of mail arriving out of logical order might detract from the usefulness of the information.

Another example of the importance of ordered message delivery is the case of maintaining replicated data items, where operations are encapsulated by messages. As long as each process maintaining a replicated data object receives the messages (processes the operations) in the same order, they are able to maintain consistent copies of the object[PBS89].

1.2 Ordered Multicast Communication

If the distributed system had a perfectly synchronized global clock, the natural method of determining global event order would be to date each event by assigning a date or *timestamp* to it and compute event orderings by comparing the timestamps of events. The *clock* is the procedure that allots a timestamp to each event.

Even though the system has no global clock providing absolute ordering on system events, necessary global event ordering can be achieved for many distributed algorithms by *simulating* a global clock, using the clock to timestamp messages, and implementing message ordering protocols [SES89, RST91, BSS90]. Global clock simulation using logical clocks, *logical time*, was originally proposed in 1978 by Lamport[Lam78].

Lamport characterizes the fundamental nature of distributed computation as the *potential causality relationship* on distributed system events:

> An event structure is a pair (E,$\prec$), where E is a set of events, and '$\prec$' is an irreflexive partial order on E called the causality relation. e $\prec$ e' if one of the following conditions hold:
>
> **1.** e and e' are events in the same process and e precedes e'
>
> **2.** e is the sending event of a message and e' the corresponding receive event.
>
> **3.** $\exists$ e'' such that e $\prec$ e'' and e'' $\prec$ e'

The '$\prec$' relationship is often referred to as '$\rightarrow$', or the *happens before* relationship:

$$\text{send(m)} \prec \text{send(m')} \Longrightarrow \text{send(m)} \rightarrow \text{send(m')}.$$

Causal message ordering protocols extend to message reception events the causal relations existing on their send events[Ray90]:

$$\text{send(M1)} \rightarrow \text{send(M2)} \rightarrow \text{send(M3)} \Longrightarrow \text{recv(M1)} \rightarrow \text{recv(M2)} \rightarrow \text{recv(M3)}$$

Message ordering protocols are usually implemented by subdividing the receive event into two events: 1) *message reception* at a node or site by an ordering process, and 2)

message delivery at the destination process by the ordering process when the protocol ordering constraints on received messages are satisfied. The *causally ordered message delivery protocol* abstraction extends, to message delivery events, the causal relations existing on their send events:

$$\text{send(M1)} \rightarrow \text{send(M2)} \rightarrow \text{send(M3)} \Longrightarrow \text{dlvr(M1)} \rightarrow \text{dlvr(M2)} \rightarrow \text{dlvr(M3)}$$

Event ordering message delivery protocols have been implemented in user-level distributed application programming toolkits and found to be extremely helpful when programming distributed applications [BC90].

Distributed application programmers have also found the multicast and the multicast group abstraction useful[BCG91]. The *multicast* abstraction supports one process sending one message to multiple processes 'simultáneously'[1]. A *multicast group* is a collection of processes that are the destinations of the same message. It is convenient for the sender of a multicast message to address the message to a group identifier of one process, rather than keeping track of a list of destinations. The group could be thought of as a list of destination addresses. Multicast group addressing is portrayed in Figure 1.3 where Px sends a message (i.e. multicasts) to group Y and multiple copies of the message are sent transparently to P1-P3.

The *causal multicast* communication abstraction combines the causally ordered message delivery protocol and multicast group abstractions.

1.3 Distributed Operating System Support for Causal Multicast Communication

Distributed operating systems designers have recognized the importance of multicast-type communication and have provided various multicast group communication abstractions with varying message delivery semantics.

[1] Messages might not be delivered at the destinations simultaneously due to scheduling and indeterminate message passing delays.

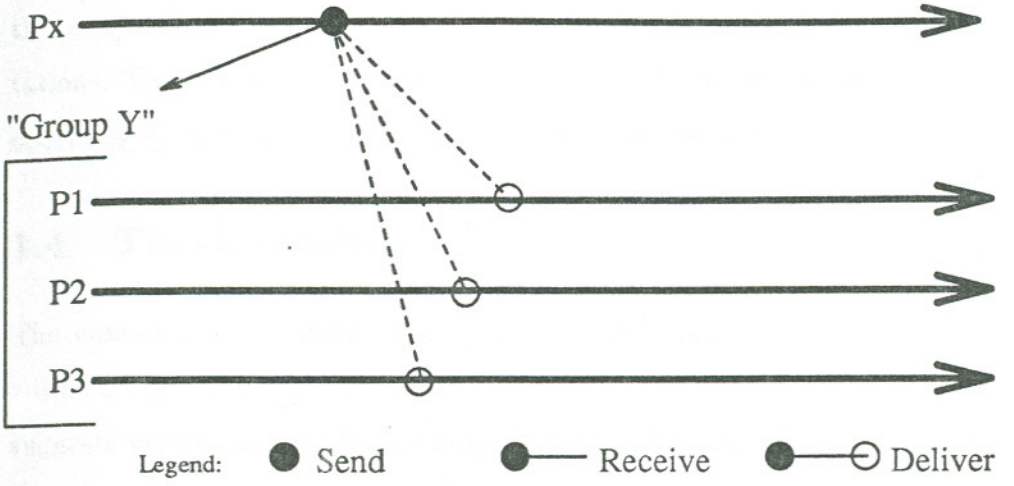


Figure 1.3: Multicast group communication and addressing.

We argue that it is useful to consider extending these message delivery semantics to include causal ordered message delivery. Benefits of doing so include the following:

1. Adding causal multicast support in the kernel to the port group message delivery functionality might reduce or eliminate the necessity of applications relying on application level ordered communication programming libraries, therefore saving application code complexity and size.

2. Communication performance can potentially be improved by moving causal multicast communication into the kernel, especially if causal communication is important and frequently used.

3. The ability of multicast software to directly access underlying hardware support for multicast would be improved.

In this thesis we identify the functionality necessary to support distributed application causal multicast communication and we investigate the architectural alternatives to implement this functionality in a micro-kernel architecture.

We construct an experimental system to study various architectural approaches. Each part of the experiment locates causal multicast support at different architectural levels corresponding to the implementation choices. By analyzing the performance results of

these experiments, we are able to predict relative costs of similar micro-kernel implementations. The results of this thesis contribute to the understanding of where distributed operating system causal multicast support should be located.

1.4 Thesis Outline

The communication abstractions provided by distributed operating systems and distributed application programming libraries are reviewed in Chapter 2. Chapter 2 also suggests various approaches to supporting causal multicast in distributed applications. Chapter 3 presents the experimental architectures used to investigate different approaches to causal multicast support. The experimental performance results, analysis, and predicted performance of micro-kernel implementations are presented in Chapter 4. We summarize and conclude in Chapter 5.

Chapter 2

Distributed Operating System and Application Abstractions for Communication

Both operating systems and application programming libraries hide the complexity of lower-level inter-process communication mechanisms by providing higher-level abstractions. By utilizing these abstractions, distributed application programmers can design, code, and reason about program behavior more efficiently than if each application provided its own primitive-hiding abstractions. Furthermore, the portability of application programs can be improved by coding with well-known and widely supported programming abstractions.

In this chapter we review the low-level, basic communication abstractions provided by typical distributed operating systems and we survey research in defining and implementing communication abstractions found convenient for programming distributed applications. As we conduct the review, we are particularly interested in:

1) identifying approaches to kernel supported causal multicast communication, and

2) identifying key causal multicast functionality necessary to support distributed applications.

2.1 Distributed Operating System Communication Abstractions

A recent trend in operating system development is the structuring of the operating system as a modular set of system servers built on top of a minimal micro-kernel, rather than using the traditional monolithic kernel structure[GLOR91]. This requires a stateless mechanism of inter-process communication, whether the processes are user processes or system servers. Limited connection-oriented resources such as pipes or sockets are not appropriate for this style of computing since one service might be provided by many servers for reasons such as reliability or load balancing. Micro-kernel operating systems, typically based on message passing, satisfy this requirement; messages go from process to process, and the physical location of these processes does not matter since the message passing system takes care of location. Indirect communication allows greater flexibility and robustness of operating system services. Indirection is usually implemented through facilities such as ports. These facilities are implemented in a manner that provides uniform local and remote inter-process communication. Since operating system services are also processes, calling of system services and communication between user processes both reduce to the uniform semantics of message-passing.

The Chorus system is an example of such an architecture. It relies on a minimal kernel that integrates distributing processing and communication at the lowest level, and it provides subsystem servers for standard operating system interfaces[GLOR91]. Besides Chorus, other examples of micro-kernel oriented distributed operating systems include Mach[ABB+86], Amoeba[MvRT+90], the V kernel[CZ85], and the x-Kernel[HP91].

2.1.1 Port and Message Abstractions

The kernel manages the exchange of messages between ports attached to processes. A message is a typed or untyped collection of data objects used in communication between processes. A port is a logical queue of messages protected by the kernel. Only one process can receive messages from a port, but all processes that have access to the port

(i.e. hold port send rights) can send messages to the port.

Ports are also used to represent kernel objects like processes, and user programs can manipulate any process, for which they have send rights, using remote procedure call (RPC) messages to the ports. This approach allows applications such as debuggers to manipulate other processes, even remotely, through the standard kernel interface.

The inter-process communication (IPC) interface is transparent because to the sender and receiver of a message, port rights carry no location information; the sender and receiver of a message know nothing about each other. Intermediate servers are allowed to transparently extend the IPC facility. Server-level extension servers, called network message servers, guarantee IPC message semantics across networks and take advantage of the transparency when forwarding messages across a network. Transparency also allows programs to intercept messages sent to others. Mach provides system calls to extract and insert port rights from/to messages. A debugger process could monitor all message traffic allowed by a port send right[Dra90].

Distributed operating systems usually provide services for network-wide port name registration and lookup functionality.

2.1.2 Communication Group Abstractions

The Chorus *port group* abstraction extends message-passing semantics between processes by allowing messages to be directed to an entire group of processes. This allows providers of a service to be selected from among members of a port group[MVF+90]. Ports can be dynamically grouped into port groups providing multicast or functional addressing facilities. A port can be a member of several port groups.

The following modes of addressing are available when sending messages to port groups:

- Broadcast to all ports in the group.
- Send to any one port in the group.
- Send to one port in the group, located at a specific site.

- Send to one port in the group, located on the same site as a given identifier.

The Mach *port set* abstraction is a group of ports that can *received* in parallel. If a thread receives on a port set, it receives the first message that appears on any of the ports in the set. Messages cannot be sent to a port set. A port can only be a member of one port set. Port sets let a server manage hundreds or thousands of ports. The Mach 3.0 single server UNIX subsystem uses approximately 2000 ports. Assigning a thread to every port managed would consume a prohibitive amount of space for stacks and other data structures [Dra90].

The V kernel[CZ85] provides multicast communication by allowing processes to join groups to which a member may multicast to other group members.

Psync is a low-level protocol designed to support a variety of high-level protocols and distributed applications[PBS89]. Psync is based on a *conversation* abstraction that provides a shared message space through which a collection of processes exchange messages. A well-defined set of processes, or participants, explicitly open a conversation, exchange messages through it, and close the conversation.

Communication Protocol Abstractions

While systems such as the V-kernel mandate a particular protocol or protocol suite, and Mach and Chorus move protocol responsibility into a network server, the x-Kernel provides explicit support for implementing network protocols. The x-Kernel views a protocol as a specification of a communication abstraction through which a collection of participants exchange a set of messages during a *session.* X-kernel[HP91] experience suggests that providing the right primitives in the kernel plays a major role in being able to implement protocols efficiently. In general, it is desirable to recognize tasks common to many protocols, and to provide efficient support routines that can be applied to those tasks.

Psync[PBS89] defers to higher levels functionality that not all applications need, such as fault-tolerance. Psync experience suggests that mechanisms that preserve timing

information should be implemented within the communication system, but the policy that dictates how the timing information is used to enforce various synchronization constraints belongs in the application. The Psync experience suggests that combining a low-level message timing information mechanism with a collection of library routines offers a simple and elegant solution to the communication needs of a broad spectrum of distributed applications.

2.2 High-Level Ordered Multicast Communication Abstractions

In this section we review ordered communication abstractions found to be useful for programming distributed applications.

2.2.1 Multicast Communication

There has been considerable interest in relieving the distributed programmer of the tedium in dealing with low-level inter-process communication mechanisms. Joseph and Birman in [JB89] review several popular distributed programming abstractions and come to the conclusion that a multicast communication abstraction is fundamental to distributed programming. The multicast abstraction is convenient for distributed programs that exhibit a high degree of interdependence. Multicast also allows the possibility of taking advantage of hardware broadcasting capabilities that could significantly reduce overall IPC message-traffic.

Multicast can be to some or all system sites. As defined in [JB89] we use the term *multicast* to refer to the delivery of a message to a list of destinations and the term *broadcast* to refer to the delivery of a message to all network sites, whether they are involved in the application or not.

2.2.2 Multicast Group Addressing

A *multicast group* is a collection of processes that are the destinations of the same message. It is convenient for the sender of a multicast message to address the message to a group of processes, rather than keeping track of a list of destinations. The group could be thought of as a list of destination addresses. Multicast group membership can be classified as: *static*—processes are assigned to groups before the application begins executing, or *dynamic*—processes may join or leave groups at will.

According to Cheriton and Zwaenepoel[CZ85], group communication cannot be implemented satisfactorily by sending single messages to group members for the following three reasons:

- The identity of all the group members may not be known to the sender of the message. With group multicast, a well-known group identifier can be used to represent the group membership.
- Transmitting a single message to each group member is less efficient than a corresponding group multicast. Numbers of packets exchanged can be reduced with group multicast. Group inter-process communication provides an application level abstraction of network multicast.
- Sending single messages to each group member does not provide the degree of potential concurrency among group members possible with group multicast.

In the V Kernel environment, a group of processes may be viewed as a single logical entity and operations may be performed on this entity. The V kernel uses process groups when implementing servers and the V kernel itself. The V system servers have been organized into groups according to the service they provide [CZ85].

2.2.3 Ordered Multicast Protocols

Group multicast ordering properties can be classified in order of increasing strength[GMS89]:

- **Single source ordering.** All messages from a single source process to the multicast group are received by the multicast group members in the same order *as sent.*

- **Multiple source ordering.** All messages sent from multiple source processes to a multicast group are received by the multicast group members in the same order.

- **Multiple group ordering.** All messages sent from multiple source processes are delivered in the same relative order even if they are addressed to different but overlapping multicast groups.

Schmuck [Sch88] classifies multicast protocols based upon ordering and reliability properties. Ordered multicast protocols guarantee message delivery ordering at different destination sites. Reliable multicast protocols guarantee messages will reach all destinations even if failures occur during the multicast.

Causal multicast protocols guarantee that causally related multicast messages are delivered at all destinations in an order that does not violate the causality relationship between corresponding multicast send events. Causal protocols typically require only one message exchange. Causal protocols can be made efficient in terms of total messages passed, and can operate in an asynchronous manner[Ray90]. The partial ordering provided by causal protocols can be extended to a total ordering (e.g. resolve ordering conflicts based upon sending process identifiers).

Atomic multicast protocols guarantee that single-source *and* multiple-source messages will be delivered in the same order at all destination sites. The potential causality relationship between communication events is not necessarily guaranteed. Two-phase protocols are typically used, requiring at least two message exchanges. They are synchronous in nature and provide a total ordering on events.

Combinations of the above protocols are possible. **Causal-atomic** multicasts provide a global message delivery ordering that also respects causality. These protocols could be constructed using an atomic protocol on top of an underlying causal primitive[BJ87][BSS90].

For efficiency, a multicast communication protocol is needed which guarantees the message delivery ordering to multicast groups, enforcing global event ordering only when necessary. Because atomic protocols order all events, they are more costly than causal protocols as they require synchronous communication whereas causal protocols can operate asynchronously.

Birman and Joseph[JB89] and Schmuck[Sch88] have found most classes of distributed algorithms to be implementable correctly using causal protocols and the remainder to be correctly implementable using atomic protocols. Schmuck has also shown causal protocols can often be used in place of atomic protocols where atomic protocols are specified[Sch88].

Reliable Multicast Protocols

Reliable multicast protocols guarantee that every message sent will eventually be received by all operational sites, despite processor failures. This property, *atomic message delivery*, is defined by Schmuck in [Sch88] as:

> If processor p sends a message m to a set D of destination sites, then the system will eventually reach one of the following two states:
>
> 1. For all $q \in D$: q has received m or q has crashed.
> 2. Processor p has crashed, and for all $q \in D$: q has crashed or q will never receive m.

Unordered multicasts can be made reliable by *flooding* or *message diffusion* and turned into reliable FIFO multicasts by adding sequence numbers to every message. Causal multicasts can be made reliable by flooding along with *piggy-backing* system-wide dependency information onto messages.

Due to the requirement that all processes must agree on the total ordering of atomic multicasts, reliable atomic multicast can only be achieved if assumptions about the asynchrony of the system are relaxed [Sch88]. Due to the fundamental system assumption

of unbounded message delays, a processor failure is indistinguishable from a very slow message channel. A process failure detection mechanism or a message-passing system with finite delays is necessary to implement reliable atomic multicast protocols. This thesis does not investigate issues of reliable multicasts.

2.2.4 ISIS Toolkit

Kenneth Birman's group at Cornell has developed a large package of distributed programming tools called the ISIS Toolkit. ISIS experience has shown that the key elements of distributed applications support include multicast communication, multicast group addressing, and ordered message delivery protocol abstractions. ISIS differs from other multicast group based systems because it integrates group membership changes with communication. For this reason and because ISIS is widely used and readily available, we have selected ISIS as the starting point for our experimental system.

ISIS Multicast Groups

ISIS depends on a coordinator process running at each site to take care of group membership changes. Although considerable high-level ordered communication work has been done [SES89, PBS89, LL86, CM84, GMS89], it appears only ISIS has addressed the problem of groups with potentially overlapping membership[BSS90]. Refer to figure 2.1. Group A's multicast precedes group B's multicast and should be delivered in that order at both P2-P3, which are members of both groups.

ISIS Communication Protocols

ISIS supports both causal and atomic multicast protocols.

