GROUP COMMUNICATION

IN DISTRIBUTED

COMPUTER SYSTEMS

M.F. Kaashoek

VRIJE UNIVERSITEIT

GROUP COMMUNICATION IN DISTRIBUTED COMPUTER SYSTEMS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit te Amsterdam,
op gezag van de rector magnificus
dr. C. Datema,
hoogleraar aan de faculteit der letteren,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der wiskunde en informatica
op maandag 14 december 1992 te 10.30 uur
in het hoofdgebouw van de universiteit, De Boelelaan 1105

door

MARINUS FRANS KAASHOEK

geboren te Leiden

Centrale Huisdrukkerij Vrije Universiteit

Amsterdam 1992

Promotor: prof.dr. A.S. Tanenbaum
Copromotor: dr. H.E. Bal
Referent: prof. W. Zwaenepoel Ph.D

© 1992 M.F. Kaashoek

Parts of Chapters 1 have been published in *Computer Communications*.

Parts of Chapter 2 have been accepted for publication in *ACM Transactions on Computer Systems*.

Parts of Chapter 3 have been published in the *Proceedings of the Eleventh International Conference on Distributed Computing Systems*.

Parts of Chapters 4 have been published in *IEEE Computer*, in *IEEE Transactions on Software Engineering*, and in the *Proceedings of the 1990 International Conference on Computer Languages*.

Parts of Chapter 5 have been published in the *Proceedings of the USENIX Symposium on Experiences with Building Distributed and Multiprocessor Systems III* and in the *Proceedings of the Fifth ACM SIGOPS European Workshop*.

Stellingen

1. Een besturingssysteem voor een gedistribueerde architectuur zou naast remote procedure call ook groepscommunicatie moeten ondersteunen, waardoor grote klassen van programmatuur eenvoudiger te programmeren zijn en ook duidelijk betere prestaties leveren.

   Lit: dit proefschrift.

2. De meest economische supercomputer van de toekomst zal een parallelle computer zijn, bestaande uit een verzameling van multiprocessor-machines verbonden door een snel datanetwerk.

3. Het Shared Data-Object Model is een beter model voor gedistribueerd gemeenschappelijk geheugen dan het op pagina's gebaseerde Virtual Shared Memory.

   W. Levelt, F. Kaashoek, H. Bal en A. Tanenbaum, ''A Comparison of Two Paradigms for Distributed Shared Memory,'' Software−Practice and Experience, 1992.

4. De vraag of microkernen beter zijn dan monolitische kernen kan nu definitief met ''ja'' beantwoord worden.

   D. Golub, R. Dean, A. Forin en R. Rashid, ''Unix as an Application Program,'' Proc. Usenix 1990 Summer Conference, June 1990.

   F. Douglis, F. Kaashoek, J. Ousterhout en A. Tanenbaum, ''A Comparison of Two Distributed Systems: Amoeba and Sprite,'' Computing Systems, Vol. 4, No. 4, Fall 1991.

5. Het succes van de ouderwetse en primitieve programmeertaal FORTRAN is opmerkelijk: na tientallen jaren één van de meest gebruikte programmeertalen te zijn geweest voor sequentiële programmatuur, lijkt het nu ook de meest gebruikte parallelle programmeertaal te zijn.

6. De prestaties van veel computerprogramma's kunnen met een paar zeer eenvoudige wijzigingen enorm verbeterd worden.

   A. Tanenbaum, F. Kaashoek, K. Langendoen en C. Jacobs, ''The Design of Very Fast Portable Compilers,'' SIGPLAN Notices, Vol. 24, No. 11, Nov. 1989.

7. Het succes van expertsystemen kan afgelezen worden uit het succes van Sky Radio en Radio 10, die in plaats van de diskjockey een rule-based computerprogramma de muziekkeuze laten bepalen.

8. De definitie van langzaam verkeer is in grote steden als Amsterdam onjuist: de fiets is in dergelijke steden een sneller vervoermiddel dan de auto. Door aanpassing van de definitie zou de fietser dan ook in meer verkeerssituaties voorrang kunnen krijgen op de automobilist.

9. Bij het invoeren van het AiO/OiO-stelsel zijn de Nederlandse Overheid en universiteiten niet ver genoeg gegaan met het kopiëren van het Amerikaanse systeem.

   a) Een AiO/OiO moet als student gezien worden. Deze keuze ontslaat de AiO/OiO van de verplichting om kostbare tijd te verspillen aan onderwijs geven en heeft als neveneffect een bezuiniging op de overheidsuitgaven.

   b) De verdediging van het proefschrift moet plaats vinden vóór het drukken van het proefschrift. Deze keuze maakt de verdediging zinvol en verhoogt de kwaliteit van het proefschrift.

   c) De AiO/OiO moet niet verplicht zijn het proefschrift te laten drukken. De universiteit zou de dissertatie alleen dán in grotere aantallen moeten uitgeven, als er voldoende belangstelling is. Deze keuze is financieel aantrekkelijk voor de AiO/OiO en heeft een positief effekt op het milieu.

10. Wetenschap is in menig opzicht vergelijkbaar met topsport. Zo hanteren beide disciplines eenvoudige maatstaven voor succes, tellen zowel de individuele prestaties als de resultaten van een team, is een goede begeleiding van jong talent essentieel en is een jarenlange training vaak noodzakelijk. De beste prestaties worden in het algemeen jong geleverd. Tot slot spelen in beide takken de bobo's een belangrijke rol.

This page intentionally left blank

1

INTRODUCTION

Computer systems consisting of multiple processors are becoming commonplace. Many companies and institutions, for example, own a collection of workstations connected by a local area network (LAN). Although the hardware for distributed computer systems is advanced, the software has many problems. We believe that one of the main problems is the communication paradigms that are used. This thesis is concerned with software for distributed computer systems. In it, we will study an abstraction, called *group communication* that simplifies building reliable efficient distributed systems. We will discuss a design for group communication, show that it can be implemented efficiently, and describe the design and implementation of applications based on group communication. Finally, we will give extensive performance measurements. Our goal is to demonstrate that group communication is a suitable abstraction for distributed systems.

Distributed computer systems, or distributed systems for short, consist of hardware and software components. The term ‘‘distributed system’’ is used for many hardware configurations. In this thesis a distributed computing system is defined as follows:

> A distributed computing system consists of multiple autonomous machines that do not share primary memory, but cooperate by sending messages over a communication network.

This definition includes architectures like the hypercube [Athas and Seitz 1988], a set of workstations connected by a LAN, or a set of computers geographically distributed over the world connected by a wide-area network (WAN). It does not include multiprocessors, because the processors in a multiprocessor can communicate through shared memory. However, the nodes in a distributed system may themselves be multiprocessors. In effect, the key property of a distributed system is that there is no physically shared memory present.

Software for distributed systems can be divided between application software (user programs, text editor, etc.) and system software (operating system, compilers, etc.). In this thesis we will focus mainly on system software. The task of an operating system for a distributed system is to hide the distribution of the hardware from the user. No matter where a user logs in, he† should be able to access his files. When a user starts a process, he should not be aware where the process is running. To summarize:

> The goal of a distributed operating system is to make a collection of computers look like a single computer to the user. It is the operating system's responsibility to allocate resources to users in a transparent way [Tanenbaum 1992].

To understand what distributed operating systems are, it is useful to compare them with the more common *network operating systems*. In a network system each user has a computer for his exclusive use. The computer has its own operating system and may have its own disks. All commands are normally run locally on the user's computer. If a user wants to copy a remote file, he explicitly has to tell where the file is located. If a user wants to start a program on another computer, he has to start the command explicitly on the remote computer. More advanced network systems may provide some degree of transparency by, for example, having a shared file system, but the key point is that the user is always aware of the distribution of the hardware. Unlike true distributed systems, network systems are very common.

Both network and distributed systems are an active area of research. One of the main problems is that existing network and distributed operating systems do not provide adequate support for writing distributed applications. In particular, many distributed applications can profit from an operating system primitive that allows for sending a message from 1 process to *n* processes. Few operating systems, however, provide primitives that make the hardware facilities for 1-to-*n* communication available to applications. The main thesis of the work presented here is that a distributed operating system should provide support for group communication (1-to-*n* communication), because:

- It simplifies distributed programming.
- It potentially gives substantial performance benefits.

To validate this thesis we have designed new protocols for doing group communication, and implemented them as part of the Amoeba distributed operating system. Then, using this group communication, we have designed and implemented both parallel and fault-tolerant applications, showing that group communication can be used effectively in distributed systems.

The outline of the rest of this chapter is as follows. In Section 1.1 we discuss distributed applications, as the goal of systems research is to provide the appropriate

---

† ''He'' should be read throughout this thesis as ''he or she.''

hardware and software base for these applications. In Section 1.2, we motivate why group communication is needed. In Section 1.3, we state the problems that are solved in this thesis and preview the solutions. Like most research, the work presented here builds on previous work, so we discuss related research in Section 1.4. This thesis is *not* a theoretical thesis. Everything described has been implemented as part of the Amoeba distributed system, which is in day-to-day use. To understand the design decisions and performance measurements, it is required that the reader has some knowledge of Amoeba, so we review Amoeba in Section 1.5. In Section 1.6, we outline the rest of the thesis.

1.1. Applications for Distributed Computer Systems

Applications that make use of a distributed system can be categorized in four different types: functional specialization, parallel applications, fault-tolerant applications, and geographically distributed applications (see Fig. 1.1) [Bal et al. 1989b]. We will discuss each class of applications in turn below.

Type of application	**Example**
Functional specialization	File service
Parallel	Parallel chess program
Fault-tolerant	Banking system
Geographically distributed	Electronic mail

Fig. 1.1. Categorization of applications that can use a distributed system.

Some distributed applications can be written as a set of specialized services. For example, a distributed operating system may be written as a collection of services, such as a file service, a directory service, a printer service, and a time-of-day service. In a distributed system it makes sense to dedicate one or more processors to each service to achieve high performance and reliability.

Because distributed systems consist of multiple processors, they can also be used to run parallel applications. The goal in a parallel application is to reduce the turnaround time of a *single* program by splitting the problem into multiple tasks, all running in parallel on separate processors. A chess program, for example, can be written to allow each processor to evaluate different parts of the game tree simultaneously.

Distributed systems are potentially more reliable than centralized systems, because each processor is independent; if one fails, it need not affect the others. Reliability can therefore be increased by replicating an application on multiple processors. If one of the processors fails, another one can finish the job. A banking system, for example, can be made fault-tolerant by replicating all bank accounts on multiple disks

and processors. If a disk becomes unreadable or if one of the processors fails, the banking system can use one of the other disks or processors to complete a transaction.

Finally, there are applications which are inherently distributed in nature. A good example of such an application is the international electronic mail system. Mail daemons spread all over the world cooperate to deliver messages from one user to another.

