Efficient Multicast Design in a Microkernel Environment

Concurrency Practice and Experience, 10:7, 497-548, 1998

A. Averbuch A. Cohen
Department of Computer Science
School of Mathematical Sciences
Tel Aviv University
Tel Aviv 69978, Israel

Abstract

Multicast has become popular in recent years with the introduction of new, very fast networks. Existing solutions of the multicast design and implementation problems are either inefficient in microkernel environments or too expensive in terms of the host operating system overhead involved. In our search for a new solution we investigate various aspects of the problem.

Exploring the desired semantics, we come to the conclusion that control functions, group maintenance algorithms, and various ordering semantics can be implemented on top of the basic services, and that the efficiency of these implementations is less critical than that of the basic services.

We describe a few naive solutions to supporting multicast in microkernels and show their limitations. Then we suggest a new solution to the problem and show analytically that it is significantly better than other solutions.

The solution we decided to implement is a hybrid solution supporting control operations in user space and data operations in the kernel. It enables the semantics of multicast or group communication abstractions to be separated from the operating system support and mechanisms used to implement those abstractions.

Since we propose an efficient solution while introducing minimal changes to the Mach kernel, our hybrid solution suggests splitting the group communication burden between the microkernel and a Group Server (GS). In order to implement our hybrid solution the Mach kernel was modified.

Finally, we present an implementation of our new solution for the Mach 3.0 microkernel environment and show a very significant measured speedup in performance of multicast operations over naive methods for multicasting.

1 Introduction

1.1 Motivation

The problems related to the interactions between microkernels, high-speed networks and multicasting are difficult and complex. Most of them have not been addressed by researchers. The interaction between the host operating system and network hardware has been investigated, but the problems related to the special nature of microkernel based operating systems and their influence on network hardware interface still need further investigation. Multicast implementation has been explored in relation to network hardware, and to some extent in relation to microkernels, but the focus of these explorations has been either the semantics or the network protocol issues related to multicasting. No research known to us has dealt with the performance issues related to the implementation of multicast in microkernel environments optimized to exploit an underlying high-speed network. This is the subject of our work.

We explore different existing and possible approaches in designing multicast mechanisms for microkernels and high-speed networks and analyze them in terms of efficiency, scalability, integrability, flexibility and transparency. Once the key issues are better understood we suggest a new solution to the problem and show analytically that a significant speedup can be achieved by using our solution. We present an implementation of this solution for the Mach 3.0 microkernel environment and show a very significant measured speedup in performance of multicast operations over naive methods for multicasting.

We describe a few naive solutions to supporting multicast in microkernels and show their limitations. Then we suggest a new solution to the problem and show analytically that it is significantly better than other solutions. This paper discusses alternative designs for the addition of a multicast primitive to the Mach microkernel.

The implemented solution is a hybrid solution (see section 3.6) supporting control operations in user space and data operations in the kernel. From our performance measurements and analysis we believe this is the best solution because it separates the semantics of multicast or group communication abstractions from the operating system support and mechanisms used to implement those abstractions. The former have been thoroughly investigated by [Ahamad85], [Liang90], [Birman91], [Birman87]) and the focus of this work is still an open issue. As explained in section 3.1.1., our main concern in designing a multicast or group communication mechanism was efficiency.

We believe that the host operating system should be the main target for optimization. It is modified to cope with the new demands of high-speed networks and group communication oriented applications in two major aspects: Design and implementation of efficient interfaces between applications or user-level servers and the microkernel and development of internal microkernel mechanisms to manipulate multicast data in a highly optimized manner.

Since we propose an efficient solution while introducing minimal changes to the Mach kernel, our hybrid solution suggests splitting the group communication burden between the microkernel and a Group Server (GS). In order to implement our hybrid solution the Mach kernel was modified.

It should be modified to cope with the new demands of high-speed networks and group communication oriented applications.

Since we want to have the efficient solution while introducing minimal changes to the Mach kernel, our hybrid solution suggests splitting the group communication burden between the microkernel and a Group Server (GS).

In order to implement our hybrid solution the Mach kernel should be modified as follows:

1. Mach IPC should support destination port lists.
2. A new system call (or a modified version of an existing system call) should be introduced. Applications will send a message to a group of ports using Mach IPC and the new/modified system call.

Finally, we present an implementation of our new solution for the Mach 3.0 microkernel environment and show a very significant measured speedup in performance of multicast operations over naive methods for multicasting. The purpose of the paper is to show that hybrid mechanism outperforms all the other solutions. Problems related to local aspects (within the host operating system) are discussed and alternative designs for the addition of a multicast primitive to the Mach microkernel suggested.

The work focuses entirely on the overheads at the sender side up to the point that the message is queued for delivery.

The proposed mechanism is intended for high-speed networks but no experiments using a high-speed network were performed to verify whether the proposed modifications to the Mach 3.0 microkernel achieves speedup in the end-to-end multicast performance. Only the difference in local message queuing overhead is taken into account when comparing the alternative solutions.

This section briefly introduces multicast, microkernel technology, and high-speed networks, the three components of our research. The purpose of our work is better understood once the problems raised by each component and the interactions between the components are explained.

1.2 Multicast

Multicast communication has been recognized as an important and useful mechanism since the mid-seventies, when network technology developers incorporated some degree of support for multicast in their hardware design of Ethernet ([Metcalfe76, Ethernet80]) and Token Ring [Clark78]. In the following years many researchers recognized the need for Group Communications in distributed applications. Group Communications is an operating system-level abstraction in contrast to multicast, which is an abstraction of network communication [Liang90]. For convenience we will use both terms interchangeably and emphasize the difference between them where applicable.

Multicast can be defined, in a non-formal way, as a message transmission mechanism that delivers a message from a single source to a set of destinations (one-to-many communication) [Liang90]. A formal definition of multicast is given in section 1.2.2. Special forms of multicast are unicast (one-to-one communication) and broadcast (one-to-all communication). Concast (convergence multicast) is considered when there is traffic in the opposite direction (many-to-one communication) [Doar93]. A concast may follow a multicast (or a broadcast) delivery.

Finally, multicast should be logically viewed as a many- to-many communication (sometimes termed 'multipoint' communication [Cox91]). In this work we use the term multicast to describe a one-to-many communication unless otherwise noted.

Group communication is considered beneficial due to the following reasons:

1. When the underlying hardware supports multicast (or even broadcast), group communication can take advantage of this hardware capability and deliver messages more efficiently while significantly reducing overall network message-traffic and diminishing host software overhead. Many popular network hardware platforms support multicast to some degree: Ethernet, token ring, FDDI (Fiber Distributed Data Interface) [Katz93], and ATM [CCITT90].

2. Group communication is a natural abstraction for distributed programming [Joseph89]. Applications such as distributed commit protocols [Gray78], the contract net protocol [Smith79], elections [Garcia82], reliable storage [Bernstein85], distributed or replicated databases or object managers, conferencing, distributed parallel computations, contract bidding, name/time services, applications and systems supporting fault tolerance, all need group communication support. Recently, new demanding applications of multicast and group communication have evolved. Video conferencing is very demanding in terms of bandwidth requirements. So is wide scale distribution of audio, video and data via commercial television digital networks (video on demand), multi-user games, on line multimedia databases and remote tutoring.

3. User level programs can use group communication abstractions in a transparent way, without dealing with the underlying complexities of message delivery, group membership, etc.

In general, group operations cannot be implemented satisfactorily using multiple single-process operations (or one-to-one communication) [Cheriton85]. This is due to four major reasons:

1. The identity of all the processes in the group may not be known to the invoker of the operation.

2. Using multiple single-process operations is less efficient than using group operations.

3. Some frameworks of one-to-many communication can be satisfied by less reliable delivery mechanisms than those provided by many one-to-one communication implementations.

4. Multiple single-process operations do not provide the concurrency that is possible and desirable with group operations. When messages are received in parallel at the destination endpoints they can be processed concurrently. Exploiting this concurrency potential of multicast communication is one of the goals of our design.

1.2.1 Classification of Group Communication and Definition of Basic Terms

The following classification and definitions are derived from a variety of sources ([Ahamad85, Liang90, Birman91] and others). Some of them map directly to existing systems which will be reviewed later (in section 2.)

Group Membership

Groups can be *static* (created and never changed) or *dynamic* (members can join and leave the group dynamically). If the operation of sending a message m (we will use the shorthand notation of *m-send operation*) to a dynamic group delivers the message only to recipients that have joined the group prior to the send operation then this operation is called a *pre-members m-send operation.* If a member can receive a message even if it joins the group after the message has been sent then we call it *all-members m-send.* Sending a message to a static group can be viewed as a special case of *pre-member m-send.* The term, *membership atomicity*, is used by some researchers [Birman91] to describe systems providing the illusion of group membership that changes instantaneously as members join, leave or fail. This is similar to the previously mentioned *pre-member m-send* operation.

Groups can be *closed* (only group members can send messages to the group [Ladin90, Peterson89] while outsiders may be able to send messages to individual group members) or *open.* Sometimes a member in a group can be a member in one or more additional groups. This is called *overlapping groups.*

In *peer groups*, all members are equal. *Hierarchical groups* have some structural hierarchy, for instance, one member is a coordinator and the others are workers.

Reliability (Atomicity)

In most group communication systems, messages either arrive correctly at all members of the group or at none of them. This property is called *atomicity* or *atomic broadcast.* Ahamad uses the term *reliable* instead of *atomic* [Ahamad85]. He also defines a weaker send operation: *k-reliable m-send* operation which is a send operation that guarantees that at least k members of the group receive the message. An *unreliable* send operation (also termed *best-effort* datagram send) is a special case of *k-reliable m-send* (with $k = 0$). Atomic or reliable delivery is not a special case of *k-reliable m-send.*

Synchronization

When sending a message, the sender may block until all recipients get the message (*synchronous m-send*), or until at least k recipients get it (*k-synchronous m-send*). If $k = 0$, the sender will not block and the send operation will be *asynchronous. Synchronous send* is not a special case of *k-synchronous m-send* because the sender may not know the number of group members. *k-reply-synchronism* means the sender blocks until it receives k replies. This is a more severe synchronism and the sender may block forever (since recipients may choose not to reply). It is important to note that reliability and synchronism are independent of each other. Nevertheless, they should be applied cleverly. For instance, if an *m-send* operation is unreliable then it should be asynchronous, otherwise, the sender may be blocked forever because no process may receive the message.

Ordering

Time ordering is another important issue in group communication. *Global ordering* is defined as follows: If two processes send one message each, then all recipients of both messages receive them in the same order. When dealing with overlapping groups, the global ordering is usually within each group and not among multiple groups.

A weaker constraint is *local ordering* which keeps the order of messages when they are sent from the same source, but does not assure any ordering of messages from different sources.

Another weaker constraint is *consistent ordering*: The system will deliver messages to all recipients in the same order, but this order is not necessarily the order in which they were sent.

In Isis [Birman87], which is described in section 2.1.1., the CBCAST protocol guarantees correct ordering of messages only for messages that are *causally* related. This is termed a *virtually synchronous system*. The term is misleading since the issue is the ordering of messages, not the nature of synchronization. We will use the term *causality ordering*.

Group members can either be processes (or threads or tasks) or ports. When ports are being used, all communication involves ports instead of processes or threads. This is an important distinction because in some systems several processes (or threads) can read from one port. We shall discuss ports in section 3.3.1.

1.2.2 Formal definitions of group communication and its semantics

A formal definition of a multicast group is given by Doar in his Ph.D. thesis [Doar93].

A multicast group is a relation $R \subset \{S \times D\}$, where S is the set of possible sources and D is the set of possible destinations. There must be at least one member in each set, i.e. $|S| \geq 1, \ \ |D| \geq 1$. For every member, d_i, of D there is at least one member, s_i, of S, such that $(s_i, d_i) \in R$. Likewise, for every member, s_j, of S there must be a member, d_j, of D such that $(s_j, d_j) \in R$. The graph created by S, D, and R must be directed, acyclic and have only one connected component. Some multicast group members' hosts may be in both S and D, but their instances will be different. When $|S| = |D| = |R| = 1$, the multicast group is an ordinary unicast pair.

We need a definition for the semantics of multicast communication to describe and discuss both existing and proposed group communication implementations. The following is a definition of the semantics of multicast communication, based on the work of Ahamad [Ahamad85] with slight modifications:

The semantics of an *m-send* operation is defined by a tuple in $S \times R \times M \times O$ where S, R, M, and O are the set variables (representing Synchrony, Reliability, Membership and Ordering respectively) which can be defined as follows:

$$
\begin{aligned}
S &= \{\textit{k-synchronous}, \textit{synchronous}, \textit{k-reply}, \textit{synchronous}\} \\
R &= \{\textit{k-reliable}, \textit{reliable}\} \\
M &= \{\textit{pre}, \textit{all}\} \\
O &= \{\textit{no}, \textit{consistent}, \textit{causal}, \textit{local}, \textit{global}\}
\end{aligned}
$$

1.3 Microkernel Technology

An important trend in operating-system development is the restructuring of the traditional monolithic operating system kernel into independent servers running on top of a minimal microkernel [Guillemont91]. These servers communicate messages to each other and to the microkernel using an Inter-Process Communication (IPC) mechanism. This model is a natural basis for building modular distributed systems and for distributing existing applications or even existing operating systems. The restructuring of operating systems such as UNIX to suit a microkernel environment forced the developers to define better and cleaner interfaces between different parts of the system and to separate machine dependent code from machine independent code, thus making it easier to port the system to other hardware platforms. The restructuring also made it necessary to define some sort of hierarchy within system services. The most generic services, which are independent of a particular operating system, have been implemented in the microkernel while abstractions which are specific to a particular operating system have been implemented in an independent server (also called "operating system personality"). This server uses the microkernel when it needs basic system services.

Since many servers can run concurrently (or in a time-sharing mode, when a single CPU is used), many different personalities may coexist. Some microkernel architectures go even further and enable a per-task or per-personality selection of policies. For instance, each task can use a different page replacement algorithm [McNamee90] while using the same microkernel or even the same CPU.

This structure makes it easier to integrate new architectures and ideas into existing systems. Since a large part of the system code resides out of the microkernel, development, debugging and maintenance of the system code has become easier. Services and functional parts of the system may be easily replaced without any effect on other components. This modular design and the ability to change the placement of servers and/or subsystems transparently serve as a major factor in the success of microkernels.

The above mentioned features and the client/server model on which microkernels are based are expensive in terms of message-handling and context switching. Thus, with microkernels it is difficult to provide performance comparable to that of monolithic kernels (at least when running on monolithic architectures). This was one of our major concerns and it is further discussed in the following sections.

1.3.1 Microkernels and multicast

The distributed nature of microkernels renders them a perfect choice for applications using group communication. The notion of supplying a service while hiding the internal structure and location of the server is mandatory when implementing reliable storage, fault tolerant systems and fault tolerant applications using group communication. Most microkernel architectures already implement a certain degree of multicast or group communication support. The most important ones will be reviewed in section 2.

1.4 High-speed Networks

While processor speeds and disk storage capacities have been constantly growing during the last 15 years, network bandwidth has remained rather constant. Five years ago a 10 megabits per second (Mb/s) local area network (LAN) such as Ethernet and a 1.5 Mb/s long-distance line were both considered fast [Partridge93]. Recently this situation has changed. Fiber optic signaling has evolved during the last decade and new LAN and wide area network (WAN) technologies are approaching data rates of 2 gigabits per second or even higher.

B-ISDN (Broadband Integrated Services Digital Network), also known as the Information Superhighway, is a digital telephone system supporting multimedia traffic: voice telephone, audio, fax, videophone, video on demand, high definition TV (HDTV) and computer communication. ATM (Asynchronous Transfer Mode) is one of the services supported by B-ISDN. The standards are still under preparation by CCITT (recently renamed ITU-TSS), ANSI, the ATM Forum and other organizations. CCITT defined ATM as "A transfer mode in which information is organized into cells; it is asynchronous in the sense that the recurrence of cells containing information from a particular user is not necessarily periodic" [CCITT90]. ATM's main differences with networks such as Ethernet, FDDI and Token Ring are its fixed cell length (of 53 bytes) and its connection-oriented nature.

Until the telephone companies provide B-ISDN globally and replace the last copper wires between customers and the local telephone exchange, ATM can be used as a LAN only. These copper wires, known as the local loop, are the bottleneck which makes it impossible to achieve transmission rates higher than 128Kbps (kilobits per second) with ISDN (also called Narrowband-ISDN or N-ISDN). When B-ISDN becomes common, the distinction between WANs and LANs will probably be eliminated. Once this happens, computers and operating systems will have to cope with ever increasing bandwidth demands. Group communication over high-speed networks will make the demands even higher.

The paper has the following structure: This section briefly introduced group communication and multicast, microkernel technology, and high-speed networks. Section 2 describes existing implementations of group communication in several operating systems, and support for multicast in common network protocols and hardware. This review points out the unresolved issues in existing implementations. In section 3 we define our specific goals, explore the desired semantics, and present our design ideas. Several possible solutions are investigated and analyzed. Section 4 explains the implementation of our solution. In section 5 we present the results and discuss them.

2 Existing Implementations

In this section we introduce existing implementations of group communication in operating systems, followed by a review of multicast support in common network protocols and hardware. We believe that familiarity with existing solutions and implementations is the only way to understand many of the problems which are unique to multicasting. Finally we focus on unsolved issues, some of which are the core of our research.

2.1 Systems Supporting Multicast

2.1.1 Isis

ISIS ([Birman87, Birman91]-from Cornell) is a toolkit for building distributed systems and applications. It is not an operating system per se but rather a set of tools and programs which can run on top of existing operating systems. It has been used both experimentally and commercially (e.g. for stock trading coordination in Wall Street firms).

ISIS defines three primitives for group communication: ABCAST, CBCAST and GBCAST. ABCAST provides an approximation of synchronous communication and guarantees that all messages are delivered to all recipients in the same order. GBCAST is a group membership maintenance mechanism similar to ABCAST. Since ABCAST is complicated and expensive, ISIS offers another important primitive: CBCAST. This primitive implements group communication with *causal ordering*, which guarantees ordered delivery only for messages that are causally related. The implementation uses message-counting in vectors with slots for each group member. It also supports overlapping groups.