Causal multicast is supported over direct interconnections (via UNIX BSD sockets) between group members for efficiency[BSS90, BSS91] as shown in figure 2.2. Messages along with piggy-backed vector timestamping[SES89] information are sent directly to group members, bypassing a coordinator process. Vector timestamps contain the logical timestamps of each group member of the groups the sender belongs to. Message delivery

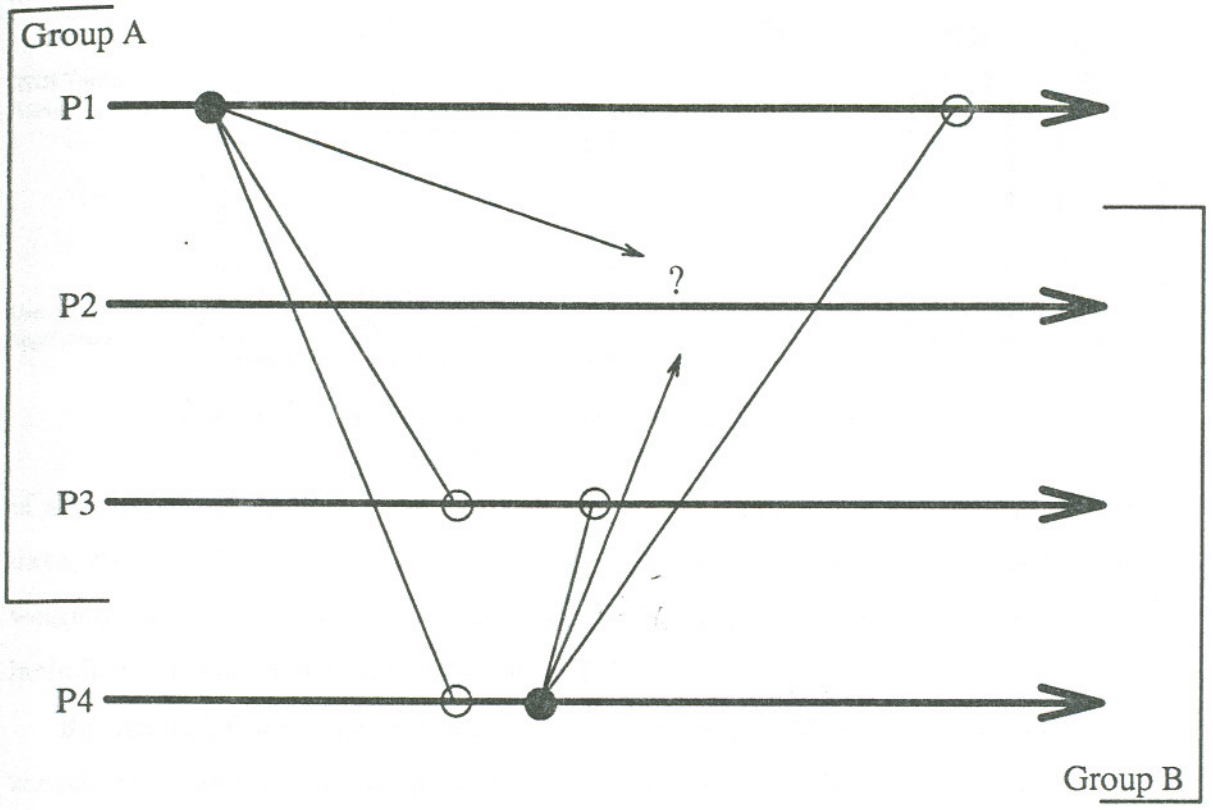


Figure 2.1: Overlapping multicast group communication ordering problem.

is delayed, *if necessary*, at the receiving process until causality would not be violated. The user-level ISIS library contains the multicast and causal protocol support necessary to do this and is linked to the user-level application.

ISIS implements a "fast atomic multicast" protocol by coupling causal multicast with a token passing scheme for assigning total orderings on message exchanges[BSS91].

ISIS Experience

ISIS experience has shown that moving protocols implementing communication groups nearer to communication hardware is important. This can help avoid unnecessary context switching on message reception, and ordering and timing of communication thread execution can be scheduled more efficiently[Fou91].

The trend in multicast communication research is to develop modular structures

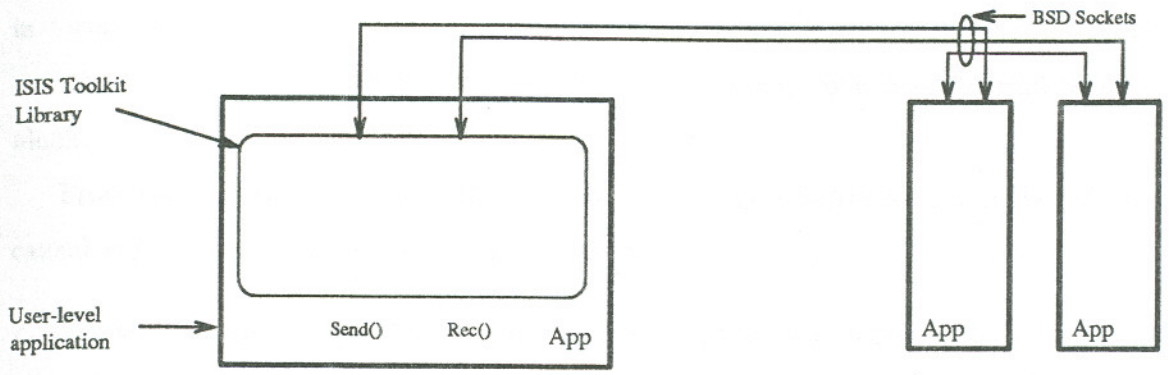


Figure 2.2: ISIS causal multicast communication architecture.

of separable facilities for group view management, causality enforcement, transporting data, etc.[BSS90]. Distributed systems programmers would select just the facilities and weights needed. This compositional programming style has been advocated by others, including x-Kernel and Psync developers[PBS89].

Birman and Cooper suggest in [BC90] what parts of ISIS would benefit from being kernelized. These pieces include the failure detection mechanism, the default multicast transport protocol, and certain aspects of the causal multicast protocol. ISIS is currently implementing a new architecture under Mach and Chorus. The architecture design envisioned is around a small module implementing multicast groups of ports, and a causal communication protocol. The minimal facilities would be chosen for the particular application.

2.3 Communication Abstraction Summary

In this chapter we have identified three key low-level and high-level communication abstractions. **Multicasting** of messages is a convenient communication abstraction for distributed application programming. **Multicast group addressing** is a convenient abstraction for distributed application programs and supports the transparent provision of distributed services. **Causal communication protocols** are adequate for most distributed application needs, can operate asynchronously, and can be made efficient

in terms of total messages passed. Communication protocols with stricter ordering semantics can be built using causal communication protocols as a fundamental building block.

From the survey described in this chapter several approaches to distributed system causal multicast support have become apparent:

1. Locate all support at the user-level in application programming libraries, as presently done in ISIS.

2. Locate all support in the kernel by extending operating system multicast port group abstractions with causal message ordering protocols.

3. Use a combination of kernel and application support, with kernel supported multicast and application supported message ordering.

It has also been learned that:

1) The kernel should provide primitives for supporting communication protocols but not necessarily the protocols themselves.

2) Placing multicast functionality in the kernel can improve application access to underlying multicast hardware.

Chapter 3

Salon: System for Architectural Experimentation in Causal Multicast Support

The main goal of our work is to experiment with the location of causal multicast support at various operating system levels. In Chapter 2 we identified the key causal multicast functionality necessary to support distributed applications efficiently. Also identified were three different approaches to providing applications with causal multicast support. The purpose of our experiments is to study the relative performance of these three approaches. The experiment will also verify that functionality *can* be moved out of the application level and causal multicast group communication still can be supported.

The experiment plan is to implement causal multicast functionality in a programming library, then use the library to implemental the three different architectural approaches to causal multicast support. The parts of the experiment are designed to model the following three cases: user-level, kernel-level, and split-level support.

The use of a common library supporting multiple levels of implementation also allows a more accurate comparison of architectural approaches. Upon measurement and analysis of the performance results for these three different architectures we will be able to prove or disprove our hypothesis that kernel support for causal multicast will show better performance than strictly application level support.

In this chapter we describe the communication abstraction functionality selected

for our experiment programming library, discuss the library design and application interfaces, and present the three experimental architectures designed using the library support.

The causal multicast support library we have built is called "Salon"[1].

3.1 Salon Library

The Salon library implements key causal multicast functionality identified in Chapter 2:

1) a causally ordered message delivery protocol,

2) multicast communication, and

3) multicast group addressing.

For simplicity, we do not support dynamic group membership changes or multiple overlapping groups, even though we believe these features are important for real-world use. The focus of the Salon library is to provide support for architectural experiments in a flexible manner, rather than a polished end-user interface library.

3.1.1 Salon Causally Ordered Communication Protocol

The causal communication protocol supported by the Salon library uses a *vector clock* timestamping mechanism and a causal multicast message delivery protocol. The resultant vector time based causal protocol is asynchronous in nature and efficient in terms of message ordering overhead.

We review each component of the Salon causal communication protocol separately:

[1] "A fashionable assemblage that is held by custom at the home of a usually prominent person and takes its character from the kind of notables who frequent it."— *Webster's 3rd New International Dictionary.* To us the term "salon" appears to capture both characteristics of membership and communication amongst membership and when used in the context of this thesis refers to a group of processes communicating in a causally ordered fashion.

Vector Clock Mechanism

The vector clock mechanism, an extension of Lamport's logical clocks, was developed independently by Mattern[Mat89] and Fidge[Fid88] in 1988, and reviewed in [BSS91].

A vector clock for a process p_i, denoted $\mathrm{VC}(p_i)$, is a vector of length n, (where n is the number of processes in our causally communicating distributed application). The vector clock is indexed by process-id, initialized to zeros.

- For each message m and corresponding send(m) at p_i, $\mathrm{VC}(p_i)[i]$ is incremented by 1.
- Each multicast message m sent by process p_i is timestamped with the incremented value of $\mathrm{VC}(p_i)$.
- When a process p_j *delivers* message m from p_i containing $\mathrm{VC}(m)$, p_j modifies its vector clock in the following manner: $\forall k \in 1..n : VC(p_j)[k] = max(VC(p_j)[k], VC(m)[k])$

Intuitively, the vector clock timestamp entries represent the number of messages sent to the destination. This characteristic is central to the message delivery protocol having enough information immediately upon message reception to know which messages should be delayed instead of delivered.

When a message is received it holds logical time information from other processes. Each process gets an optimal approximation of global time. The global view of system vector time provided to an outside observer is shown in Figure 3.1.

The rules for comparing vector timestamps are as follows:

1) $VC_1 \preceq VC_2 \iff VC_1[i] \leq VC_2[i],\ \forall i$

2) $VC_1 \prec VC_2 \implies VC_1[i] \leq VC_2[i]$ *and* $\exists i : VC_1[i] < VC_2[i]$

It has been shown that the vector clock mechanism represents a partial order which represents event causality precisely[Mat89]:

$$m \prec m' \iff VC_m < VC_{m'}.$$

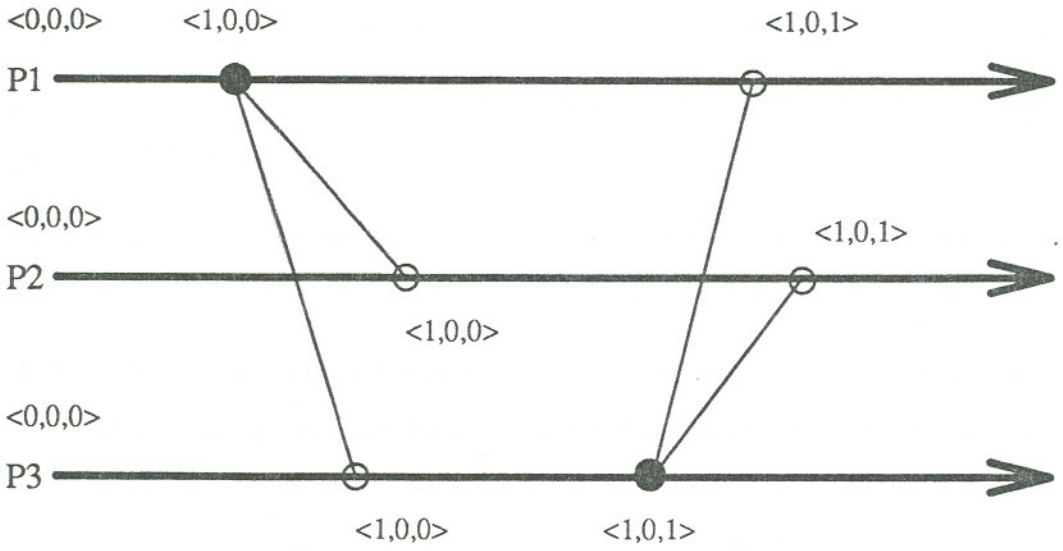


Figure 3.1: Global view of vector time.

Causal Message Delivery Ordering Algorithm

Schiper, Eggli, and Sandoz in [SES89] proposed a point-to-point causal message delivery protocol based upon the vector clock mechanism. This point-to-point message delivery algorithm has been extended to a multicast message delivery algorithm in [BSS91] as follows:

> On reception of message m sent by p_i and timestamped with VC(m), process $p_j \neq p_i$ delays delivery of m until:
>
> $$\forall k : 1..n : \begin{cases} VC(m)[k] = VC(p_j)[k] + 1 & \text{if } k{=}i \\ VC(m)[k] = VC(p_j)[k] & \text{otherwise} \end{cases}$$
>
> Process p_j need not delay messages received from itself.

Delayed messages are kept on a queue. When the message is delivered, the receiving process's logical clock entry in its vector clock array is updated by the vector clock mechanism so delivery of delayed messages might be possible.

Salon Causal Communication Protocol Implementation

A simplified vector-timestamping causal communication protocol can be realized by implementing two functions:

timestamp() This function increments the invoking process's logical clock entry in its vector clock array and piggybacks the result onto outgoing messages.

deliver() When a message is received, this function compares the message's piggybacked timestamp information with the destination process's vector clock. If causality would not be violated, the message is delivered to the process and the process's vector clock is updated. Otherwise delivery is delayed. After updating the vector clock contents, this function checks for delayed messages that might have become deliverable.

3.1.2 Salon Communication Group Abstraction

For simplicity, only one static non-overlapping group is supported by the Salon library. A group membership list is kept within each application's context data structure.[2] These lists associate a process ID with an operating system port identifier. When an application is started, it initializes an operating system port for itself. The application subsequently associated its process ID with the port ID and registers the name with the network name service.

Upon receiving a "group_init" message on its port from a distinguished group member, an application polls the network name service for port IDs of the group members, filling its group list.

Extensions of this static group semantic to dynamic over-lapping group membership would probably be necessary for more complex distributed system application needs. Examples of how this can be done are given in [BCG91].

[2] In a manner similar to Xt/X11 Athena Widget [MAS87][Ase88] design, Salon library applications declare a context data structure defined by the Salon library. All Salon library functions take an argument pointer to this context, and execute appropriate functionality based upon the context contents.

3.1.3 Salon Multicast Abstraction

For simplicity, all multicasts are to all members of the multicast group, including the sender itself. *Synchronous* and *asynchronous* multicasts are supported. Receivers of synchronous multicasts are expected to reply directly to the initiator of the synchronous multicast. The multicast sender can listen for as many replies as it wishes. There has been no attempt to avoid convergence of multicast replies as in [Ste91] and replies are not ordered. Issuers of asynchronous multicasts do not wait for replies and receivers do not send replies.

3.1.4 Salon Library Design

The Salon library is designed to simplify the task of transferring functionality between various distributed system levels.

All library functions are stateless, state being held in the invoker's context structure, and therefore amenable to multi-threading use. The Salon library is comprised of the following modules:

Msg This module contains the application program interface to the Salon library. It is operating system independent.

Proto This module contains the causal protocol functionality. It contains the timestamping and delivery functions for messages. It is operating system independent.

OS The micro-kernel operating system dependent message interface routines are held here. Message multicasting functionality is included in this module as well as port initialization, registration, name lookup, and functions to receive messages on ports.

X11 This optional module holds application GUI functionality. It contains simple X11 widget functionality for application use. Examples of support include scrollable text widgets, arbitrary command buttons and X11 event handling. This module is operating system independent.

Salon Library Interface

Applications, whether at the user or server-level, initiate a multicast with the **Send(cxtp, datap, length, type)** function call, where **cxtp** is a pointer to the invoker's context data structure, **datap** is a pointer to application data to be sent, **length** is the number of bytes of data, and **type** is defined as either synchronous or asynchronous.

Applications use the library function **Rec(cxtp, block)** to receive messages on their specific group portid context, where **cxtp** is as above, and **block** is defined to be either true or false depending on whether the library should block on the receiving port or not.

In contrast to the ISIS Toolkit multi-tasking approach, the Salon library leaves the responsibility and management of the user's receive port to the application. ISIS users register application functions to be invoked by the ISIS tasking system when messages are received.

The receiver of a synchronous message [3] (i.e. the sender expects a reply message) uses the Salon library

Reply(cxtp) function to issue a reply. Again in contrast to the ISIS Toolkit, the Salon library does not receive and buffer reply messages in the communication library for an application. For simplicity, all replies contain zero bytes (null messages) and Salon applications invoke the **RecReply(cxtp, block)** library function for collecting replies to synchronous multicasts.

Located in the OS module of the Salon library, **Multicast(cxtp, msgp)** is not normally invoked by the application program directly, instead it is invoked as one of the last functions in the library execution of the application's **Send()** invocation.

The Salon library **Timestamp(cxtp, msgp)** function is also not designed to be used by the application program, being located in the library protocol module. It is responsible for updating and timestamping outgoing messages with the application context's vector clock contents.

---

[3] Salon messages have a small header of protocol information with one of the fields specifying the type of the message, where type can be synchronous, asynchronous, group_init, or exit.

The library **Deliver(cxtp, msgp)** is called automatically by the library **Rec()** function when a message is received on the receiving port. The **Deliver()** function is part of the protocol module and is responsible for carrying out the message delivery ordering functionality of the library causal protocol. If delivery of the message to the application invoking **Rec()** is possible without causality violations, **Deliver()** calls an application receive message function pointer that has been initialized by the application, passes the received message, and updates the application's vector time context. Since updating the receiving application's vector clock context means some other delayed messages might be deliverable, **Deliver()** then checks and delivers additional deliverable delayed messages to the application.

3.1.5 Visualization Tools for Debugging and Demonstration

Early in the experiment, during the Salon library development phase, two servers were built. These servers provided project developers with debugging and visualization tools that were particularly useful when implementing and testing the ordered message delivery protocol of the Salon library. The servers are briefly described here, see the appendix for more details and a screen dump of application widget functionality.

Plot Server

The core of this server is a widget displaying Mattern's vector time event diagrams[Mat89]. The Plot Server widget displays a global view of system progress in a distributed system event diagram. The Salon library echos communication events to the Plot Server transparently to applications.

Perturber Server

The Perturber Server allows the user to delay messages sent from selected origins to selected destinations. Use of the Perturber Server allows developers and demonstrators to "reliably" perturb the delivery of individual messages, simulating actual system delays.

Salon Development Environment

Interactive development and testing of the Salon library was carried out using the Plot and Perturber servers with "multicast group" application widgets. The application widgets allowed developers to generate synchronous or asynchronous communication events on demand, locally or remotely, using the X11 Window System facilities [Sch87].