All these classes of applications, except for geographically distributed applications will show up in this thesis. This chapter, for example, describes how the Amoeba operating system is built out of specialized services. Chapter 4 discusses a number of parallel applications and Chapter 5 discusses two classes of fault-tolerant applications.

1.2. Why Group Communication?

Most current distributed operating systems are based on *Remote Procedure Call* (RPC) [Birrell and Nelson 1984]. The idea is to hide the message passing, and make the communication look like an ordinary procedure call (see Fig. 1.2). The sender, called the *client*, calls a *stub routine* on its own machine that builds a message containing the name of the procedure to be called and all the parameters. It then passes this message to the driver for transmission over the network. When it arrives, the remote driver gives it to a stub, which unpacks the message and makes an ordinary procedure call to the *server*. The reply from server to client follows the reverse path.

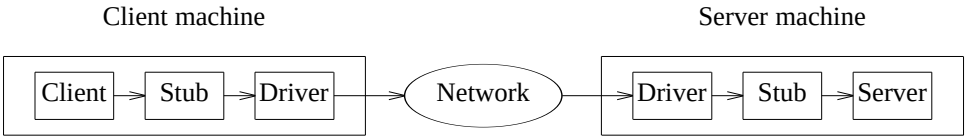


Fig. 1.2. Remote procedure call from a client to a server. The reply follows the same path in the reverse direction.

RPC is very useful, but inherently point-to-point. Many applications also need 1-to-*n* communication [Chang 1984; Gehani 1984; Dechter and Kleinrock 1986; Ahamad and Bernstein 1985; Cheriton and Zwaenepoel 1985; Joseph and Birman 1986; Liang et al. 1990; Cristian 1991]. Consider, for example, a parallel application. Typically in a parallel application a number of processes cooperate to compute a single result. If one of the processes finds a partial result (e.g., a better bound in a parallel branch-and-bound program) it is desirable that this partial result be communicated immediately to the other processes. By receiving the partial result as soon as possible, the other processes do not waste cycles on computing something that is not interesting anymore, given the new partial result.

Now consider a second application: a fault-tolerant storage service. A reliable storage service can be built by replicating data on multiple processors each with its own disk. If a piece of data needs to be changed, the service either has to send the new data to all processes or invalidate all other copies of the changed data. If only point-to-point communication were available, then the process would have to send $n - 1$ reli-

able point-to-point messages. In most systems this will cost at least $2(n-1)$ packets (one packet for the actual message and one packet for the acknowledgement). If the message sent by the server has to be fragmented into multiple network packets, then the cost will be even higher. This method is slow, inefficient, and wasteful of network bandwidth.

In addition to being expensive, building distributed applications using only point-to-point communication is often difficult. If, for example, two servers in the reliable storage service receive a request to update the same data, they need a way to consistently order the updates, otherwise the data will become inconsistent. The problem is illustrated in Figure 1.3. The copies of variable *X* become inconsistent because the messages from Server 1 and Server 2 are not ordered. What is needed is that all servers receive all messages in the same order.

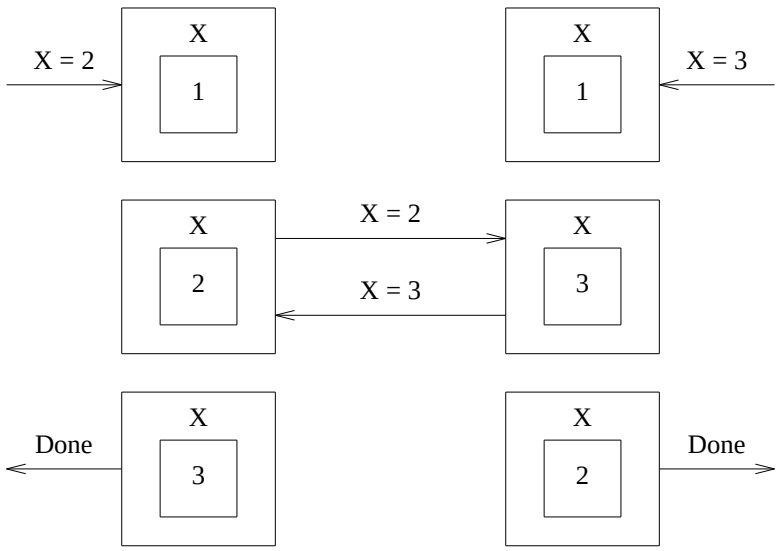


Fig. 1.3. Server 1 and 2 receive at approximately the same time an update message for the shared variable *X*. The copies of *X* become inconsistent because the messages from Server 1 and Server 2 are not ordered.

Many network designers have realized that 1-to-*n* communication is an important tool for building distributed applications; *broadcast* communication is provided by many networks, including LANs, geosynchronous satellites, and cellular radio systems [Tanenbaum 1989]. Several commonly used LANs, such as Ethernet and some rings, even provide *multicast* communication. Using multicast communication, messages can be sent exactly to the group of machines that are interested in receiving the message. Future networks, like Gigabit LANs, are also likely to implement broadcasting and/or multicasting to support high-performance multi-media applications [Kung 1992].

The protocol presented in this thesis for group communication uses the hardware multicast capability of a network, if one exists. Otherwise, it uses broadcast messages or point-to-point messages, depending on the size of the group and the availability of

broadcast communication. Thus, this system makes any existing hardware support for group communication available to application programs.

A word on terminology. The terms ‘‘broadcasting,’’ ‘‘multicasting,’’ and ‘‘group communication’’ are often confused. We will refer to the abstraction of a group of processes communicating by sending messages from 1 to *n* destinations as ‘‘group communication.’’ We consider ‘‘broadcasting’’ and ‘‘multicasting’’ as two hardware mechanisms that can be used to implement the abstraction of group communication. A broadcast message on a network is received by all processors on that network. A multicast message on a network is only received by the processors on that network that are interested in receiving it. We will often use the term ‘‘broadcasting’’ to refer to any of the three terms, as this is the term generally used in the literature.

1.3. Problems and Solutions

The goal of this thesis is to present a system that simplifies distributed programming. Fundamental to our proposal is the usage of group communication. The system can best be described using a three layer model (see Fig. 1.4). The bottom two layers deal with multicast and group communication. The top layer uses group communication to implement applications, such as a run-time system for parallel programming or a fault-tolerant file service. To validate our approach, this thesis discusses the design, implementation, usage, and performance of each layer. We will now overview the system, starting at the bottom layer.

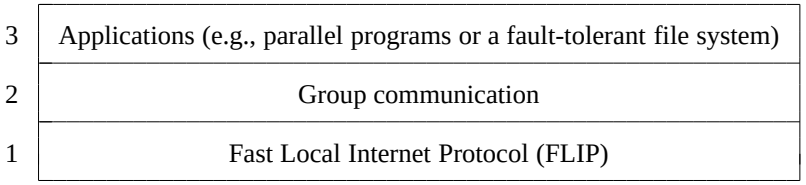


Fig. 1.4. Structure of the proposed system.

Layer 1, the Fast Local Internet Protocol (FLIP), implements a new network protocol specifically designed to meet the requirements of distributed systems. Current network protocols, like IP [Postel 1981a; Comer 1992], have been designed for network systems. To meet distributed system requirements using IP, new protocols have been invented for each subset of requirements. For example, the Internet Control Message Protocol (ICMP) has been introduced to implement dynamic routing and to cope partially with network changes [Postel 1981b]. The Address Resolution Protocol (ARP) has been introduced to map IP addresses on hardware addresses [Plummer 1982]. The Reverse Address Resolution Protocol (RARP) has been introduced to acquire an IP address [Finlayson et al. 1984]. Internet Group Management Protocol (IGMP) has been introduced to implement group communication [Deering 1988]. The Versatile Message Transport Protocol (VMTP) has been introduced to meet the

requirements for group communication and a secure, efficient, and at-most-once RPC protocol [Cheriton 1986].

One of the main contributions of this thesis is a protocol (FLIP) that tries to address the distributed system requirements that we have identified in a clean, simple, and integrated way. FLIP is designed to support efficient RPC communication, group communication, location transparency, security, and process migration. One key idea is that FLIP addresses do not identify locations (machines), as in IP, but logical entities such as processes or process groups.

Layer 2 implements a new abstraction for efficient reliable group communication. This layer offers a primitive that guarantees delivery of a message to a group of processes, even in the face of communication failures. Furthermore, it guarantees that every process receives all messages, and in the same order. If required, the protocol can also recover from arbitrary processor failures at some cost in performance.

The group communication primitives are based on a protocol that guarantees total ordering and which is much more efficient than sending *n* unordered reliable point-to-point messages. If two application processes, on two different machines, simultaneously broadcast two messages, *A* and *B* respectively, then our primitive guarantees that either (1) every process in the group receives first *A* and then *B* or (2) every process in the group receives first *B* and then *A*. Under no conditions do some processes get *A* first and others get *B* first. By making the broadcast both reliable and indivisible in this way, the user semantics become simple and easy to understand.

Applications that use group communication run in layer 3. One possible application is a run-time system for a form of distributed shared memory (DSM), called the Shared Data-Object Model [Bal 1990; Bal et al. 1992a]. A data-object is an instance of an abstract data type that can be shared among multiple processes, typically running on different processors, without requiring physically shared memory. Shared objects are replicated in the private memory of the processors and are kept consistent using the group primitives of layer 2. By using group communication and objects to store shared data, we are able to achieve a more efficient implementation of distributed shared memory than a model based on virtual memory pages [Levelt et al. 1992]. We have used the shared object model successfully to run a number of parallel applications, such as the Traveling Salesman Problem [Lawler and Wood 1966], parallel Alpha-Beta search [Knuth and Moore 1975], the All Pairs Shortest Path Problem [Aho et al. 1974], and Successive Overrelaxation [Stoer and Bulirsch 1983].

Although it is efficient and simple to replicate data using group communication, some data structures are better not replicated at all. For example, when a shared integer is mostly written by one processor *p* and almost never accessed by another processor, it is better to store the integer at processor *p*. The problem is to detect such cases automatically without bothering the programmer. We have investigated an approach based on run-time support combined with compile-time information.

In addition to parallel programs, many other applications can profit from group communication. We discuss two kinds of fault-tolerant applications: fault-tolerant

parallel applications and fault-tolerant distributed services. When a number of processors work in parallel on a problem, the chance that one of them will fail increases with the number of processors. In most systems a failure of one processor will terminate the complete application. For applications that have to meet a deadline, such as weather prediction, this is unacceptable. We have developed a new scheme to make parallel applications that do not perform I/O, fault-tolerant in a transparent and efficient way.

The second class of fault-tolerant applications that we discuss are distributed services. We demonstrate that group communication is important for this class of applications by showing that a fault-tolerant directory service is easier to implement with groups than with RPC and is also more efficient. The directory service exemplifies distributed services that provide high reliability and availability by replicating data.