2.1.2 Amoeba

Amoeba is a distributed operating system from Vrije University ([Mullender90], [Tanenbaum90]). Amoeba process groups are closed. The way to communicate with a group is to do RPC to one of its members. That member then uses group communication within the group, if necessary. This is a sort of transparency: group communication is hidden behind RPC which is the chosen primitive of communication in Amoeba.

Processes can be members of several groups. When a group is created using $CreateGroup()$, the degree of fault tolerance (how many dead members the group must be able to withstand and still continue to function correctly) is determined by a parameter. When joining or leaving a group, one of the parameters points to a short message which is sent to all group members.

Message delivery to group members in Amoeba is reliable (atomic) and the ordering semantics is *global-time ordering.*

The semantics of group communication in Amoeba can be summarized as:

$$(synchronous, reliable, pre, global)$$

The synchronization is achieved using a sequencer: each message is sent to an elected sequencer which then "broadcasts" it to all group members. The sending process is blocked until it sees the broadcast message. (Broadcast here does not necessarily mean hardware-broadcast, it means sending the message to all members.) The Amoeba group communication protocol is also fault tolerant. It can survive the loss of k processors, where k is the resilience degree selected when the group is created. Amoeba uses a two-phase recovery algorithm ([Kaashoek91, Garcia82]): at phase 1 a coordinator is elected and at phase 2 the coordinator rebuilds the group.

In summary, group communication in Amoeba is very robust but is expensive in terms of performance.

2.1.3 Chorus

Chorus [Rozier91] is a microkernel-based distributed operating system from Chorus Systems, originally written in INRIA in 1980. Chorus uses ports (units of addressing) for communication. Ports are intermediate entities between senders and receivers and may be shared by several threads. Actors (units of resource allocation, essentially multi-threaded processes) may have several ports. Ports are identified by both a Local-Identifier and a Unique-Identifier. Messages carry the UI of the port (or port group, described below) to which they are sent.

Group communication in Chorus is implemented using port groups. Port groups are similar to ports in that they are globally named by UIs. Port group UIs can be statically allocated and maintained over an extended period to provide well-known service names. When a port group is created it is initially empty; ports can be subsequently added and deleted. A port can be a member of several groups.

The nucleus (the Chorus kernel) provides the creator of a group with its group manipulation key. This key must be specified to modify the contents of the group and may be freely transmitted between consenting actors.

Group UIs and group manipulation keys are related as follows:

$$groupUI = f(key)$$

where f is a non-invertible function known by the nucleus.
Chorus provides two IPC mechanisms:

Asynchronous IPC: The semantics of asynchronous IPC in Chorus is:

$$(0 - synchronous, 0 - reliable, pre, no).$$

The sender is blocked only during local processing. The system does not guarantee that the message will be received by the destination port. If the destination port is not valid the sender is not notified and the message is discarded.

RPC: A client/server model. RPC guarantees a response from the server.

Sending messages to port groups can be done using four addressing modes:

1. Broadcast to all ports in the group (not applicable to RPC requests)
2. Send to any one port in the group
3. Send to one port in the group, located on a given site
4. Send to one port in the group, located on the same site as a given UI.

2.1.4 The V-Kernel

The V kernel ([Cheriton85, Cheriton84]) is an implementation of a transparent message-based communication system between processes executing on machines connected by a local area network. It has been used by the Stanford Distributed Systems Group as a base for the distributed V-System [Berglund88], which is a completely distributed operating system.

Group communication is supported by the V kernel using process groups. Processes in a V process group are equal. Process groups are identified by a group identifier (*group-id*), which is identical in syntax and similar in semantics to process identifiers.

Groups are dynamic and open. A new group is dynamically created by

$$group - id = CreateGroup(initialPid, type)$$

which returns a unique *group-id*. The process designated by *initialPid* becomes the first member of the group. The *type* parameter specifies the attributes of the group as:

global: members can reside on any host in the V domain

local: members can reside on the same host as the *initialPid*

restricted: restricts membership to processes with the same user number (or authorization)

unrestricted: any process can join the group

There is also a range of statically allocated reserved group-ids which are used for well-known services.

Each V-kernel maintains a hash table accessed by group identifiers recording the group membership of all processes local to this kernel. Thus, membership information for a group is distributed across all the hosts that have members in the group. In particular, each kernel only maintains information about its own processes.

To add a process to a group or to remove a process from a group the $JoinGroup(group-id, pid)$ or $LeaveGroup(group-id, pid)$ calls can be used. If *pid* is a *group-id*, these operations can be used to join or remove members of different groups.

The $QueryGroup(group - id, pid)$ operation can be used to check if a process would be allowed to join a group, if it is already a member of the group, and if the group does not exist. It also returns the (approximate) number of processes in the group.

V supports 3 semantics of group communication:

1. $(0 - reply - synchronous, 0 - reliable, -, -)$ is an unreliable multicast to the group.

2. $(1 - reply - synchronous, 1 - reliable, -, -)$ assures reliable delivery to at least one process. Further replies are discarded and there is no indication of how many other processes are received/replied to the message.

3. $(k - reply - synchronous|k \geq 1, k - reliable|k \geq 1, -, -)$. is a delivery where the message sender is blocked until the first reply arrives or a timeout occurs. Additional replies can be collected using the *GetReply* operation. It is thus left to the sender to decide how many reply messages are of interest and how long it is willing to wait for them. All replies to a message are discarded when the process sends again.

   If a process sends a message to a group in which it is a member, it will not get a copy of its own message. This prevents the deadlock problem of the sender blocking while waiting for a reply from itself.

Processes can handle (reply or forward) messages without knowing if they were addressed to a single process or to a process group. They can also check this information and receive messages sent to specific groups using the *ReceiveSpecific* operation.

2.2 Support of Multicast in Existing Networks

Networks based on bus or ring architectures have an intrinsic support for broadcast. Most of them also support local area (within the same subnet) multicast without requiring replication of data frames. Since frames can be read off the wire by all participating hosts (the decision is made at each host's interface hardware or higher level protocols) all that is needed is a standard way to designate frames as multicast frames. Most of the widely used local-area network hardware, namely Ethernet, FDDI, and Token Ring, define multicast addresses and support multicasting. This support will be discussed in sections 2.2.2. through 2.2.4.

The local-area solution can be extended to some wide-area networks, such as satellite or radio networks, where all recipients can hear everything and multicast is easily supported without using any complicated routing mechanisms. However extending multicasting capabilities to other wide-area networks and internetworks is non-trivial. Multicast support at the internetwork level has been achieved in recent years primarily due to the work of Deering ([Deering88, Deering89, Deering91]). His work has become part of the TCP/IP protocol suite and IP multicast is being widely used in the Internet. There are other protocols currently in use or under research, such as

- XTP (Xpress Transfer Protocol) [Chesson92]
- ST-II (Experimental Internet Stream Protocol Version 2) [Topolcic90]
- MTP (Multicast Transport Protocol) [Armstrong92], and
- VMTP (Versatile Message Transaction Protocol) [Cheriton88].

In the following section we will present the IP multicast model. The ATM multicast model is discussed in section 2.2.5. We will not review other protocols since none of them are widely used and most of them raise problems similar to those discussed in section 2.3.

2.2.1 IP Multicast

IP multicast is the transmission of a datagram packet to a set of zero or more destination hosts, with a single address specifying the set of destination hosts [Deering89]. The IP multicast model extends the name space of Internet addresses to include addresses of host groups.

Host Groups and their Semantics

Host groups are defined by the following semantics:

$$(0 - synchronous, 0 - reliable, pre, no)$$

Packets sent to host groups are delivered with a "best effort" reliability to all members, i.e., they are not guaranteed to arrive intact at all members of the destination group or in the same order.

The sender need not be a member of the destination group (the group is open). Dynamic management of group membership provides flexible binding of Internet addresses to hosts. Hosts may join and leave groups over time. A host may belong to more than one group at a time.

A portion of the 32-bit IP address space is reserved for host group addresses.

Host groups may be permanent (well-known) or transient. A permanent group has a well-known IP address which has been assigned by the IANA (Internet Assigned Numbers Authority). It should be emphasized that the address is permanent, not the membership of the group.

IP Multicast Routing and IGMP

The Internet Group Management Protocol (IGMP) [Deering89] is used between IP hosts and their immediate-neighbor multicast agents to support allocation of transient group addresses, and to allow a task (or a process, or a program executed by the operating system) to join or leave a group on a certain interface of the host. IGMP is a required part of all IP implementations and is encapsulated in IP datagrams, with an IP protocol number of 2. IGMP is designed to avoid flooding the local network with messages and minimize collisions due to simultaneous replies triggered by a query.

The TTL (Time-to-live) field in the header is used to restrict propagation of multicast messages to distant networks.

Deering also proposed a method for routers to figure dynamically how to forward multicast datagrams. The method is based on the Distance Vector Multicast Routing Protocol (DVMRP) [Waitzman88], which is based on work by Deering (described in his Ph.D. thesis [Deering91]) and influenced by earlier work of Dalal and Metcalfe [Dalal78]. The basic idea is that, when a node (a multicast router) receives a multicast message from a network interface, it forwards the message to all its other relevant interfaces if and only if it came from the interface which the node would use to route messages to the source. Otherwise the message is discarded. The result of this algorithm is delivery trees, from the sender to all group members, which are generated dynamically by the routers. The IGMP protocol allows networks to advertise through their routers whether they wish to receive datagrams sent to a particular multicast group. As explained above, multicast routers forward messages only to the relevant interfaces (for each specific group destination).

One of the nice things about this approach is that both routing information for a particular multicast group and its membership data are shared. Delivery trees are implicitly computed when multicast messages arrive, and there is no need to build explicit delivery trees for each sender. As Jacobson [Jacobson94] describes it: "The network takes care of delivering data from senders to all interested receivers. Receivers only have to announce their interest. Senders just send. In everyone, both senders and receivers, this scales $O(\log R)$ where R is the number of receivers." This method works because multicast addresses have global meaning and they provide network-level identity for sessions. One does not have to know who the recipients are in order to talk to them. IP multicast has been used for videoconferencing over the Internet [Turletti93] and for both video and audio multicasting (described below).

The MBONE

The MBONE (Multicast Backbone) is a virtual network layered on top of the Internet supporting multicast and used for experimentation. The network is composed of islands that can directly support IP multicast. Since many IP routers do not support multicast, the MBONE links multicasting LANs by virtual point-to-point links called tunnels. A tunnel is a method for sending datagrams between routers separated by gateways that do not support multicast routing. It acts as a virtual network between two routers. Tunnels are considered a temporary solution until more routers support multicasting, or other protocols are employed. Old versions of the IP multicast software used IP Loose Source and Record Route option to support tunneling. Since the presence of IP options may cause modern router hardware to forward the tunnel packets through a slow software processing path, a new method was implemented in March 1993: IP multicast packets were encapsulated for transmission through tunnels, so that they resembled normal unicast datagrams to routers along the tunnel.

The routing protocol used in the MBONE is implemented by the *mrouted* daemon and is based on modified DVMRP. It is derived from the Routing Information Protocol (RIP) [Hedrick88], a distance-vector protocol from which it maintains topological knowledge. DVMRP implements a multicast forwarding algorithm called Truncated Reverse Path Broadcasting on top of RIP. DVMRP suffers from scaling problems typical of any distance-vector routing protocol. The MBONE also uses MOSPF, a more scalable protocol, which is described in section 2.2.1.4.

The IETF (Internet Engineering Task Force) has transmitted several conferences (voice and video) over the Internet. The first transmission [Casner92] in March 1992, spanned 20 sites within 16 time-zones. The traffic observed during these transmissions was in the range of 100-500Kbps (very low, compared to the traffic anticipated on high-speed networks). It is possible to carry MBONE traffic over slower lines by manipulation of the tunnel threshold. This threshold is compared to the time-to-live (TTL) value of the IP packet, and reducing it will block some of the traffic. By convention in the IETF multicasts, higher bandwidth sources, such as video, transmit with a smaller TTL, so they can be blocked while lower bandwidth sources, such as compressed audio, are allowed through [Casner94].

Improvements to IP Multicast

Multicasting across wide area networks [Deering90] is a problem that has not been completely solved yet. One of the main questions is which routing algorithm should be used.

Distance vector algorithms are simple, as compared to link-state algorithms [Callon87], but as Waitzman, Partridge and Deering [Waitzman88] suggest, a link-state algorithm, which already maintains much of the state needed, might prove a better basis for Internet multicast routing and forwarding. Multicast Open Shortest Path First Protocol (MOSPF) [Moy94] is an enhancement to the link-state routing protocol OSPF [Moy94B]. MOSPF provides the ability to forward multicast datagrams from one IP network to another. An IP multicast packet is routed based both on the packet's source and its multicast destination (source/destination routing). Each MOSPF router in the path of a multicast packet bases its forwarding decision on the contents of a data cache which is called the forwarding cache (it has a separate entry for each source/destination pair). A forwarding cache entry is actually built from two components. The first is called the local group database. It is built by the IGMP protocol and indicates the group membership of the router's directly attached networks. The local group database enables the local delivery of multicast packets. The second component is the packet's shortest path tree (built using Dijkstra's algorithm [Aho83]) which describes the shortest path to each multicast destination. During packet forwarding, any commonality of paths is exploited; when multiple hosts belong to a single multicast group, a multicast packet will be replicated only when the paths to the separate hosts diverge. This is a very important feature: it reduces both operating system and network traffic load. MOSPF adds a new link state advertisement describing the location of multicast destinations. A multicast packet's path is then calculated by building a pruned shortest-path tree rooted at the packet's IP source. These trees are built on demand, and as described above, the results of the calculation are cached for use by subsequent packets.

While MOSPF optimizes the path to any given group member, it does not necessarily optimize the use of the internetwork as a whole. To do so, instead of calculating source-based shortest-path trees, something similar to a minimal spanning tree (containing only the group members) would need to be calculated. This type of minimal spanning tree is called a Steiner tree and its calculation is known to be NP-complete [Karp72]. The problem is further complicated by the dynamic nature of group membership and the need to maintain optimal multicast trees while members are joining and leaving the group.

MOSPF handles inter-area multicasting by defining forwarders (bridges), which forward a summary of the group membership information to the backbone nodes. MOSPF also eliminates the use of tunnels and since it can interoperate with DVMRP (and IGMP), it can reduce the overall number of tunnels needed to support DVMRP.

Recently, another approach to multicasting called *core-based trees* has been proposed [Ballardie93]. Core-based trees have designated multicast servers, which serve as the core of a multicast distribution system. Data to be multicast is sent to the servers for multicast distribution. The developers of core-based trees have suggested this scheme may scale better to large internetworks than Deering's approach. Doar [Doar93] discusses the applicability of these approaches as well as some additional heuristic solutions ([Wall80], [Kou81] and [Rayward86]) to ATM networks. He claims that in the long-term operation of any real network which provides multicast capability, the dynamic nature of multicast applications is such that the existing algorithms are likely to provide solutions whose success is only ephemeral.

There are still many unresolved problems in the IP multicast model. Reliable multicast support, multicasting to heterogeneous recipients (recipients with different Quality of Service requirements), network security issues, and handling flow control are just a few of the open issues which are not in the scope of this discussion.

2.2.2 Multicast in Ethernet

Ethernet directly supports the sending of local multicast frames by using ethernet addresses with the first byte of 01 Hex. Initially each network interface attached to ethernet is configured to receive both its own address and the broadcast address (all ones: $ff:ff:ff:ff:ff:ff$). Configuring the interface to listen to additional addresses in the multicast address-range ($01:xx:xx:xx:xx:xx$) enables multicast messages for that address on the particular interface to be received. Unfortunately, most ethernet interface implementations allow a very limited number of multicast addresses to be configured in hardware. They rely on opening up the address filter to accept all multicast packets during those periods when the number of addresses exceeds the limit of the hardware filter, and letting higher level protocols (such as IP) filter incoming packets and discard unwanted addresses.

IP multicast uses ethernet multicast capabilities, where applicable, by mapping class D IP addresses to ethernet multicast addresses. This means that the mapping is not unique: 32 different IP multicast addresses map to each ethernet multicast address. This implies that IP (or the device driver) should exercise additional filtering capabilities even if the interface performs perfect hardware multicast filtering. Stevens [Stevens94] suggests that despite this imperfect address mapping and hardware filtering, multicasting is still better than broadcasting.

2.2.3 Multicast in FDDI

FDDI networks support multicasting in the form of group addresses. An IP multicast address is mapped to an FDDI group address by placing the low order 23 bits of the class D IP address into the low order 23 bits of the FDDI group address $01-00-5E-00-00-00$ (in "canonical" order).Thus FDDI uses the same mapping as Ethernet between the class D IP address and the 48-bit hardware address [Katz93].

2.2.4 Multicast in Token-Ring

The Token-Ring Network Architecture Reference [IBM89] provides several types of addressing mechanisms, one of which is group addresses (multicast). However, because of limitations in most token ring controllers, Token-Ring cannot use the same mapping as ethernet and FDDI from IP multicast to 48-bit Token-Ring hardware multicast addresses [Pusateri93]. A special subtype of group addresses is called functional addresses which are indicated by a bit in the destination MAC address. There are a limited number of functional addresses and therefore several unrelated functions must share the same functional address. It should be noted that since there are only 31 possible functional addresses, there may be other protocols that are assigned this functional address as well. Therefore, just because a frame is sent to this functional address does not mean that it is an IP multicast frame. While this method of addressing requires filtering at higher levels, it is preferable to the all-rings broadcast address method, which is more expensive in terms of overhead, of course.

2.2.5 ATM multicast model

In ATM, there are several types of connections which are supported by the signaling protocols (at the UNI - User-Network Interface - level, which is layer 3 in ATM parlance) based on the ATM Forum work [ATM93]. This work supplements the broadband signaling protocol standards (identified as Q.93B [ITU-TSS93]) to support 'Point-to-Multipoint Connections' (the term used to describe a multicast connection in ATM). It is defined as a collection of associated ATM VC or VP links with associated endpoint nodes, with the following properties:

- One ATM link, called the Root Link, serves as the root in a simple tree topology. When the root node sends information, all of the remaining nodes on the connection, called Leaf Nodes, receive copies of the information.
- Each of the leaf nodes on the connection can send information directly to the root node. The root node cannot distinguish which leaf is sending information without additional (higher-level) information.
- The leaf nodes cannot communicate directly with each other with this connection type.