3.2 Causal Multicast Experimental Architectures

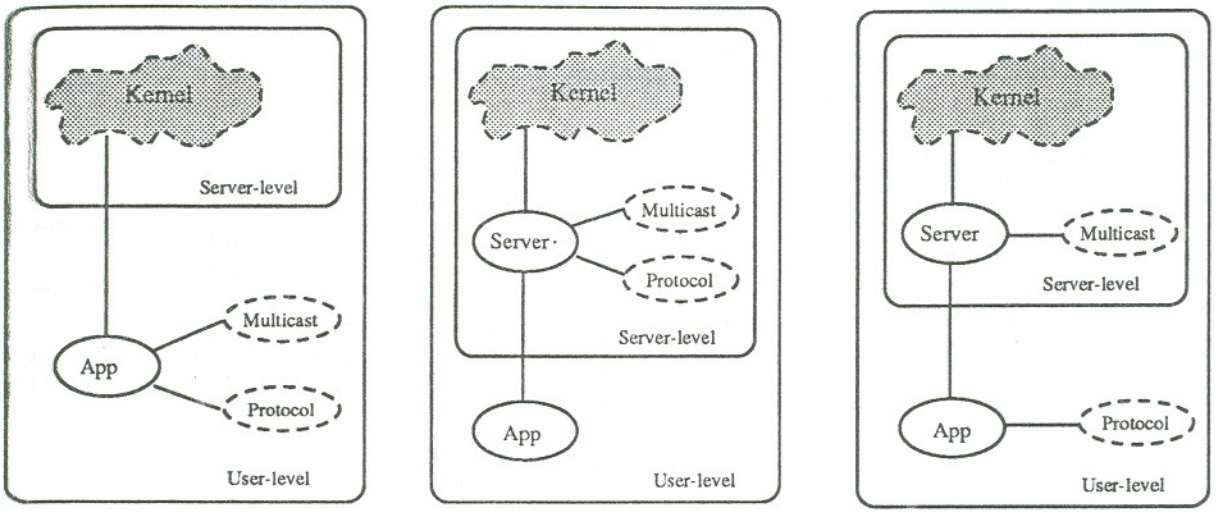
This section presents the three experimental architectures we have designed using the Salon library. We have varied the placement of library protocol and multicast functionality as shown in Figure 3.2. Each architecture was instrumented to identify the component costs of causal multicast communication operation and we use these results to reason about where support for causal multicast should be placed. Performance testing is discussed in Chapter 4.

In the first part of the experiment, the Salon library protocol and multicast support is linked into the application at the user-level. This architecture is similar to current popular approaches(e.g. ISIS) and its performance serves as a benchmark for the other two experimental architectures.

In the second part of the experiment, a single salon server is located on each node which provides all support for causal multicast. Each server exports an interface that allows its local clients (group members) to send and receive messages to/from the multicast group. To an application, this server-level architecture would be comparable to the kernel providing full support for causal multicast communication.

The third part of the experiment is a split-level salon with multicast functionality at the server-level, while protocol functionality remains at the user-level. This part of the experiment simulates kernel-level implementation of multicast delivery to port groups with user-level protocol support.

The following sections describe each of the experiment architectures in more detail.



User-Level Salon
Part I

Server-Level Salon
Part II

Split-Level Salon
Part III

Figure 3.2: Experimental architectures investigated.

3.2.1 User-Level Salon Implementation

The part I experimental architecture models distributed applications utilizing a user-level programming library similar to the ISIS Toolkit for causal multicast communication support.

The user-level implementation is depicted in Figure 3.3. After declaring and initializing their Salon context structure, applications register their ports, and send and receive multicasts by invoking Salon library functions. For simplicity reply ports and nullreply message paths are not shown.

3.2.2 Server-Level Salon Implementation

The purpose of the server-level experimental architecture is to investigate moving all causal multicast support out of distributed applications and into the kernel. This approach models the extension of the port group abstraction with causal multicast message delivery semantics.

The server-level salon implementation consists of one server salon per node and zero or more user-level applications. All protocol and multicast functionality is located at the

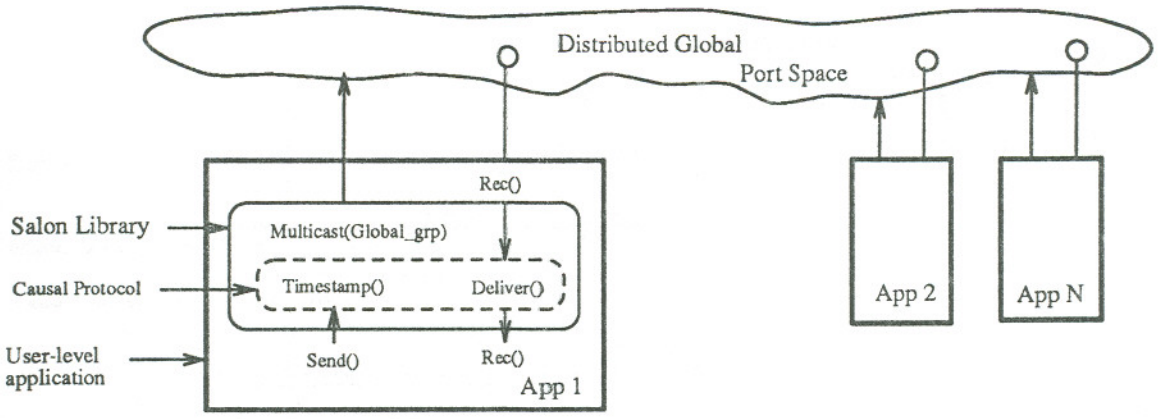


Figure 3.3: User-level Salon architecture.

server-level application as shown in Figure 3.4.

Operation of the server-level architecture is explained by tracing an application multicast. The application interface to the Salon library is identical to Part I. In this case the user-level **Send()** invocation will send the user-level message to the local Salon port (the server salon's well known local port). The user-level **Rec()** function receives on the application's receive port as before.

The server has a local and global thread which share a vector clock context for local applications. The local thread receives on its receive port, the server salon local port, by invoking **Rec()**. When a message is received on the local thread's port, the library calls the **timestamp()** function, and multicasts the timestamped message to the group members of the local thread's group context. In the case of the local thread, the group context is a global list of all the other server salons.

The server salon's global thread similarly calls the library's **Rec()** function to handle the reception of global messages. The **Rec()** function invokes library protocol to handle message delivery to local applications. When local message delivery is appropriate, the received global message is multicast locally to the local group members All user-level applications are members of the same group, all communication is multicasting to the full group membership so only one vector clock context needs to be kept for each group on each site. Individual messages sent locally are assumed to be received in FIFO order by

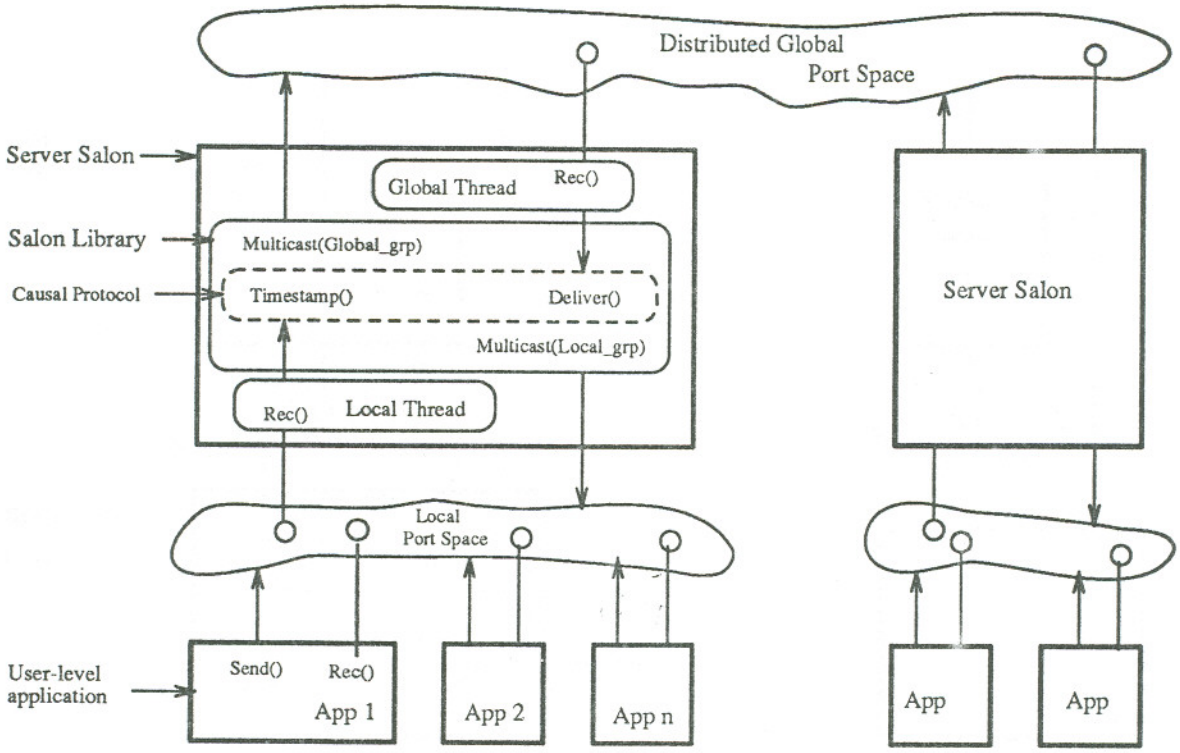


Figure 3.4: Server-level Salon architecture.

the message-passing system. This means causal message delivery orderings maintained by the Salon protocol at the server-level will be transferred to each local application process in the same order.

3.2.3 Split-Level Salon Implementation

We investigate the combination of user-level protocol and kernel-level multicast functionality with our third experimental architecture. The split-level experiment models a kernel-level multicast port group abstraction combined with an application level protocol. The multicast servers shown in Figure 3.5 represent port groups with FIFO ordered multicast semantics. Causal protocol support is linked to applications at the user-level.

The architectural differences are again transparent to the user-level application—they call on **Send()** and **Rec()** just as in the other architectures.

The Salon library user-level invocation of the **Send()** function does not multicast

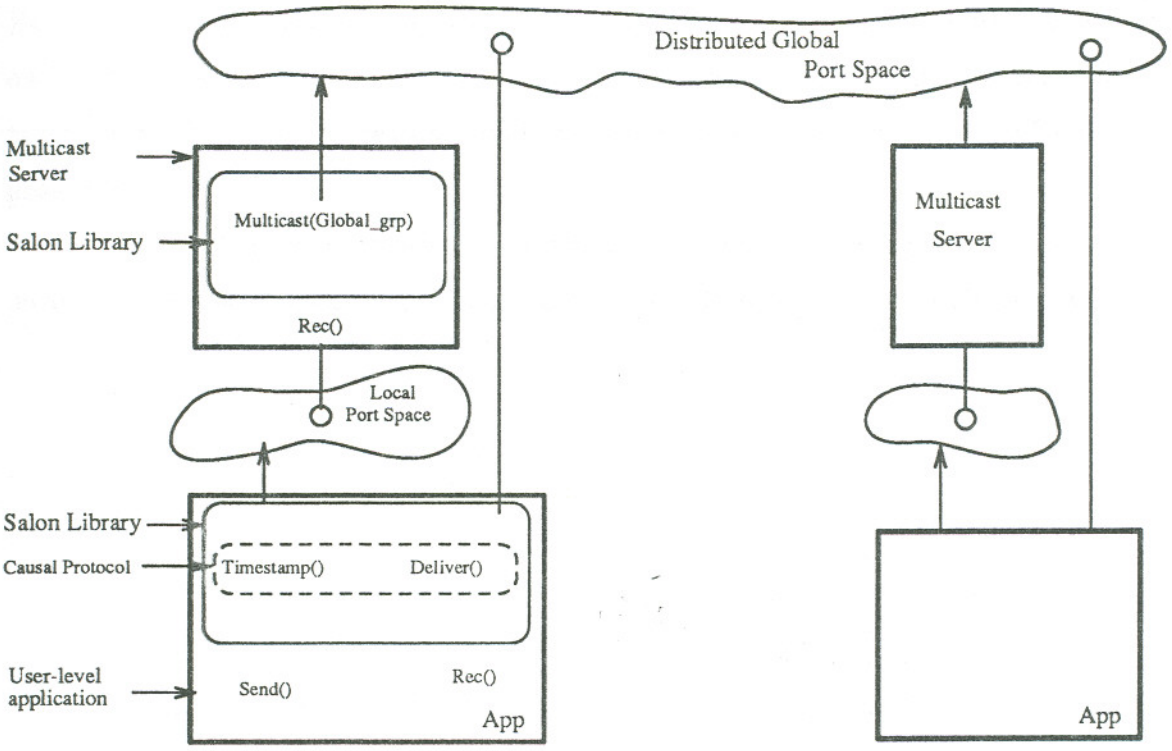


Figure 3.5: Split-level Salon architecture.

the message to a group, but sends the timestamped message to the local multicast server port. The user-level applications receives on its receive port which in this case is globally known. Received messages are delivered when delivery is appropriate as before.

The multicast server calls the library **Rec()** function which immediately sends the message on to the multicasting function of the library. The group context of the multicast server is a global list of the user-level applications.

3.3 Summary of Experimental Setup

We have introduced an experiment to investigate the relative performance of different approaches to causal multicast support. We have motivated the use of the Salon causal multicast programming library and presented its design, functionality, and application interfaces. Three experimental architectural approaches to providing operating system support for causal multicast have been designed and implemented using our Salon library.

Each experimental architecture models one of the causal multicast support approaches outlined at the end of Chapter 2. Analysis of the performance results for these three architectures allows us to reason about the relative merits of these three different approaches.

In the next chapter, we review the metrics for the experiment, present and analyze the measured performance results, then project the results to real-world implementations.

Chapter 4

Results & Analysis

In this chapter we present the results of our experimentals and explain their significance for micro-kernel based implementations of causal multicast. From this analysis we will be able to predict the relative performance of kernel supported causal multicast.

Our implementation environment consisted of four Microvax II workstations running Mt Xinu Mach 2.6, connected by a 10 Mb/sec ethernet. Since only 4 nodes were available for performance measurements, architecture scaling performance was studied by running multiple processes per node. We did not find references to such types of scaling performance measurement in our literature review. Due to the inherent nature of distributed programming, distributed applications are usually designed to be located on discrete nodes. We will comment on the scaling performance results, but we will limit most of our analysis to the "normal" distributed programming environment case of one application per node.

The selection of performance test types was influenced by previous work benchmarking ISIS performance[BSS91]. Both types of multicast, synchronous and asynchronous, were studied for each architectural experiment. Synchronous(RPC) performance testing indicates round-trip communication *delay* including message emission and reply reception. It is useful to estimate total multicast group communication overhead. However, synchronous causal multicasting is not typically used by distributed applications; indeed asynchronous operation is one of the main advantages of causal communication protocols. Asynchronous performance testing gives an estimate of communication *latency* or how much time applications can expect to be delayed when issuing multicasts. The

causal multicast system bandwidth, or maximum number of messages per second, is also indicated by asynchronous testing. Asynchronous performance gives a better indication of the multicast group communication delay at the sender due to the causal multicast protocol itself. Both rate of emission at sender (send rate) and rate of reception at receiver (throughput) were measured in the asynchronous case.

ISIS v2.1 performance figures were measured for reference purposes and are included in Appendix D. These figures were used to verify the reliability of early performance testing results.

4.1 Results

For each architecture, the fundamental group size results along with scaling performance results are given. Fundamental results, made up of simple combinations of group membership and location, are presented in tabular form for ease of reference and will be used later when we derive component cost figures in the analysis section. Scaling results are presented in graphical form to visually assist the reader in identifying relative performance between the experimental architectures for varying application data packet sizes. Relative point-to-point architectural performance differences as well as local and remote causal multicast overhead trends can be observed upon viewing the scaling plots.

Workload

For all Salon measurements multicasts were sent to all group members, including the sender of the multicast. For each architecture the test procedure was the same: (1) start up the particular number of salon server and receiver(slave) processes, (2) start up a distinguished multicast sending process(master), (3) initialize the master's multicast group list, (4) the master multicasts a group_init message to the group, (5) the master runs the synchronous or asynchronous multicast test loop, averaging appropriately and logging results. It should be noted, we are not purposefully delaying message transmissions and receptions or otherwise creating a communication sequence of events which

violate causality. In fact, we doubt causality will be violated during our testing as all multicasts originate from the same source. If causality were to be violated, the testing would also reflect the overhead due to handling delayed messages by the Salon protocol module. This approach is similar to ISIS performance measurements in [BSS91].

The master process, group member one, is responsible for initiating multicasts, timing performance, and logging results. In the asynchronous case, throughput is measured and logged by one distinguished receiver. For each type of performance test the size of application messages was varied. Application data consisted of either 0 bytes (Null vm_allocated data pointer) one vm_allocated page(Microvax vm_page_size = 4096 bytes) or two vm_allocated pages (8192 bytes). Mach MSD2.6 allows data to be sent out-of-line(i.e. pointers to data are passed in messages). For each individual multicast, memory was vm_allocated, written to ensure allocation, multicast out-of-line, and upon reception written. Since the Mach message-passing system implements copy-on-write semantics, writing the out-of-line data ensures both local and remote data copying.

Two main test loops were built to gather synchronous and asynchronous performance results. Both loops saved the time-of-day, sent out a number of multicasts, got the time-of-day again, then calculated and logged performance results for the loop run. All testing was performed running in multi-user UNIX operating system mode, therefore involuntary context switching performance could not be entirely avoided. Machine loading was "unofficially" minimized as much as possible, though. Successive runs of the loops were repeated with short sleep intervals to reduce effects of involuntary context switching by allowing time for other system services and possible applications to run.

When testing scaling performance, first the total number of processes was varied from 1 to 12 on one node. Then two nodes were used, alternating the startup location of each new process. This procedure was repeated with three nodes then four nodes, generating for each experiment a total of 48 data points.

For fundamental performance measurements, servers were not started on each node unless application processes were being tested on that node. For scaling performance measurements, the total number of servers running equaled the total number of nodes

under test in all cases. For example, when measuring the distribution overhead by running four nodes, with a total of three application processes running it would only be necessary to run servers on three nodes, nonetheless we chose to start a server on the fourth node in these types of cases anyway.

Collecting the scaling performance measurements proved the most challenging, requiring extensive use of remote shell programming in the remote application cases. Testing was controlled by shell scripts, started after the work day, and running overnight. In retrospect, process startup locations should have probably been shuffled to average out individual node performance differences due to different memory sizes , node loading, etc.

Additional shell script tools were developed to extract data results, and gather simple statistical means and variances. Discussion of the significance of our measurements is continued in Appendix E. The mean results of all individual measurements made inside of all test loops of all test runs made over our scheduled testing period of one month are presented in the tables and scaling plots.

Time Measurement Accuracy

The gettimeofday(2) function call was used as a the wall-clock for generating elapsed time results. The gettimeofday() clock resolution is 10ms for a Microvax II. This means the a clock reading for a 10ms real-time interval is between 5ms and 15ms, for effective overall accuracy of +/- 50%. By increasing the real-time interval being measured, the effective accuracy is increased: 100ms yields +/- 5%, 200ms +/- 2.5%, etc., With large group sizes, especially in the remote cases, clock accuracy has less impact than measurement accuracy. But with small group sizes, where communication time is short (e.g. less than 50ms), clock readings only +/- 10-50% accurate reduce confidence in measurements. It is necessary to increase the number of multicasts during small measurement intervals to increase measurement confidence. However the time interval can only be increased to something less than 100ms, the typical UNIX timeslicing interval. A context switch while conducting a timing measurement can have a significant effect on measurement

accuracy, especially when measuring elapsed time values of less than 100ms.