In summary, this thesis makes the following research contributions:

- A new network protocol, FLIP, that meets the communication requirements of a distributed system.
- An efficient protocol for reliable and totally-ordered group communication.
- A new model and implementation for DSM based on compiler optimizations, group communication, and shared data-objects.
- An efficient scheme to make a broad class of parallel applications fault-tolerant in a transparent way.
- A fault-tolerant directory service based on group communication, demonstrating that fault-tolerant services based on group communication are easier to implement and more efficient than the same service implemented using RPC.

Although on first sight some of the contributions may seem unrelated, a single theme runs through all of them: group communication is a fundamental abstraction in a distributed operating system. All these contributions relate to group communication and its implementation.

1.4. Related Work

The research discussed in this thesis builds on previous work. There are four other kinds of work with which we can compare ours:

- Network protocols for distributed systems.
- Broadcasting as an operating system service.
- Distributed shared memory.
- Systems providing fault tolerance.

We will briefly compare our work to each of these four in turn. Subsequent chapters will contain more detailed comparisons.

Most of the research in communication protocols has resulted in new protocols on top of IP. The key idea in FLIP is *not* to build on existing protocols to meet new requirements, but to design a single protocol that integrates all the desired functionality. The result is a protocol that is better for distributed systems than existing ones. The main disadvantage of our approach is that it is incompatible with existing protocols. We believe, however, that a Ph.D. thesis should *not* be restricted to research that is backward compatible with work that was done 10 years ago.

The second related area is systems that provide broadcasting as a user primitive. A number of systems (e.g., Isis, Psync, and V) provide group communication in some form. Isis is a toolkit especially designed to implement fault-tolerant distributed applications. The toolkit provides a number of group communication primitives that allow the user to program as if the system were in virtual synchrony [Birman and Joseph 87]. Our system also provides support for fault-tolerant applications, but in addition it provides a very cheap broadcast primitive for applications that are not concerned with fault tolerance.

Psync is a communication mechanism based on *context graph*s [Peterson et al. 1989]. Psync provides the programmer with a number of communication primitives on which more powerful primitives can be built. One of Psync's library functions contains a primitive that provides reliable and totally-ordered group communication, built out of the basic primitives. The performance of this primitive is worse than ours, because it is based on a less efficient protocol.

Another system that provides group communication as an operating system primitive is the V system [Cheriton and Zwaenepoel 1985; Cheriton 1988b]. The main difference between the V system and our system is that V provides unreliable and unordered group communication, while ours provides reliable and totally-ordered group communication. Our interface to the programmer is therefore easier to understand and to use. The advantage of the V approach is that different group communication interfaces (e.g., reliable and totally-ordered) can be layered over their basic interface, each interface exactly matching the requirements of the program and giving optimal performance. We prefer, however, to have one simple and powerful interface to avoid an explosional number of interfaces. In general, having many slightly different interfaces results in programs that are hard to understand and therefore are error-prone. We are willing to pay performance to win uniformity in software.

The third area that relates to our research is distributed shared memory. In general, the approach taken in DSM research is to use the hardware memory management unit (MMU) to implement the illusion of one virtual memory that is shared among multiple processors, even though there is no physically shared memory present. Typically, these systems divide the shared virtual memory in fixed-size pages and only use point-to-point communication [Li and Hudak 1989; Nitzberg and Lo 1991]. The main goal in implementing DSM systems is to reduce the number of messages sent, because sending a message is orders of magnitude more expensive than accessing a value in private memory. Our approach to DSM does not require any special hardware support, and is

completely implemented outside the operating system kernel. This has the advantage that the shared memory can be divided in pieces that correspond to the requirements of the application, reducing the interprocess communication for shared memory. Furthermore, our approach allows for extensive compile-time optimization, reducing the costs for interprocess communication even more.

The fourth and last related area is systems that support fault-tolerant applications. Most efforts in research on fault tolerance are focused on making an arbitrary application fault-tolerant. This can either be done transparently to the programmer as in [Strom and Yemini 1985; Johnson 1989], or require involvement of the programmer as in [Liskov 1988; Spector et al. 1988]. The advantage of having the programmer involved is that the programmer can specify which parts of the program have to be fault-tolerant, which can result in better performance. Our approach is simple, transparent to the programmer, and efficient. We achieve these properties by restricting our application domain to computationally intensive parallel programs that do not perform I/O with the outside world. Although we are considering only a restricted domain, it includes a large set of important applications.

Very little research has been published that compares different methods for building fault-tolerant applications. Elnozahy, Johnson, and Zwaenepoel give performance figures for different methods of making a consistent checkpoint [Elnozahy et al. 1992]. In this thesis, we compare two implementations of a fault-tolerant application: one based on RPC and one based on group communication.

1.5. Experimental Environment: Amoeba

All the research contributions of this thesis have been implemented as part of the Amoeba distributed operating system [Mullender 1985; Van Renesse 1989; Mullender et al. 1990; Tanenbaum et al. 1990]. Amoeba is a true distributed system; to the user the complete system looks like a single computer. Before describing how Amoeba achieves this goal, it is useful to describe for which hardware configuration Amoeba was designed, because it differs from what most companies and institutions currently have.

The driving force behind the system architecture is the need to incorporate large numbers of CPUs in a straightforward way. In other words, what do you do when you can afford 10 or 100 CPUs per user? One solution is to give each user a personal 10-node or 100-node multiprocessor. However, we do not believe this is an effective way to spend the available budget. Most of the time, nearly all the processors will be idle, which by itself is not so bad. However, some users will want to run massively parallel programs, and will not be able to harness all the idle CPU cycles, because they are in other users' personal machines.

Instead of this personal multiprocessor approach, we believe that the processor pool model, shown in Figure 1.5, is a better approach. In this model, all the computing power is located in one or more *processor pools*. Each processor pool consists of a substantial number of CPUs, each with its own local memory and its own network con-

nection. At present, we have a prototype system operational, consisting of three standard 19-inch equipment racks, each holding 16 single board computers (MC68020 and MC68030, with 3-4 Mbyte RAM per CPU), a number of Intel 386s, and a number of Sun 3/60s. Each CPU has its own Ethernet connection. Our model does not assume that any of the CPUs share physically memory, in order to make it possible to scale the system. While it would be easy to build a 16-node shared memory multiprocessor, it would not be easy to build a 1000-node shared memory multiprocessor. However, if shared memory is present, it can be utilized to optimize message passing by just doing memory-to-memory copying instead of sending messages over the LAN.

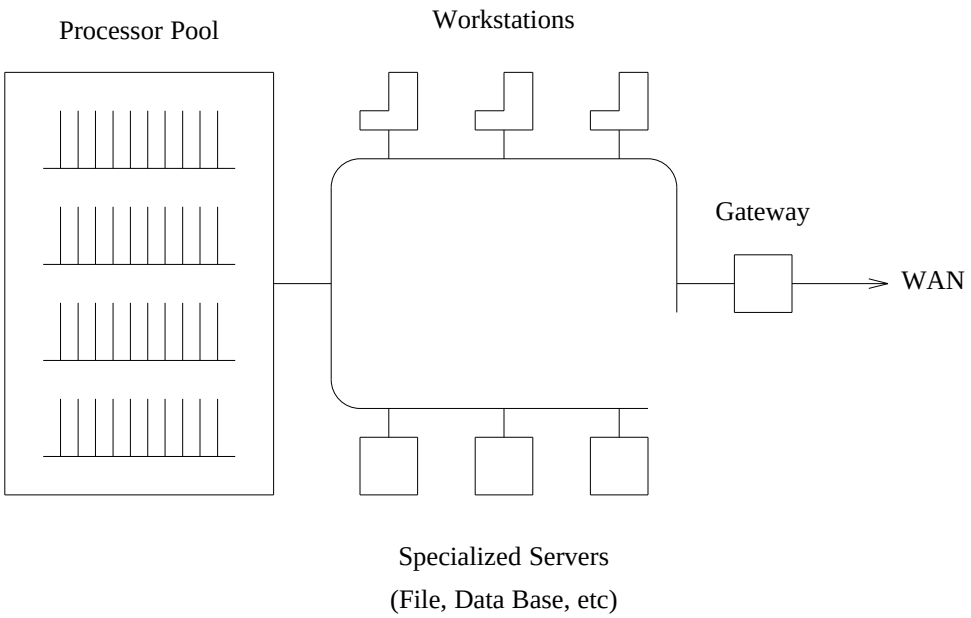


Fig. 1.5. The Amoeba System Architecture.

Pool processors are not ‘‘owned’’ by any one user. When a user types a command, the Amoeba system automatically and dynamically allocates one or more processors for that command. When the command completes, the processors are released and go back into the pool, waiting for the next command, which may be from a different user. If there is a shortage of pool processors, individual processors are timeshared, with new jobs being assigned to the most lightly loaded CPUs. The important point to note here is that this model is quite different from current systems in which each user has exactly one personal workstation for all his computing activities. The pool processor model is flexible, and provides for good sharing of resources.

The second element in the Amoeba architecture is the workstation. It is through the workstation that the user accesses the system. Although Amoeba does not forbid running user programs on the workstation, normally the only program that runs there is the window manager. For this reason, X-terminals can also be used as workstations.

Another important component of the Amoeba configuration consists of specialized servers, such as file servers, which for hardware or software reasons need to run

on a separate processor. Finally, we have the gateway, which interfaces to wide-area networks and isolates Amoeba from the protocols and idiosyncrasies of the wide-area networks in a transparent way.

Amoeba Microkernel

All the machines in Amoeba run the same microkernel. A *microkernel* is an operating system kernel that contains only the basic operating system functionality. In Amoeba it has four primary functions:

- Manage processes and threads.
- Provide low-level memory management support.
- Support communication.
- Handle low-level I/O.

Let us consider each of these functions in turn.

Like most operating systems, Amoeba supports the concept of a process. In addition, Amoeba also supports multiple threads of control within a single address space. A process with one thread is essentially the same as a process in the UNIX† system. Such a process has a single address space, a set of registers, a program counter, and a stack.

In contrast, although a process with multiple threads still has a single address space shared by all threads, each thread logically has its own registers, its own program counter, and its own stack. In effect, a collection of threads in a process is similar to a collection of independent processes in UNIX, with the one exception that they all share a single common address space.

A typical use for multiple threads might be in a file server, in which every incoming request is assigned to a separate thread. That thread might begin processing the request, then block waiting for the disk, then continue work. By splitting up the server in multiple threads, each thread can be purely sequential, even if it has to block waiting for I/O. Nevertheless, all the threads can, for example, have access to a single shared software data cache. Threads can synchronize using semaphores or mutexes to prevent two threads from accessing the shared cache simultaneously.