Some special messages are used, in addition to the ones defined for point-to-point connections, to support point-to-multipoint connections and connection control. These calls are:
ADD PARTY
ADD PARTY ACKNOWLEDGE
ADD PARTY REJECT
DROP PARTY
DROP PARTY ACKNOWLEDGE

A point-to-multipoint connection is set up by first establishing a point-to-point connection between the root node and a leaf node. Additional leaf nodes can be added to the connection by ADD PARTY requests from the root node. A leaf node may be added or dropped from a point-to-multipoint connection at any time after the establishment of the connection.

In a formal definition of cell relay services [Minoli94], presented to the ANSI T1 Committee [Minoli93] unidirectional and bidirectional point-to-multipoint services are defined. Minoli suggests that the cells submitted by the origin should be delivered to the specified destinations in order, without errors, and with low delay and low variation in intercell delay. In addition, the network should provide (distributed) replication for optimized delivery of cells without requiring the origin to supply duplicate copies of the cells. It is also stated that the issues of whether destinations can be added during the information transfer phase and who authorizes the adds and how a connection is identified are for further study. However, users can abandon the call autonomously at any time. The issue of whether the other participants of the call need to be informed of the defection and how to accomplish this, is for further study. From the above description it is apparent that the semantics and control operations of an ATM multicast service are not well defined and understood yet. It seems that the sender has to know every receiver, create a call to one of them, then add others to the call.

CMAP (Connection Management Access Protocol) suggests different group management semantics and mechanism [deHart92] and has been put forward to the ANSI T1S1 technical sub-committee. A proposal has also been put forward to the ATM Forum for a signaling protocol based upon Q.93B with extensions inspired by CMAP.

CMAP defines the notion of ownership of a call, which can be transferred during the existence of a call. Calls can be specified as open (where any end-point may join) or closed (where end-points can only be invited by the source to join). There is also a verify option, where the source must give permission to an end-point for it to join. CMAP also handles different bandwidth request possibilities (including a "best-effort" option).

No matter which protocol is used, it is clear that hardware support is needed to provide multicast in ATM networks. Since ATM does not use shared medium to transfer information - space division switches are used instead - replication of cells must be explicitly designed with the hardware to support multicast. Doar [Doar93] claims that this is an extra level of complexity which has not been catered for in all switch designs. This will slow the use of multicast in the first generation of ATM networks.

Doar also suggests that the first area where the amount of multicast traffic increases in newer networks will be in the local area. Wide area multicast will be much less frequent in the near future. The reason is that LAN ATM switches (which are relatively small, at most 32×32 inputs and outputs) do not pose severe hardware scaling problems for multicast support, while in the wide area, ATM switches (which may be as large as 1024×1024) will have far more difficulties in providing multicast capabilities on each and every port.

Considering the above description of ATM's multicast model, including hardware and control considerations, it is clear that multicast support in ATM scales $O(R^2)$, where R is the number of receivers, as compared to $O(\log R)$ in IP multicast. It is also clear that ATM's multicast model is incompatible with the IP multicast model and that the connection oriented nature of ATM poses flow control and QOS problems on multicast realization.

In light of the above discussion, Turner's claim that "cell networking has become important due to several reasons, one of which is its ability to provide easier support for multicasting" [Turner86] should be reconsidered.

2.3 Unsolved Issues

Braudes and Zabele [Braudes93] have analyzed several existing multicast protocols and determined that there are significant problems with all of them if reliable, adaptive multicast transport of large data items is needed. The problems include inadequate address management, excessive processing of control information, poor response to network congestion, inability to handle high priority traffic, and suboptimal error recovery and retransmission procedures. They developed a high-level notion of the requirements for a service that addresses these issues and suggested that three basic components are required to provide a reliable multicast service on top of the existing network protocols (such as IP multicast):

- The Multicast Group Authority (MGA)
- The Reliable, Adaptive Multicast Protocol (RAMP), and
- Modified routing algorithms.

While these suggestions address protocol and networking software issues, none of them discusses problems such as the interface between protocols and operating systems and the efficiency within the end-points (their hardware, operating system, memory management, moving data between protocol layers, etc.). Such problems have not been given enough consideration by both the operating system research and network research communities. These unsolved problems are one of the reasons why the ultimate throughput delivered to user applications has not increased as rapidly as network speeds or host processing speed. Some of these problems are the main concern of this work.

3 Design Ideas

3.1 Goals

The definition of our goals is based on the following three observations and assumptions:

1. When connecting a workstation or even a server to a high-speed (gigabit) network, the host becomes bottleneck in terms of throughput. This is due to both interface hardware weakness and unsuitable operating system design.
2. Group communication (over high-speed networks or in microkernel architectures) tends to worsen the throughput situation.
3. Microkernels are dangerous traps where performance and efficiency are important. They tend to be less efficient than monolithic systems, unless very delicate design decisions are made while implementing services on top of them.

3.1.1 Efficiency

Our main concern in designing a multicast or group communication mechanism is efficiency. Most existing systems have well defined semantics, good support of group membership operations, and nice user and programming interfaces. Some of them decomposed the multicast facility into subsystems which can be implemented separately (in user space libraries, in servers and/or in the kernel). The main concern of the latter implementations has been modularity and simplicity. None of the systems known to us has made efficiency a major goal.

We claim that existing solutions emphasize the control and management functions of group communication while ignoring some of the efficiency aspects or all of them. Research or implementation concerned with efficiency issues deals mainly with the network load and the overhead of group communication in terms of network traffic.

The need for efficient group communication mechanisms within each host hasn't emerged until recently because networks were considered the bottleneck. This however is changing now. Since new high-speed networks tend to be much more powerful, in terms of throughput, than hosts connected to them we believe that the host operating system is the main target for optimization. It should be modified to cope with the new demands of high-speed networks and group communication oriented applications. Our point of view is the path that data has to follow in order to get multicast to a group of recipients. We will try to minimize the cost of data handling within a system. This can be done by isolating and defining the most basic sort of group communication services needed and designing their integration into microkernels. We believe that control functions, group maintenance algorithms, and various ordering semantics can be implemented on top of the basic services, and that the efficiency of these implementations is less critical than that of the basic services.

Another aspect of efficiency is employment of potential concurrency when messages are received in parallel at the destination endpoints. A good design of a multicast facility should avoid serialization of transmissions thus disabling concurrency.

3.1.2 Scalability

Scalability is a major issue in multicast or group communication. Our design should scale in three dimensions:

- It should scale to long distances (measured in network subnet-boundaries that have to be crossed). This is critical when considering group communication across wide area networks. Delayed replication of messages is an example of a good mechanism which enhances the scalability (and efficiency) of multicast across networks because it exploits commonality of paths leading to a group of destinations.

- It should also scale to large group sizes (the group size is the number of recipients of a multicast message). Implementations allowing for concurrency while delivering a message to group members (or to subgroups) enhance scalability in the group-size dimension.

- Finally, it should scale to many process groups.

As explained in section 2.2.5., ATM multicast is a good example of a design which doesn't scale as well as IP multicast. We want to avoid such design flaws.

3.1.3 Integrability

The design should lend itself to integration with existing mechanisms and abstractions. We want to be able to add stronger functionality and semantics to existing operating system interfaces rather than defining a new interface.

3.1.4 Flexibility

Our group communication mechanism should be flexible enough to use several network and/or hardware protocols at the same time so that processes on different machines, running different operating systems and using different network protocols and hardware will be able to multicast among each other. While the Integrability goal deals with the interface between our design, the operating system on which it is implemented and the applications using it, the Flexibility goal mainly refers to the lower level interfaces (network and network hardware - the lowest three layers in the ISO model).

3.1.5 Transparency

We want to be able to transparently replace an application with another one for debugging or fault tolerance reasons (e.g., replicating an application transparently). All seemingly normal communication performed by the application will be sent to a group and processed by its members without even changing the original application.

3.2 Choosing the Suitable Semantics

As explained in section 3.1.1., our main goal is minimizing the cost of data handling within a system by isolating and defining the most basic sort of group communication services needed and designing their integration into microkernels. If these services are cleverly designed it will be possible to implement higher level abstractions and mechanisms on top of them. If the basic microkernel services are not too restrictive semantically they can accommodate a wide range of control functions, group maintenance algorithms, and ordering semantics. Performance compromises such as those observed by Watson and Mamrak [Watson87] between application-level and transport-level abstractions can be avoided by using partial evaluation techniques [Massalin92] or protocol compilation techniques [Felten93].

Thus we use the following guidelines when determining the semantics:

1. Implement the minimal mechanism needed to achieve maximal performance. If no performance gain is achieved by implementing an abstraction within the microkernel - do not implement it there!

2. Implement the least restrictive semantics within the kernel. Stronger semantics can be built on top of a weaker one in user-level servers, while the opposite is not always true and is very inefficient (e.g., implementing an asynchronous multicast communication protocol over an RPC layer would cause severe congestion at the sender [Birman91]). It is also important that the chosen semantics should not prevent employment of mechanisms such as delayed delivery and concurrency, though it doesn't have to define and handle such mechanisms.

In the following sub-sections we consider the four major components of the semantics - group membership, reliability/atomicity, synchronization, and ordering - in light of the above mentioned guidelines.

3.2.1 Group membership:

We should not restrict the structure of groups, their representation, distribution and membership rules in any way. These should be left to the operating system or the network protocol or both. Implementing groups or membership policies and mechanisms within the microkernel would not increase the efficiency of the solution. It may hurt its scalability because the implemented policies may not scale well in certain environments. It will certainly harm the Integrability of the solution because an internal microkernel implementation is inherently operating-system dependent. Thus, our design will not handle group membership. This doesn't mean that our design can ignore group membership altogether: it should allow for distributed group membership maintenance at higher (or lower) levels, such as IP multicast host groups.

3.2.2 Reliability/Atomicity:

Reliable delivery can be implemented on top of an unreliable one. Thus, we should not provide atomicity as a part of our design while allowing for it to be implemented at higher levels.

3.2.3 Synchronization:

Should we support synchronous or asynchronous multicast as a basic microkernel service? As explained in section 1.2., a *synchronous m-send* operation blocks the sender until all recipients get the message. *k-reply synchronism* means the sender blocks until it receives k replies. When k is equal to the multicast group size this synchronism becomes a pure RPC mechanism. We believe that such synchronous mechanisms are inherently inefficient since the presence of a delay is something intrinsic to them. While synchronous mechanisms do not require reliability they can cause severe deadlocks in an unreliable environment: senders may block forever because one or more recipients may not receive the message or reply to it.

Asynchronous operations are a key to good performance in distributed systems, regardless of the underlying communication primitive [Birman91]. Moreover, asynchronous operations may be used to implement higher level synchronous models. They also lend themselves better to concurrency mechanisms. Thus choosing to implement asynchronous multicast primitives will be the least restrictive choice.

3.2.4 Ordering:

The above decision (i.e., to implement asynchronous multicast primitives) is related to the choice of ordering constraints. Since asynchronous operations create causal delivery obligations, Birman et al. [Birman91] claim that group communication should respect causality. They prefer using CBCAST as the core communication protocol in their process group architecture because it can be used to implement causal ABCAST (causal consistent ordering) or even stronger ordering mechanisms such as total ordering.

3.3 Mach as a Representative Microkernel Platform

In this section we explain why we decided to use the Mach 3.0 microkernel as a platform for our design experiments. We also describe some Mach 3.0 abstractions and mechanisms which are relevant to our work.

The issues related to the design of an efficient group communication support are not specific to a single microkernel architecture. Therefore, we believe that problems and solutions identified in this work are not unique to any single microkernel. We used the Mach 3.0 microkernel for testing ideas and solutions and for measurement purposes. It has been chosen as a platform for our design experiments for the following reasons:

1. It formally has no support for multicast or group communication. This frees us of pre-existing mechanisms and makes it easier to design unbiased solutions.

2. It is a good and mature example of modern microkernel architecture.

3. Its documentation and source code are available and easy to obtain.

4. It is a popular architecture supported by several vendors and organizations.

3.3.1 Mach ports

The most important abstractions on which Mach IPC primitives operate are messages and ports [Loepere92]. The message primitives are used to send messages to ports and receive messages from ports. Mach messages are delivered reliably and local ordering is preserved (i.e., messages sent by a single thread are delivered in the order sent). They also support sending large blocks of data Out-Of-Line (OOL) using a well known memory management optimization of Mach: Copy-On-Write (COW). To send data efficiently with this kind of optimization, Mach uses pointers to memory objects. We will study messages and their structure in section 4.2. In this section we focus on Mach ports.

Ports in Mach are best described as mailboxes linked to message queues. To send or receive a message, a task has to exercise *capabilities* (also known as *port rights*) for the port to which the message is being sent to or received from. Send rights to a port can be granted to many tasks while a receive right is held by a single task (all threads within a task can use the send/receive rights). Port rights can be transferred between tasks using messages. Transferring a receive right removes the right from the sender and grants it to the receiver. There is also a send-once right which allows the holder to send a single message to the port.

Mach 3.0 provides both an asynchronous IPC mechanism and a synchronous one (RPC) [Draves90]. Both sending and receiving messages is done via a system call (*mach_msg_trap*) which is discussed in section 4.3. The IPC interface supports remote procedure call as an important special case operation. The second parameter to this call contains a bit which can be turned on to specify a send operation and another bit which can specify a receive operation. RPC is supported by turning on both of these bits, which causes an RPC-optimized path to be followed within the kernel when handling this call. RPC uses send-once rights extensively: when a client performs an RPC, it provides a send-once right to a reply port. This right is carried in the message sent by the client and may be used by the server for replying.

Mach ports do not have user-visible global names. The Mach microkernel maintains a table of all ports to which the task has access rights (the port *name space*). Ports are referenced within the microkernel as object pointers (pointers to the port table). Within each task, ports are referred to as port names. Port names are indexes into a port *name space* and they represent the task's view of a port (This is similar to file descriptors in Unix.) All threads within a task share the task's port name space (and its associated capabilities). The same port may have (and usually has) a different name in different tasks. The IPC message handling primitives have to translate per-task names to kernel object pointers when copying-in a message from user to kernel space, and vice versa.

Mach supports another abstraction, the port set, which can be used to receive messages from several ports (this is similar to the *select*() system call in Unix). Though tempting, the implementation and semantics of port sets do not allow for easy incorporation of group communication into the Mach microkernel.

Mach IPC does not support global naming or access to ports. Thus, to send a message to a remote port a special mechanism should be used. The original mechanism has been implemented as a user-level server called NetMsgServer. This server creates local proxy ports for remote ports and uses conventional protocols (such as TCP) to send messages to a remote NetMsgServer, which, in turn, delivers them to the remote port.

3.4 Naive Solutions

In this section we suggest several possible solutions to the problem of designing an efficient multicast mechanism in a microkernel environment. We analyze these solutions theoretically before considering their possible implementations. There are several benefits to this method:

1. It is independent of any specific hardware or Mach microkernel release.

2. It can be easily adapted to other microkernels.

3. It will help us analyze the performance issues related to multicast implementations.

4. It will help us decide which tests or microbenchmarks should be conducted and what exactly should be measured.

5. It should match the measured results, and in case of a mismatch it will indicate some sort of an error in the design, in the benchmark or in the measurements.

3.4.1 Solution number 1: Everything in user space

The integrability goal suggests using mach ports and Mach IPC for implementing group communication. Mach ports and Mach IPC are heavily used in Mach and they seem to be a natural mechanism for group communication as well. MACH IPC messages are complex enough to justify a special Mach Interface Generator, MIG [Boykin93], which eliminates the need for the programmer to code the packing, sending, receiving and unpacking of MACH

IPC messages. Thus, MIG can be augmented to support group communication details while the programmer can still use high level MIG syntax to get group services.

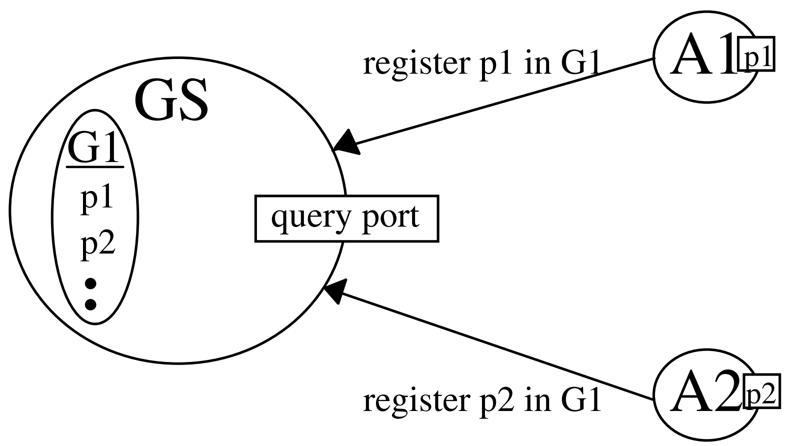

Figure 1: A User space Group Server (GS)

Ignoring the transparency goal (for the time being) we can solve the group communication problem by implementing a dedicated user space group server (GS). It will perform group membership maintenance, but actual send operations will be left to the application itself. The group membership list will be a list of ports (of size n) which have been registered at the GS (see Figure 1) by applications (such as A1 and A2 in Figure 1). The application will acquire port send rights from the GS via the query port. Given a group name (G1) port send rights $(p_1, p_2, \ldots, P_n)$ will be returned, and the application will send the message n times using Mach IPC (see Figure 2).

The roles of such a group server are:

1. Maintaining group membership: To create a group or to join an already existing group, an application registers one of its ports under the group name via a call to the GS. The GS maintains membership lists (lists of ports owned by various applications) for each group. All queries to the GS are performed via a single well-known port.

2. Providing a global name service for groups: When an application wants to send a group message it has to query the GS to obtain send rights to all ports which are members in the group (ports owned by group members). Delivery of messages to group members is performed by the application itself, using microkernel (MK) services, without involving the GS (see Figure 2).