The combined effects of the clock accuracy and involuntary context switching measurement accuracy means we have varying degrees of confidence in the *absolute* value of our individual performance measurements. When measuring fundamental group size synchronous performance, the number of multicasts per timing loop is selected to keep the total measured time interval greater than 50ms and less than 80ms. Therefore, the corresponding fundamental synchronous figures we report are accurate to at least +/- 10%. The majority of detailed analysis is based on these figures. When testing synchronous scaling performance, where measured values are typically higher, clock accuracy is less of a problem than context switching effects. We have not attempted to quantify these context switch impacts on communication performance. If we only compare performance across architectures by looking for *relative* performance differences, we claim these measurement accuracy impacts tend to balance out.

In the asynchronous performance measurement tests, we have sent a fixed number of multicasts per loop regardless of how much time has elapsed. It would not make sense to delay multicasts when we are trying to measure the maximum number we can send per second. Certainly, context switching will have some impacts, but as we are primarily interested in relative performance comparisons, they also are expected to balance out.

Results Presentation Key

For each architecture we present results for fundamental group sizes and locations in detail and provide scaling results in a visual manner via graphic plotting methods. The table Dest.(multicast destination) column is explained as follows:

Self Master multicasts to the group of which it is the only member. Replies to itself in synchronous cases.

Loc/Self Master multicasts to a group size of two—itself and a local process, and collects replies from each in synchronous cases.

1Rem/Self Master multicasts to a group size of one—itself and 1 remote process, and collects replies from each in synchronous cases.

2Rem/Self Master multicasts to a group size of three—itself and 2 remote processes, and collects replies from each in synchronous cases.

3Rem/Self Master multicasts to a group size of four—itself and 3 remote processes, and collects replies from each in synchronous cases.

Displayed in the tables is the mean measurement time for various group sizes and process locations. In the synchronous case the times indicate the average time it takes to allocate application data, build a message, multicast the message to all group members (Dest.) and collect replies from all group members including the multicast sender. Receivers of messages modify data to ensure copying before replying to multicasts. Writing all received data pages by the receiver contradicts Mach's lazy message-passing approach but is the only way to make sure we are measuring the performance impact of increased application data packet sizes.

In the asynchronous case, the numbers indicate the average multicasts per second send rate for the multicast sender. Application data is allocated, a message is built, and multicast to all group members, including the multicast sender for each message sent. Receivers do not reply, but modify data received to ensure data copying.

Scaling performance for each experiment case is presented in the accompanying graphics plots. One axis represents the total number of nodes the applications are distributed across, therefore gives an indication of the distribution communication overhead. Another axis represents the total number of applications participating in the multicast(group membership size). The performance along this axis indicates local communication overhead. Averaged mean measurement values for synchronous round-trip message time (RPC time) and asynchronous multicasts per second are displayed on the vertical axis.

4.1.1 Experimental Synchronous Performance

For each experimental architecture, the mean time in milliseconds of how long it takes to multicast a message to each destination process and receive a null reply at the sending process from each receiver is presented. Tables 4.1, 4.2, and 4.3 present fundamental synchronous performance following by synchronous scaling performance in Figures 4.1, 4.2, and 4.3.

Detailed results discussion takes place in the analysis section, but a few performance comments will be given here: The noticeable jump in all fundamental synchronous 3Remotes/Self, 0 byte cases is due to context switching and is discussed in the analysis section. From the synchronous scaling plots, it is seen:

- One process/node performance is similar across architectures.
- Local(one node) performance is similar across architectures.
- The step between one node and two node performance across architectures indicates remote message-passsing overhead increased cost.
- The server-level salon scales better remotely(due to reduced overall message traffic) but has fixed overhead when processes are not running on a node.

Table 4.1: Synchronous user-level salon fundamental performance.

Dest.	App Data in Bytes		
	0	4096	8192
	ms	ms	ms
Self	4.31	20.13	27.06
Local/Self	8.51	32.64	44.01
1 Remote/Self	40.33	113.15	167.83
2 Remotes/Self	60.25	215.44	270.12
3 Remotes/Self	118.48	302.13	361.78

Table 4.2: Synchronous server-level salon fundamental performance.

Dest.	App Data in Bytes		
	0	4096	8192
	ms	ms	ms
Self	12.55	34.39	42.38
Local/Self	16.36	46.68	57.96
1 Remote/Self	50.45	131.30	177.52
2 Remotes/Self	69.24	227.75	287.71
3 Remotes/Self	117.74	309.54	374.36

Table 4.3: Synchronous split-level salon fundamental performance.

Dest.	App Data in Bytes		
	0	4096	8192
	ms	ms	ms
Self	8.41	26.75	33.88
Local/Self	12.45	39.25	50.43
1 Remote/Self	44.80	118.00	173.11
2 Remotes/Self	64.93	204.94	275.18
3 Remotes/Self	117.62	297.12	370.30

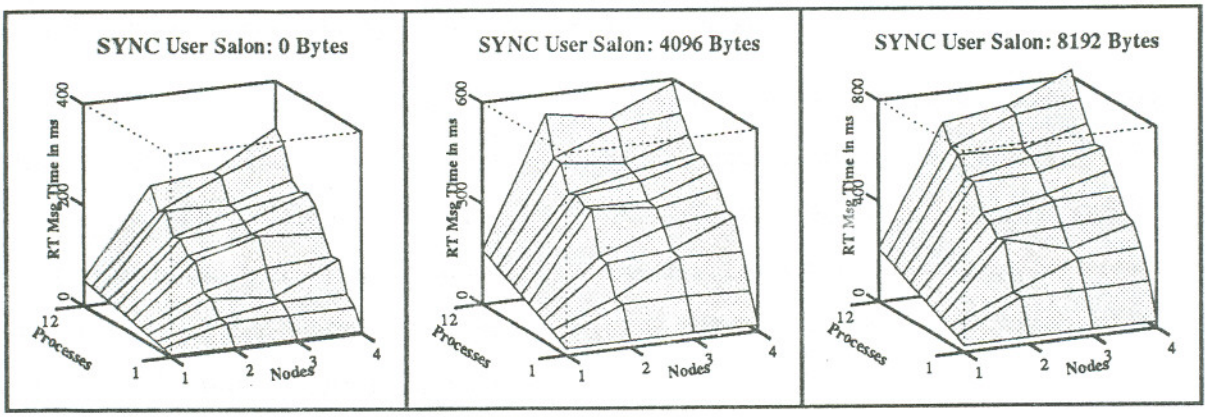


Figure 4.1: Synchronous user-level salon scaling performance.

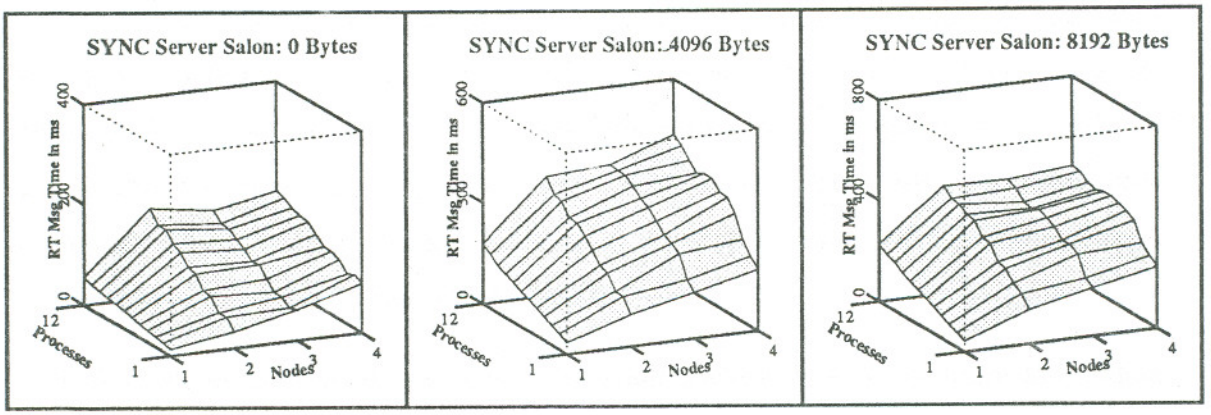


Figure 4.2: Synchronous server-level salon scaling performance.

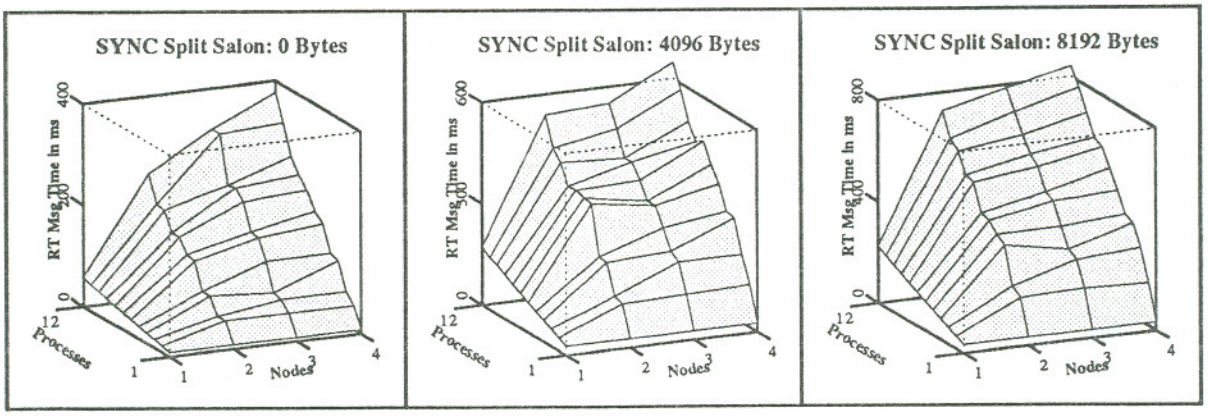


Figure 4.3: Synchronous split-level salon scaling performance.

4.1.2 Experimental Asynchronous Performance

Fundmental asynchronous performance for the various experiments is presented in Tables 4.4, 4.5, and 4.6. Both send rate and throughput measurement results are presented in the tables. The *SRate* is the mean number of multicasts per second emitted and its reciprocal is the time in milliseconds per multicast. The mean *TPut* figure represents message throughput or message receive rate at one group member; sender if self, other member is local or remote.) is presented.

In Figures 4.4, 4.5, and 4.6 asynchronous scaling performance results in Mcasts per sec, or multicasts per second, are presented in graphical form for the various experimental architectures and application data packet sizes. The actual number of application messages sent during one Mcast per sec interval equals the corresponding group size times the Mcast per sec figure. In all asynchronous cases the sender does not wait for replies after each multicast and the receivers do not send replies. Note: The orientation of the processes axis and nodes axis has been changed in hopes of making viewing easier.

Asynchronous scaling plots reveal:

- Overall, asynchronous type communication performance scales more flatly than the synchronous type. We attribute this effect to the absence of replies.
- The server-level architecture scales better remotely but has reduced performance when processes are not running on a node due to fixed server overhead.

Table 4.4: Asynchronous user-level salon fundamental performance.

Dest.	App Data in Bytes								
	0			4096			8192		
	SRate		TPut	SRate		TPut	SRate		TPut
	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s
Self	283.50	3.53	293.51	52.26	19.14	54.05	38.14	26.22	39.17
Loc/Self	170.19	5.88	170.23	32.92	30.38	34.37	24.39	41.01	25.20
Rem/Self	42.80	23.37	41.33	13.25	75.46	13.09	7.43	134.63	7.33
2Rem/Self	20.05	49.38	17.72	7.64	130.89	7.52	5.76	173.61	5.77
3Rem/Self	12.12	82.50	11.81	6.03	165.84	5.89	4.25	235.29	4.12

Table 4.5: Asynchronous server-level salon fundamental performance.

Dest.	App Data in Bytes								
	0			4096			8192		
	SRate		TPut	SRate		TPut	SRate		TPut
	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s
Self	83.33	12.00	86.23	29.54	33.86	30.31	24.21	41.30	24.87
Loc/Self	69.70	14.35	70.06	22.03	45.40	22.60	17.63	56.72	18.26
Rem/Self	32.31	30.95	30.78	10.36	96.55	10.01	6.57	152.10	6.07
2Rem/Self	19.30	51.81	20.35	7.48	133.69	7.29	5.12	195.31	4.92
3Rem/Self	11.15	89.69	10.26	5.84	171.23	5.60	4.10	243.90	3.89

Table 4.6: Asynchronous split-level salon fundamental performance.

Dest.	App Data in Bytes								
	0			4096			8192		
	SRate		TPut	SRate		TPut	SRate		TPut
	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s
Self	132.67	7.54	139.52	38.17	26.20	39.19	29.89	33.46	30.78
Loc/Self	97.22	10.29	96.82	26.47	37.78	27.16	20.31	49.24	20.86
Rem/Self	37.11	26.95	35.33	11.99	83.37	11.53	7.21	138.61	6.70
2Rem/Self	15.62	64.02	15.18	7.88	126.90	7.51	5.73	174.52	5.52
3Rem/Self	9.68	103.31	9.18	5.31	188.32	5.21	4.25	235.29	4.05

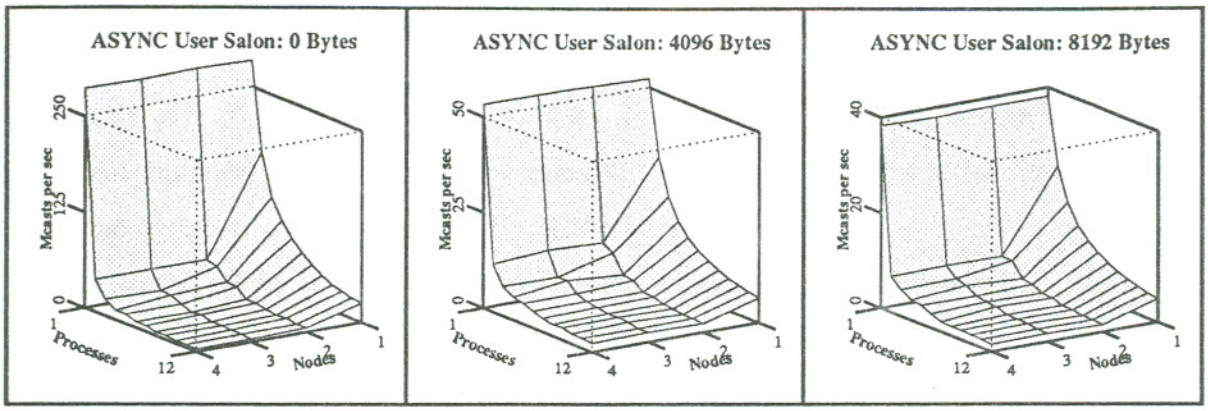


Figure 4.4: Asynchronous user-level salon scaling performance.

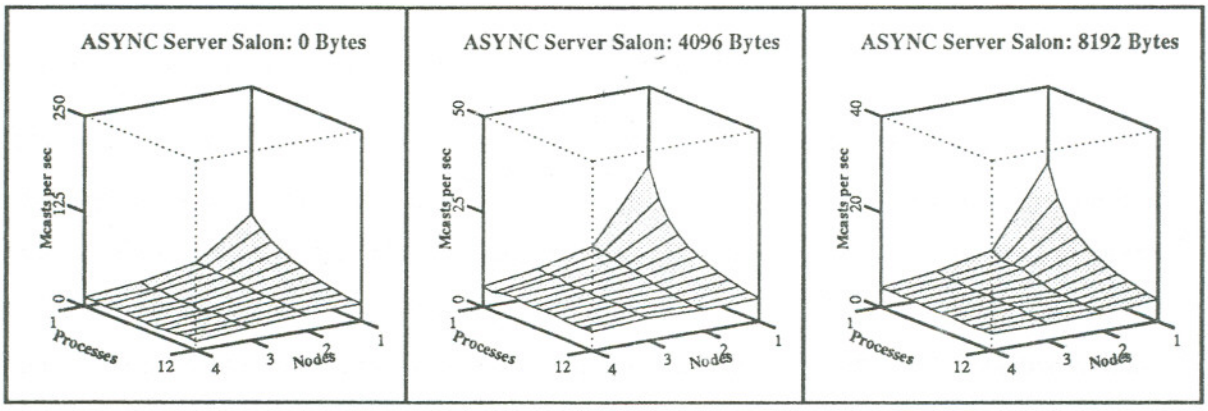


Figure 4.5: Asynchronous server-level salon scaling performance.

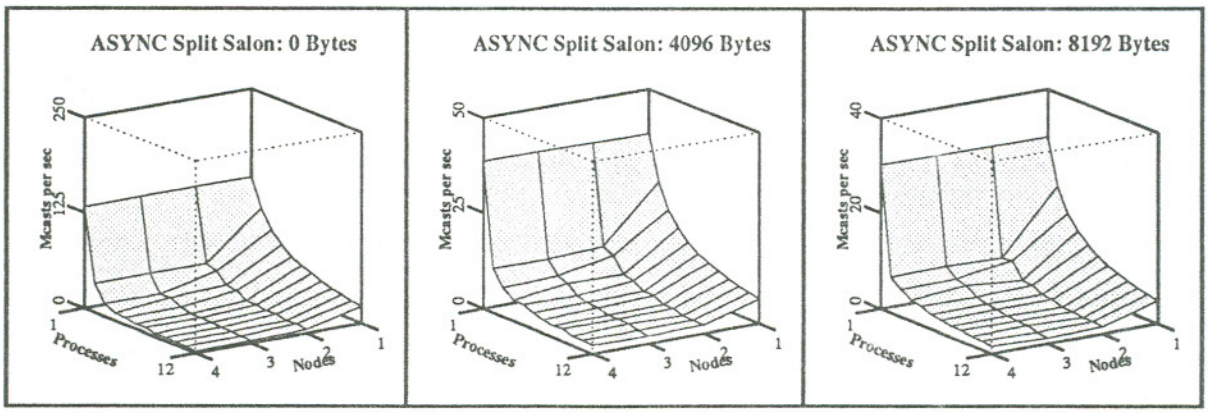


Figure 4.6: Asynchronous split-level salon scaling performance.

4.2 Performance Analysis

The goals of our performance analysis are to: 1) identify component costs of causal multicast support, and 2) predict the relative performance benefits of locating causal multicast support functionality directly in the kernel.

Our Salon experimental architectures and performance results give a basic understanding of communication patterns and functionality overhead necessary to support simple causal group communication. The experimental architecture performance results combined with the results of additional Mach message-passing performance tests enable us to quantify causal multicast component costs.

4.2.1 Performance Analysis of User-Level Salon

The cost of strictly application-library-based support is made up of two pieces: the message-passing system cost and the causal functionality cost. We begin by running tests to find the Mach MSD2.6 message-passing cost. In these tests we re-used as much Salon library code, looping, averaging, timing, and logging code as possible, simplifying the implementation and increasing the correlation to Salon experiment measurements.

To help isolate individual communication component costs, we measured the performance of point-to-point local and remote message exchanges. In addition to the fundamental Salon test cases, we measured the performance of the following cases:

Local Master sends to a local process and receives the reply in synchronous case.

Remote Master sends to a remote process and receives the reply in synchronous case.

Mach overhead scaling performance was not measured. As in the Salon test cases, tests were run for application data sizes of 0, 4096, and 8192 bytes (i.e. 0,1,and 2 virtual memory pages). New vm_allocated pages were allocated and written for each multicast and written on reception to ensure Mach data copying.

Mach Overhead—No Protocol Headers

First the overhead of passing only vm_allocated pages with no protocol headers was measured. The synchronous and asynchronous results are shown in Tables 4.7 and 4.8.