The second task of the kernel is to provide low-level memory management. Threads can allocate and deallocate blocks of memory, called *segments*. These segments can be read and written, and can be mapped into and out of the address space of the process to which the calling thread belongs. A process must have at least one segment, but it may have many more of them. Segments can be used for text, data, stack, or any other purpose the process desires. The operating system does not enforce any particular pattern on segment usage. Normally, users do not think in terms of segments, but this facility can be used by libraries or language run-time systems.

---

† UNIX is a Registered Trademark of AT&T.

The third job of the kernel is to handle interprocess communication. Two forms of communication are provided: RPC communication and group communication. Group communication is the main topic of this thesis.

Both the point-to-point message system and the group communication make use of a specialized protocol called FLIP. This protocol is a network layer protocol, and has been specifically designed to meet the needs of distributed computing. It deals with both point-to-point communication and multicast communication on complex internetworks. FLIP is another topic of this thesis.

The fourth function of the kernel is to manage low-level I/O. For each I/O device attached to a machine, there is a device driver in the kernel. The driver manages all I/O for the device. Drivers are linked with the kernel, and cannot be loaded dynamically.

Amoeba Objects

Amoeba is organized as a collection of objects (essentially abstract data types), each with some number of operations that processes can perform on it. Objects are generally large, like files, rather than small, like integers, due to the overhead in accessing an object. Each object is managed by an object server process. Operations on an object are performed by sending a message to the object's server.

When an object is created, the server returns a *capability* to the process creating it. The capability is used to address and protect the object. A typical capability is shown in Figure 1.6. The *Port* field identifies the server. The *Object* field tells which object is being referred to, since a server may manage thousands of objects. The *Rights* field specifies which operations are allowed (e.g., a capability for a file may be read-only). Since capabilities are managed in user space the *Check* field is needed to protect them cryptographically, to prevent users from tampering with them.

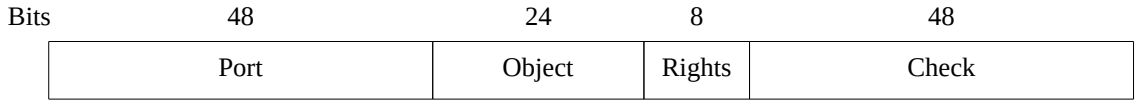


Fig. 1.6. A typical capability.

The basic algorithm used to protect objects is as follows [Tanenbaum et al. 1986]. When an object is created, the server picks a random *Check* field and stores it both in the new capability and inside its own tables. All the rights bits in a new capability are initially on, and it is this *owner capability* that is returned to the client. When the capability is sent back to the server in a request to perform an operation, the *Check* field is verified.

To create a restricted capability, a client can pass a capability back to the server, along with a bit mask for the new rights. The server takes the original *Check* field from its tables, EXCLUSIVE ORs it with the new rights (which must be a subset of the rights in the capability), and then runs the result through a one-way function. Such a

function, $y = f(x)$, has the property that given x it is easy to find y, but given only y, finding x requires an exhaustive search of all possible x values [Evans et al. 1974].

The server then creates a new capability, with the same value in the *Object* field, but the new rights bits in the *Rights* field and the output of the one-way function in the *Check* field. The new capability is then returned to the caller. In this way, processes can give other processes restricted access to their objects.

Summarizing, Amoeba's hardware configuration is based on the processor pool model. The software of Amoeba is organized as a microkernel and a collection of objects. Each object can be transparently accessed through capabilities. Because access to objects is location-independent, the user has the illusion that he is working on a single computer, although the system is distributed over many processors.

1.6. Outline of the Thesis

This thesis is structured bottom-up (see Fig. 1.7). In the next chapter we will discuss the design and implementation of FLIP. We will first describe the requirements that a protocol to support a distributed system must address. Then, we will introduce FLIP and describe how it is used in Amoeba and what its performance is. Also, we will evaluate how well it succeeds in meeting the requirements and compare it to other work.

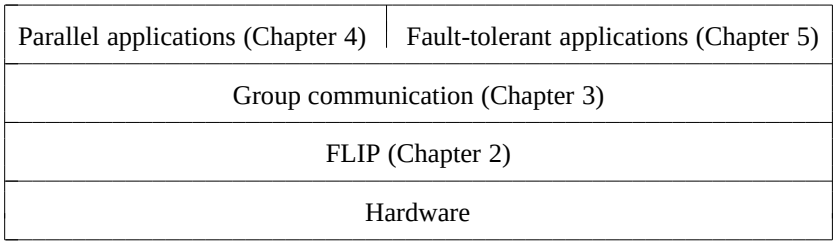


Fig. 1.7. Structure of this thesis.

In Chapter 3 we will introduce a new abstraction for group communication. Then we will discuss the design issues for group communication and which choices have been made for Amoeba. The main part of this chapter is devoted to the protocols that allow us to provide efficient, reliable, and total-ordered group communication. To validate the efficiency of the protocols, we discuss the results of extensive performance measurements. The last part of Chapter 3 discusses and compares our work with other protocols.

Chapter 4 is the first of two chapters that are concerned with applications. In this chapter we will look at parallel applications. We first discuss existing architectures for parallel computing, and then introduce a hybrid form that is based on shared data-objects, group communication, and compile-time analysis. To validate this new software architecture, we discuss the implementation and performance of three parallel

applications, each with different communication needs. Like the previous chapters, this chapter also ends with a discussion and comparison of related work.

Chapter 5 is the other chapter concerned with applications; its topic is fault-tolerant applications. We will first look at how to make parallel applications fault-tolerant. This is an interesting problem, because a compromise between two conflicting goals is needed. The main goal of a parallel application is to achieve speedup by using cycles of multiple processors and not to waste cycles on implementing fault tolerance. On the other hand, parallel applications tend to run for a long time and on many processors, so there is a real chance that due to software or hardware problems one of the processors fails, terminating the whole application unsuccessfully. In the first part of Chapter 5, we will discuss a compromise for these conflicting goals, an implementation, and the performance costs.

The second part of Chapter 5 is devoted to more traditional fault-tolerant applications. The main idea is to show that for these applications group communication is a better approach than RPC. We will do so by comparing a fault-tolerant directory service based on group communication with one based on RPC. The service chosen exemplifies the class of services that are made fault-tolerant by replicating data on more than one processor and updating the copies consistently when data is changed.

In the last chapter we will summarize the main results of this thesis.

Notes

The introduction and Section 1.1 are partly based on [Van Renesse 1989; Bal et al. 1989b; Tanenbaum 1992]. The section describing Amoeba contains material from the paper by Tanenbaum, Kaashoek, van Renesse, and Bal published in *Computer Communications* [Tanenbaum et al. 1991].

2

FAST LOCAL INTERNET PROTOCOL

Most network protocols are designed to support a reliable bit stream between a single sender and a single receiver. For applications such as remote login sessions or bulk file transfer these protocols are adequate. However, distributed operating systems have special requirements such as achieving transparency, specific RPC semantics even in the face of processor crashes, group communication, security, network management, and wide-area networking. Furthermore, applications on distributed operating systems often use a complex local *internetwork* (a network of networks) of communication subsystems including Ethernets, high-speed multiprocessor buses, hypercubes, and optical fibers. These kinds of communication are not well supported by protocols such as TCP/IP, X.25, and OSI TP4.

As part of our ongoing research on the Amoeba distributed operating system, we have designed, implemented, and evaluated a new internet protocol that, in many respects, is better suited for distributed computing than existing protocols. This new protocol, called FLIP (Fast Local Internet Protocol), is the subject of this chapter.

Although the ISO OSI protocols are not widely used, the OSI model is convenient for describing where functionality can be put in a protocol hierarchy [Zimmerman 1980]. In Figure 2.1, we show the OSI model, along with TCP/IP and FLIP protocol hierarchies. Very briefly, FLIP is a connectionless (datagram) protocol, roughly analogous to IP, but with increased functionality and specifically designed to support a high-performance RPC protocol rather than a byte-stream protocol like TCP or OSI TP4.

The outline of the rest of this chapter is as follows. In Section 2.1 we will describe the requirements that a distributed operating system places on the underlying protocol. In Section 2.2 we will discuss the FLIP service definition; that is, what FLIP provides. In Section 2.3 we will discuss the interface between FLIP and higher layers. In Section 2.4 we will discuss the protocol itself. In Section 2.5 we will discuss how FLIP is implemented. In Section 2.6 we present measurements of its performance. In

Level	OSI	TCP/IP	FLIP
7	Application	User-defined	User-defined
6	Presentation	User-defined	Amoeba Interface Language (AIL)
5	Session	Not used	RPC and Group communication
4	Transport	TCP or UDP	Not needed
3	Network	IP	FLIP
2	Data Link	E.g., Ethernet	E.g., Ethernet
1	Physical	E.g., Coaxial cable	E.g., Coaxial cable

Fig. 2.1. Layers of functionality in OSI, TCP/IP, and FLIP.

Section 2.7 we will compare it to related work. Finally, in Section 2.8 we will draw our conclusions. Appendix A describes the protocol itself in detail.

2.1. Distributed System Requirements

Distributed systems place different requirements on the operating system than traditional network systems do. Network systems run all of a user's applications on a single workstation. Workstations run a copy of the complete operating system; the only thing that is shared is the file system. Applications are sequential; they make no use of any available parallelism. In such an environment, file transfer and remote login are the two basic applications that the communication mechanisms in the operating system must support. In a distributed system the situation is radically different. A user process may run anywhere in the system, to allow efficient sharing of computing cycles. Applications are rewritten to take advantage of the available parallelism. For example, distributed systems can provide a version of the UNIX *make* program that allows compilations to run in parallel. Other applications may be rewritten to provide fault tolerance by using the redundancy of hardware. In such an environment file transfer is only one of the many applications that depend on the communication mechanisms provided by the operating system.

In this section, we will investigate the requirements for communication in a distributed system and outline the approach taken by FLIP. We identify six requirements: transparency, remote procedure call, group communication, security, network management, and wide-area networking. We discuss each of these requirements in turn. It should be noted that many existing network and distributed systems meet all or a subset of the requirements, but in this chapter we argue that the implementation of these systems can often be simplified by using a better network protocol.

Transparency

An important goal for distributed systems, such as Amoeba [Tanenbaum et al. 1990; Mullender et al. 1990], Chorus [Rozier et al. 1988], Clouds [Dasgupta et al. 1991], Sprite [Ousterhout et al. 1988], and V [Cheriton 1988b], is transparency. Distributed systems are built from a large number of processors connected by LANs, buses, and other communication media. No matter where a process runs, it should be able to communicate with any other process in the system using a single mechanism that is independent of where the processes are located. The communication system must be able to route messages along the ''best'' route from one process to another. For example, if two processes can reach each other through a LAN and high-speed bus, the communication system should use the bus. The users, however, should not have to specify which route is taken.