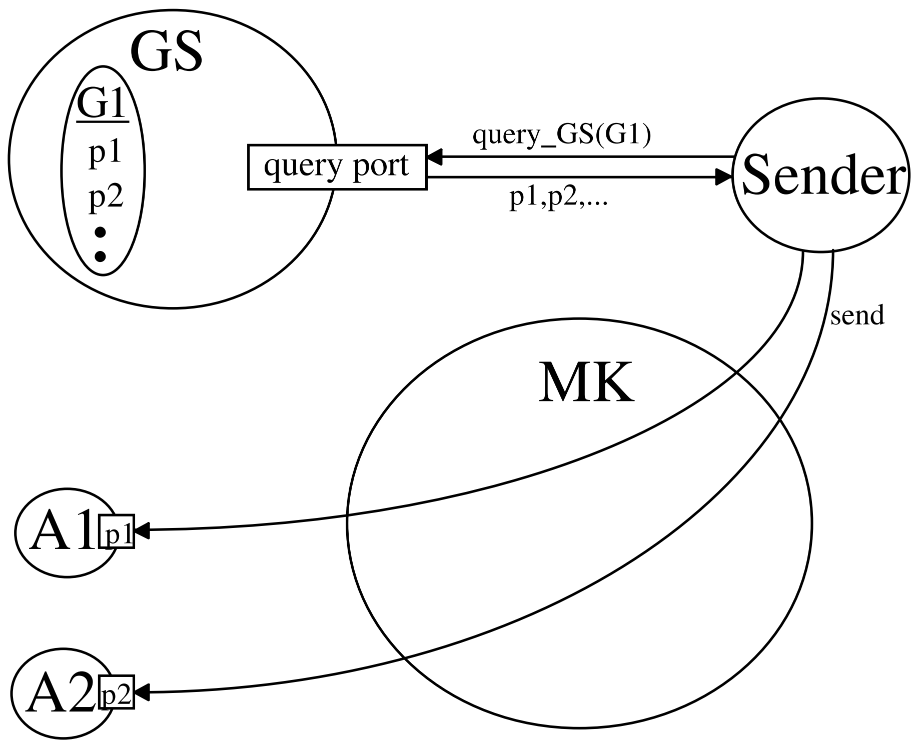

Figure 2: An *m-send* operation using Solution number 1

3. Interfacing to other GSs, on other hosts, via the network thus maintaining a global namespace of groups by GSs on different hosts. The GS is not involved in interfacing to various network communication protocols and media for delivery and receipt of group messages. Its only role is to provide name services.

Analysis of solution number 1

We will base the calculation of the costs on decomposition of the group communication operation into its elements, looking at the basic steps of execution.

Step 1. Query the GS for a membership list:

Given a message to multicast, obtain a representation of its planned recipients. If the recipient group is *static* this stage will be performed once per group in each application. If the group is *dynamic* this stage will either have to be performed before each and every multicast operation or, in case of a more clever design, only when group membership is modified. For our analysis we will assume *pre-members m-send* semantics and *dynamic* groups. The cost of this operation, which is actually a Mach IPC RPC call to the GS, is C_{get_group}. C_{get_group} can be further decomposed into the cost of an optimized *mach_msg* trap (the RPC optimization which costs C_{msgrpc}), a context switch to the server (C_{cs}) and another context switch to the querying application (C_{cs}). The size of the group will determine the size of the message returned by the server ($c + kn$). Thus, the cost is

$$C_{get\_group} = C_{msgrpc} + 2C_{cs} + c + kn.$$

We can reduce the cost of this phase significantly by using shared memory between the GS and the application. A somewhat different idea has been implemented by Reynolds and Heller [Reinolds91]. They used shared memory to pass data between a device driver and a user level protocol server. This idea has been used by Thekkath et al. [Thekkath93] for implementing network protocols at user level. Using a similar mechanism between user level applications (a GS and another application) is even simpler. It will eliminate the need to query the GS before each group communication operation. The application will have a correct representation of the group in its address space at all times. The cost of such an arrangement is an initial setup at startup, and the cost of synchronization (mutual exclusion when accessing the shared memory). We can ignore both these costs because they are negligible.

Step 2. Distribute the message using Mach IPC

For each group member, send a Mach IPC message to the appropriate port. Assuming asynchronous group communication, this means n *mach_msg_trap* operations (system calls).

Each system call can be detailed as follows:

- The trap causes entry into kernel space.
- The message header is copied from user space to kernel space.
- The message body is copied from user space to kernel space. If the message is sent out of line using the Mach Copy-On-Write (COW) optimization, there is no need to copy the message body but a COW operation within the kernel is required instead.

It is important to note here that all group messages are sent in this manner, including those intended for remote hosts. The distinction between local and remote recipients is made later, within the kernel. It should be noted also that this step is where the replication of the message occurs. Sending the message n times includes n replications of its header and n COW operations on its body.

The cost of this step is $C_{distribute1}$ where

$$C_{distribute1} = nC_{syscall} = n(C_{trap} + C_{hcopy} + C_{cow})$$

Step 3. The kernel delivers n messages to their destinations

The cost of this step depends on the protocols used to deliver the messages: some of them may be local messages and they will be queued in the receiving port's queue. Others will use various communication protocols such as TCP/IP, which is implemented in user space (in the Unix server). We note however the following property of the cost ($C_{deliver1}$):

$$C_{deliver1} \geq nC_{local\_delivery}$$

A local delivery of a single message consists of:

- Queuing the message in its recipient task's queue (C_{queue}).
- One of the recipient task's threads gets the message. We will not include this operation in our analysis.

Thus the cost of this step is:

$$C_{deliver1} \geq nC_{local\_delivery} = nC_{queue}$$

The total cost of solution 1 is:

$$C_{solution1} = C_{get\_group} + C_{distribute1} + C_{deliver1}$$

Ignoring the cost of getting a group representation (as discussed above) we get:

$$C_{solution1} \geq n(C_{trap} + C_{hcopy} + C_{cow} + C_{queue})$$

Advantages of solution 1

The implementation of a group server in user space seems to have strong advantages:

– It is relatively simple.

– It fits within existing semantics.

– It does not require any kernel changes. Code residing in user space is easier to develop and maintain.

– It conforms to the spirit of microkernel technology which tries to minimize the kernel and implement any possible service in user space.

Disadvantages of solution 1

- It is not efficient. Sending a group message to a group of size n costs n message send operations. This is very expensive. It also doesn't scale to large group sizes. In addition, the GS has to be queried prior to each group send operation in order to get the latest membership list, unless special setup and mutual exclusion mechanisms are used, as described above.

- It is not transparent to the application. It is transparent to the microkernel, though, but this is hardly considered an advantage. This lack of transparency has two implications:

  1. When an application sends a group message it has to do all the work by itself (except the maintenance of group membership).
  2. No transparent service replication (e.g., for fault tolerance) is possible. The application must be aware of the fact that it is sending a message to a group.

- It is hardly a multicast mechanism. This is the biggest disadvantage of solution number 1. Since the kernel has no idea that a multicast operation is being performed, and thus it delivers n independent messages to their destinations, the group communication operation is reduced into n unicast operations and no optimization such as delayed replication, or using hardware multicast (if provided) can be employed. It also causes loss of potential concurrency since the operations are serialized. This means that solution 1 doesn't scale to long distances as well.

- It is not transparent to the application. It is transparent to the microkernel, though, but this is hardly considered an advantage.

3.4.2 Solution number 2: Transparency in user mode

The transparency and integrability goals suggest a representation of process groups by ports. Since Mach ports are "owned" by a single task (we have seen that only a single task can have receive rights for a port) we introduce a mediator: a dedicated user space group mediator (GM). The GM is illustrated in Figure 3.

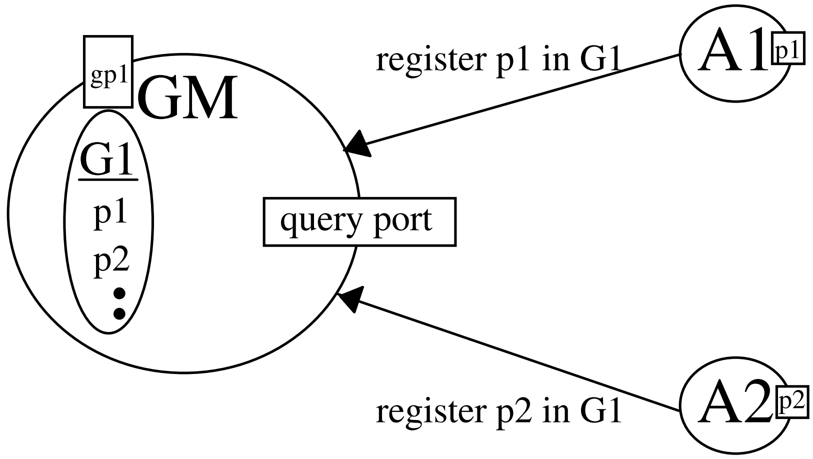

Figure 3: A User space Group Mediator (GM)

The GM's roles are as follows:

1. Creating ports which represent groups: To create a group, an application or the GM itself registers a group name at the GM. A port is allocated by the GM to represent the new group. Such a port is denoted as $gp1$ in figure 3.

2. Maintaining group membership: To join an already existing group, an application registers one of its ports under the group name via a call to the GM. The GM maintains membership lists (lists of ports owned by various applications) for each group.

3. Providing a global name service for these ports: When an application wants to send a group message it has to query the GM (using its query port) to obtain a send right to the port representing the group (a port owned by the GM).

4. Delivering messages to group members: On receipt of a message destined for one of the above ports, deliver the message to all group members (see figure 4).

5. Interfacing to other GMs, on other hosts, via the network thus maintaining global namespace of group ports on different hosts. Interfacing to various network communication protocols and media for delivery and receipt of group messages.

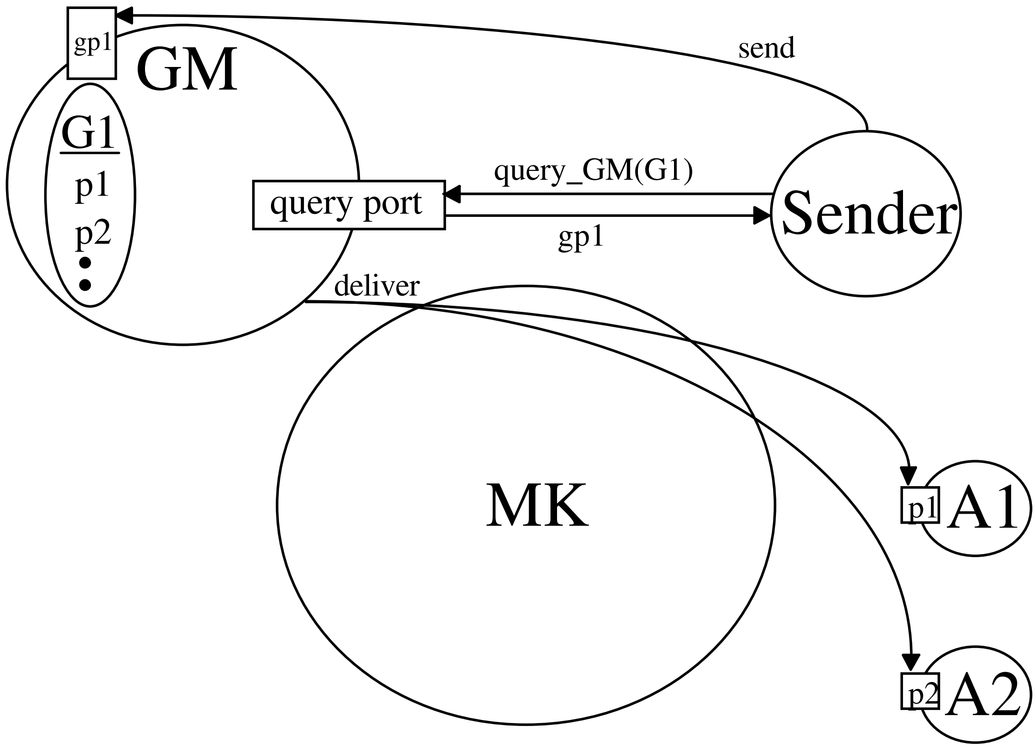

Figure 4: An *m-send* operation using Solution number 2

Analysis of the solution number 2

Step 1. Query the GM for a group port:

Given a message to multicast, obtain a port send right to the port representing the group (port rights have been described in section 3.3.1.). This step is performed once per group in each application, regardless of the group membership semantics (*static* or *dynamic*). The cost of this operation, which is actually a Mach IPC RPC call to the GM, is C_{query2}. C_{query2} is independent of the group size. Since it is performed only once during the lifetime of an application, when it sends the first message to the group, we will ignore this constant cost in our analysis.

Step 2. Send the message to the GM

The application sends the message (using a single Mach IPC message) to the group port owned by the GM (such as *gp1* in figure 4). This is actually a *mach_msg_trap* operation (system call) and consists of the following stages:

- A trap causing entry into kernel space.
- A preparation of a message header and a copy operation of the message header from user space to kernel space.
- Copying of the message body from user space to kernel space. If the message is sent out-of-line using the Mach Copy-On-Write (COW) optimization, there is no need to copy the message body, but a COW operation within the kernel is required instead.
- Finally, the message is delivered to a local task's port. This local delivery mainly consists of queuing the message in a port's queue. (We ignore the work done by the receiving thread when it gets the message.)

The cost of this step is:

$$C_{send2} = C_{trap} + C_{hcopy} + C_{cow} + C_{queue}$$

Step 3. Distribute the messages to group members

The GM selects the local destination ports and sends a Mach IPC message to each one of them. The remaining destinations (remote ports) are reached by the GM via network protocols, and possibly via other servers. If the GM has a built-in functionality to directly send messages to remote ports (in a similar way to the NetMsgServer), optimizations can be made at this stage: Instead of sending n messages separately, the GM can send one message for each subgroup of ports, where all members of a subgroup reside on the same host, to the GM on the other host. This is a sort of delayed replication.

The important observation here is that in the remote hosts, all messages are delivered by the remote GMs on a one-per-one basis. In the trivial case of n messages destined to local ports, n messages are sent by the GM using n local Mach IPC messages. Since this work is not concerned with evaluation of communication protocols or network delivery mechanisms, we will use this trivial case for our analysis, keeping in mind that all other cases have to perform at least n "local" deliveries, though not necessarily on the same host. Assuming asynchronous group communication, n local deliveries mean n *mach_msg_trap* operations (system calls). As explained above, each system call requires:

- A preparation of a message header (possibly omitting the group port owned by the GM and specifying the port owned by the real individual recipient) and a copy operation of the message header from user space to kernel space.
- If the message is sent out-of-line using the Mach Copy-On-Write (COW) optimization, there is no need to copy the message body n times, but still n COW operations within the kernel are required.

Again, it should be noted that this step is where the replication of the message occurs. Sending the message n times includes n replications of its header and n COW operations on its body.

The cost of this step is $C_{distribute2}$ where

$$C_{distribute2} = n(C_{trap} + C_{hcopy} + C_{cow})$$

Another possible optimization in the GM is using hardware multicast, if provided, to send a message to all ports accessible by a single hardware multicast message, but this type of optimization makes the GM aware of low-level communication layer details - something which should be left to the communication protocols instead.

In our analysis we will assume the trivial case: all messages are destined for local ports. We will discuss the possible optimizations later on.

Step 4. The kernel delivers n messages to their destinations

In the case of n messages destined for local ports, n local deliveries are made by the local kernel. In the more complex case, n local deliveries are also performed, but on different hosts. The cost is:

$$C_{deliver2} \geq nC_{local\_delivery}$$

As we have seen, a local delivery of a single message consists of:

- Queuing the message in its recipient task's queue (C_{queue}).
- One of the recipient task's threads gets the message. We will not include this operation in our analysis.

Thus, the cost of this step is:

$$C_{deliver2} \geq nC_{local\_delivery} = nC_{queue}$$

The total cost of solution 2 is:

$$C_{solution2} = C_{query2} + C_{send2} + C_{distribute2} + C_{deliver2}$$

Omitting C_{query2}, this reduces to:

$$C_{send2} + C_{distribute2} + C_{deliver2} = (n+1)(C_{trap} + C_{hcopy} + C_{cow} + C_{queue})$$

Advantages of solution 2

Representing groups by ports, as described above, has four advantages:

- It is simple and elegant. It nicely integrates with existing Mach semantics and abstractions such as ports and messages.
- It is flexible since it does not depend on any specific network hardware or network protocol.

- It is transparent to the application: It makes it possible to replace an existing Mach application with one that has been replicated for fault-tolerance, for instance. This can be done transparently: the users of the application will use its original interface without ever noticing that all its communication is being sent to a group.

- It relieves the application of the tedious duty to query the server for a membership list before sending a group message. A single query per group is sufficient now because the evaluation of the group membership list is deferred to a later stage and is performed within the GM itself.

Disadvantages of solution 2

- It is not scalable. Sending a group message to a group of size n costs n+1 message-send operations. This is very expensive. Like solution 1, this solution causes loss of potential concurrency since the operations are serialized.

- It is not efficient and it exhibits high latency: The group send operation cannot occur until a single message is sent and delivered to the GM. Thus, the latency is at least twice the latency of a regular Mach IPC message.

3.5 Solution Number 3: Everything in Kernel Space

The solutions presented so far are unacceptable due to the following reasons:

- They actually reduce a group communication operation to many unicast operations. This happens because the Mach kernel has no knowledge of group operations, group membership, etc. Everything has been done by the application or the group mediator (GM), in user space.

- The operation of sending a multicast message to a group of size n requires $O(n)$ system calls (traps) to the Mach kernel.

It is very difficult, if not impossible, to solve these problems in the framework of solutions number 1 and 2. A totally different solution is to place everything in the kernel. It is not difficult to see why this solution opens a wide variety of optimization possibilities. We can perform a group communication operation using a single Mach IPC call. This call will pass a single copy of the message to the kernel which, in turn, may perform optimized deliveries (using hardware multicast facilities). There is no need to obtain a group membership list because the kernel handles membership as well. We can actually implement the GM inside the kernel (see figure 5) and eliminate the costs of the interactions between the GM and the kernel, observed in solution number 2.