Table 4.7: Synchronous Mach MSD2.6—no Salon header.

Dest.	App Data in Bytes		
	0	4096	8192
	ms	ms	ms
Self	2.24	17.58	24.72
Local/Self	5.08	29.06	40.18
Remote/Self	36.08	100.57	150.36
Local	3.02	18.14	25.46
Remote	33.46	91.07	164.40

Table 4.8: Asynchronous Mach MSD2.6—no Salon header.

Dest.	App Data in Bytes								
	0			4096			8192		
	SRate		TPut	SRate		TPut	SRate		TPut
	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s
Self	696.42	1.44	714.30	59.40	16.83	61.06	41.12	24.32	42.30
Loc/Self	381.88	2.62	374.00	37.79	26.46	39.65	27.07	36.94	28.34
Rem/Self	49.54	20.18	46.44	14.35	69.67	13.89	8.35	119.69	8.60
Loc	696.45	1.44	593.08	63.01	15.87	62.82	48.47	20.63	48.98
Rem	54.06	18.00	49.14	16.52	60.52	15.39	8.68	115.25	8.40

From the Mach results without a protocol header we can estimate some fundamental message-passing overhead expenses. It costs 2.24ms to send a synchronous message to yourself including the null reply cost. The asynchronous(no reply) send rate to either yourself(or local) is 1.44ms. Subtracting the asynchronous result from the synchronous result equals a null reply to yourself cost of 0.80ms.

1 Mach local send = 1.44ms

1 Mach local reply = 0.80ms

Other than the Mach message header cost, the majority of the null reply cost is

system call overhead; the time it takes for a process to enter and leave the kernel. This gives an idea of system call overhead. We modified our synchronous test loop to execute a getpid() system call instead of issuing a Salon synchronous multicast. Getpid() does nothing but return the caller's process identifier. Ousterhout in [Ous90] reports this test indicates the cost of entering and leaving the operating system kernel. Ousterhout measured 207us per getpid() call done on a MicroVaxII running Ultrix 3.0. We measure a time of 340us for a MicroVaxII running Mach MSD2.6.

Therefore, the majority of the reply to yourself cost must be two kernel switches, one for sending the reply and one for receiving the reply or two kernel entry/exit times or 0.68ms.

In the same manner, subtracting the asynchronous cost of sending a 0 byte message locally, 1.44ms, from the synchronous cost of 3.02ms gives the cost of sending a local reply, 1.58ms. Sending a local message and a local reply involves a context switch between application processes. The result of subtracting the cost of a reply to self, 0.80ms, from the local reply cost of 1.58ms, 0.78ms, indicates the cost of context switching to and back from a local processes that are blocked on their receive ports (and are therefore already in kernel mode). We can estimate the cost of switching to another context at one half that of 0.78ms, 0.39ms. Context switching back and forth between non-blocked processes costs one/half additional kernel entry/exit time at each process. This makes the total cost of a user mode to user mode context switch approximately twice that plus the one kernel entry/exit time or 2 * (0.39ms) + 0.34ms = 1.12ms.

1 context switch(one way) = 0.39ms

1 user mode back to user mode context switch = 1.12ms

An estimate of copying expense can be made using the asynchronous results. The difference between the destination self and local send rate performance is the overhead due to copying received data into the user space of the receiver. This cost amounts to 16.83ms - 15.87ms = 0.96ms in the 4096 byte case and 24.32ms - 20.63ms = 3.69ms in the 8192 byte case. Ousterhout in [Ous90] reports memory bandwidth figures of 3.5 Kbytes/ms for a Microvax II.

Mach Overhead—With Protocol Headers

For simplicity all messages sent in our Salon experiment contain 4 fields (except for null replies), although in some cases not all fields are filled:

1 Required Mach msg_header_t data structure,

2 Salon message data header (20 byte length—describes Salon specific protocol information: Salon message type, sender, destination),

3 Salon message vector timestamp (56 byte length—fixed size for our experiment's maximum group size of 12 processes. 8 bytes for protocol overhead + 4 bytes per process), and

4 Salon application data pointer. Application data in messages is passed out-of-line, even though we are forcing data to be copied by writing all pages. Data copying between the application and Salon layers is reduced by passing pointers to application data.

A Mach msg_type_t field is also required for each Salon field in the experiment's Mach message structure to type the associated information for the Mach message-passing system.

Items 2 and 3 can be considered Salon causal protocol overhead so we performed another fundamental Mach benchmark test to separate out the message-passing overhead expense of adding message fields for Salon causal protocol support. The message structure for this test holds all of the above four fields. The Mach with Salon header results are shown in Tables 4.9 and 4.10.

Fundamental Salon message passing costs can be calculated by combining the results of Mach with and without the protocol header. Zero byte results show accompanying all Salon messages (except replies) with the 76 byte Salon protocol header costs 2.62ms - 2.24ms = 0.38ms or approximately 0.40ms.

Salon protocol message header cost = 0.40ms

The main difference between user-level and Mach w/header performance is the addition of the Salon causal protocol overhead. We can estimate the cost of adding the

Table 4.9: Synchronous Mach MSD2.6—Salon header.

Dest.	App Data in Bytes		
	0	4096	8192
	ms	ms	ms
Self	2.62	17.88	25.40
Local/Self	5.82	29.68	40.90
Remote/Self	37.36	96.20	152.60
Local	3.38	18.68	25.58
Remote	34.92	97.32	132.44

Table 4.10: Asynchronous Mach MSD2.6—Salon header.

Dest.	App Data in Bytes								
	0			4096			8192		
	SRate		TPut	SRate		TPut	SRate		TPut
	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s
Self	550.00	1.82	611.12	58.21	17.18	59.67	40.10	24.94	41.19
Loc/Self	319.24	3.13	298.78	36.78	27.19	38.84	26.63	37.55	28.31
Rem/Self	46.76	21.39	44.38	14.30	69.93	13.95	7.56	132.31	7.23
Loc	550.00	1.82	539.90	61.84	16.17	61.90	48.04	20.82	48.31
Rem	51.03	19.60	47.67	16.26	61.49	15.13	9.19	108.83	8.56

Salon causal protocol by comparing destination self Mach w/header costs from Table 4.9 and user-level Salon costs from Table 4.1. These include the timestamping and delivery costs. We see that adding the protocol adds 4.31ms - 2.62ms = 1.69ms in the 0 byte case. Analyzing profiling data for these test runs shows the protocol overhead approximately evenly split between timestamping and delivery cost. One might expect delivery costs to increase when distributed applications have multiple originators of multicasts. In these cases it is probable that messages would be occasionally (possibly frequently) delayed to maintain causal communication amongst the group, although we did not estimate the increased overhead.

Salon causal protocol cost = 1.70ms

Timestamp() cost = 0.85ms

Deliver() cost = 0.85ms

The cost of multicasting 0 bytes to local/self is the Mach w/header cost plus one timestamp at the sender and a deliver at each receiver. If we subtract the Mach cost from the measured result, another estimate of protocol cost can be made: 8.48ms - 5.82ms = 2.66ms. Dividing costs equally between timestamping and delivery gives a measured cost per timestamp or deliver of 2.66ms / 3 = 0.89ms. We did not attempt to analyze the remote case results.

Sending a synchronous remote message takes 33.46ms, asynchronous remote takes 18.50ms, therefore a remote reply takes around 14.96ms, or 15ms.

1 Mach remote reply = 15.00ms

Remote message-passing costs dominate the remote/self destination case. If remotely destined messages were always sent before local messages, remote processes could begin their processing of messages slightly earlier, therefore replying sooner in the synchronous case.

As a check on our reasoning, let us consider the Mach no-header, destination local/self cost. The cost of synchronously multicasting to a local process and yourself should be about the equal to the cost of the same send in the self case added to the local case. For 0 byte packet sizes our results add up to 5.26ms. Adding the fundamental component costs separately gives 2 local sends + 2 local replies + 2 one-way context switches = 2.82ms + 1.60ms + 0.78ms = 5.20ms. The measured result is 5.08ms which is within +/- 5% of our calculation.

Local/self results are not equal to local + self results for all packet size cases because we only allocate and build one message for each multicast. Working backwards from our measured results with this finding we can estimate our test application's fundamental Mach allocation and message building costs. Salon test applications use identical functionality when allocating and building messages so this cost is worthwhile to calculate.

Mach no-header, 0 byte, self + local = 2.24ms + 3.02ms = 5.26ms. This figure less the measured local/self result equals the overhead of declaring and building a Mach message with no application data: 5.26ms - 5.08ms = 0.18ms. Another way to look at it is that 0.18ms represents the savings per additional multicast destination over a simple

linear increase in overhead.

Basic Mach message structure overhead = 0.18ms

We can cross check on our local reply cost figure now: The 0.80ms local reply cost is approximately equal to the basic mach message overhead of 0.18ms plus 2 kernel entry/exits of 0.68ms or 0.86ms.

Due to the large variance of remote communication measurement especially when sending pages of data, it is difficult to analyze remote vs. remote/self figures. (See Appendix E for a discussion of our data.) In order to do so with any confidence would require more stringent data collection techniques, characterizing simultaneous network loading and factoring it out, and monitoring lower-level communication layer statistics.

We can get an idea of how the user-level salon architecture scales locally by examining one node scaling performance. User-level salon 0 byte synchronous overhead is increased by the local/self cost, 8.51ms, minus the self cost, 4.31ms, or 4.20ms per local process. We can examine x node/x process performance to get an idea of how the architecture scales remotely. The first remote node adds one remote copy send, 19.60ms, plus a remote reply, 15ms, to the cost of sending to self with protocol to get a 41.96ms total cost for a multicast. The next remote node adds approximately another remote copy send cost and a small indeterminate remote reply cost to the self plus one remote node cost to get a 60.98ms total cost.

This formula does hold adding a fourth node. Adding the same remote overhead for the fourth node puts the sending process into jeopardy of involuntary context switching time as the estimated figure is approaching the end of the typical 100ms UNIX process time slice interval. Upon examining the logged data output[1], we found that the involuntary context switching rate more than doubled for the sending process. We attribute the 39.40ms non-linear jump in overhead when adding the fourth node in the 0 byte, 3 Remotes/self case shown in Table 4.1 to this additional cost.

[1]The UNIX getrusage(2) system call was used to log resource utilization for the sending process along with the Mach MSD2.6 host_ipc_statistics(2) system call being used to log IPC message traffic statistics.

4.2.2 Performance Analysis of Server-Level Salon

The kernel-level and split-level approaches share additional message-passing and server process overhead that is not present in the user-level approach. Application messages are doubly-redirected by servers.

Depending on the remote message-passing copy semantics, there might be additional message copying at the server-server level even though servers do not read or modify pages of applications data. Local destination multicast application data does not have to be copied from the source to destination until it is written at the local Because of this we need to be able to estimate the Mach cost of sending and receiving a message without data copying. We modified our basic Mach overhead testing code to only allocate one message and the corresponding application data packet per test run, instead of per multicast. Receivers of this multicast do not modify the application data so in local destination cases there is no application data copying. Remote destination multicast application data is still copied due to Mach remote message-passings semantics [WT88]. Evidence of this remote copying semantic can be seen by comparing Table 4.10 remote figures with Table 4.12 remote figures.

Table 4.11: Synchronous Mach MSD2.6—Salon header, no application data copying.

Dest.	App Data in Bytes		
	0	4096	8192
	ms	ms	ms
Self	2.40	4.94	4.92
Local/Self	5.31	10.40	10.46
Remote/Self	38.45	92.87	143.68
Local	3.06	5.48	5.60
Remote	35.99	89.76	144.48

Because the server-based architecture functionality is provided by the same Salon library code as in the user-level architecture, we build the server analysis upon the user-level analysis. Mach kernel, context switching, message-passing, and Salon message and protocol costs are the same. However, the server-based architectures do introduce a new

Table 4.12: Asynchronous Mach MSD2.6—Salon header, no application data copying.

Dest.	App Data in Bytes								
	0			4096			8192		
	SRate		TPut	SRate		TPut	SRate		TPut
	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s	Mc/s	ms/Mc	Mc/s
Self	618.05	1.62	696.44	241.88	4.13	255.51	234.95	4.26	247.85
Loc/Self	348.21	2.87	323.40	130.59	7.66	133.82	127.25	7.86	129.61
Rem/Self	46.74	21.39	43.92	17.41	57.45	15.46	9.39	106.53	8.60
Loc	611.10	1.64	576.38	258.25	3.87	243.27	258.02	3.88	239.60
Rem	50.82	19.68	46.08	16.96	58.96	15.25	9.79	102.16	8.82

cost— overhead due to the server processes themselves.

Functionally, local and global server-level threads and split-level threads are the same; They listen on a port, receive and multicast messages. The server-level threads also do some protocol work but we have already estimated this cost. We will analyze split-level server performance to estimate the cost of a server thread; having only one thread, it should make estimation easier.

To arrive at an estimate for server thread cost we will work backwards from information known so far and the 0 byte case destination self split-level measurement result shown in Table 4.3, 8.41ms. The additional overhead added by the split-salon architecture over the user-salon architecture is one local no-copy message, 1.62ms, and one full-context switch, 1.12ms. Therefore, whatever cost remains after subtracting this known overhead from the measured split-level result indicates the thread cost, which works out to be 1.35ms.

Server thread cost = 1.35ms

With this server thread cost figure, we can identify the component costs of the 0 byte, destination self and local/self cases. In addition to the user-level cost of 4.32ms and 8.51ms respectively, we are adding the costs of two local no-copy message, 3.24ms, two full context switches, 2.24ms, and two server thread expenses of 2.70ms. These cost estimates add up to a total self cost of 12.50ms (measured 12.55ms) for the self case and 16.79ms (measured 16.36ms) for the local/self case.

The additional overhead in the remote/self case is equal to the additional overhead in the self and local/self cases plus the cost of one copy message from the remote server to the remote destination and the context switch from the same. User-level cost 40.33ms + 2 context switches 2.70ms + 2 no-copy messages 3.24ms + 2 threads 2.24ms + 1 remote context switch 1.12ms + 1 remote copy message 1.82ms = 51.45ms (measured 50.45ms).

The same analysis can be applied to the 4096 and 8192 byte destination self and local/self cases to yield estimated self performance of 33.33ms (measured 33.88ms) and 40.52ms (measured 42.38ms) and local/self performance of 45.84ms (measured 46.68ms) and 57.47ms (measured 57.96ms). We will not attempt to estimate remote 4096 and 8192 byte performance.

Again, we can get an idea of how the server-level salon architecture scales locally by examining one node scaling performance. Server-level salon 0 byte synchronous overhead is increased by the local/self cost, 16.36ms, minus the self cost, 12.55ms, or 3.81ms per local process. This 3.81ms is made up of one local copy send cost, 1.82ms, one context switch, 1.12ms, and one local reply, 0.80ms, for approximately 3.74ms. Adding a third node to remote/self server-level performance adds basically one remote copy send cost, 19.60ms, to the remote/self communication overhead, 50.79ms, accounting for the measured 3 node/3 process performance result of 68.24ms. But again, adding the fourth remote node puts us into involuntary context switching jeopardy, and we attribute the additional 32.86ms overhead jump to this impact. Logged experiment resource data verifies that the context switching rate has increased approximately five-fold in this case.

4.2.3 Performance Analysis of Split-Level Salon

The split-level salon architecture is simpler than the server-level architecture and its performance falls between the strictly user-level and the strictly server-level architectures. The additional overhead in the split-level case is about one half the amount of server-level additional overhead due to one half the message indirection by server threads.

User-level performance plus one half the additional server-level overhead equals 8.41ms (measured 8.41ms), 26.73ms (measured 26.75ms), and 33.79ms (measured 33.88ms) in

the 0, 4096, and 8192 byte destination self cases and 12.59ms (measured 12.45ms), 39.24ms (measured 39.25ms) and 50.74ms (measured 50.43ms) in the destination local/self cases. Estimated remote/self 0 byte split-level performance is 44.42ms (measured 44.80ms).

Finally, we examine the split-level salon architecture local scaling performance as before. Split-level salon 0 byte synchronous overhead is increased by the local/self cost, 12.45ms, minus the self cost, 8.41ms, or 4.04ms per local process. This 4.04ms is made up of one local copy send cost, 1.82ms, one context switch, 1.12ms, one local reply, 0.80ms, plus one deliver, 0.85ms, for approximately 4.59ms.

Adding a third node to remote/self server-level performance again adds basically one remote copy send cost, 19.60ms, to the remote/self communication overhead, 46.01ms, which accounts for the measured 3 node/3 process performance result of 62.85ms. Again, the 35ms additional jump when adding the fourth remote is attributed to context switching.

4.3 Predicted Implementation Performance

In this section we will predict the relative performance of the following three approaches to providing support for causal communication in distributed operating systems: 1) all support in user-level libraries, 2) kernel-level multicast port groups with causal multicast message delivery semantics (modeled by our server-level Salon architecture), and 3) kernel-level multicast port groups with user-level causal message delivery protocol located in application (modeled by our split-level Salon architecture).

We base our predictions on the premise that the real-world kernel-level architecture and functionality would be similar to our experimental architectures'. Realizing actual implementation details might differ, we do feel our projected systems reasonably represent the core functionality necessary to support causal multicast.

4.3.1 Projected Causal Multicast Application Library Performance

We have directly implemented this case and reported on its performance. Without operating system multicast support, we predict the performance of other implementations of user-level causal communication will be similar to our user-level salon results. The ISIS causal multicast performance is said to be near the underlying limits of non-multicasting hardware [BCG91] and Salon user-level performance is comparable to our measured ISIS performance included in the appendix.

Small improvements might be made by optimizing message multicasting to yourself, or in the protocol efficiency area by optimizing vector clock access and reducing timestamp lengths, but the dominant cost is point-to-point message-passing overhead, especially in the remote destination cases.

4.3.2 Projected Port Group Multicast with Causal Protocol Performance

Intuitively, a kernel-level approach is most desirable in cases of multiple group members residing on individual nodes as the protocol delivery overhead is reduced and there is an overall reduction in message traffic overhead, especially in the remote cases, due to server-server high-level traffic instead of point-to-point application message passing. This effect can most readily be seen upon examination of the asynchronous, multi-node/multi-process cases. Another advantage of centralizing protocol support is only one delayed message queue needs to be located on a node versus duplication of message queues by each group member on the node. This could result in memory savings due to reduced duplication of messages on a node.

A hypothetical kernel-level causal multicast system is shown in figure 4.7. The approach is to add causal protocol functionality for timestamping and delivering messages at the kernel-level at all nodes. Messages suitable for causal delivery would be forwarded to a port group comprised of causal multicast port group member ports.