Most communication protocols do not provide the transparency that is required by applications running on a distributed system. Addresses in these protocols identify a host machine instead of a process. Once a process is started on a machine, it is tied to that machine. For example, if the process is migrated to another processor, the process has to inform its communication partners that it has moved. To overcome such problems, distributed systems require that an address identifies a *process*, not a *host*.

Remote Procedure Call (RPC)

Distributed operating systems are typically structured around the client-server paradigm. In this model, a user process, called the *client*, requests another user process, called the *server*, to perform some work for it by sending the server a message and then blocking until the server sends back a reply. The communication mechanism used to implement the client-server model is called RPC [Birrell and Nelson 1984].

The RPC abstraction lets the programmer think in terms of normal procedure calls, which are well understood and have been around for a long time. This is in sharp contrast with, for example, the ISO OSI model. In this model, communication is treated as an input/output device, with user primitives for sending messages and getting indications of message arrivals. Many people think that input/output should not be the central abstraction of a modern programming language. Therefore, most distributed system builders, language designers, and programmers prefer RPC.

Group Communication

Although RPC is a good abstraction for the request/reply type of communication, there is a large body of applications that require a group of several processes to interact closely. Group communication allows a message to be sent reliably from 1 sender to *n* receivers. As discussed in Chapter 1, many applications can profit from such a communication primitive and many networks provide mechanism to do broadcast or multicast at the data-link layer.

One of the difficulties in making a protocol that allows user applications to use the data-link broadcast or multicast capability of a network is routing. A group address

has to be mapped on one or more data-link addresses, possibly on different networks. The protocol has to make sure that messages will not loop and that a minimum number of messages are used to transmit user data to the group. Groups may change over time, so routing tables have to be dynamically updated. Furthermore, to achieve good performance, the routing protocol should use a data-link multicast address to send a message to a number of receivers whenever possible.

Security

Although security cannot be provided by a communication protocol alone, a good protocol can provide mechanisms to build a secure, yet efficient distributed system. With current protocols, addresses can often be faked, making it possible for a process to impersonate an important service. For example, in many systems a user process can impersonate the file server once it knows the address of the file server (which is typically public knowledge). Most protocols do not provide any support for encryption of data. Users must decide whether or not to use encryption. Once they have decided to do so, they have to encrypt every message, even if both source and destination are located in the same secure room. A protocol provides much better performance by avoiding encryption if it knows a network is trusted, and using encryption if the network is not trusted.

Network Management

In an environment with many processors and networks, it often happens that a processor has to be taken down for maintenance or a network has to be reconfigured. With current software, reconfiguring a network typically requires manual intervention by a system administrator to assign new network numbers and to update the configuration files. Furthermore, taking some machines down often introduces communication failures for the rest of the machines. Ideally, a protocol makes it possible that network management can be done without any manual intervention.

Wide-Area Networking

Most processes in a distributed system communicate with services that are located nearby. For example, to read a file, users normally do an RPC with their local file server and not with a file server in another domain on another continent. Although communication with another domain must be possible, it should not introduce a performance loss for the more common, local case.

Why a New Protocol?

None of the current protocols addresses the requirements for distributed systems and applications adequately. The TCP and OSI protocols are connection-oriented and require a setup before any message can be sent. In a distributed system, processes are often short-lived and perform mostly small RPCs. In such an environment the time spent in setting up a connection is wasted. Indeed, almost none of the current RPC

implementations are based on connections. Although IP is a connectionless protocol, it still has some serious disadvantages. Because addresses in IP identify hosts instead of processes, systems based on IP are less transparent, making certain functionality, such as network management, harder to implement.

To meet the distributed system requirements using IP, a new protocol was invented for each subset of requirements. Protocols like ICMP, ARP, RARP, IGMP, and VMTP are examples. A big advantage of this approach is that one can adjust to new requirements without throwing away existing software. However, it is sometimes better to start from scratch. FLIP addresses the requirements in a clean, simple, and integrated way. The following FLIP properties allow us to achieve the requirements:

1. FLIP identifies entities with a location-independent 64-bit identifier. An entity can, for example, be a process.
2. FLIP uses an one-way mapping between the ‘‘private’’ address, used to register an endpoint of a network connection, and the ‘‘public’’ address used to advertise the endpoint.
3. FLIP routes messages based on the 64-bit identifier.
4. FLIP discovers routes on demand.
5. FLIP uses a bit in the message header to request transmission of sensitive messages across trusted networks.

In the next sections we will present FLIP, discuss our experience using it, and its performance in the Amoeba distributed system. FLIP is the basis for all communication within Amoeba and is in day-to-day use.

2.2. Flip Service Definition

This section describes the services that FLIP delivers. Communication takes place between *Network Service Access Points* (NSAPs), which are addressed by 64-bit numbers. NSAPs are location-independent, and can move from one node to another (possibly on different physical networks), taking their addresses with them. Nodes on an internetwork can have more than one NSAP, typically one or more for each entity (e.g., process). FLIP ensures that this is transparent to its users. FLIP messages are transmitted unreliably between NSAPs and may be lost, damaged, or reordered. The maximum size of a FLIP message is $2^{32}-1$ bytes. As with many other protocols, if a message is too large for a particular network, it will be fragmented into smaller chunks, called *fragment*s. A fragment typically fits in a single network *packet*. The reverse operation, re-assembly, is (theoretically) possible, but receiving entities have to be able to deal with fragmented messages.

The address space for NSAPs is subdivided into 256 56-bit address spaces, requiring 64 bits in all. The null address is reserved as the broadcast address. In this thesis we will define the semantics of only one of the address spaces, called the *stan-*

dard space, and leave the others undefined. Later these other address spaces may be used to add additional services.

The entities choose their own NSAP addresses at random (i.e., stochastically) from the standard space for four reasons. First, it makes it exceedingly improbable that an address is already in use by another, independent NSAP, providing a very high probability of uniqueness. (The probability of two NSAPs generating the same address is much lower than the probability of a person configuring two machines with the same address by accident.) Second, if an entity crashes and restarts, it chooses a new NSAP address, avoiding problems with distinguishing reincarnations (which, for example, is needed to implement at-most-once RPC semantics). Third, forging an address is hard, which, as we will see, is useful for security. Finally, an NSAP address is location-independent, and a migrating entity can use the same address on a new processor as on the old one.

Each physical machine is connected to the internetwork by a *FLIP box*. The FLIP box can either be a software layer in the operating system of the host, or be run on a separate communications processor. A FLIP box consists of several modules. An example of a FLIP box is shown in Figure 2.2.

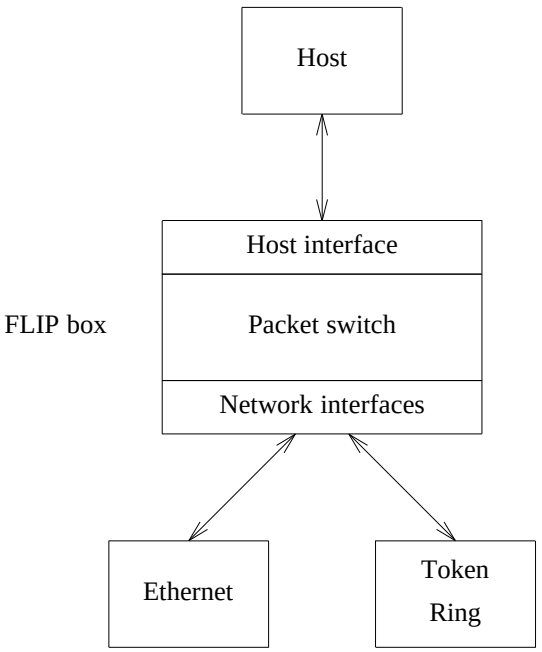


Fig. 2.2. A FLIP box consists of an host interface, packet switch, and network interfaces.

The *packet switch* is the heart of the FLIP box. It transfers FLIP fragments in packets between physical networks, and between the host and the networks. It maintains a dynamic hint cache mapping NSAP addresses on data-link addresses, called the

routing table, which it uses for routing fragments. As far as the packet switch is concerned, the attached host is just another network. The *host interface* module provides the interface between the FLIP box and the attached host (if any). A FLIP box with one physical network and an interface module can be viewed as a traditional network interface. A FLIP box with more than one physical network and no interface module is a router in the traditional sense.

2.3. The Host Interface

In principle, the interface between a host and a FLIP box can be independent of the FLIP protocol, but for efficiency and simplicity, we have designed an interface that is based on the FLIP protocol itself. The interface consists of seven downcalls (for outgoing traffic) and two upcalls (for incoming traffic), as shown in Figure 2.3.

Routine	**Description**
Flip_init(ident, receive, notdeliver) → ifno	Allocate an entry in the interface
Flip_end(ifno)	Close entry in the interface
Flip_register(ifno, Private-Address) → EP	Listen to address
Flip_unregister(ifno, EP)	Remove address
Flip_unicast(ifno, msg, flags, dst, EP, length)	Send a message to *dst*
Flip_multicast(ifno, msg, flags, dst, EP, length, ndst)	Send a multicast message
Flip_broadcast(ifno, msg, EP, length, hopcnt)	Broadcast *msg* hopcnt hops
Receive(ident, fragment description)	Fragment received
Notdeliver(ident, fragment description)	Undelivered fragment received

Fig. 2.3. Interface between host and packet switch. A fragment description contains the data, destination and source, message identifier, offset, fragment length, total length, and flags of a received fragment (see next section).

An entity allocates an entry in the interface by calling *flip_init*. The call allocates an entry in a table and stores the pointers for the two upcalls in this table. Furthermore, it stores an identifier used by higher layers. An allocated interface is removed by calling *flip_end.*

By calling *flip_register* one or more times, an entity registers NSAP addresses with the interface. An entity can register more than one address with the interface (e.g., its own address to receive messages directed to the entity itself and the null address to receive broadcast messages). The address specified, the *Private-Address*, is not the (public) address that is used by another entity as the destination of a FLIP mes-

sage. However, public and private addresses are related using the following function on the low-order 56 bits:

Public-Address = One-Way-Encryption(Private-Address)

The One-Way-Encryption function generates the Public-Address from the Private-Address in such a way that one cannot deduce the Private-Address from the Public-Address. Entities that know the (public) address of an NSAP (because they have communicated with it) are not able to receive messages on that address, because they do not know the corresponding private address. Because of the special function of the null address, the following property is needed:

One-Way-Encryption(Address) = 0 if and only if Address = 0

This function is currently defined using DES [National Bureau of Standards 1977]. If the 56 lower bits of the Private-Address are null, the Public-Address is defined to be null as well. The null address is used for broadcasting, and need not be encrypted. Otherwise, the 56 lower bits of the Private-Address are used as a DES key to encrypt a 64-bit null block. If the result happens to be null, the result is again encrypted, effectively swapping the result of the encrypted null address with the encrypted address that results in the null address. The remaining 8 bits of the Private-Address, concatenated with the 56 lower bits of the result, form the Public-Address.