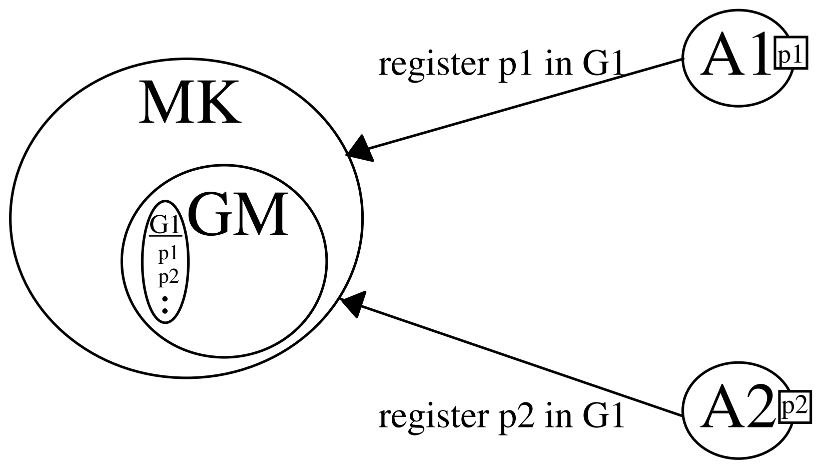

Figure 5: A Group Mediator (GM) within the Kernel

But this solution has its price. For the kernel to exploit all the possible optimizations, some non-trivial changes to Mach IPC have to be introduced. Something in the message itself should denote a group communication operation. The destination port can be interpreted as a group identifier which will be used to get the membership list, or a group name can be included in the message body and used to obtain the list. And finally, the interface between Mach IPC and the network protocols needs substantial reworking to reduce the overhead associated with going out of the kernel and into the Unix server (as done when TCP/IP is used for delivery).

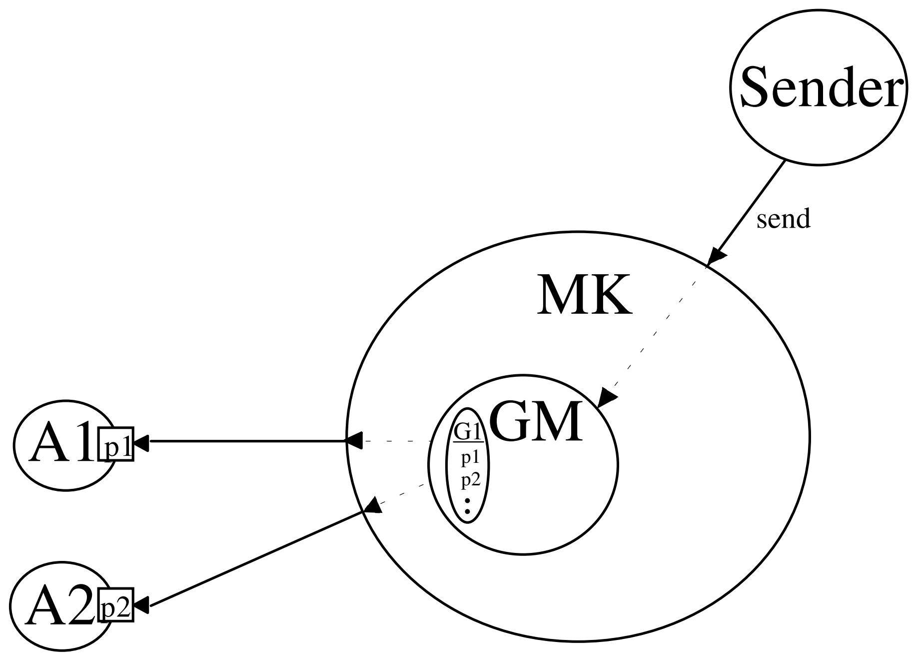

Figure 6: An *m-send* operation using Solution number 3

3.5.1 Analysis of solution 3

A single message is sent to the kernel (see figure 6), where its header is replicated n times. A replication of a header is probably cheaper than a copy operation of a header from user to kernel space. After the replication, n COW operations are performed, followed by n deliveries (we consider them local deliveries, as explained in solution 1 and 2). Thus the total cost of solution number 3 is:

$$C_{solution3} = C_{trap} + C_{hcopy} + n(C_{hreplicate} + C_{cow} + C_{queue})$$

Advantages of solution 3

- It is very efficient. We argue that the most efficient solution of the group communication problem is achieved in this way.
- It is transparent to the application.
- It is relatively scalable, depending on the way group messages are handled within the kernel. It allows for potential concurrency to be exploited since the operations are not necessarily serialized.

– It can be integrated with existing mechanisms and semantics. We did not discuss the interface between applications and the kernel providing group communication facilities; The interface can be via a well known kernel port or using a new/modified Mach IPC trap (system call). The first solution is more transparent than the second one.

– It is flexible in the sense that it can interface with any network device or protocol already implemented.

Disadvantages of solution 3

– It is difficult to implement.

– It is opposed to the main concept of microkernels: keeping the microkernel compact and simple. Introducing profound changes to microkernel code is considered harmful.

We abandon this efficient solution because we believe that a good hybrid solution can provide similar efficiency without sacrificing design principles.

3.6 The Hybrid Solution: The Kernel/Server Model

The solution we decided to implement is a hybrid solution supporting control operations in user space and data operations in the kernel. We believe this is the best solution because it makes it possible to clearly separate the semantics of multicast or group communication abstractions from the operating system support and mechanisms used to implement those abstractions. The former have been thoroughly investigated by researchers ([Ahamad85], [Liang90], [Birman91], [Birman87]) and we feel that the important issues are well understood while the latter are still an open issue. As explained in section 3.1.1., our main concern in designing a multicast or group communication mechanism was efficiency.

We believe that the host operating system should be the main target for optimization. It should be modified to cope with the new demands of high-speed networks and group communication oriented applications in three major aspects:

1. Design and implementation of efficient interfaces between applications or user-level servers and the microkernel.

2. Development of internal microkernel mechanisms to manipulate multicast data in a highly optimized manner.

3. Better support of modern networking hardware via innovative software (and hardware) design of microkernel/hardware interfaces and device drivers.

Our design and implementation focus on the first two aspects. While the third aspect has been a concern of hardware manufacturers [Banks93] it cannot be handled without involving specific hardware or networking protocol issues and thus is not further discussed in this work.

Since we wish to enjoy the efficiency of solution number 3 while introducing minimal changes to the Mach kernel, our hybrid solution suggests splitting the group communication burden between the microkernel and a Group Server (GS). This should be done with the following considerations in mind:

1. Decisions concerning semantics of group communication should not be part of the kernel because there is no apparent performance gain in placing them within the kernel and because we may need to support several different semantics at the same time.

2. Group membership maintenance can be performed by different algorithms using different semantics. We don't want to commit ourselves to any specific approach; thus we prefer handling these issues out of the kernel. There is also no apparent performance gain in handling these issues within the kernel.

3. The main problem in the first two solutions was loss of kernel awareness to the fact that group communication is being handled. The kernel view of this communication process was that of multiple unicast messages. To solve this problem we have to enhance the microkernel with the notion of group communication. The application will send a single message tagged as a group communication message to the kernel (reducing the interface between the application and the kernel into a single system call - a huge performance improvement by itself). The kernel, now aware of group communication, will recognize the special message as a message to a group, and perform the replication and delivery tasks, possibly optimizing them.

4. User level servers (maybe the GS's themselves) can still participate in relaying group messages between hosts, but the final delivery within each host will be done efficiently, using a single system call.

In order to implement our hybrid solution the Mach kernel should be modified as follows:

1. Mach IPC should support destination port lists.

2. A new system call (or a modified version of an existing system call) should be introduced. Applications will send a message to a group of ports using Mach IPC and the new/modified system call.

As for the GS used in this solution, it can be almost identical to the one presented in solution number 1 (see figure 1), but the message will be sent to the kernel only once (see figure 7) using the new Mach kernel primitive for group communication (i.e., the new, or modified, system call).

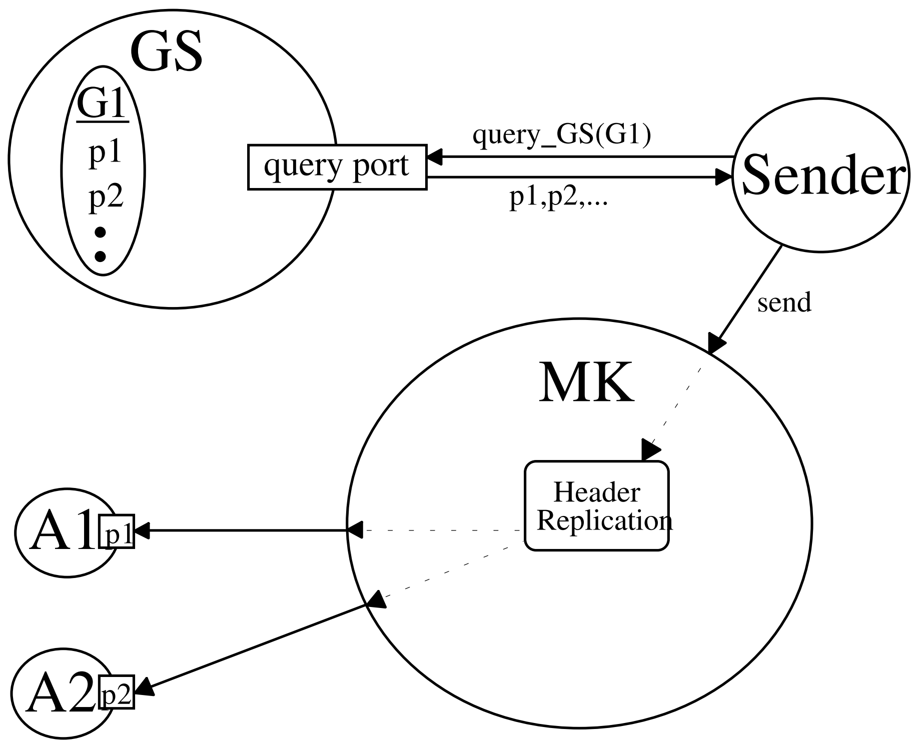

Figure 7: An *m-send* operation using the Hybrid Solution

3.6.1 Analysis of the Hybrid Solution

Step 1. Query the GS

Given a message to multicast, obtain a representation of its intended recipients (a list of ports owned by the group members) from the GS. The considerations mentioned in solution number 1 pertaining to the group being static or dynamic are valid here as well. We use the same cost:

$$C_{get\_group} = C_{msgrpc} + 2C_{cs} + c + kn$$

As discussed in the description of solution number 1, we can eliminate the need to query the GS before performing each group communication operation by using shared memory between the GS and the kernel. The application will use a group name (groups are checked in by names at the GS). The kernel will have, at all times, a correct representation (list of port rights) of all groups in its address space. This representation is maintained by the GS with a negligible cost of synchronization operations (mutual exclusion when accessing the shared memory). We will ignore this cost in our analysis.

Step 2. Send the group message once using modified Mach IPC

Use the new, or modified, kernel trap (Mach system call) to send the message. The breakdown of this operation is as follows:

- A trap causes entry into kernel space.
- The message header is copied **once** from user space to kernel space. Port rights in the recipient list are translated from a task representation to a kernel representation (object pointer).
- The message body is copied into kernel space. If the message is sent out of line using the Mach Copy-On-Write (COW) optimization, there is no need to copy the message body, but a COW operation within the kernel is required.

The cost of this step is C_{send_hybrid} where

$$C_{send\_hybrid} = C_{syscall} = C_{trap} + C_{hcopy} + C_{cow}$$

Step 3. The kernel delivers n messages to their destinations

- For each destination, replicate the header of the message (possibly omitting the group membership list).
- For each destination perform a COW operation of the message body (assuming Out-Of-Line messages).
- For each destination queue the message in its recipient task's queue.
- In each recipient task, one of its threads gets the message. We will not include this step in our analysis.

Assuming the case of n messages destined for local ports, the kernel makes n local deliveries. If some of the destinations are ports on other hosts, further protocol-dependent steps are required. Since the kernel now knows that a multicast operation is being performed, it can optimize deliveries using hardware multicast on interfaces supporting it, sending a single copy to many ports on the same machine, etc. Some of these optimizations can be done by servers in user space, maybe even the Group Server itself, though very delicate performance considerations should be taken when implementing such services in user space.

The cost of this step depends on the protocols used to deliver the messages; some of them may be local messages which will be handled by local Mach IPC. Others will use various communication protocols such as TCP/IP, which is implemented in user space (in the Unix server). For our analysis purposes we will take into account only the steps up to the message queuing within the kernel. This is in accordance with our goal to investigate the efficiency of the operating system and not that of the protocols themselves.

The cost of this step is:

$$C_{deliver\_hybrid} \geq n(C_{hreplicate} + C_{cow} + C_{local\_delivery}) = n(C_{hreplicate} + C_{cow} + C_{queue})$$

The total cost of our hybrid solution is:

$$C_{get\_group} + C_{send\_hybrid} + C_{deliver\_hybrid}$$

Omitting C_{get_group} as explained above yields the following total cost of the hybrid solution:

$$C_{hybrid} = C_{trap} + C_{hcopy} + C_{cow} + n(C_{hreplicate} + C_{cow} + C_{queue})$$

3.7 Comparative Analysis of the Solutions

In this section we perform analytic comparisons of the solutions suggested above. Since the difference between the cost of solution number 1 and solution number 2 is within a constant, and the difference between the cost of solution number 3 and the hybrid solution is within a constant as well, we will compare solution number 1 to the hybrid solution.

To calculate the expected speedup gained by using an enhanced multicast mechanism as presented in the hybrid solution over the naive multicast approach detailed in solution number 1, we use Amdahl's Law [Patterson90]:

$$speedup = \frac{execution\ time\ of\ a\ naive\ multicast\ operation\ as\ presented\ in\ solution\ 1}{execution\ time\ of\ an\ enhanced\ multicast\ operation\ as\ presented\ in\ the\ hybrid\ solution}$$

Thus, the expected speedup is:

$$speedup = \frac{C_{solution1}}{C_{hybrid}} = \frac{n(C_{trap} + C_{hcopy} + C_{cow} + C_{queue})}{C_{trap} + C_{hcopy} + C_{cow} + n(C_{hreplicate} + C_{cow} + C_{queue})}$$

$$\lim_{n \to \infty} speedup = \frac{C_{trap} + C_{hcopy} + C_{cow} + C_{queue}}{C_{hreplicate} + C_{cow} + C_{queue}}$$

3.8 Performance Speedup Prediction

Predicted speedup can be roughly estimated using results obtained by other researchers. In [Leprau93], a breakdown of a non-trapping RPC test program is presented. The measurements were made on a single HP9000/730 with a 67 MHz PA-RISC 1.1 processor running the Mach 3.0 kernel (NMK13) and the OSF/1 Single server (version 1.0.4b1). Table 1 presents results (measured in cycles) of the non-trapping RPC test program which are relevant to our research:

Stage	Description	Cycles
1	Call to *mach_msg_trap*, trap into kernel	350
2	*mach_msg_trap* operations before copyin	18
3	*Copyin* message to kernel address space	205
4	"Red tape" (port name lookup, generating rights, locking and unlocking kernel structures, etc.)	159
5	Context switch from client task to server task	178

Table 1: Measurements of a non-trapping RPC test program

We can map the first two stages to our definition of C_{trap}. Thus, for this specific test, $C_{trap} = 368$. Stage 3 comprises copying of both the header and the message body (no COW operations have been used in this test). Since stage 5 can happen only after the message has been queued for delivery, we conclude that Stage 4 contains the queuing operation. Thus, for certain message sizes (above some unknown minimal size):

$$C_{hcopy} + C_{cow} + C_{queue} \leq stage3 + stage4 = 364$$

Since $C_{hreplicate} < C_{hcopy}$ (the former happens within the kernel itself while the latter entails moving data from user space to kernel space), we conclude that:

$$C_{hreplicate} + C_{cow} + C_{queue} < 364$$

In summary, our predicted speedup for very large n's is at least $\frac{368+364}{364} = 2.01$. This means that our hybrid solution will exhibit group communication which is at least double in performance than naive solutions such as solution number 1. We would expect even a bigger speedup due to the inequalities in the above calculations.

4 Implementation of the Hybrid Solution

4.1 Implementation Guidelines

Our aim was to investigate the feasibility and advantages of the hybrid solution: control operations in a server, data operations in the kernel. We wanted to measure the speedup of this solution as compared to solution number 1. As noted in section 3.4.4., at least two kernel modifications are needed in order to implement the hybrid solution:

1. Mach IPC should support destination port lists.
2. A new system call (or a modified version of an existing system call) should be introduced. Applications will send a message to a group of ports using Mach IPC and the new/modified system call.

The following guidelines led us while designing the modifications:

1. Slight modifications to the existing message structure are better than inventing a new message format especially designed for group messages. A new message structure, specifically designed for group communication, will necessitate numerous changes both in the kernel and outside. Since Mach relies heavily on MIG, to support a new message format, MIG will have to be modified thoroughly.

2. Modifications to the existing message structure should be as minimal as possible. This will make it easier to incorporate support for the new message structure into MIG and in various kernel routines interacting with Mach messages. If the modifications are carefully planned, kernel routines which do not have to understand or manipulate destination port lists, can remain untouched.

3. If possible, the modifications should be transparent to ignorant kernels (those who do not understand group messages). Such kernels should be able to deliver the message to a single destination or at least understand that the message is not a simple one and return an error to the sender.

4. Parsing of group destination lists should be as efficient as possible.

5. Data movement should be as efficient as possible.

The above guidelines led us to the following decisions:

1. Group destination lists should be a part of the message data. Ignorant kernels will either treat them as regular data or ignore them.

2. Something in the message header should denote that the message contains, among other things, group membership lists.

4.2 Adding destination port lists support to Mach IPC messages

Each Mach IPC message begins with a single message header structure (*mach_msg_header_t*) describing the overall message structure and contents. The data being sent is preceded by a data type specifier (the *mach_msg_type_t* structure). A message may contain multiple data type specifiers, each followed by a chunk of data. Figure 8 illustrates the idea:

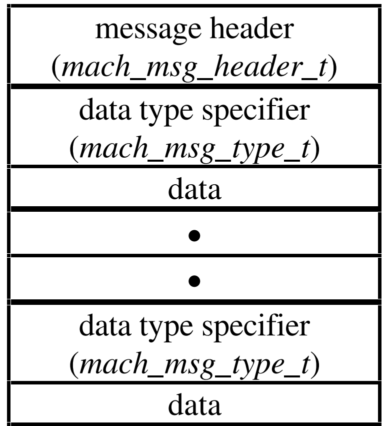

Figure 8: Mach IPC message with multiple data chunks

To support destination port lists in Mach IPC messages we had to modify two structures: The Mach IPC message header (*mach_msg_header_t*) should include a flag denoting that the message is a group message (in accordance with decision 2 above) and the data type specifier (*mach_msg_type_t*) should have provisions to incorporate group destination lists as data within the message (in accordance with decision 1 above).

The original structure of the Mach IPC message header is presented in Figure 9. It should be noted that bits 16 through 26 are unused in the original Mach 3.0 message header.