An advantage of using the port group abstraction is that underlying hardware multicast can be utilized transparently by applications. Applications typically do not have

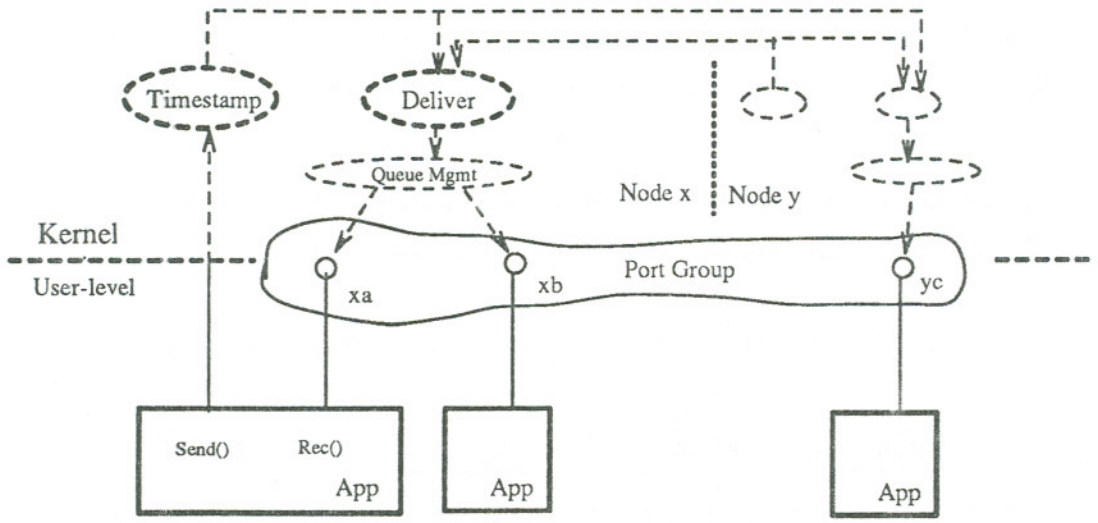


Figure 4.7: Hypothetical kernel-level causal communication support.

access to underlying support for hardware multicast.

In this approach, application code would be simplified but at the expense of kernel complexity. Placing causally ordered communication policy in the kernel seems to disagree with micro-kernel philosophy as not all message exchanges would need or use causal multicast guaranteed ordering.

In terms of performance, savings over the user-level approach include reduced kernel entry/exit costs and some reduction in message-passing overhead. We feel there should be some additional small reductions in message-passing overhead due to the kernel being able perform similar operations back-to-back instead of in response to a series of system calls when multicasting. We can only estimate that this will be a small value, possibly about the cost of a basic Mach message expense, 0.18ms. When there is more than one process per node there is a reduction in protocol overhead.

Destination self performance is identical to user-level salon performance for all data sizes. One protocol deliver cost, 0.85ms, is saved in all local/self destination cases over user-level salon performance. This is because messages are not handed off to the port group multicast function until delivery is acceptable.

Because the kernel is now responsible for multicasting, there will be a reduction

of one kernel entry/exit 0.34ms plus the estimated kernel efficiency savings estimated above 0.18ms. The savings will only be realized when sending messages to the second and subsequent group members.

Estimated synchronous performance is shown in Table 4.13. Figures in parenthesis represent percentage of relative performance improvement over the user-level approach.

Table 4.13: Estimated synchronous causal multicast port group performance.

Dest.	App Data in Bytes		
	0	4096	8192
	ms	ms	ms
Self	4.31 (0)	20.13 (0)	27.06 (0)
Local/Self	7.14 (16.10)	31.27 (4.20)	42.64 (3.11)
1Remote/Self	39.81 (1.29)	112.63 (0.46)	167.31 (0.31)
2Remotes/Self	59.21 (1.73)	214.40 (0.48)	269.08 (0.39)
3Remotes/Self	116.92 (1.32)	300.57 (0.52)	360.22 (0.43)

The only case in Table 4.13 where the percentage improvement over user-level performance is greater than 2% is destination local/self, which we consider to be unimportant. Overall, we consider the meager projected performance improvements shown in the table insufficient to justify consideration for kernel implementation.

4.3.3 Projected Port Group Multicast with User-Level Causal Protocol Performance

Our multicast port group/user-level protocol architecture was designed to investigate real-world implementation of a high-level causal protocol and low-level multicast service such as provided by Chorus. A hypothetical real-world implementation based on utilizing the port group multicast abstraction for message multicasting is shown in figure 4.8.

The model is conceptually cleaner than the kernel model. This approach has the advantage of not burdening the kernel with causally ordered communication support. With the multicast functionality in the kernel, there again is the opportunity for the kernel to exploit a hardware multicast mechanism.

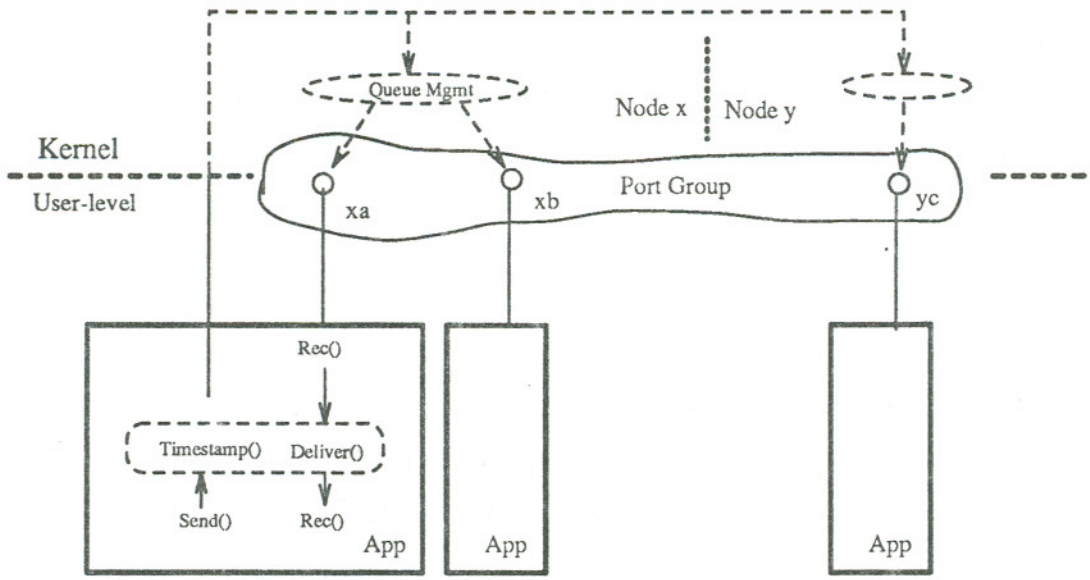


Figure 4.8: Hypothetical multicast port group/user protocol support.

Predicted destination self performance is identical to user-level salon performance for all data sizes. For the other destination cases, for each multicast message sent to a group size greater than one, one kernel entry/exit switch, 0.34ms, is saved plus one basic Mach message cost, 0.18ms.

Projected performance is shown in Table 4.14.

Table 4.14: Estimated synchronous port group multicast with user-level protocol performance.

Dest.	App Data in Bytes		
	0	4096	8192
	ms	ms	ms
Self	4.31 (0)	20.13 (0)	27.06 (0)
Local/Self	7.99 (6.11)	32.12 (1.60)	43.49 (1.81)
1Remote/Self	39.81 (1.29)	112.63 (0.46)	167.31 (0.31)
2Remotes/Self	59.21 (1.73)	214.40 (0.48)	269.08 (0.39)
3Remotes/Self	116.92 (1.32)	300.57 (0.52)	360.22 (0.43)

Interestingly enough, except for the uninteresting multi-process per node cases, projected performance of the split-level approach is the same as the strictly kernel-level

approach. We can conclude from comparing Table 4.13 and 4.14 performance results that there is no performance improvement made by adding causal protocol support in the kernel. The minimal improvements in performance are due only to kernel supported multicast.

4.4 Summary

For the one application process per node process distributions we have assumed to be the most important for distributed applications, we have been unable to prove that either of our kernel supported causal multicast approaches has superior performance over keeping all causal multicast functionality at the user-level. (However, for other process per node distributions, other conclusions might be possible.)

We feel we have made significant progress toward understanding where the appropriate location is for causal multicast support. The most important factor to deciding where causal multicast functionality should be located is the multicasting facility. Presently the majority of expense to causal multicast is not the protocol but message-passing expense, especially in the remote cases. This finding is confirmed by our statistical analysis reported in the appendix. The reason our kernel multicast implementation experiments had low user-level performance is they still serialize the transmission of messages. If messages could be passed in parallel to remote locations with a hardware multicast facility, multicast communication performance would be enhanced greatly.

Both the kernel-level and split-level approaches would provide easy access to this hardware multicast facility. Application located multicasting would probably not be able to access this hardware as easily and would need to continue to simulate multicast with serialized point-to-point messages. Moving multicast functionality from the application into the kernel will result in a reduction of application code size/complexity and duplicated functionality between distributed application members.

We have found the overhead introduced by the causal protocol is small, 1.70ms. Even if we assumed the asynchronous message-passing performance of a hardware multicast

assisted multicast to be a flat 20ms/multicast as shown in the remote destination case of Table 4.12, the causal protocol would be only 1.70ms/21.70ms = 7.8% of the total cost. We would question whether the potential improvement would justify its addition of application-specific complexity to the kernel.

Chapter 5

Conclusions

The fundamental goal of this thesis is to gain an understanding of where causal multicast support should be located in distributed operating systems. We began by reviewing communication abstractions found useful when programming distributed systems at the operating system level and at the application level.

Multicast communication, including the port group multicast abstraction, has been found to be convenient at the distributed operating system level and causal multicast group communication has been found to be a convenient programming abstraction at the application level. Key distributed application causal multicast functionality includes multicast message-passing, multicast group addressing, and causally ordered message delivery protocols.

We identified three different real-world approaches to providing support for causal multicast:

1) locating multicasting and causal protocols at the user-level in programming libraries,

2) locating multicasting and causal protocols at the kernel-level combined into a port group causal multicast abstraction, and

3) a combination of locations with kernel-level port group multicasting and user-level causal protocol support.

An experiment was designed to investigate different approaches to causal multicast support. The first step was to design and build a user-level programming library with

causal multicast support functionality for a distributed operating system environment. Using our library, three different architectures modeling the three real-world approaches were built and performance tested. We performed a detailed performance analysis on each experimental architecture, identifying fundamental costs of each approach. Understanding causal multicast component costs allowed us to project the user-level experiment results to the real-world approaches for kernel supported causal multicast. Implementation of our experimental architectures also showed that causal multicast support could be provided outside the application level.

Our experiment showed the majority of causal multicast expense to be serialized message-passing overhead. Locating multicast support in the kernel improves access to hardware multicast facilities which, if used, would result in significant reductions in that overhead. Multicast is a generally useful mechanism for distributed operating systems and applications. Kernel supported multicast would improve the efficiency of kernel support for IPC, one of the basic mechanisms micro-kernels are responsible for. The micro-kernel philosophy of placing mechanism in the kernel would not be violated by incorporating multicast support in the kernel, and in fact, some distributed operating systems already provide such support. Additional benefits include reduced application complexity, code size, and code redundancy.

For our assumed typical one process per node distributions, our results showed no performance benefit from locating causal protocol support in the kernel. Placing causal protocol support in the kernel appears to violate the micro-kernel philosophy of keeping policy out of the kernel. Keeping protocol support at the application level eases modification of protocol functionality.

In summary, we find the appropriate location for multicast support is the kernel, while causal protocol support should remain in applications. This finding is also reinforced by the fundamental distributed system philosophy of placing mechanism in the kernel(multicast) and policy(protocol) in the application.

We have attained our goal of gaining an understanding of what fundamental causal multicast functionality is and where it should be located. A causal multicast library has

been developed that has features of general use when programming distributed systems. We have also contributed a detailed low-level performance analysis of a distributed operating system. Local and remote Mach MSD2.6 message-passing performance has been measured and analyzed, including the passing virtual memory mapped application data in messages out-of-line.

Less significant contributions but which have been extremely useful in our work include the development of our Salon protocol development environment which includes the visual, interactive widgets for generating, perturbing, and plotting communication events. Several generally useful distributed system servers, both single and multi-threaded have been implemented. Distributed tools have been developed for remote job execution and control.

5.1 Limitations of Our Work

Our work has been limited to static, non-overlapping group membership. We have chose not to investigate these features but feel they are important in the real world.

5.2 Suggestions for Future Work

The message-passing efficiency of the multicasting function has been shown to be a critical part of causal multicast performance. The experiment we described here could be repeated on systems that supported multicast messages such as Chorus. Our results could be re-evaluated with known multicasting performance figures, or alternatively, parts of our experiment could be used to evaluate the performance of the multicasting functionality itself.

Our primary interest was causal multicast for distributed computing applications, therefore our fundamental one process per node assumptions limited the analysis our our experimental results. Should those assumptions be modified, we believe our experimental results, especially scaling performance, might be used to investigate causal multicast support for different sorts of computing problems, possibly in the area of non-shared

memory, parallel computing.

Bibliography

[AAH+85] M. W. Alford, J. P. Ansart, G. Hommel, L. Lamport, M. Liskov, G. P. Mullery, and F. B. Schneider, editors. *Distributed Systems—Methods and Tools for Specification, An Advanced Course*, volume 190 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[ABB+86] Michael J. Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael W. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, July 1986.

[Ase88] Paul Asente. Simplicity and productivity. *Unix Review*, 6(9):57–63, September 1988.

[BC90] Kenneth P. Birman and Robert Cooper. The ISIS project: Real experience with a fault tolerant programming system. Technical Report Cornell Technical Report TR90-1138, Department of Computer Science, Cornell University, July 1990.

[BCG91] Kenneth P. Birman, Robert Cooper, and Barry Gleeson. Programming with process groups: Group and multicast semantics. Technical Report TR91-1185, Department of Computer Science, Cornell University, January 1991.

[BJ87] Kenneth P. Birman and Tommy A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BSS90] Kenneth P. Birman, André Schiper, and Pat Stephenson. Fast causal multicast. Technical Report Cornell Technical Report TR90-1105, Department of Computer Science, Cornell University, April 1990.

[BSS91] Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):273–314, August 1991.

[CM84] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[CZ85] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[DKSK91] Fred Douglis, M.Frans Kaashoek, Andrew S.Tanenbaum, and John K.Ousterhout. A comparison of two distributed systems:amoeba and sprite. Technical Report Technical Report IR-230, Dept. of Mathematics and Computer Science, February 1991.

[Dra90] Richard Draves. A revised IPC interface. In *USENIX Mach Workshop Proceedings, Burlington, VT*, pages 101–121. USENIX, October 1990.

[Fid88] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*. University of Queensland, february 1988.

[Fou91] Open Software Foundation. The uses of Mach in the research community. Technical report, Open Software Foundation, January 1991.

[GLOR91] Marc Guillemont, Jim Lipkis, Doug Orr, and Marc Rozier. A second-generation micro-kernel based UNIX; lessons in performance and compatibility. In *Proceedings of the Winter 1991 USENIX Conference, Dallas, TX*, pages 13–22. USENIX, January 1991.

[GMS89] Hector Garcia-Molina and Annemarie Spauster. Message ordering in a multicast environment. In *Proceedings of the 9th International Conference of Distributed Computing Systems*. IEEE, June 1989.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[JB89] Tommy A. Joseph and Kenneth P. Birman. Reliable broadcast protocols. In Sape J. Mullender, editor, *Distributed Systems*, pages 293–317. Addison-Wesley, 1989.

[Lam78] L. Lamport. Time,clocks,and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LL86] Barbara Liskov and Rivka Ladkin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the fifth ACM Symposium on Principles of Distributed Computing*, pages 29–39. ACM SIGOPS-SIGACT, August 1986.

[MAS87] Joel McCormack, Paul Asente, and Ralph Swick. X Toolkit Library—C language interface. Technical report, Massachusetts Institute of Technology, 1987.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–216. Elsevier Science Publishers B.V.(North Holland), 1989.

[MVF+90] M.Rozier, V.Abrossimov, F.Armand, I.Boule, M.Gien, M. Guillemont, F.Herrmann, C.Kaiser, S.Langlois, P.Léonard, and W.Neuhauser. Overview of the Chorus distributed operating system. Technical Report Technical Report CS/TR90-25, Chorus Systémes, April 1990.

[MvRT+90] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba— a distributed operating system for the 1990s. *Computer*, pages 44–53, May 1990.

[NHJ+75] Norman H. Nie, C. Hadlai Hull, Jean G. Jenkins, Karin Steinbrenner, and Dale H. Bent. *SPSS-Statistical Package for the Social Sciences.* McGraw-Hill,Inc., 1975.

[Ous90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference, Anaheim, CA*, pages 247–256. USENIX, June 1990.

[PBS89] Larry L. Peterson, Nick C. Bucholz, and Richard Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–256, August 1989.

[PH86] G. Perlman and F. L. Horan. Report on |STAT: Compact data manipulation and analysis programs for UNIX and MSDOS. *Behavior Research Methods, Instruments, & Computers*, 18(2):168–176, 1986.

[Ray88] Michel Raynal. *Networks and Distributed Computation.* MIT Press, 1988.

[Ray90] Michel Raynal. Order notions and atomic multicast in distributed systems: A short survey. Technical Report 524, Institut De Recherche en Informatique et Systemes Aleatoires, March 1990.

[RST91] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processsing Letters*, (39):343–350, 27 September 1991.

[Sch87] Robert W. Scheifler. X Window System protocol. Technical report, Massachusetts Institute of Technology, 1987.

[Sch88] Frank Bernard Schmuck. *The Use of Efficient Broadcast Protocols in Asynchronous Distributed Systems.* PhD thesis, Cornell University, August 1988. TR 88-928.

[SES89] André Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the 3rd International Workshop on Distributed Algorithms, Lecture Notes on Computer Science, 392*, pages 219–232. Springer-Verlag, 1989.

[Ste91] Patrick Stephenson. *Fast ordered Multicasts.* PhD thesis, Cornell University, February 1991. TR 91-1194.

[WT88] Linda R. Walmer and Mary R. Thompson. A programmer's guide to the Mach system calls. Technical report, School of Computer Science, Carnegie Mellon University, February 1988.

Appendix A

Event Plot Server

The *event plot server* was developed to graphically display the vector time relationships between communication events in the Salon environment. At the core of the event plot server is an Athena viewport widget consisting of a plot widget. The plot widget plots system events in a manner consistent with event diagram techniques as described by Mattern[Mat89] and Fidge[Fid88]. Event diagrams can be used to provide a view of the system vector time to an observer external to the distributed system and can be used to display the concurrency or ordering of events.

As the legend in Figure A.1 shows, our plot widget displays (S)ynchronous and (A)synchronous multicast send events and reception and delivery of message events. Internal events are represented by small dark dots (not shown in figure). The plot widget utilizes viewport support for up/down left/right scrolling of the event displays. Buttons are provided for resetting (clearing), redrawing, and screen dumping the event diagram display. The debug button enables scrolling of ascii debugging information about messages received and server operations in the dialog widget area.

Figure A.1 shows an event diagram consisting of four multicast group members P1-P4. The following sequence of system events are displayed: 1) P1 sends an asynchronous multicast A1 which is received at each node P1-1 and subsequently delivered D1 to all processes, 2) P2 sends its A1 which is received P2-1 at all nodes and delivered D2 at all process, 3) P3 sends its A1, etc., and 4) P4 etc.

The plot server was developed using the Salon library functionality for OS, message,

and X11 support. The operation of the plot server is transparent to Salon library applications except for the plot enable flag in the application context structure. Plot messages are generated and sent to the plot server by the Salon library functions if the application context plotting flag is set to true.

The plot server has also been found useful as a tool for demonstration of causal protocol message ordering properties.