Flip_register encrypts a Private-Address and stores the corresponding Public-Address in the routing table of the packet switch. A special flag in the entry of the routing table signifies that the address is local, and may not be removed (as we will see in Section 2.4). A small EP-identifier (End Point Identifier) for the entry is returned. Calling *flip_unregister* removes the specified entry from the routing table.

There are three calls to send an arbitrary-length message to a Public-Address. They differ in the number of destinations to which *msg* is sent. None of them guarantee delivery. *Flip_unicast* tries to send a message point-to-point to one NSAP. *Flip_multicast* tries to send a message to at least *ndst* NSAPs. *Flip_broadcast* tries to send a message to all NSAPs within a virtual distance *hopcnt.* If a message is passed to the interface, the interface first checks if the destination address is present in the routing table and if it thinks enough NSAPs are listening to the destination address. If so, the interface prepends a FLIP header to the message and sends it off. Otherwise, the interface tries to locate the destination address by broadcasting a LOCATE message, as explained in the next section. If sufficient NSAPs have responded to the LOCATE message, the message is sent away. If not, the upcall *notdeliver* will be called to inform the entity that the destination could not be located. When calling one of the send routines, an entity can also set a bit in *flags* that specifies that the destination address should be located, even if it is in the routing table. This can be useful, for example, if the RPC layer already knows that the destination NSAP has moved. Using the *flags* parameter the user can also specify that security is necessary.

When a fragment of a message arrives at the interface, it is passed to the appropriate entity using the upcall *receive*.

This interface delivers the bare bones services that are needed to build higher-level protocols, such as RPC. Given the current low error-rates of networks, we decided not to guarantee reliable communication at the network level, to avoid duplication of work at higher levels [Saltzer et al. 1984]. Higher-level protocols, such as RPC, send acknowledgement messages anyway, so given the fact that networks are very reliable it is a waste of bandwidth to send acknowledgement messages at the FLIP level as well. Furthermore, users will never call the interface directly, but use RPC or group communication.

2.4. The Flip Protocol

A FLIP box implements unreliable message communication between NSAPs by exchanging FLIP fragments and by updating the routing table when a fragment arrives. In this section, we will describe the layout of a FLIP fragment and tell how the routing table is managed.

2.4.1. The FLIP Fragment Format

Similar to fragments in many other protocols, a FLIP fragment is made up of two parts: the FLIP header and the data. The general format of a FLIP header is depicted in Figure 2.4. A header consists of a 40-byte fixed part and a variable part. The fixed part of the header contains general information about the fragment. The *Actual Hop Count* contains the weight of the path from the source. It is incremented at each FLIP box with the weight of the network over which the fragment will be routed. If the *Actual Hop Count* exceeds the *Maximum Hop Count*, the fragment will be discarded. The *Reserved* (*Res*.) field is reserved for future use.

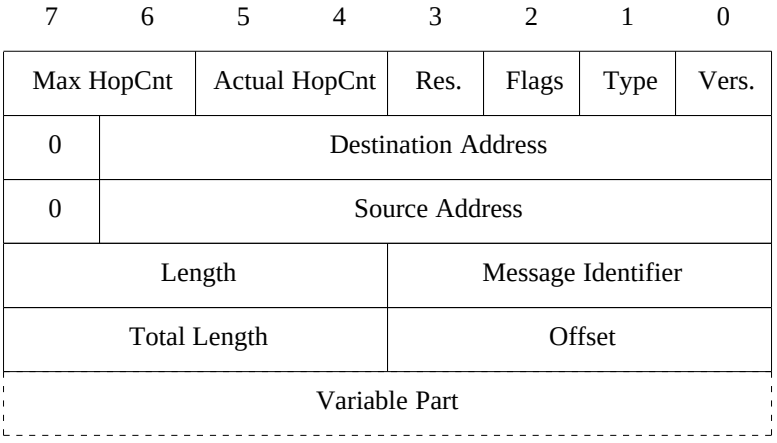


Fig. 2.4. General format of a FLIP fragment.

The *Flags* field contains administrative information about the fragment (see Fig. 2.5). Bits 0, 1, and 2 are specified by the sender. If bit 0 is set in *Flags*, the integer fields (hop counts, lengths, Message Identifier, Offset) are encoded in big endian (most significant byte first), otherwise in little endian [Cohen 1981]. If bit 1 is set in *Flags*, there is an additional section right after the header. This *Variable Part* contains *parameters* that may be used as hints to improve routing, end-to-end flow control, encryption, or other, but is never necessary for the correct working of the protocol. Bit 2 indicates that the fragment must not be routed over untrusted networks. If fragments only travel over trusted networks, the contents need not be encrypted. Each system administrator can switch his own network interfaces from trusted to untrusted or the other way around.

Bit	Name	Cleared	Set
0	Endian	Little endian	Big endian
1	Variable Part	Absent	Present
2	Security	Not required	Don't route over untrusted networks
3	Reserved		
4	Unreachable	Location unknown	Can't route over trusted networks only
5	Unsafe	Safe	Routed over untrusted network(s)
6	Reserved		
7	Reserved		

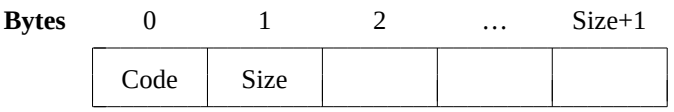
Fig. 2.5. Bits (4 input and 4 output) in the *Flags* field.

Bits 4 and 5 are set by the FLIP boxes (but never cleared). Bit 4 is set if a fragment that is not to be routed over untrusted networks (bit 2 is set) is returned because no trusted network was available for transmission. Bit 5 is set if a fragment was routed over an untrusted network (this can only happen if the *Security* bit, bit 2, was not set). Using bits 2, 4, and 5 in the *Flags* field, FLIP can efficiently send messages over trusted networks, because it knows that encryption of messages is not needed.

The *Type* field in the FLIP header describes which of the (six) messages types this is (see below). The *Version* field describes the version of the FLIP protocol; the version described here is 1. The *Destination Address* and the *Source Address* are addresses from the standard space and identify, respectively, the destination and source NSAPs. The null *Destination Address* is the broadcast address; it maps to all addresses. The *Length* field describes the total length in bytes of the fragment exclud-

ing the FLIP header. The *Message Identifier* is used to keep multiple fragments of a message together, as well as to identify retransmissions if necessary. *Total Length* is the total length in bytes of the message of which this fragment is a part, with *Offset* the byte offset in the message. If the message fits in a single fragment, *Total length* is equal to *Length* and *Offset* is equal to zero.

The *Variable Part* consists of the number of bytes in the *Variable Part* and a list of parameters. The parameters are coded as byte (octet) strings as follows:



The (nonzero) *Code* field gives the type of the parameter. The *Size* field gives the size of the data in this parameter. Parameters are concatenated to form the complete *Variable Part*. The total length of the *Variable Part* must be a multiple of four bytes, if necessary by padding with null bytes.

2.4.2. The FLIP Routing Protocol

The basic function of the FLIP protocol is to route an arbitrary-length message from the source NSAP to the destination NSAP. In an internetwork, destinations are reachable through any one of several routes. Some of these routes may be more desirable than others. For example, some of them may be faster, or more secure, than others. To be able to select a route, each FLIP box has information about the networks it is connected to.

In the current implementation of FLIP, the routing information of each network connected to the FLIP box is coded in a *network weight* and a *secure flag*. A low network weight means that the network is desirable to forward a fragment on. The network weight can be based, for example, on the physical properties of the network such as bandwidth and delay. Each time a fragment makes a hop from one FLIP box to another FLIP box its *Actual Hop Count*† is increased with the weight of the network over which it is routed (or it is discarded if its *Actual Hop Count* becomes greater than its *Maximum Hop Count*). A more sophisticated network weight can be based on the type of the fragment, which may be described in the *Variable Part* of the header. The *secure* flag indicates whether sensitive data can be sent unencrypted over the network or not.

At each FLIP box a message is routed using information stored in the routing table. The routing table is a cache of hints of the form:

(Address, Network, Location, Hop Count, Trusted, Age, Local)

---

† *Hop Count* is a misnomer, but it is maintained for historical reasons.

Address identifies one or more NSAPs. *Network* is the hardware-dependent network interface on which *Address* can be reached (e.g., Ethernet interface). *Location* is the data-link address of the next hop (e.g., the Ethernet address of the next hop). *Hop Count* is the weight of the route to *Address*. *Trusted* indicates whether this is a secure route towards the destination, that is, sensitive data can be transmitted unencrypted. *Age* gives the age of the tuple, which is periodically increased by the FLIP box. Each time a fragment from *Address* is received, the *Age* field is set to 0. *Local* indicates if the address is registered locally by the host interface. If the *Age* field reaches a certain value and the address is not local, the entry is removed. This allows the routing table to forget routes and to accommodate network topology changes. The *Age* field is also used to decide which entries can be purged, if the routing table fills up.

The FLIP protocol makes it possible for routing tables to automatically adapt to changes in the network topology. The protocol is based on six message types (see Fig. 2.6). The precise protocol is given in Appendix A; here we will give a short description. If a host wants to send a message to a FLIP address that is not in its routing table, it tries to locate the destination by broadcasting a LOCATE message†. LOCATE messages are propagated to all FLIP boxes until the *Actual Hop Count* becomes larger than the *Maximum Hop Count*. If a FLIP box has the destination address in its routing table, it sends back an HEREIS message in response to the LOCATE. User data is transmitted in UNIDATA or in MULTIDATA messages. UNIDATA messages are used for point-to-point communication and are forwarded through one route to the destination. MULTIDATA messages are used for multicast communication and are forwarded through routes to all the destinations. If a network supports a multicast facility, FLIP will send one message for all destinations that are located on the same network. Otherwise, it will make a copy for each location in the routing table and send point-to-point messages.

If a FLIP box receives a UNIDATA message with an unknown destination, it turns the message into a NOTHERE message and sends it back to the source. If a FLIP box receives a UNIDATA message that cannot be routed over untrusted networks (as indicated by the *Security* bit), and that cannot be routed over trusted networks, it turns the message into an UNTRUSTED message and sends it back to the source just like a NOTHERE message. Moreover, it sets the *Unreachable* bit in the message (regardless of its current value). For a message of any other type, including a MULTIDATA message, if the *Security* bit is set, and the message cannot be routed over trusted networks, it is simply dropped. If, for a NOTHERE or an UNTRUSTED message, a FLIP box on the way back knows an alternative route, it turns the message back into a UNIDATA message and sends it along the alternative route. If, for a NOTHERE message, no FLIP box knows an

† We assume that a network has a broadcast facility. For networks that do not have such a facility, we are considering adding a name server.