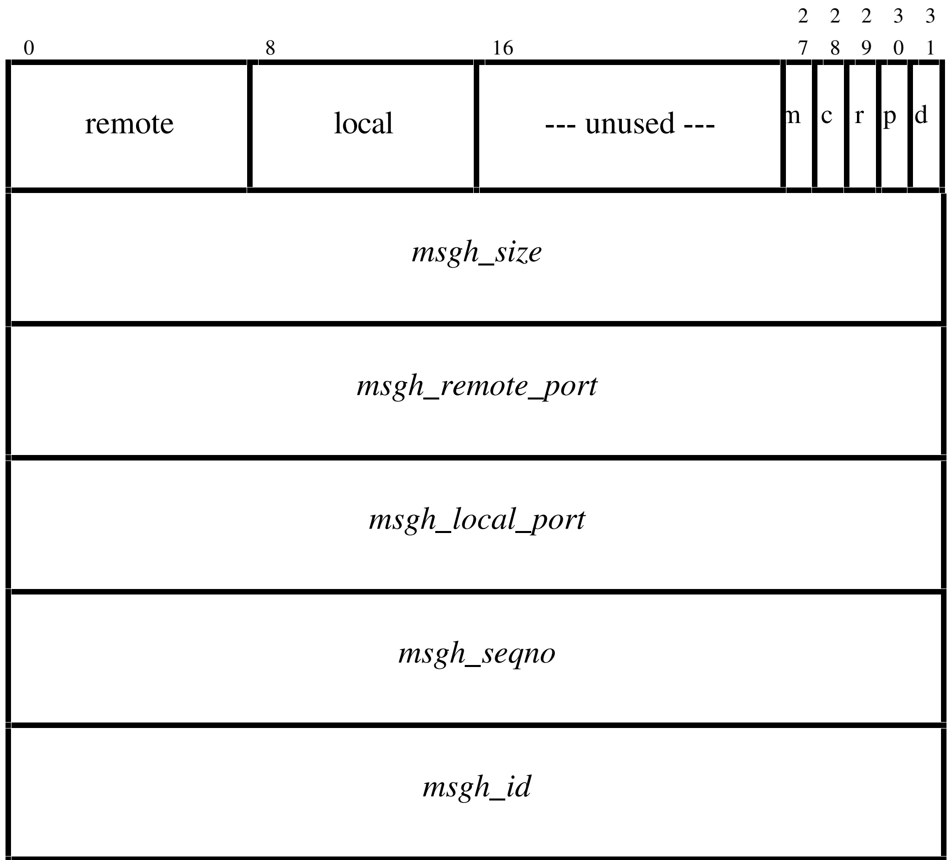

Figure 9: Original Mach IPC message header

We used one of the 11 unused bit flags in the Mach IPC message header to denote a group message:

#define MACH MSGH BITS GROUP   0x00080000

#define MACH MSGH BITS UNUSED   0x07ff0000

The MACH MSGH BITS GROUP bit is bit 19 in the header. It is shown as the 'g' bit in figure 10.

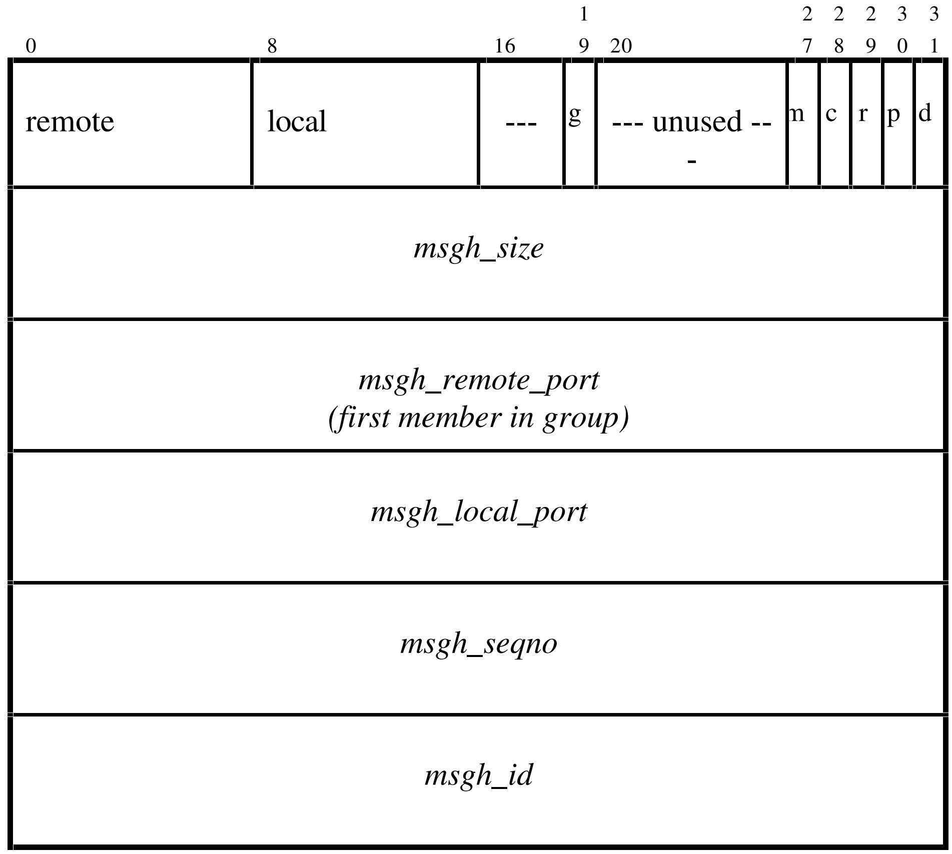

Figure 10: Modified Mach IPC message header with group support

When a group message is sent, the 'g' bit is turned on and the *msgh_remote_port* field contains the first member of the destination group. This setting makes the message compatible with ignorant kernels (those which do not support group communications), theoretically ensuring that at least one group member receives the message. Regrettably, this cannot work without introducing a slight modification: Since current Mach IPC code insists on bits 16 through 26 being clear, the message will not be delivered to any recipient unless the receiving kernel is modified to ignore the value of the g bit (or all of the bits 16 though 26).

The original structure of the data type specifier (*mach_msg_type_t*) is presented in Figure 11.

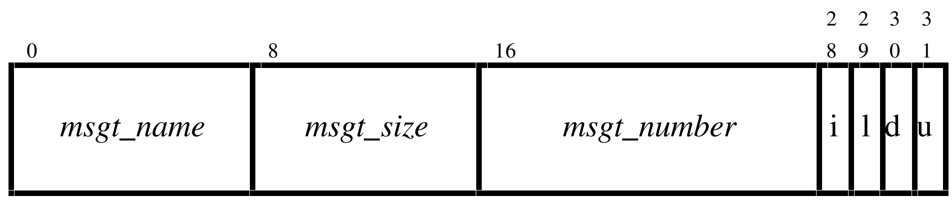

Figure 11: Original Mach IPC data type specifier (*mach_msg_type_t*)

The *msgt_name* field specifies the type of the elements in the chunk. Each data element is of size *msgt_size* (specified in bits). The *msgt_number* field specifies the number of data elements in the chunk. If *msgt_inline* (bit 'i') is TRUE, the data follows the type descriptor in the body of the message. If *msgt_inline* is FALSE, then a pointer to the data should follow the type descriptor, and the data is sent out-of-line. In this case, if *msgt_deallocate* (bit 'd') is TRUE, then the out-of-line data is moved (instead of being copied) into the message. Bit 'u' is unused.

If the value to be stored in the *msgt_size* or *msgt_number* fields is larger than fits in the field, MACH IPC defines an alternate structure to describe the data: the *long-form* version. When *msgt_longform* (bit 'l') is TRUE, the type descriptor is actually a *mach_msg_type_long_t*. The structure of the longform data type specifier (*mach_msg_type_long_t*) is presented in Figure 12.

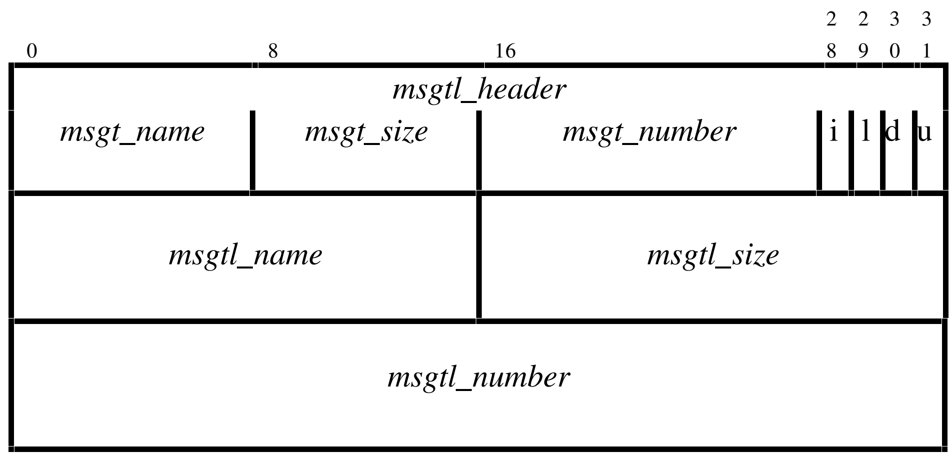

Figure 12: Original Mach IPC long data type specifier (*mach_msg_type_long_t*)

When the longform structure is used the *msgt_name*, *msgt_size* and *msgt_number* fields in the *msgtl_header* should be zero. We use these fields for group messages as described below.

If group communication is enabled, *msgt_longform* should be TRUE and the fields are used as follows: *msgt_name*, *msgt_size* and *msgt_number* describe the group of remote ports (*msgt_number* is the number of members in the group). *msgtl_name*, *msgtl_size*, and *msgtl_number* describe the data. *msgt_deallocate* is ignored: the data is always moved (and not copied). *msgt_inline* should be FALSE (in spite of the fact that inline data is used to describe the group members). The pointer to the data should follow the type descriptor (it should appear before the group member list). The group member list should contain at least one member - the remote port that appears in *msgh_remote_port*. Figure 13 illustrates the body of a group message using our modifications.

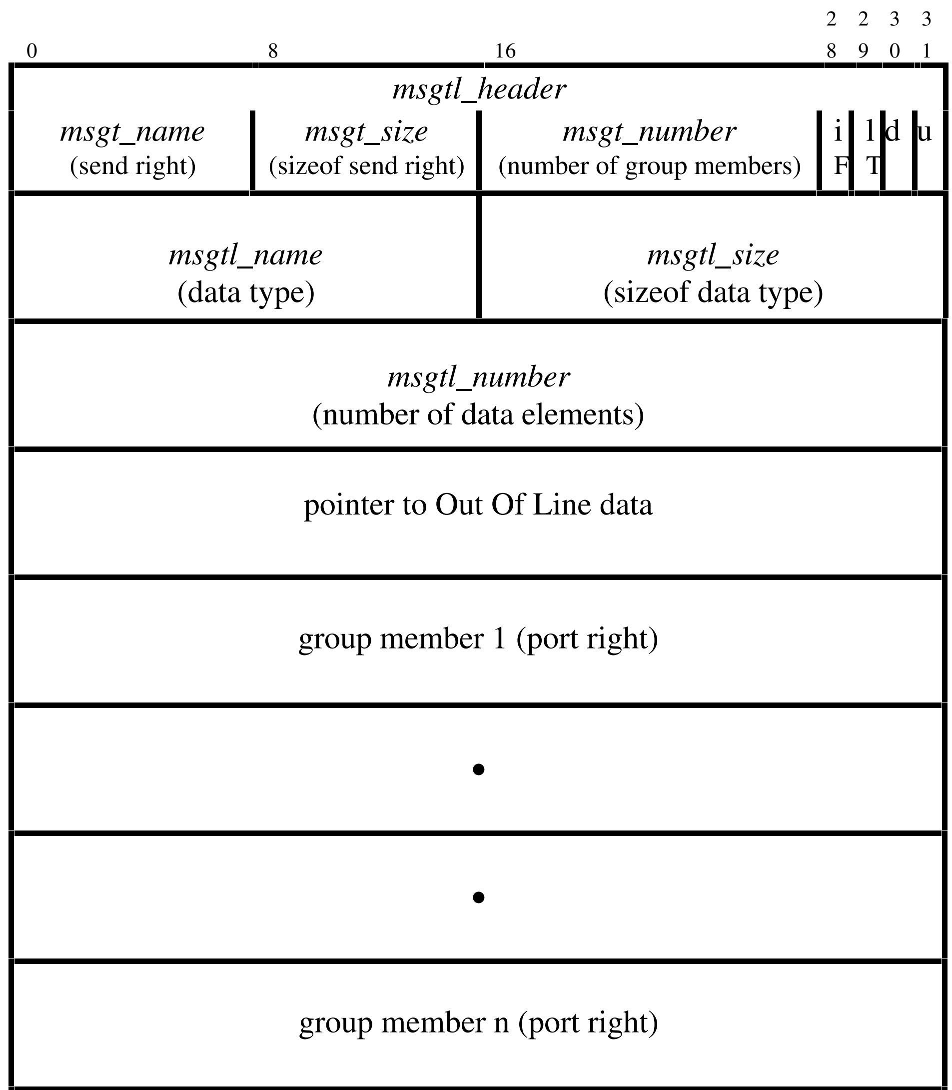

Figure 13: Modified Mach IPC message body with group list support

4.3 System Call Modifications

The first dilemma was: whether we should incorporate changes into an existing system call or whether we should introduce a new system call.

There are two important advantages to a new system call:

1. It is easier to develop and debug. It cannot harm the operation of the kernel since it

doesn't interfere with existing system call code thus eliminating side-effect issues.

2. It necessitates fewer (or no) changes to existing IPC message format; there is no need to identify a message as a group message since the system call handling it is dedicated to group messages.

But there are many disadvantages as well:

1. A whole set of routines must be crafted to handle group communication while many interactions with group messages are identical to interactions with regular messages. This means that many lines of code for handling group messages have to be replicated, thus making the microkernel larger.

2. Ignorant kernels will not be able to handle group messages since they will not have the new system call installed.

3. This solution gives application programs a new responsibility: they must know if a message is a group message or regular message and use the suitable system call accordingly.

4. It is natural to see a group message as an extension of a regular message (a message intended for a single recipient). If a new system call is introduced, the trivial case of a message destined for a group of size 1 will be handled differently from a regular message. This is somewhat unnatural.

We decided to make slight modifications to the existing system call. This is easier said than done since the immediate problem is how to debug the modified kernel. Too many factors depend on the correct functioning of the *mach_msg_trap* (the system call handling mach messages). If we introduce the slightest bug in the system call, the system may halt or function oddly.

Thus we looked for a solution in which bugs will not impair functionality other than the tested mechanisms. We decided to replicate the *mach_msg_trap* code and rename it *mach_gmsg_trap* during the development phase. Once the new code has been debugged it can be merged back.

The system call used to send or receive a message (or both, using the same message buffer when optimizing for RPC) in Mach is *mach_msg*() (which is mapped to the kernel function *mach_msg_trap*()).

The *mach_msg_trap*() function uses additional kernel functions to perform its role. Figure 14 shows a simplified representation of the work done by those functions when a message is being sent.

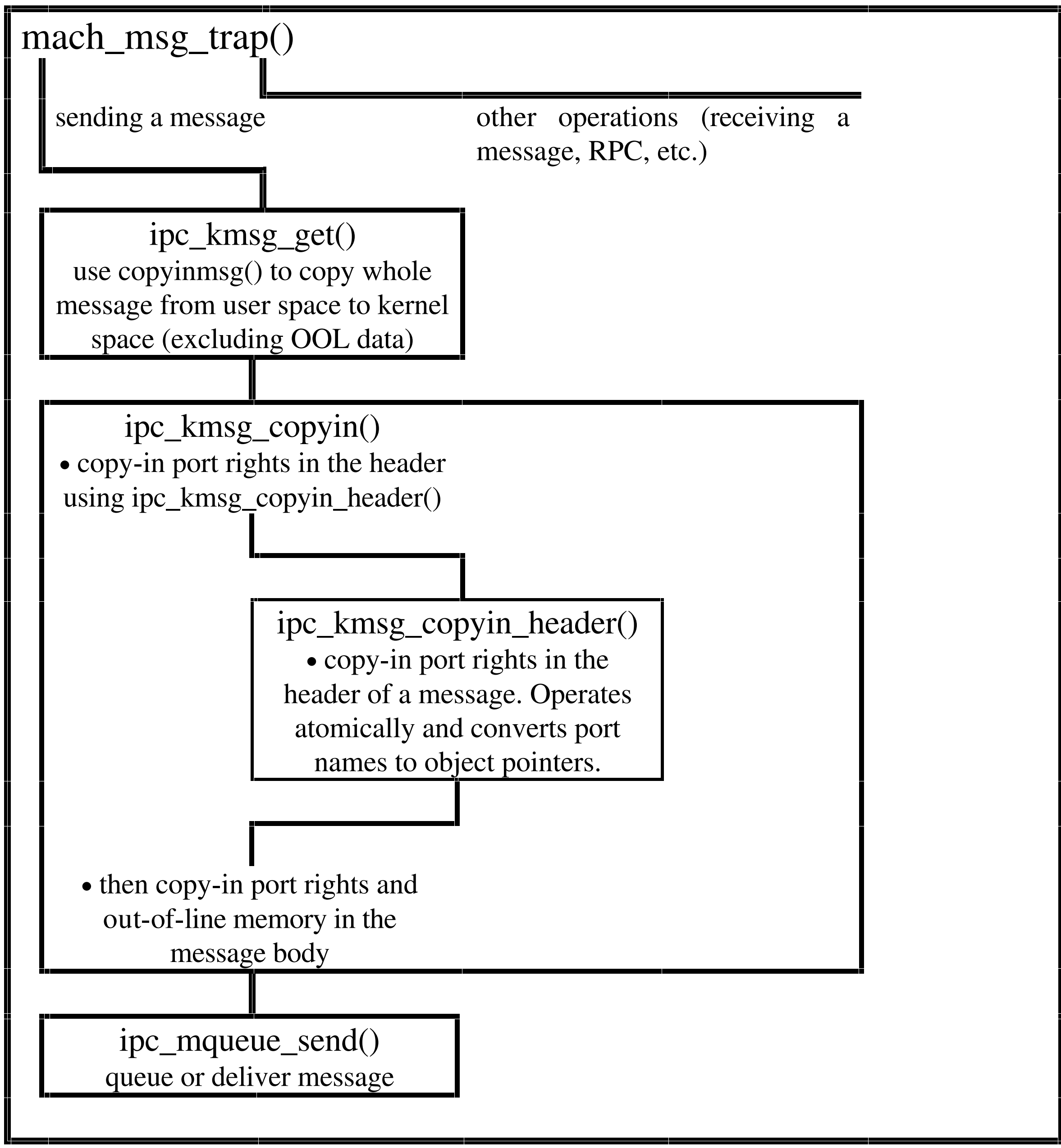

Figure 14: Sending a message using *mach_msg_trap* and its associated kernel functions

A more detailed explanation is needed in order to explain our design decisions. When sending a message the kernel has to perform several "copy-in" operations:

The first operation (performed by *ipc_kmsg_get*()) is simply copying raw bytes from user space to kernel space. During this phase the whole message, including the header, the data-type specifiers and the data chunks (the whole structure described in Figure 8) is copied to kernel space. It is important to understand that pointers are copied without change. The (out-of-line) data indicated by these pointers is not copied. Port names (which are task namespace dependent) are copied without changes as well.