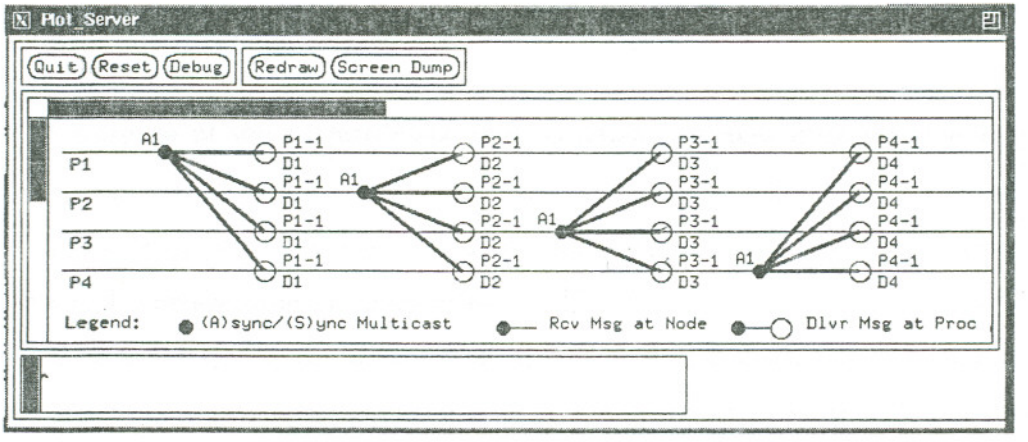


Figure A.1: Salon plot event server display.

Appendix B

Perturber Server

The purpose of the *perturber server* is to delay messages from selected senders to selected receivers. This allows an operator the means to reliably alter the message delay characteristics of our message passing system and can be used to test communication protocol message ordering properties.

The perturber server takes advantage of the port transparency characteristics of the distributed operating system chosen as an implementation platform. The server widget is made up of groups of buttons representing the process identifiers of the multicast group members. The user clicks on selected buttons to delay messages from the selected source to the selected receivers. The button is highlighted after a message is received and delayed, therefore the system message handling stream is "perturbed". Upon clicking the highlighted button, the delayed message is sent on to the receiver. The Salon library's OS, message, and X11 support was used to develop this server.

The following series of screen dumps illustrate the perturbation functionality this server provides. Figures B.1, B.2, and B.3 show the perturber server display together with the plot server display discussed elsewhere. In this case the plot server will be used to plot the "perturbed" communication events due to perturber server operation.

In Figure B.1 a normal non-perturbed multicast P1-A1 has been sent by P1 to all group members P1-P4. After this multicast, button "To:4 x From:1" is clicked (not shown). Figure B.2 shows button "To:1 x From:1" highlighted which means the P1-A2 multicast message destined to P4 has been delayed by the perturber server, and not received at P4 as shown by the plot server display. After P2 sends asynchronous multicast

A1 to all group members which gets received at all processes (currently the perturber delay queue is one message long), the highlighted button is clicked which releases the P1-A1 message destined to P4 as shown in Figure B.3.

Buttons are provided for enabling debug message scrolling in the dialog widget area and for initializing a list of process port identifiers to perturb.

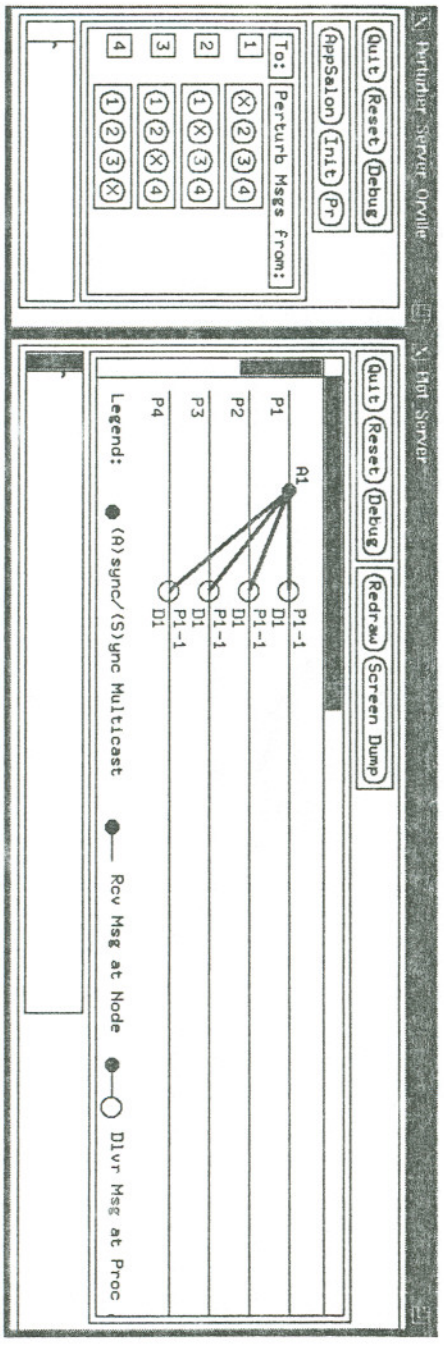


Figure B.1: Perturber and plot servers with no message perturbation.

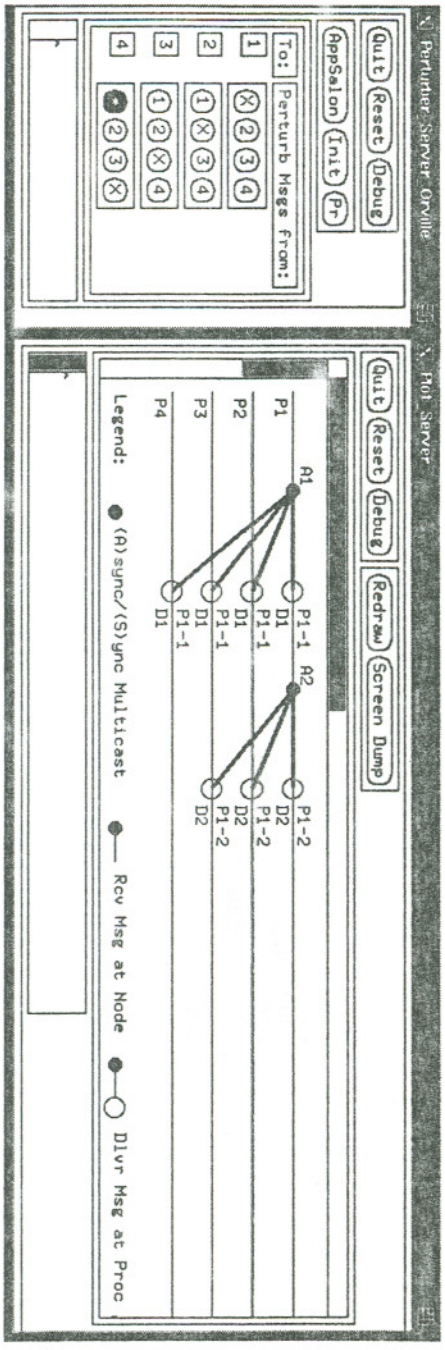


Figure B.2: Perturber and plot servers with message delayed at perturber.

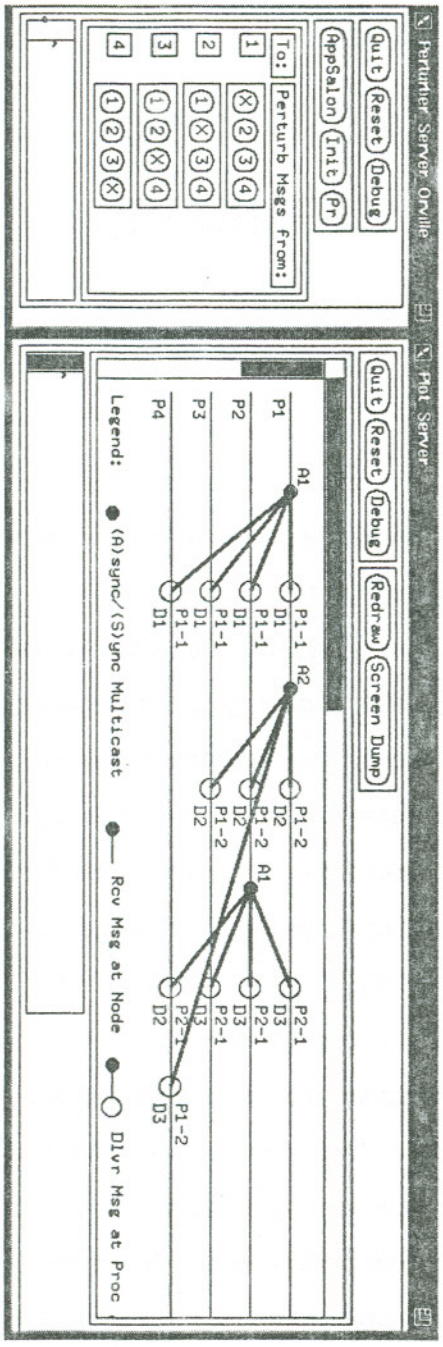


Figure B.3: Multicast followed by delayed message release.

Appendix C

Salon Protocol Visual Debugging Environment

In this appendix we describe how the Salon library and various architectures were developed and tested in a visually based development environment.

An application level widget was developed using Salon X11 support and distributed across our nodes utilizing the distribution features of X11r4. Each application widget has its own dialog window allowing application events to be viewed in their own context instead of serialized and interleaved at one console. The application widgets along with the plot and perturber servers create a development environment where we can generate real-time application multicasts on-demand, perturb multicast reception, and visually plot corresponding distributed communication events in a global manner.

The screen dump in Figure C.1 is typical of the tools usage during the Salon library and architecture development work. Four application widgets are shown, each running on a different node but using one X11 display. The plot and perturber servers also might be running on different nodes, it does not matter. Widget buttons are included for asynchronously generating various types and data sizes of multicasts, along with group membership actions, internal events, and environment initialization and control. A button is provided for changing the type of the application widget to of either application-level salon (user-level Salon architecture) or server-level salon (server-level architecture). Other buttons enable plotting and perturbation of communication events and enabling debug statement printing to the dialog area.

The user-level architecture was first developed in this environment, followed by the server-level architecture. These facilities were used right up to the point of actually measuring Salon architecture performance. The application widget associated overhead of polling and/or blocking of both X11 events plus OS message events would have impacted our performance results so for performance testing all X11 widget support was turned off in applications.

C.1 User-Level Salon Development

The library was first developed with the protocol module stubbed to always deliver messages as soon as received. Violations of causally ordered message delivery could be reliably generated using the perturber server facility and displayed using the plot server facility. Figure C.1 shows an example of a causality violation where messages P1-1 and P2-1 are delivered in different order at P4 than at P1-P3. The purpose of the P2 internal event is to reduce clustering and make viewing of plot events easier.

After the above Salon library functionality was implemented and debugged, the protocol module was implemented and functionality was visually debugged and tested. A button for the protocol was enabled in the application widget to turn on the causal protocol on-demand Figure C.2 shows the same sequence of communication events as in Figure C.1 except now the protocol has causally ordered the message delivery of P1-1 at P4 so the delivery order of P1-1 and P2-1 is the same at P1-P4.

C.2 Server-Level Salon Development

In the server-level architecture, all protocol functionality is removed from the application-level and located at the server-level. A screen dump of the Salon environment being used to demonstrate the server-level architecture is shown in Figure C.3. A server-level application was developed and distributed (S1-S4) to serve as a GUI front-end for the server-level architecture experiment servers. The server widgets have many of the same buttons as the application widgets. Each server thread also has its own box containing

a dialog widget and group membership and environment initialization buttons. Having a debug dialog window for each server thread of control was extremely useful when the server was moved from its single-threaded development stage to its dual-threaded final stage.

We illustrate the non-ordering of concurrent events property of the Salon causal protocol with the following server examples. Figure C.3 displays a sequence of communication events where P1-1 message reception was delayed, using the perturber server (not shown), at P4 to allow P4 to generate a P1-concurrent multicast. With the server protocol disabled, P4-1 is delivered D1 at P4. P1-1 is delivered D2 at P4.

Figure C.4 shows the same sequence of events but with the Salon causal protocol enabled. Even though the Salon protocol does not delay P1-1 delivery at P4 because P1-A1 and P4-A1 are concurrent events, causality has not been violated.

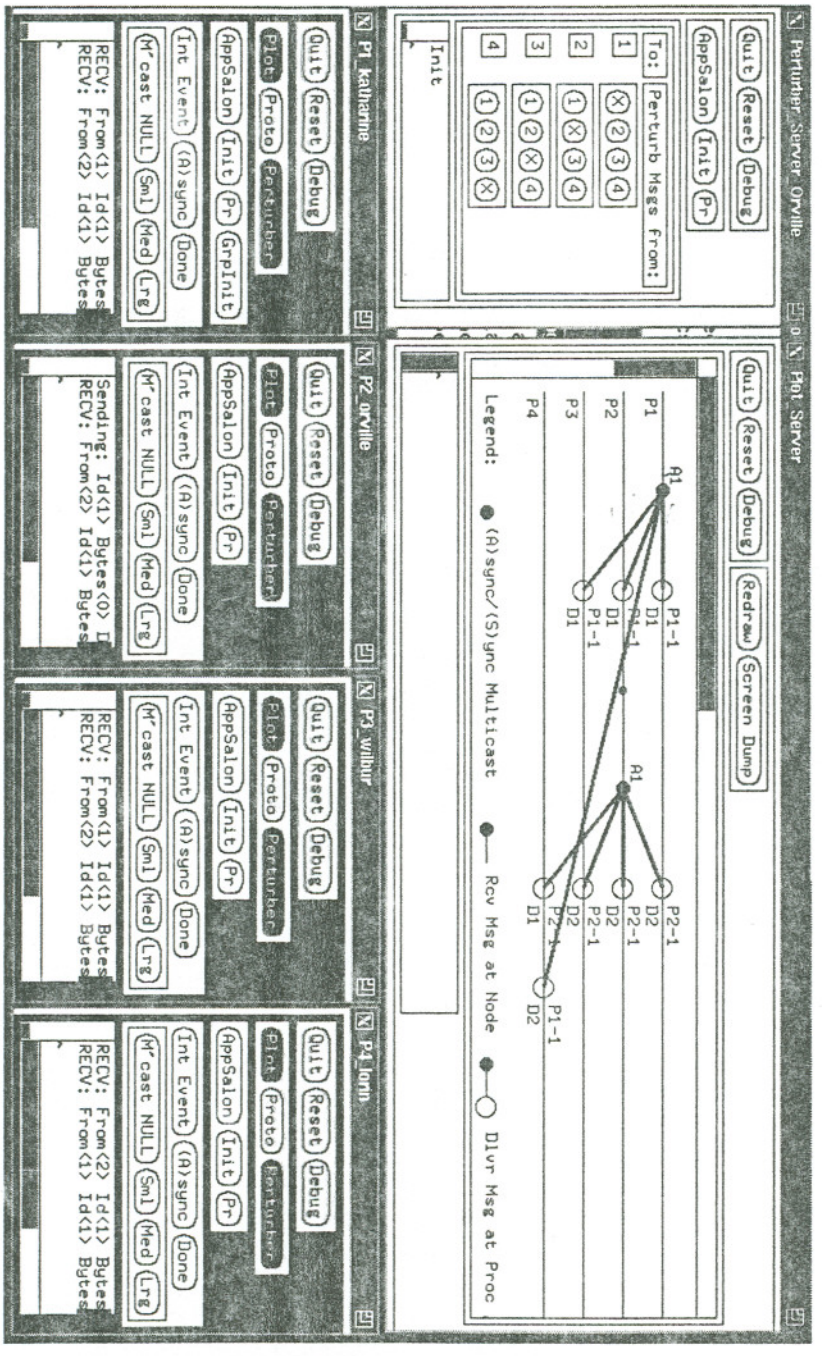


Figure C.1: User-level Salon distributed application without causally ordered protocol.

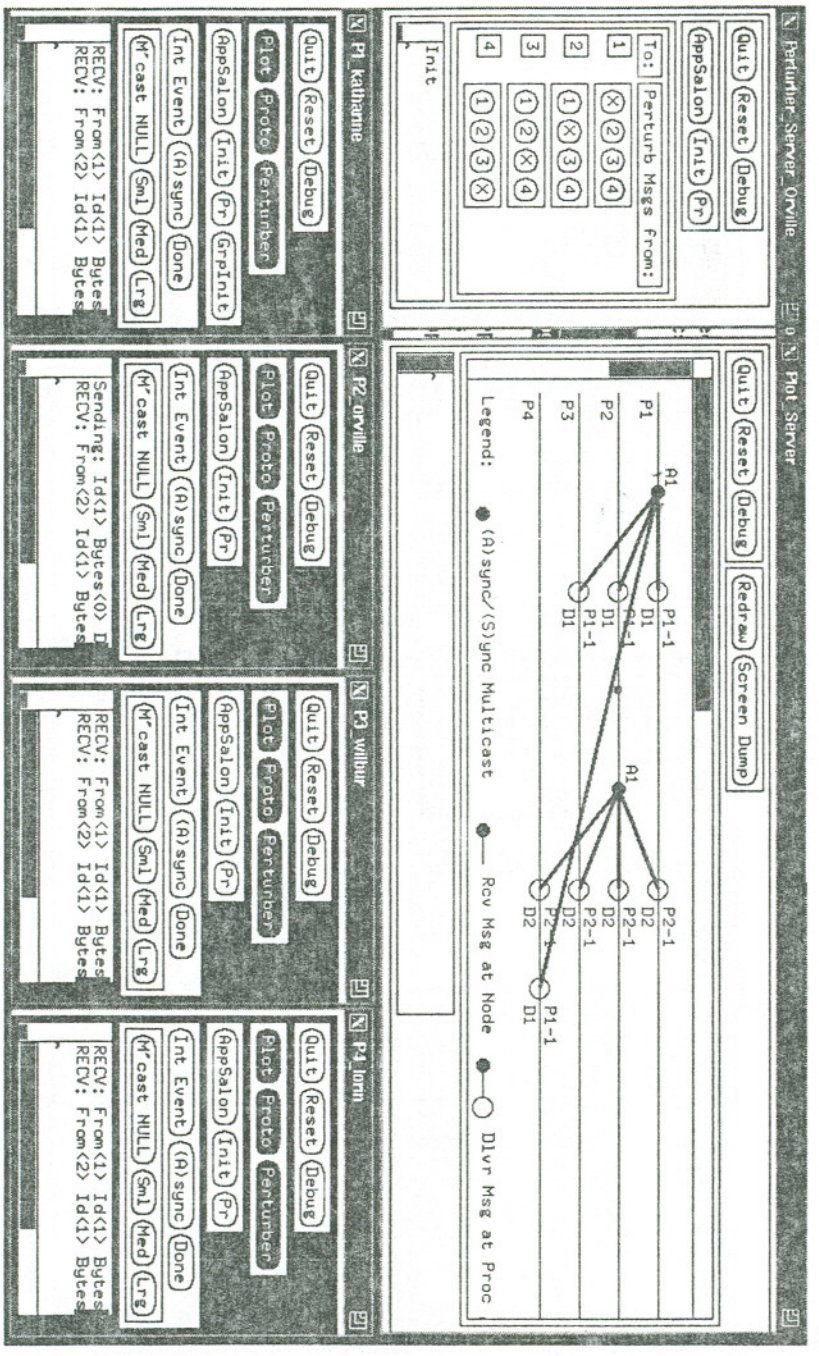


Figure C.2: User-level Salon distributed application with Salon protocol enabled.