Type	Function
LOCATE	Find network location of NSAP
HEREIS	Reply on LOCATE
UNIDATA	Send a fragment point-to-point
MULTIDATA	Multicast a fragment
NOTHERE	Destination NSAP is unknown
UNTRUSTED	Destination NSAP cannot be reached over trusted networks

Fig. 2.6. FLIP message types.

alternative route, the message is returned to the source NSAP and each FLIP box removes information about this route from the routing table.

LOCATE messages must be used with care. They should be started with a *Maximum Hop Count* of one, and incremented each time a new locate is done. This limits the volume of broadcasts needed to locate the destination. Even though the hop counts are a powerful mechanism for locating a destination and for finding the best route, if routing tables become inconsistent, LOCATE messages may flood the internetwork (e.g., if a loop exists in the information stored in the routing tables in the internetwork). To avoid this situation, each FLIP box maintains, in addition to its routing table, a cache of (Source Address, Message Identifier, Offset, Destination Network, Location) tuples, with a standard timeout on each entry. For each received broadcast message, after updating the routing table, it checks whether the tuple is already in the cache. If not, it is stored there. Otherwise, the timeout is reset and the message is discarded. This avoids broadcast messages flooding the network if there is a loop in the network topology.

To illustrate how the FLIP box works, let us look at an example of how the RPC layer sends a point-to-point message to another process in the network topology depicted in Figure 2.7. The topology consists of three machines. It is not very realistic, but it allows us to explain some important properties of FLIP using a simple internetwork. When a FLIP box boots, it reads information about its configuration from a table (i.e., the type of networks it is connected to and information about these networks, such as the maximum packet size). This information tells the machine how many interfaces it has, the type of the interfaces, and some network dependent information, such as the weight of the network and whether the network has a multicast facility. After a FLIP box is initialized, it starts running with an empty routing table.

The example network topology contains two network types: a VME-bus and an Ethernet. Because a VME-bus is faster than an Ethernet, the weight given to the

VME-bus is lower than the weight given to the Ethernet. Every FLIP box is reachable from another host through different routes. There is, for example, a path of weight 1 from *A* to *B*, but also a path of weight 4 (from *A* to *C* over the Ethernet and then from *C* to *B* over the VME-bus).

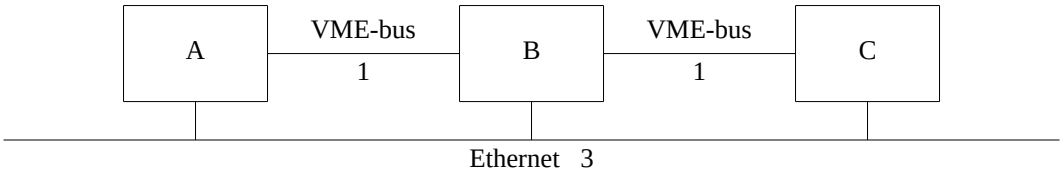


Fig. 2.7. An example network topology. FLIP box *A* and *C* both have two network interfaces: one for the VME-bus and one for the Ethernet. FLIP box *B* has 3 network interfaces: two to the VME-bus and one to the Ethernet. The VME-bus has weight 1 and the Ethernet has weight 3.

Let us now consider the case that the RPC layer sends a message from process P_1 on host A to process P_2 running on host *B*. When both processes start, the RPC layers register the FLIP addresses for the processes with their own FLIP box. The RPC layer of P_1 sends a message by calling *flip_unicast* with the public address of P_2 as the destination address (we assume that P_1 knows the public address of P_2). Because the address of P_2 is not initially present in the routing table of *A*, *A* buffers the message and starts to locate P_2 by sending a LOCATE message with *Max Hop Count* set to 1. *A*'s FLIP box will forward this message on the VME-bus, but not on the Ethernet, because to forward a message across the Ethernet the *Maximum Hop Count* must be at least 3. When the LOCATE message arrives at *B*, the FLIP address of P_1 will be entered in *B*'s routing table along with the weight of the route to P_1, the VME-bus address of *A*, and the network interface on which *A* is reachable. Because the public address of P_2 is registered with *B*'s routing table, *B* will return an HEREIS message. When the HEREIS message arrives at *A*, *A* enters P_2's public address in its routing table and sends the message that is waiting to be sent to P_2. Lower layers in the FLIP box will cut the message in fragments, if necessary. *B* receives the message for P_2 from the VME-bus and will forward it to the RPC layer of P_2 by calling *receive*. From now on, the routes to both P_1 and P_2 are known to *A* and *B*, so they can exchange messages without having to locate each other.

Now, assume that P_2 migrates to host *C*. The RPC layer unregisters the address at host *B* and registers it at host *C*. Thus, P_2 has removed its address from *B*'s routing table and has registered it with C's routing table. The next FLIP UNIDATA message of a message that arrives at *B* from *A*, will be returned to *A* as a FLIP NOTHERE message, because the address of P_2 is not present in *B*'s routing table. When *A* receives the NOTHERE message, it will invalidate the route to P_2. As *A* does not know an alternative route with the same or less weight to P_2, it will pass the NOTHERE message to the interface. The interface forwards the message to P_1's RPC layer by calling *notdeliver*.

P_1's RPC layer can now retransmit the message by calling *flip_unicast* again. As the route to P_2 has been invalidated, the interface will buffer the message and start by locating P_2 with *Max Hop Count* set to 1. After a timeout it will locate with *Max Hop Count* set to 2. Then, it will find a route to P_2: a hop across the VME-bus to *B* and another hop across the VME-bus from *B* to *C*. It will enter this new route with weight 2 in its routing table and forward the message across the VME-bus to *B*. When *B* receives the message, it will forward the message to *C*. From then on, P_1 and P_2 can exchange messages without locating each other again.

If the topology changes and, for example, *A* is disconnected from the VME-bus, the route to P_2 in *A*'s routing table will be removed after a period of time, because no messages will arrive via the VME-bus and therefore the age field of the entry in the routing table will reach the value that causes it to be removed. If P_1 then tries to send a message to P_2, the interface will again start by locating P_2's public address. This time it will find a route with weight 3; one hop across the Ethernet. If P_1 sends a new message before the route to P_2 is purged from *A*'s routing table, A will forward the message across the VME-bus and the message will be lost (assuming that the driver for the VME-bus does not return any error). In this case, the RPC layer has to tell the interface explicitly (using the *flags* parameter of flip_unicast) to purge the routing table entry. It does this, for example, if it did not receive an acknowledgement after a number of retrials.

Finally, assume that instead of *A*, *B* is disconnected from the VME-bus. A will first use its route with weight 2 and send the message to B across the VME-bus. If B does not know yet that the route over the VME-bus to C disappeared, it will forward the message over the VME-bus and the message will be lost. Otherwise, it will send the message as a NOTHERE message back to A, because the *Max Hop Count* is set by *A* to 2. In both cases, *A* will send the message to *C* using the Ethernet, possibly after doing another locate.

2.5. Using Flip under Amoeba

FLIP is the basis for all communication within the Amoeba distributed system. The configuration at the Vrije Universiteit is depicted in Figure 2.8. The pool processors, the I80486 router, and the specialized servers run the Amoeba kernel (see Fig. 2.9). The workstations and the SPARC router run either Amoeba or a version of UNIX containing a FLIP driver, so UNIX and Amoeba processes can communicate transparently. All the 70 machines are connected through 3 Ethernets and the processors in the MC68030 pool are also connected by VME-buses. We also implemented FLIP across TCP/IP and UDP/IP, so that we can use TCP/IP connections as a data link. This implementation is the basis for a small scale WAN project that connects multiple sites in The Netherlands, and has been tested across the Atlantic as well. We will now describe how FLIP meets each of the distributed system requirements listed in Section 2.1.

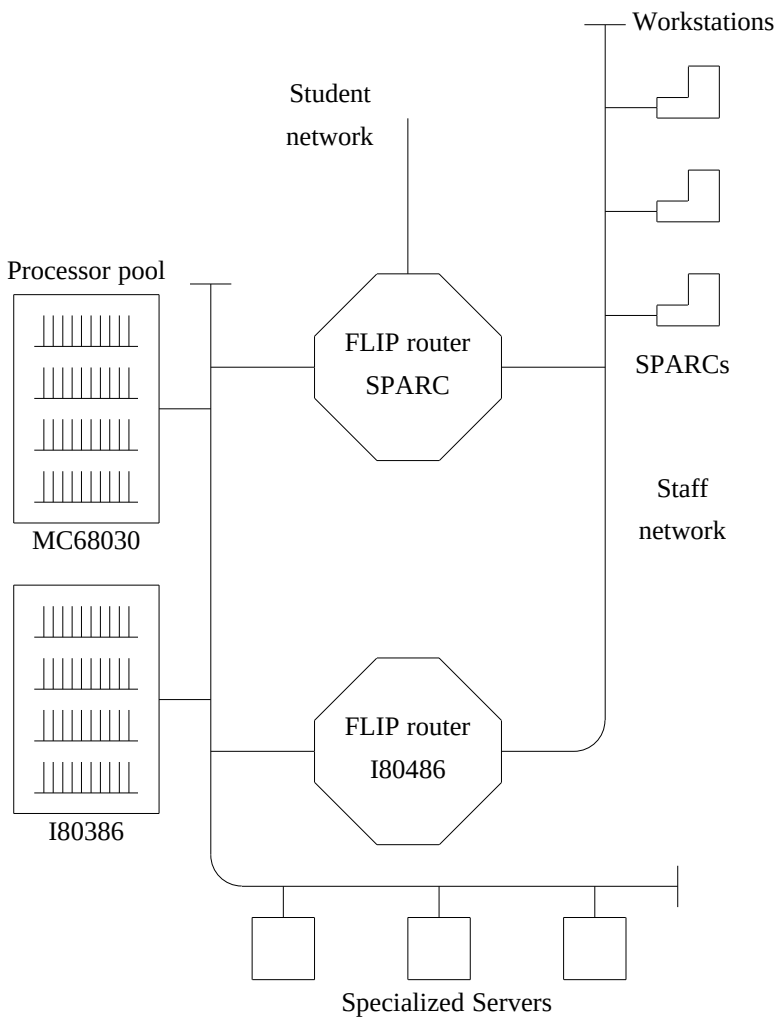


Fig. 2.8. The FLIP internetwork at the Vrije Universiteit. It contains three different machine architectures with different endianness and two types of networks: Ethernet and VME-buses. On average ten people are using the Amoeba system every day to develop system software, and distributed and parallel applications.

Transparency

The primary goal of Amoeba is to build a transparent distributed operating system (see Section 1.5). To the average user, Amoeba looks like a traditional timesharing system. The difference is that each command typed by the user makes use of multiple machines spread around the network. The machines include process servers, file servers, directory servers, compute servers, and others, but the user is not aware of any of this. At the terminal, it just looks like an ordinary time sharing system.