The next operation has to translate the above mentioned pointers and port names to a meaningful representation within kernel space. *ipc_kmsg_copyin_header*() handles the header of the message. Message headers always contain port names (source and destination) and their translation is referred to as "copy-in" of port rights, which is a misnomer in our opinion.

Finally, port names in the message body (it is possible to send port right in a message) and pointers to OOL data (as well as the OOL data itself) have to be treated. The message body is parsed, chunk by chunk, using the data type descriptor heading each chunk. If the type denotes a port right, it is being translated into an object pointer. OOL data is being copied in and the pointer updated. If the original OOL data is destined for deallocation the copy-in operation may be performed by optimized VM operations.

The translations have to be done atomically because the objects could be manipulated by other tasks concurrently. It is important to pay attention to the fact that the whole message is being scanned twice during the above mentioned stages.

Our design of group messages leaves the header almost unchanged (except the 'g' bit). All the group list information is kept in the message body. Thus when translating port names to object pointers in the header (there are two port names in each header: source and destination) we must parse the message body as well, locate the group list and translate port names to object pointers. Since all group messages must use OOL data in our design, we have to copy in the data once and then increment references to the first (and only) copy.

As explained above, during the debugging phase of our implementation of group communication, we used a special function, *mach_gmsg_trap*(). If the message being sent is not a group message, or if the operation is not sending a simple message, the original *mach_msg_trap*() is called. Otherwise, our new functions are used as specified in Figure 15.

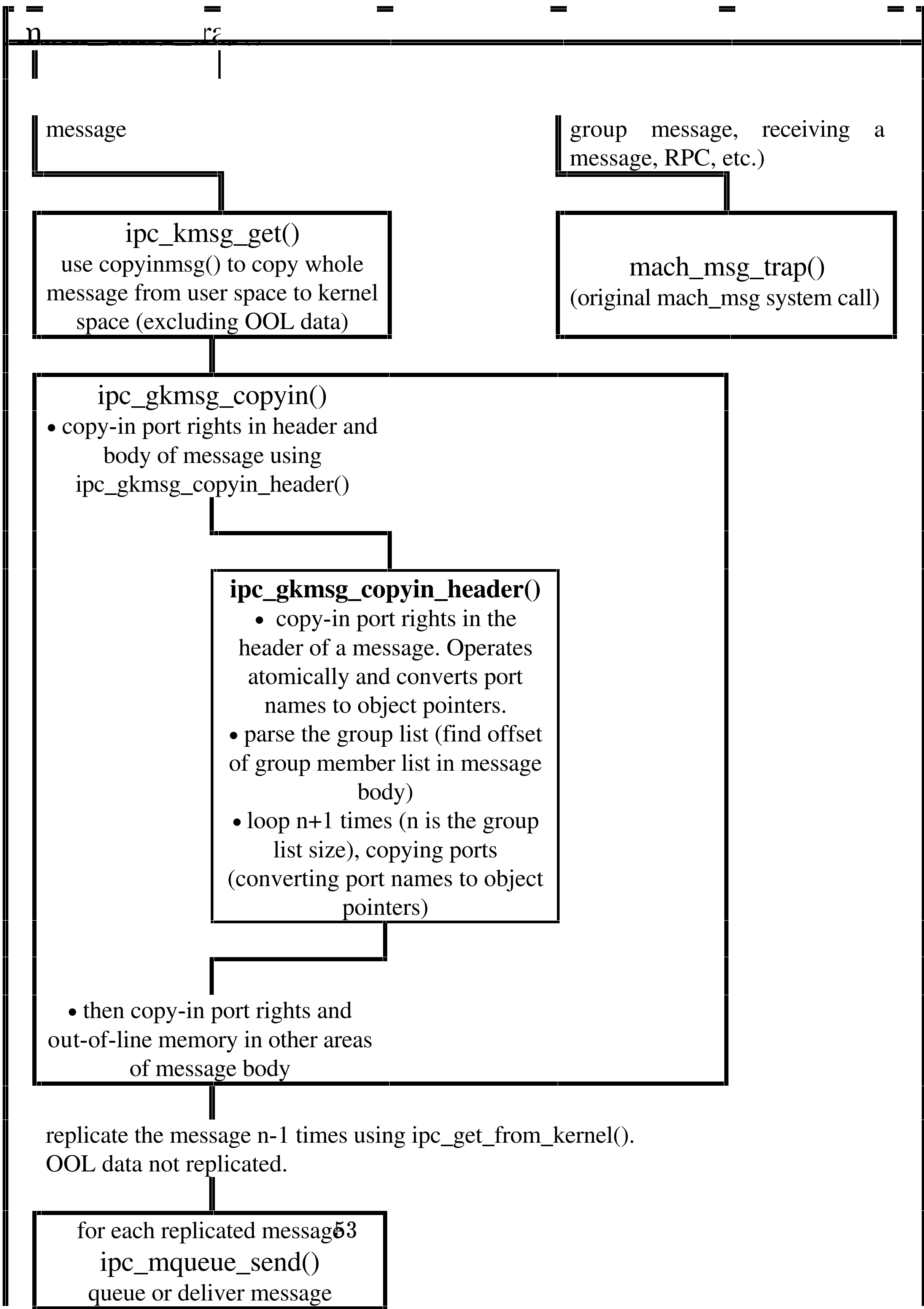

Figure 15: Sending a group message using *mach_gmsg_trap*

5 Benchmark Results

In this section we present the benchmarks we used to evaluate our implementation of the enhanced multicast operation (the hybrid solution) as compared to naive multicast operations (solution 1). We discuss some delicate issues related to performance measurement in general and the performance of our implementation in particular. Finally we introduce the results of our measurements and compare the measured speedup to the theoretically expected one.

5.1 Benchmarks used

We used a modified version of the time server and client programs presented in [Boykin93]. The programs have been converted to use Mach 3.0 conventions. The client program has been modified to send time queries to multiple ports using our modified kernel trap (*mach_gmsg_trap*). We used group sizes of 10, 20, 50, 100 and 1000. The results obtained from these tests have been compared to the results of running a client program sending multiple queries (10, 20, 50, 100 and 1000) sequentially using the original mach trap (*mach_msg_trap*).

5.2 Performance Measurements

Measuring the performance of our implementation has been a non-trivial task and we have had to overcome several problems.

We did not have hardware capable of measuring processor cycles such as the Quamachine used by Massalin [Massalin92]. His machine has measurement facilities such as an instruction counter, a memory reference counter, hardware program tracing, and an interval timer with 20-nanosecond resolution. We did not have a clock with such resolution, or even one supporting resolutions of 10 or 100 μsec. The system clock supports a resolution of 10 milliseconds only. Thus we employed a common solution widely used in chemistry or physics laboratories: to run many consecutive operations, measure the total elapsed time and calculate the (average) time per operation. To eliminate inaccuracies incurred by the clock's coarse granularity the elapsed time should be at least one or two orders of magnitude larger than the clock resolution. This means hundreds, thousands or even more messages should be sent at once to get significant measurements. Sending thousands of messages or more at once introduces additional problems:

– Since the Mach Microkernel is multithreaded, messages that have been sent are not handled sequentially: some of the messages (the first ones) are being queued and delivered while others (later ones) are still being processed by the kernel. This makes it impossible to measure times in a straightforward way, based on elapsed time. Measuring elapsed time (from the invocation of *mach_gmsg_trap* until the return from the trap) would measure not only the message-send operation but also queuing, scheduling operations and context switches related to the delivery of messages to servers and the time consumed by the servers while receiving some of the messages.

  In order to solve this problem we introduced a slight change in our modified kernel trap: we prevented *mach_gmsg_trap* from queuing messages for delivery. Allocation and copying operations still performed as usual but the kernel routine usually invoked to put a message in a queue so that the intended recipient can accept it, replaced by a null routine. This means that while taking measurements no group messages could be delivered.

In order to compare multicast messages to unicast ones our client test program calls *mach_gmsg* for unicast messages as well, but without turning on the MACH_MSGH_BITS_GROUP bit in the header of the message. Our *mach_gmsg* routine has been modified to handle such messages in the same way handled by *mach_msg* except that delivery to the recipient is never performed.

Since no messages are delivered, the buffer space consumed by replicated messages is never freed. This leads to another problem.

– We do not want to measure paging activity caused by heavy memory usage due to allocation of thousands or tens of thousands of message buffers, header data structures, etc. In such situations, paging can be a severe artifact with dramatic influence on our measurements. Even without extra paging, some memory or resource allocation algorithms become extremely inefficient in situations such as shortage of memory, depletion of kernel tables or other resources.

We solved this problem in two ways:

1. Rebooting the machine between consecutive runs of the benchmarks.

2. Running the benchmarks until a performance degradation is noticed. This means we have reached some resources' limits. We can ignore the measurements taken from that poing and onwards.

In addition to the above measurements we measured null *mach_gmsg_trap* calls (system call entry and *thread_syscall_return* exit) within each run. This can be done by calling *mach_gmsg_trap* (or the original *mach_msg_trap* call) with the option MACH_MSG_OPTION_NONE. This call causes an immediate return using *thread_syscall_return*(MACH_MSG_SUCCESS).

The reason for conducting these additional measurements was twofold:

1. It is needed to ensure that no unrelated or background activity is being performed while our tests are running. This null test can be seen as a control mechanism for our measurements.

2. We were interested in the overhead of a naked call to *mach_gmsg_trap*, the trap into the kernel and the return. The cost of these operations exactly matches our definition of C_{trap}. It is revealing to compare this cost with other costs to get some idea of the savings involved in using multicast instead of many unicast messages.

We used a PC with an Intel 486 33MHz processor and 8 MB of memory. The kernel used was Mach 3.0 MK83 with our modifications (STD+WS+gmsg) and the Unix server was UX42 (STDNFS+WS). Tests were performed on an idle system with a minimal number of servers running, no logins permitted, etc.

We conducted 4 sets of tests: two of them at full processor speed (33MHz) and two at slow speed (16 MHz). Each set consisted of 4 runs for each of the 5 group sizes tested (10, 20, 50, 100 and 100), each of them followed by a reboot of the machine (5 reboots per set).

The following table describes one set in detail:

4 consecutive runs	with group size 10 each of which consists of 1000 iterations of sending a message to a group of 10 members, followed by 10000 iterations of sending a message to a single destination, followed by 10000 iterations of null *mach_gmsg_trap* call
Reboot	
4 consecutive runs	with group size 20 each of which consists of 500 iterations of sending a message to a group of 20 members, followed by 10000 iterations of sending a message to a single destination, followed by 10000 iterations of null *mach_gmsg_trap* call
Reboot	
4 consecutive runs	with group size 50 each of which consists of 200 iterations of sending a message to a group of 50 members, followed by 10000 iterations of sending a message to a single destination, followed by 10000 iterations of null *mach_gmsg_trap* call
Reboot	
4 consecutive runs	with group size 100 each of which consists of 100 iterations of sending a message to a group of 100 members, followed by 10000 iterations of sending a message to a single destination, followed by 10000 iterations of null *mach_gmsg_trap* call
Reboot	
4 consecutive runs	with group size 1000 each of which consists of 50 iterations of sending a message to a group of 1000 members, followed by 50000 iterations of sending a message to a single destination, followed by 50000 iterations of null *mach_gmsg_trap* call

Table 2: A breakdown of a set of tests

5.3 Results

The measurements obtained in our tests are presented in the following tables. Tables 3 to 7 show results obtained at 33 MHz, with group sizes 10, 20, 50, 100 and 1000. Tables 8 to 12 show results obtained at 16MHz. The calculated speedup for each and every run is shown as well.

Iterations	Operation	Set1				Set2			
		run1	run2	run3	run4	run1	run2	run3	run4
10000	unicast (ms)	190	190	190	210	180	190	220	220
1000	multicast (ms)	80	80	80	830	80	70	90	730
	speedup	2.38	2.38	2.38	0.25	2.25	2.71	2.44	0.30
10000	Null RPC (ms)	110	100	100	120	100	110	120	120

Table 3: Measurements with group size of 10 (processor running at 33 MHz)

Iterations	Operation	Set1				Set2			
		run1	run2	run3	run4	run1	run2	run3	run4
10000	unicast (ms)	190	180	210	190	190	190	190	200
500	multicast (ms)	40	40	50	50	50	40	50	50
	speedup	4.75	4.50	4.20	3.80	3.80	4.75	3.80	4.00
10000	Null RPC (ms)	110	120	130	110	100	110	110	110

Table 4: Measurements with group size of 20 (processor running at 33 MHz)

Iterations	Operation	Set1				Set2			
		run1	run2	run3	run4	run1	run2	run3	run4
10000	unicast (ms)	180	190	200	190	190	180	190	190
200	multicast (ms)	30	30	30	40	30	30	30	30
	speedup	6.00	6.33	6.67	4.75	6.33	6.00	6.33	6.33
10000	Null RPC (ms)	110	110	110	110	100	100	100	110

Table 5: Measurements with group size of 50 (processor running at 33 MHz)

Iterations	Operation	Set1				Set2			
		run1	run2	run3	run4	run1	run2	run3	run4
10000	unicast (ms)	190	180	180	190	200	180	190	210
100	multicast (ms)	20	20	30	30	20	20	30	30
	speedup	9.50	9.00	6.00	6.33	10.00	9.00	6.33	7.00
10000	Null RPC (ms)	110	100	110	110	110	110	120	130

Table 6: Measurements with group size of 100 (processor running at 33 MHz)

Iterations	Operation	Set1				Set2			
		run1	run2	run3	run4	run1	run2	run3	run4
50000	unicast (ms)	930	910	1130	970	900	880	1050	920
50	multicast (ms)	110	120	160	400	120	130	120	380
	speedup	8.45	7.58	7.06	2.42	7.50	6.77	8.75	2.42
50000	Null RPC (ms)	530	520	630	530	510	610	510	510

Table 7: Measurements with group size of 1000 (processor running at 33 MHz)

Iterations	Operation	Set1				Set2			
		run1	run2	run3	run4	run1	run2	run3	run4
10000	unicast (ms)	840	770	780	920	860	810	880	910
1000	multicast (ms)	260	270	270	1030	260	270	300	590
	speedup	3.23	2.85	2.89	0.89	3.31	3.00	2.93	1.54
10000	Null RPC (ms)	390	400	400	390	400	390	390	400

Table 8: Measurements with group size of 10 (processor running at 16 MHz)

Iterations	Operation	Set1				Set2			
		run1	run2	run3	run4	run1	run2	run3	run4
10000	unicast (ms)	830	820	830	840	830	820	830	830
500	multicast (ms)	190	180	180	190	180	180	190	180
	speedup	4.37	4.56	4.61	4.42	4.61	4.56	4.37	4.61
10000	Null RPC (ms)	460	400	390	390	400	390	390	390

Table 9: Measurements with group size of 20 (processor running at 16 MHz)

Iterations	Operation	Set1				Set2			
		run1	run2	run3	run4	run1	run2	run3	run4
10000	unicast (ms)	820	830	830	830	780	780	780	770
200	multicast (ms)	120	120	130	130	130	120	140	130
	speedup	6.83	6.92	6.38	6.38	6.00	6.50	5.57	5.92
10000	Null RPC (ms)	390	390	400	390	400	390	400	410

Table 10: Measurements with group size of 50 (processor running at 16 MHz)

Iterations	Operation	Set1				Set2			
		run1	run2	run3	run4	run1	run2	run3	run4
10000	unicast (ms)	770	770	780	770	780	780	770	780
100	multicast (ms)	110	110	110	110	110	110	120	120
	speedup	7.00	7.00	7.09	7.00	7.09	7.09	6.42	6.50
10000	Null RPC (ms)	390	390	390	390	390	390	400	390

Table 11: Measurements with group size of 100 (processor running at 16 MHz)

Iterations	Operation	Set1 run1	Set1 run2	Set1 run3	Set1 run4	Set2 run1	Set2 run2	Set2 run3	Set2 run4
50000	unicast (ms)	3920	3930	4100	3930	4050	4090	4230	4090
50	multicast (ms)	470	480	580	560	480	480	560	550
	speedup	8.34	10.27	7.07	7.02	8.44	8.52	7.55	7.43
50000	Null RPC (ms)	1960	1980	1950	1950	1960	1950	1960	1960

Table 12: Measurements with group size of 1000 (processor running at 16 MHz)

5.4 Discussion

The detailed speedup results presented in the previous section and the average speedup (over all runs) for each group size are summarized in tables 13 and 14.