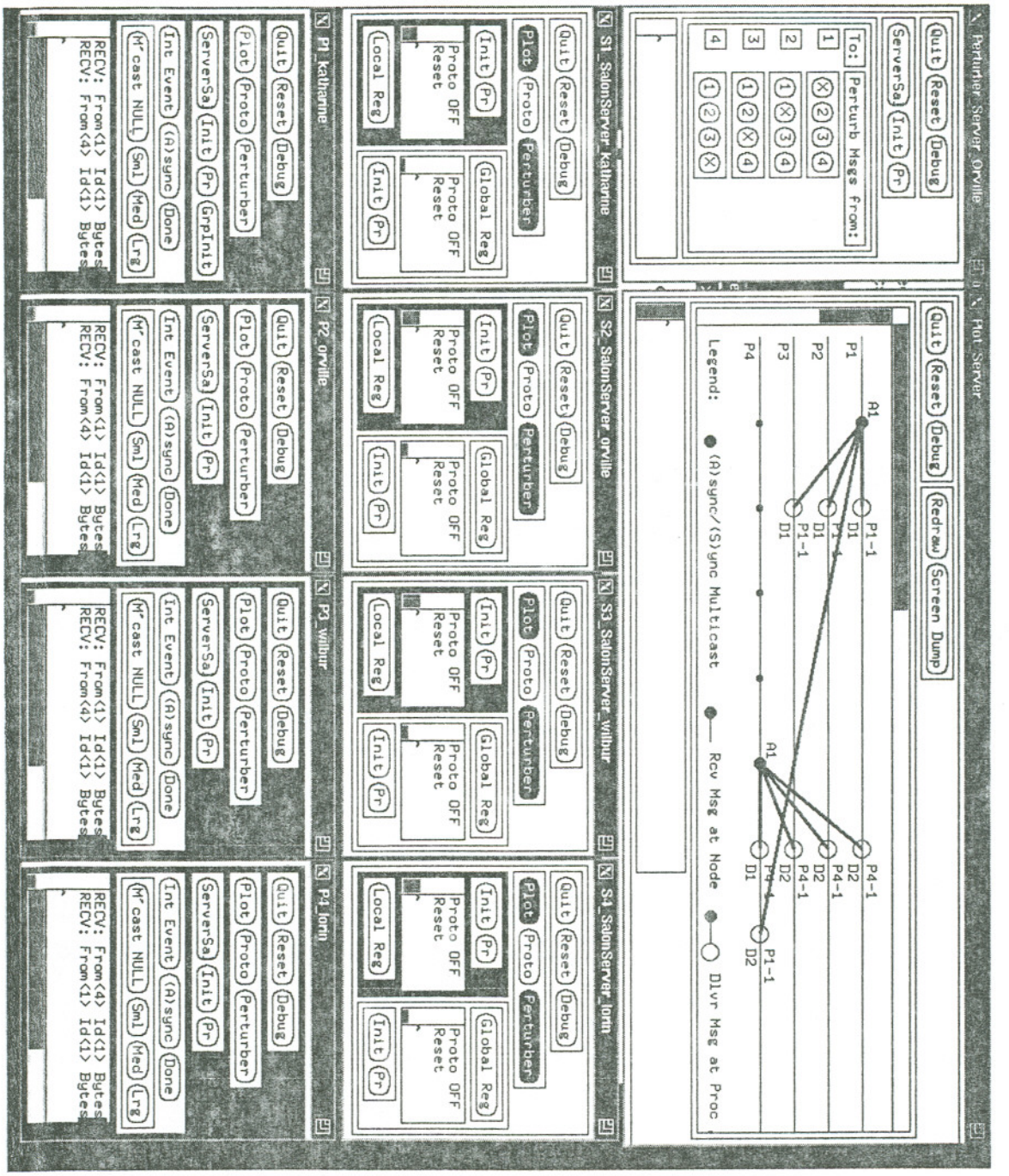


Figure C.3: Server-level Salon distributed application with Salon protocol disabled.

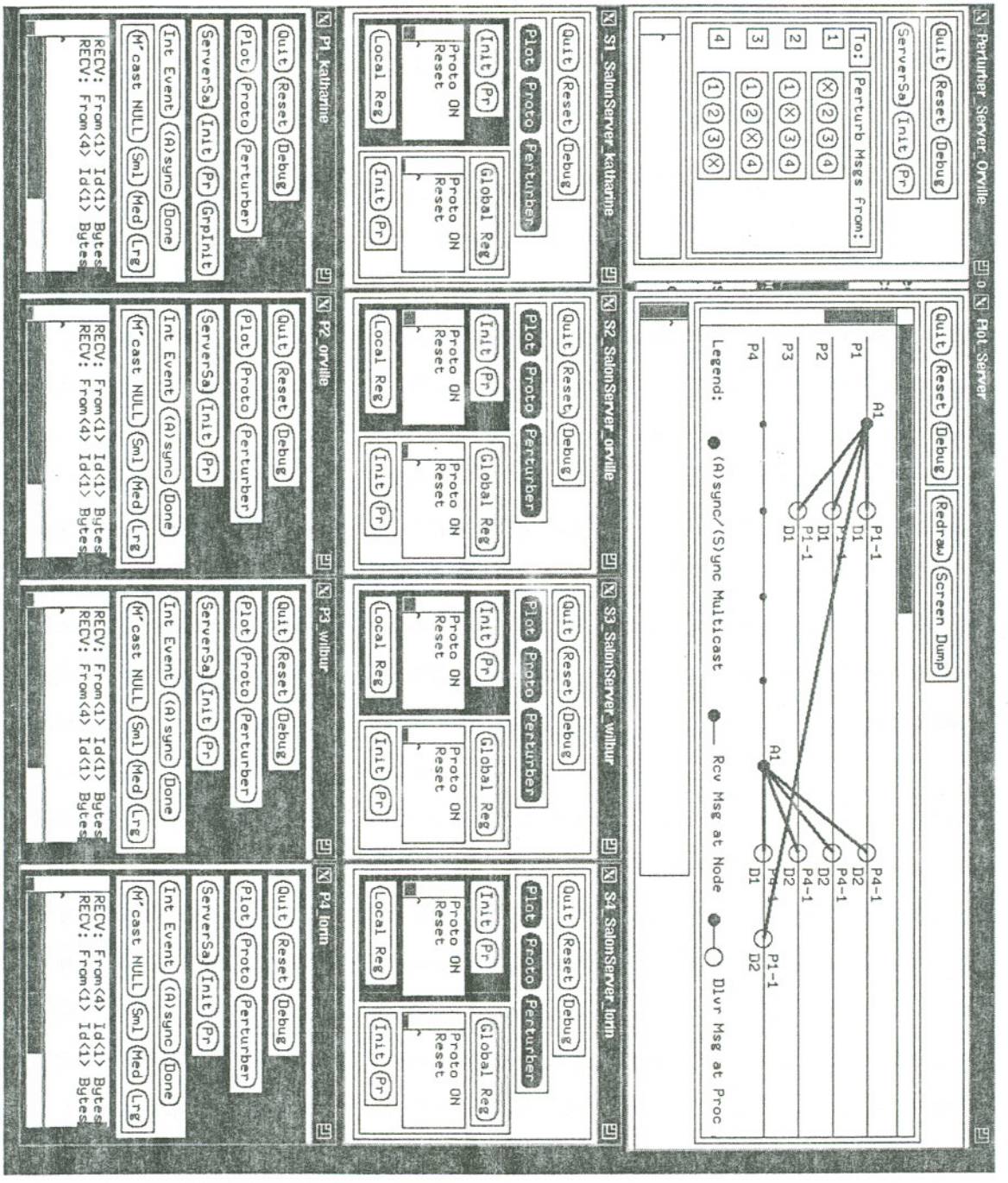


Figure C.4: Server-level Salon distributed application with Salon protocol enabled.

Appendix D

ISIS Toolkit Performance

This chapter contains the performance results we measured for the ISIS Toolkit. ISIS is a large programming toolkit with many features Salon does not have, including dynamic group membership. The ISIS figures were compiled as a reality check for our Salon figures and are not directly comparable. ISIS v2.1 fundamental and scaling performance is reported. We received the new version of ISIS (v3.0) too late in the project to report on its performance here.

Both synchronous and asynchronous multicast performance measurements were made. Synchronous(RPC) performance testing measures the round-trip *delay* from message sending to reply reception. Asynchronous performance testing gives an indication of *latency*. The rate of emission (send rate) was measured.

When testing synchronous performance for all group sizes, five multicasts were done in each averaging loop. When testing asynchronous performance, five multicasts were sent per averaging loop with group sizes of one, three were sent with size two, and one was sent in all other cases. Ten averages of the multicast loop were done in each run. Sample size for each data point is greater than 25.

ISIS application data consisted of strings of length 0 bytes(Null string), 1000 bytes, and 7000 bytes. These sizes were chosen as outlined in the ISIS work described in [BSS91]. For each test run the same string was passed, therefore avoiding application memory allocation and associated costs.

Our implementation environment consisted of the same four Microvax II workstations running Mt Xinu Mach 2.6, connected by a 10 Mb/sec ethernet that we used for the

Salon experiment testing. The averaging, timing, and scaling techniques were as before.

D.1 Results

We report fundamental and scaling performance results for ISIS v2.1 in the following tables and plots.

Synchronous ISIS Performance

Fundamental synchronous ISIS v2.1 mean performance for various application data packet sizes is shown in Table D.1.

Table D.1: Synchronous ISIS v2.1 performance.

Dest.	App Data in Bytes		
	0	1000	7000
	ms	ms	ms
Self	24.16	24.63	45.85
Local/Self	135.21	139.78	185.54
Remote/Self	81.58	86.71	156.71

Mean synchronous ISIS v2.1 scaling performance results are plotted for various packet sizes in Figure D.1. Our scaling type of overhead measurement where we run more than one process resulted in interesting situations typified by the local/self performance in Figure D.1 where local/self performance is worse than remote/self performance. We attribute this reduced ISIS performance to the ISIS multitasking architecture and our selection of benchmarking methods for RPC time measurement. As discussed in chapter 4, our technique consists of one master process multicasting to all group members, who reply in synchronous cases. This results in reasonably low overhead multiprocess/node Salon performance, as the master and slave processes simply block on their port when idle or expecting messages, and do not eat up CPU time. This technique apparently does not do well with the ISIS multitasking architecture though. A simple receiving ISIS slave application only doing null replies to messages sent uses 60-70% as much CPU time

as a master sending ISIS application who time, logs, multicasts, handles replies, etc. As more processes are added per node, the problem is amplified. It seems in [BSS91] this problem is avoided by having the master process asynchronously multicast to the group, then having group members reply with asynchronous multicasts.

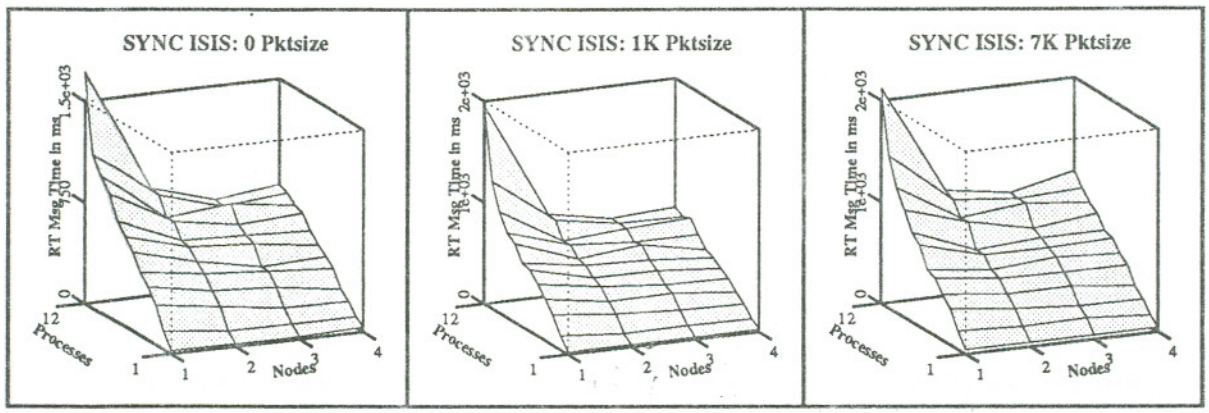


Figure D.1: Synchronous ISIS v2.1 scaling performance.

Asynchronous ISIS Performance

We report fundamental asynchronous ISIS v2.1 figures in Table D.2. Performance is shown in mean Mcasts or multicasts per second.

Table D.2: Asynchronous ISIS v2.1 performance

Dest.	App Data in Bytes		
	0	1000	7000
	SRate	SRate	SRate
	Mcasts/s	Mcasts/s	Mcasts/s
Self	223.65	129.61	39.55
Loc/Self	64.54	48.11	18.84
Rem/Self	90.08	58.01	17.10

Asynchronous ISIS v2.1 scaling performance is shown in Figure D.2. Mean Mcasts or multicasts per second results are plotted.

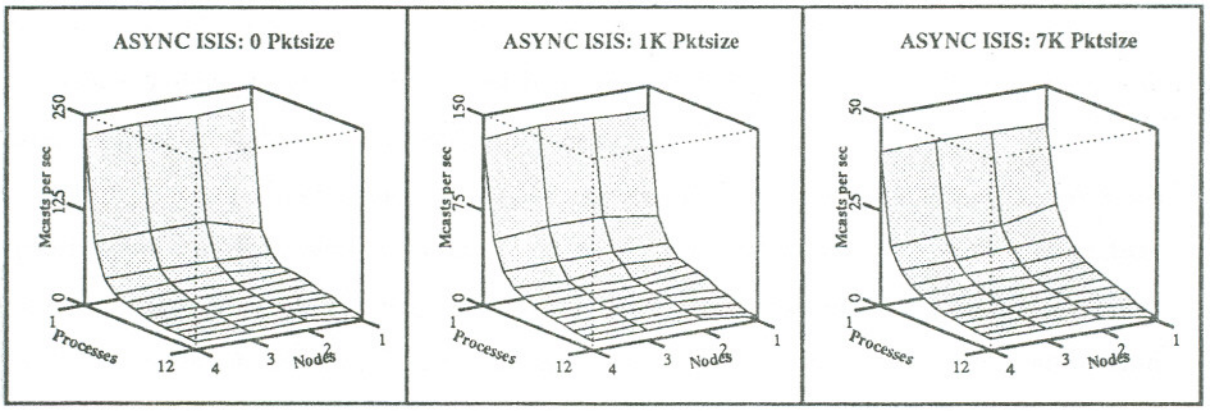


Figure D.2: Asynchronous ISIS v2.1 scaling performance.

Appendix E

Data Significance

In this appendix we briefly describe our measurement technique and comment on the significance of our measurement results.

Overall we compiled two independent groups of test results, synchronous and asynchronous. Independent variables for each group included type of architecture, data packet size, number of node, and number of processes.

Multiple tests of each group type were performed over a period of a month. For each group type, multiple tests of architecture type were performed. For each architecture type, multiple tests of different data packet sizes were performed. For each packet size type, there were 48 independent test points, one for each combination of four nodes and 12 processes. For each independent test point, multiple runs were of performance testing were performed.

A run is made up several multicasting loops. Inside of each multicasting loop, multiple multicasts were performed per clock interval to improve measurement accuracy as explained in Chapter 4 of our work. The elapsed time interval and corresponding packet size type for each multicast loop was logged into its respective group/architecture/node.proc.rawdat file.

Data extraction tools were written to gather the raw data, generate simple statistics, and generate data for visually plotting performance results.

Shell scripts were written to automate testing—test startup, data and error handling, and job control. The UNIX CRON(8) facility was used to schedule jobs overnight when system resource loading was predicted to be minimal.

E.1 Statistical Significance of Results

Our dependent variables are of the interval-level type[NHJ+75]— in the synchronous case we are measuring the round-trip delay when multicasting messages with replies and in the asynchronous case the number of multicasts per second.

Theoretically there is some absolute minimum time it takes for synchronous multicasting and some maximum number of asynchronous multicasts per second that our system can support, if only for one multicast, However due to multiple multicasts per clock timing period necessary to overcome poor clock resolution, that one minimum or maximum will get averaged into a group of multicasts for that clock period.

The experiment nodes were not connected by a private network other department users could be blocked from using, and the Mach MSD2.6 is a multiuser/multiprocessing operating system. Intuitively these two un-monitored independent variables increase the variance in both local and especially remote communication performance measurements. We have chosen for simplicity not to characterize either remote network or local multiprocessing loading, choosing instead to minimize impacts by running tests overnight and limiting personal local Microvax node processing extraneous to the experiment. This means especially in the remote communication cases, we have needed to make multiple test runs and average the results to obtain meaningful measurement results.

Due to our fundamental clock accuracy of +/- 5%, we will select a level of significance of 90%. We estimate this would correspond to normal confidence levels of 95%.

We generated two sets of results for all cases, fundamental and scaling results. The fundamental results were used for analysis and calculation. We feel it worthwhile to address their significance. The scaling results were not used for calculation and no predictions have been based on them. We do not feel it worthwhile to examine their significance. The scaling graphic plots themselves appear to indicate data significance. [1] Reasonably well behaved results can be observed with a few questionable points in

[1] We credit Kenny K.T. Toh (ktoh@mascot.berkeley.edu) for developing and sharing the "pdraw" plot drawing programs used for plotting scaling results.

all cases. We attribute these points to the uncontrollable independent variables of our experiment.

We examined the significance of our fundamental experiment results by performing a two-way, independent group t-test. A weighted means analysis was used pairwise between the architecture combinations of each group type. Only the 0 byte packet size was tested as those figures were primarily used in chapter 4 analysis. Gary Perlman's |STAT statistical program package tool, *desc*[PH86] was used to generate the t-test results.

The t-testing results are shown in Table E.1. Pairwise two-way t-test probabilities are displayed in the table for the group type and architecture type pairs shown. The probability numbers represent the probability that the pair of architecture have equivalent performance at that point denoted by the Destination column.

See Chapter 4 for a description of destination types and more detail on test conditions.

Table E.1: Pairwise t-test probability of equality between mean architecture performance.

	Probability of Equality					
	Synchronous			Asynchronous		
Dest.	I vs II	I vs III	II vs III	I vs II	I vs III	II vs III
Self	0.000	0.000	0.000	0.000	0.000	0.000
Loc/Self	0.000	0.000	0.000	0.000	0.000	0.000
1Rem/Self	0.000	0.000	0.002	0.000	0.000	0.000
2Rem/Self	0.000	0.026	0.031	0.743	0.120	0.107
3Rem/Self	0.923	0.929	0.988	0.563	0.131	0.318

As can be seen from the probabilities shown in the table, the only case our confidence is lower than 90% is in the multi-destination remote cases. The table shows there is an increased probability at these points that the mean results are equal. We have not based any calculations on these cases, as remote node communication measurements have too much variance. These results reinforce our assertion made in the thesis that communication costs, especially remote, are the majority of overhead expense, not the differences in architectures.