Group Size	Set1 run1	Set1 run2	Set1 run3	Set1 run4	Set2 run1	Set2 run2	Set2 run3	Set2 run4	Average Speedup
10	2.38	2.38	2.38	0.25	2.25	2.71	2.44	0.30	1.89
20	4.75	4.50	4.20	3.80	3.80	4.75	3.80	4.00	4.20
50	6.00	6.33	6.67	4.75	6.33	6.00	6.33	6.33	6.09
100	9.50	9.00	6.00	6.33	10.00	9.00	6.33	7.00	7.89
1000	8.45	7.58	7.06	2.42	7.50	6.77	8.75	2.42	6.37

Table 13: Speedup results obtained with processor running at 33 MHz

Group Size	Set1 run1	Set1 run2	Set1 run3	Set1 run4	Set2 run1	Set2 run2	Set2 run3	Set2 run4	Average Speedup
10	3.23	2.85	2.89	0.89	3.31	3.00	2.93	1.54	2.58
20	4.37	4.56	4.61	4.42	4.61	4.56	4.37	4.61	4.51
50	6.83	6.92	6.38	6.38	6.00	6.50	5.57	5.92	6.31
100	7.00	7.00	7.09	7.00	7.09	7.09	6.42	6.50	6.90
1000	8.34	10.27	7.07	7.02	8.44	8.52	7.55	7.43	8.08

Table 14: Speedup results obtained with processor running at 16 MHz

These results demonstrate a very significant speedup obtained by our kernel modifications. Sending multicast messages to groups of 10 to 1000 recipients using our Hybrid solution and its implementation using a special kernel trap (*mach_gmsg_trap*) can be as much as 2 to 8 times faster then other, naive solutions. This is in accordance with our analytical prediction of a double speedup at least.

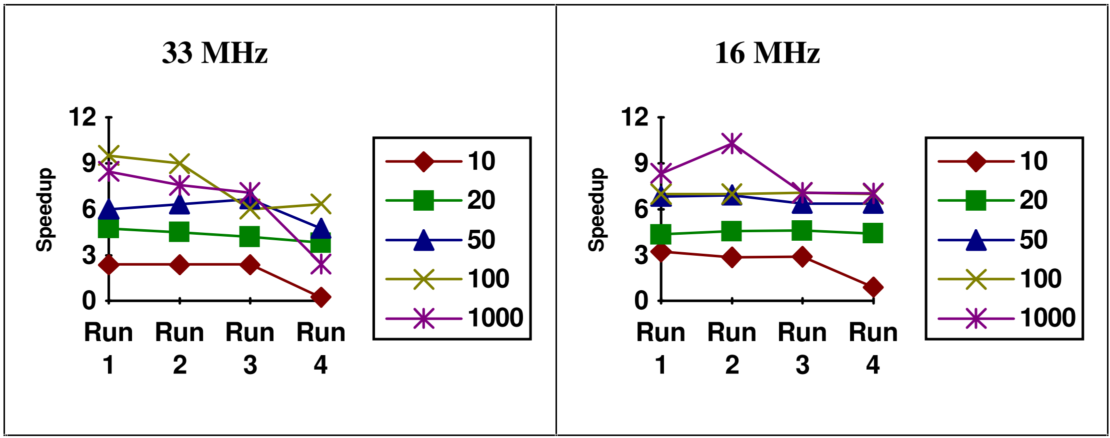

Figure 16: Speedup obtained in four runs (from set 1) for different group sizes

Figure 16: Speedup obtained in four runs (from set 1) for different group sizes

There are, however, some interesting phenomena which deserve a special observation. A glance at tables 13 and 14 and at figure 16 reveals two "anomalies":

1. In both processor speeds, run 4 exhibits a drastic drop in speedup for some group sizes. We expected such a drop due to depletion of memory resources and therefor it may be more reasonable to calculate speedups based on the first three runs in each set. Table 15 shows speedups averaged over four runs compared to those averaged over the first three runs in each set. The latter are higher of course.

Group Size	Processor running at 33 MHz			Processor running at 16 MHz		
	Average Speedup		Null Trap (μsec)	Average Speedup		Null Trap (μsec)
	2 sets of 4 runs	first 3 runs in each set		2 sets of 4 runs	first 3 runs in each set	
10	1.89	2.42	11.00	2.58	3.04	39.50
20	4.20	4.30	11.25	4.51	4.51	40.13
50	6.09	6.28	10.63	6.31	6.37	39.63
100	7.89	8.31	11.25	6.90	6.95	39.13
1000	6.37	7.69	10.88	8.08	8.37	39.17

Table 15: Comparison of Average Speedups

2. Tests run at high processor speed (33MHz) exhibit a drop in average speedup for group size 1000 (see figure 17). This does not happen in a lower processor speed. A possible

explanation could be some sort of background garbage collection activity or other system activity not related to our tests, but the results of our control measurements rule out this explanation: The steady measurement results of null *mach_gmsg_trap* calls show that the machine itself is not loaded by other activities or any sort of garbage collection activity in the background.

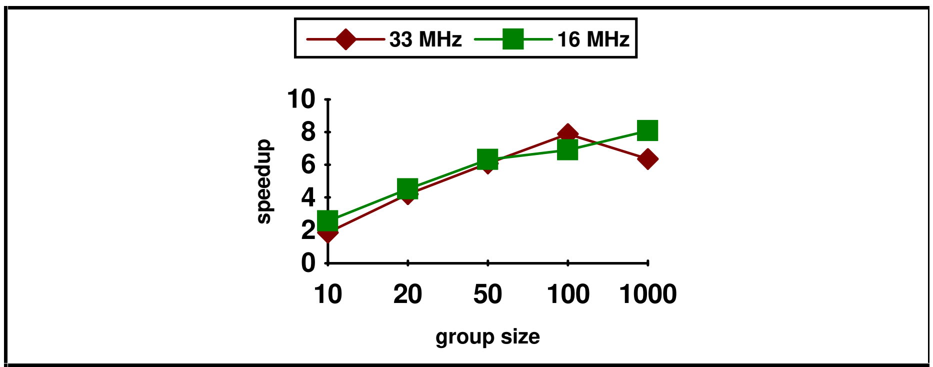

Figure 17: Average speedup obtained at different processor speeds

The difference between results obtained at 33 MHz and those obtained at 16 MHz can be explained by the fact that doubling the clock frequency does not double the speed of I/O operations in general and memory references in particular. Since the ratio of time spent on internal CPU operations to time spent on external I/O and memory references is different between multicast operations vs. multiple unicast operations, it is easy to explain why higher processor speeds may exhibit a drop in speedup for large group sizes. This becomes more significant when resources are scarce, thus it is more apparent in run 4 (and sometimes in run3 as well).

5.5 Conclusion

We have shown that clever design of multicasting mechanisms in a microkernel environment can improve the performance of group communication significantly. Our research focused on problems related to local aspects (within the host operating system). Future work based on our design may further improve the interface between the host operating system and the networking hardware, exploit hardware solutions which are optimized for better performance (such as [Banks93]), and improve other aspects of multicasting. These may include the interface between protocols and operating systems, efficient multicast message delivery at the end-points, and the design and implementation of multicasting network servers which will perform optimized inter-host delivery.

References

[Ahamad85] M. Ahamad. Multicast Communication in Dynamic Process Groups. Submitted to the 5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 1985.

[Aho83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. Data Structures and Algorithms. Addison-Wesley, 1983.

[Armstrong92] S. Armstrong, A. Freier, and K. Marzullo. Multicast Transport Protocol. RFC 1301, Xerox, Apple, Cornell University, February 1992.

[ATM93] ATM Forum. ATM User-Network Interface Specification, version 3.0, PTR Prentice Hall, September 1993.

[Ballardie93] T. Ballardie, P. Francis, and J. Crowcroft. Core Based Trees. In Proceedings of ACM SIGCOMM, San Francisco, September 1993, pp. 1-11.

[Banks93] D. Banks, and M. Prudence. A High Performance Network Architecture for a PA-RISC Workstation. IEEE Journal of Selected Areas in Communications, Vol. 10, No. 1, February 1993, pp. 191-202.

[Berglund88] E. J. Berglund. An Introduction to the V-System, IEEE Micro, August 1986, pp. 35-52.

[Bernstein85] A. Bernstein. A Loosely Coupled Distributed Algorithm for Reliably Storing Data, IEEE Transactions on Software engineering, SE-11, May 1985, pp. 446-453.

[Birman87] K. P. Birman and T. A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In Proceedings of 11th ACM Symposium on Operating Systems Principles, Austin, TX, November 1987, pp. 123- 138.

[Birman91] K. P. Birman, Cooper R. and B. Gleeson. Programming with Process Groups: Group and Multicast Semantics. TR-91-1185, Department of Computer Science, Cornell University, January 1991.

[Birman91B] K. Birman and R. Cooper. The ISIS Project: Real Experience with Fault Tolerant Programming System, Operating Systems Review, April 1991, pp. 103-107.

[Boykin93] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso. Programming under Mach. Addison-Wesley, 1993.

[Braudes93] R. Braudes, and S. Zabele. Requirements for Multicast Protocols. RFC 1458, TASC, May 1993.

[Callon87] R. Callon. A comparison of 'Link State' and 'Distance Vector' Routing Algorithms. DEC, November 1987.

[Casner92] S. Casner, and S. Deering. First IETF Internet Audiocast. ACM Computer Communication Review, vol. 22, no. 3, pp. 92-97, July 1992.

[Casner94] S. Casner. Frequently Asked Questions on the Multicast Backbone. ftp.sis.edu:mbone/faq.txt, December 1994

[CCITT90] CCITT Draft Recommendation I.113. Geneva, May 1990.

[Cheriton84] D. R. Cheriton. The V Kernel: A Software Base for Distributed Systems. IEEE Software, April 1984, pp. 19-42.

[Cheriton85] D. R Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. ACM Transactions on Computer Systems, Vol. 3, No. 2, May 1985, pp. 77-107.

[Cheriton88] D. Cheriton. VMTP: Versatile Message Transaction Protocol. RFC 1045, Stanford university, February 1988.

[Chesson92] G. Chesson. XTP Protocol Definition. Technical Report PEI 92-10, Protocol Engines Incorporated, Mountain View, California, Revision 3.6, January 1992.

[Clark78] D. Clark, K. Pogran and D. Reed. An Introduction to Local Networks. In IEEE Proceedings, Vol. 66, No. 11, November 1978.

[Cox91] J.R. Cox and J. S. Turner. Project Zeus: Design of a Broadband Network and its application on a University Campus. Technical Report WUCS91-45, Department of Computer Science, Washington University in St. Louis, July 1991.

[Dalal78] Y. K. Dalal and R. M. Metcalfe. Reverse Path Forwarding of Broadcast Packets. Communications of the ACM, Vol. 21, No. 2, December 1978, pp. 1040-1047.

[Deering88] S. Deering. Multicast Routing in Internetworks and Extended LANs. In Proceedings of SIGCOMM 1988, Stanford, California, August 1988.

[Deering89] S. Deering. Host Extensions for IP Multicasting. RFC 1112, Stanford University, August 1989.

[Deering90] S. E. Deering, and D. P. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. ACM Transactions on Computer Systems, Vol. 8, no. 2, May 1990, pp. 85-110.

[Deering91] S. Deering. Multicast Routing in a Datagram Internetwork, Stanford Technical Report, STAN-CS-92-1415, Department of Computer Science, Stanford University, December 1991.

[deHart92] J. deHart, M. Gaddis, and R. Bubenik. Connection Management Access Protocol (CMAP) Specification. Technical Report WUCS-92- 01, Applied Research Laboratory, Department of Computer Science, Washington University in St. Louis, May 1992.

[Doar93] J. M. S. Doar. Multicast in the Asynchronous Transfer Mode Environment. Ph.D. dissertation, University of Cambridge, St. John's College, January 1993.

[Draves90] R. Draves. A Revised IPC Interface. In Proceedings of Usenix Mach Workshop, The Usenix Association, Burlington, Vermont, October 1990, pp. 101-121.

[Ethernet80] The Ethernet Local Network: Three Reports. Technical Report CSL- 80-2, Xerox Palo Alto Research Center, Palo Alto, California, 1980.

[Felten93] E. W. Felten. Protocol Compilation: High-Performance Communication for Parallel Programs. Ph.D. dissertation, University of Washington, July 1993.

[Garcia82] H. Garcia-Molina. Elections in a Distributed Computing System. IEEE Transactions on Computers, Vol. 31, January 1982, pp. 48-59.

[Gray78] J. Gray. Notes on Database Operation Systems. Lecture Notes in Computer Science, 60, Springer Verlag, New York, 1978.

[Guillemont91] M. Guillemont, J. Lipkis, D. Orr and M. Rozier. A second generation micro-kernel based UNIX; Lessons in performance and compatibility. In Proceedings of Usenix Conference, The Usenix Association, Dallas, Texas, January 1991, pp. 13-22.

[Hedrick88] C. Hedrick. Routing Information Protocol. RFC 1058, Rutgers University, June 1988.

[IBM89] IBM Token-Ring Network, Architecture Reference. Publication SC30- 3374-02, Third Edition, September 1989.

[ITU-TSS93] ITU-TS draft Recommendation Q.93B, B-ISDN User-Network Interface Layer 3 Specification for Basic Call/Bearer Control. May 1993

[Jacobson94] V. Jacobson. Learning from the Present - Things that IP Got Right and ATM got Wrong. Keynote Address, Usenix Symposium on High- Speed Networking, The Usenix Association, Oakland, California, August 1994.

[Joseph89] T. A. Joseph and K. P. Birman. Reliable broadcast protocols. In S. J. Mullender, editor, Distributed Systems, Addison-Wesley, 1989, pp. 293-317.

[Kaashoek91] M. F. Kaashoek, and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In Proceedings of 11th International Conference on Distributed Computing Systems, IEEE, 1991, pp. 222-230.

[Karp72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller, and J. W. Thatcher, (Eds), Complexity of Computer Computations, Plenum Press, New-York, 1972, pp. 85-103.

[Katz93] D. Katz. Transmission of IP and ARP over FDDI Networks. RFC 1390, Cisco Systems, Inc., January 1993

[Kou81] L. Kou, G. Markowsky, and L. Berman. A Fast Algorithm for Steiner Trees. Acta Informatica, Vol. 15, 1981, pp. 141-145.

[Ladin90] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In Proceedings of 10th ACM Symposium on Principles of Distributed Computing (ACM SIGOPS- SIGACT), Quebec City, Quebec, August 1990, pp. 43-58.

[Leprau93] J. Leprau, M. Hibler, B. Ford, and J. Law. In-Kernel Servers on Mach 3.0: Implementation and Performance. In Proceedings of Mach III Symposium, The Usenix Association, April 1993, pp. 39-55.

[Liang90] L. Liang, S. T. Chanson and G. W. Neufeld. Process Groups and Group Communications: Classifications and Requirements. Computer, February 1990, pp. 56-66.

[Loepere92] K. Loepere. Mach 3 Kernel Principles and Mach 3 Kernel Interfaces. Open Software Foundation and Carnegie Mellon University, Revision 2.2, July 1992.

[Massalin92] H. Massalin. Synthesis: An Efficient Implementation of Fundamental Operating System Services. Ph.D. dissertation, Columbia University, New York, 1992.

[McNamee90] D. McNamee and K. Armstrong. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In Proceedings of Usenix Mach Workshop, The Usenix Association, Burlington, Vermont, October 1990, pp. 17-29.

[Metcalfe76] R. Metcalfe and D. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. Communications of the ACM, Vol. 19, No. 7, 1976, pp. 395-404.

[Minoli93] D. Minoli. Proposed Cell Relay Bearer Service Stage 1 Description. T1S1.1/93-136-R1 (Revision 1), ANSI Committee T1 (T1S1.1), Sommerset, N.J., June 1993.

[Minoli94] D. Minoli, and M. Vitella. ATM and cell relay service for corporate environments, McGraw-Hill, 1994.

[Moy94] J. Moy. Multicast Extensions to OSPF. RFC 1584, Proteon, Inc., March 1994.

[Moy94B] J. Moy. OSPF Version 2. RFC 1583, Proteon, Inc., March 1994.

[Mullender90] S. J. Mullender, G. Van Rossum, A. S. Tanenbaum, R. Van Renesse and H. Van Staveren. Amoeba: A Distributed Operating System for the 1990s. IEEE Computer, Vol. 23, May 1990, pp. 44-53.

[Partridge93] C. Partridge. Gigabit Networking. Addison-Wesley, 1993.

[Patterson90] D. A. Patterson, and J. L. Hennessy. Computer Architecture: A Quantitative Approach. Morgan Kaufman Publishers, San Mateo, California, 1990.

[Peterson89] L. L. Peterson, N. C. Bucholz and R. Schlichting. Preserving and using context information in interprocess communication. ACM Transactions on Computer Systems, Vol. 7, No. 3, August 1989, pp. 217-246.

[Pusateri93] T. Pusateri. IP Multicast Over Token-Ring Local Area Networks. RFC 1469, June 1993.

[Rayward86] V. J. Rayward-Smith, and A. Clare. On finding Steiner Vertices. Networks, Vol. 16, 1986, pp. 283-294.

[Reinolds91] F. Reinolds and J. Heller. Kernel support for network protocol servers. In Proceedings of Usenix Mach Symposium, The Usenix Association, Monterey California, November 1991, pp. 149-162.

[Rozier91] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuahuser. Overview of the CHORUS Distributed Operating System. CS/TR-90-25.1, Chorus systemes, February 1991.

[Smith79] R. Smith. The Contract Net Protocol: High-Level Communication and Control in Distributed Problem Solver. In Proceedings of the First International Conference on Distributed Computing Systems, October 1979, pp. 185-192.

[Stevens94] W. R. Stevens. TCP/IP Illustrated: The Protocols. Addison-Wesley, 1994, p. 177.

[Tanenbaum90] A. S. Tanenbaum, R. Van Renesse, H. Van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen and G. Van Rossum. Experience with the Amoeba Distributed Operating System. Communications of the ACM, Vol. 33, pp. 46-63, December 1990.

[Thekkath93] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing Network Protocols at User Level. IEEE/ACM Transactions on Networking, Vol. 1, No. 5, October 1993.

[Topolcic90] C. Topolcic. Experimental Internet Stream Protocol, Version 2 (ST- II). RFC 1190, CIP Working Group, October 1990.

[Turletti93] T. Turletti. H.261 software codec for videoconferencing over the Internet. TR-1834, INRIA, January 1993.

[Turner86] J. S. Turner. New Directions in Communications (or Which Way to the Information Age). IEEE Communications, Vol. 24, No. 10, October 1986, pp. 8-15.

[Waitzman88] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. RFC 1075, BBN STC, Stanford University, November 1988.

[Wall80] D. W. Wall. Mechanisms for Broadcast and Selective Broadcast. Ph.D. dissertation, Stanford University, California, June 1980. Also published as Computer Systems Laboratory Technical Report 190.

[Watson87] R. W. Watson and S. A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. ACM Transactions on Computer Systems, vol. 5, no. 2, May 1987, pp. 97-120.