To achieve this degree of transparency a two level naming scheme is used: capa-

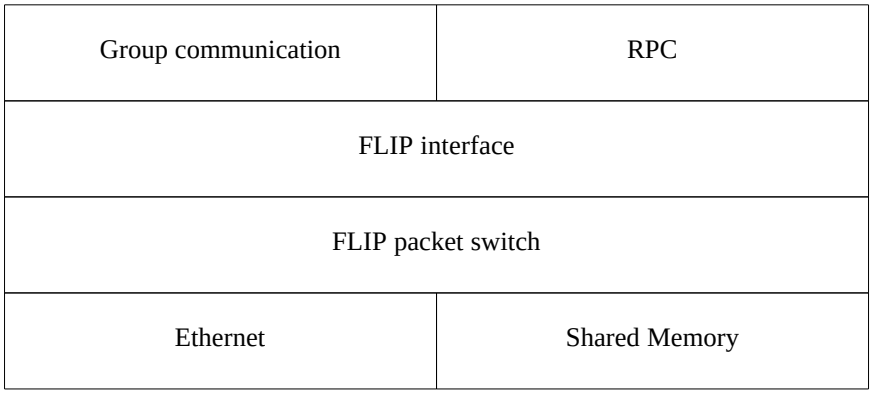


Fig. 2.9. Communication layers in the Amoeba kernel for a pool processor.

bilities and FLIP addresses. Each object (e.g., a file) is named with a capability. Associated with each object type is a service (a single process or a group of processes) that manages the object. When a client wants to perform an operation on an object, it sends a request message to the service that manages the object. The service is addressed by the port, which is part of the capability. In short, capabilities are persistent names that identify objects.

To make capabilities easy to use, users can register them with the *directory service*. The directory allows users to register capabilities under an ASCII string. Furthermore, it implements a UNIX-like access protection scheme. Details about the implementation and functionality of the directory service can be found in Chapter 5.

Within the kernel, ports are mapped onto one or more FLIP addresses, one for each server. When a client wants to perform an operation on an object, it provides the kernel with the capability of the object. The kernel extracts the port from the capability and looks in its port cache for a FLIP address that listens to the port. Using the FLIP address, the kernel sends messages, relying on the FLIP box to deliver the messages to the right location. If there is no mapping from port to FLIP address in the cache, the kernel uses *flip_broadcast* to locate the port. The FLIP addresses of the responders to the LOCATE request are stored with the port in the port cache to avoid future locates. This locate procedure has the important side effect that at the same time the FLIP boxes build up their routing tables, so a second locate at the FLIP level is avoided. In the common case that networks do not change rapidly and processes migrate infrequently, no LOCATE messages are sent.

At-Most-Once RPC

The RPC layer in the Amoeba kernel provides an interface for at-most-once RPC, so when the RPC returns the invoker knows whether (1) it was executed exactly once, or (2) it was not executed at all, or (3) it arrived at one server before contact was lost due to communication errors or a crash. One of the problems in achieving at-

most-once semantics is deciding whether a new incoming request has been executed or not. With FLIP, this problem is easily solved. Each time a server is started, the server chooses a new FLIP address. Thus, all requests sent to a crashed server will fail automatically, because the old FLIP address is unknown. During one incarnation of the server, the server can decide, based on sequence numbers in the message, whether the request was executed or not.

Our implementation of RPC is very similar to Birrell and Nelson's [Birrell and Nelson 1984], except for two important differences. First, because FLIP addresses are 64-bit large and location-independent, our implementation has no need for a unique identifier; the FLIP address is the unique identifier. Second, our implementation does not use the next request as an acknowledgement for the last reply. Instead, our implementation sends an explicit acknowledgement when the reply is received. This simplifies the implementation of the RPC layer. Furthermore, sending the acknowledgement is not in the critical path of an RPC (see the next section).

Group Communication

The group communication is based on the protocols described in the next chapter. It provides a primitive to send a message to a group of processes reliably. Furthermore, this primitive guarantees that all broadcast messages within a group are totally-ordered. The group communication protocols make heavy use of *flip_multicast*. This has the advantage that a group of *n* processes can be addressed using one FLIP address, even if they are located on multiple networks.

As explained in Section 2.4.2, we treat the ability of a network to send multicast messages as an optimization over sending *n* separate point-to-point messages. If the FLIP box discovers that a FLIP address is routed to *n* locations on the same network, it asks the network dependent layer to return a multicast address for the *n* locations. It is then up to the network layer to create such a multicast address and to make sure that the *n* locations will listen to it. After the network layer has done so, it returns to the packet switch a multicast address and a list of locations that listen to the multicast address. From then on, the packet switch can use the multicast address. The implementation of the Ethernet layer does this as soon as the FLIP box maps an address onto two locations on the same Ethernet.

Thus, the FLIP protocol does all the routing of multicast messages, and recognizes when a data-link multicast could be used to reduce the number of messages. Once it recognizes the possibility of optimization, it leaves it up to a network dependent layer to perform it. The reason that FLIP itself cannot perform the optimization is that FLIP does not know about the data-link addresses for multicast.

Security

Security in Amoeba is implemented using the FLIP support for security. Although FLIP does not encrypt messages itself, it provides two mechanisms for supporting security. First, messages can be marked sensitive by the sender (using the *Security* bit), so that they will not be routed over untrusted networks. Second, messages going through FLIP may be marked unsafe (using the *Unsafe* bit), so that the receiver can tell whether or not there is a safe route to the sender. If, based on this information, a process thinks there is a safe route to the destination, it can try to send sensitive messages unencrypted, but with the *Security* bit set. If this message is bounced with the *Unreachable* bit set, no trusted path exists after all. This can only happen due to configuration changes. The process can then encrypt the message, and retransmit it with the *Security* bit cleared.

Our implementation of secure RPC is in an experimental phase and is not yet in day to day use; we are still studying how to do secure group communication. Like many secure systems, Amoeba's secure RPCs are based on a shared key between the client and the server and its implementation is roughly similar to Birrell's [Birrell 1985]. The main difference is that our implementation uses FLIP's knowledge about trusted and untrusted networks. The Amoeba processor pools and specialized servers are located in one single room and together form a trusted network. Thus, all communication between processes in the processor pool and, for example, the file service does not have to be encrypted. However, as soon as a FLIP message leaves this network, it is guaranteed to be encrypted (if it is part of a secure RPC). This encryption is transparent to the user. Our expectation is that we can build a complete secure system with acceptable performance, because the common case does not require encryption. Furthermore, it is not necessary that all processors be equipped with encryption hardware.

Network Management

Little network management is required in Amoeba. FLIP can deal automatically with network changes: we add machines, networks, or reconfigure our systems just by plugging or unplugging cables. When a machine comes up, it does not have to send out ARP or RARP requests and wait until a server responds; instead it can be used as soon as it is plugged into the network.

The only network management that is required has to do with trusted and untrusted networks. FLIP relies on the system administrator to mark a network interface as ''trusted'' or ''untrusted,'' because FLIP itself cannot determine if a network can be considered trusted. In our implementation only the system administrator can toggle this property.

Wide-Area Communication

Although FLIP has been used successfully in small WANs, it does not scale well enough to be used as the WAN communication protocol in a large WAN. Addresses form a flat name space that is not large enough to address all the machines in the world and still ‘‘guarantee’’ uniqueness. Furthermore, the way FLIP uses broadcast makes it less suitable for a WAN. We traded scalability for functionality. Moreover, we believe that WAN communication should not be done at the network layer, but at a higher layer in the protocol hierarchy.

There are three reasons for doing so. First, most communication is local within one domain†. Thus, we decided we did not want to give up on flexibility and performance, just because a message could go to a remote domain.

A second reason to make a distinction between a local and remote domain is that protocols on a WAN link differ from protocols used in a distributed system. WAN links are mostly owned by phone companies that are not interested in fast RPCs. Furthermore, different protocols on WANs are used to cope with the higher error rates and the lower bandwidth of WAN links. Thus, making a protocol suitable for WAN communication at the network layer could very well turn out to be a bad design decision, because at the boundary of a domain the messages may have to be converted to the protocols that are used on the WAN link.

The third reason has to do with administration of domains. WAN communication typically costs more money than communicating across a LAN. Transparently paying large amounts of money is unacceptable for most people. Furthermore, even if there is no boundary at the network layer, there is still a logical boundary. Administrators control domains independently and they like to have control over what traffic is leaving and entering their domain. An administrator might want to keep ‘‘dangerous messages’’ out of his domain. If communication is transparent at the network layer, this is hard to achieve, as recently demonstrated by the worm on the Internet [Spafford 1989].

In the Amoeba system we have implemented WAN communication above the RPC layer [Van Renesse et al. 1987]. If a client wants to access a service in another domain, it does an RPC to a *server agent* in its domain. The server agent sends the RPC to the WAN server, which forwards the RPC to the WAN service in the server’s domain using the appropriate protocol for the WAN link. The WAN service in the server’s domain creates a *client agent* that executes the same RPC and it will find the server.

---

† Measurements taken at our department show that 80% of all IP messages are destined for a host on the same network, 12% stay within the department, and that 8% are destined for some other IP site.

2.6. Performance of FLIP

An important measure of success for any protocol is its performance. We have compared the performance of Amoeba 5.0 RPC (with FLIP) with Amoeba 4.0 RPC (pre-FLIP version) and with other RPC implementations on identical hardware. The delay was measured by performing 10,000 0-byte RPCs. The throughput was measured by sending maximum-size RPCs. In Amoeba 4.0 this is measured by sending 30,000-byte RPCs; in Amoeba 5.0 this is measured by sending 100,000-byte RPCs (which is still smaller than the maximum possible size); in SunOS using 8-Kbyte RPCs; in Sprite using 16-Kbyte RPCs; and in Peregrine using 48,000-byte RPCs. To make direct comparisons possible we also measured Amoeba 5.0 RPC with the sizes used for the other systems. All measurements were made on Sun 3/60s connected by an almost quiet 10-Mbit/s Ethernet.

The first row in the table in Figure 2.10 gives the performance of RPC using the protocols in Amoeba 4.0. The second row in the table gives the performance for the new RPC implementation on top of FLIP. The delay in Amoeba 4.0 is lower than in Amoeba 5.0, because Amoeba 4.0 RPC is implemented over bare Ethernet and requires all machines in a domain to be on one network, so it does not have to do routing and the implementation can be tuned for the case of one network interface. In spite of the overhead for routing, the throughput in Amoeba 5.0 is 30% higher, largely because Amoeba 4.0 uses a stop-and-wait protocol, while Amoeba 5.0 uses a blast protocol [Zwaenepoel 1985] to send large messages. This enables user processes in Amoeba 5.0 RPC to get 87% of the total physical bandwidth of an Ethernet (the FLIP and RPC protocols including headers use 90% of the total bandwidth).

RPC implementation	Delay (msec)	Maximum Bandwidth (Kbyte/s)	Amoeba 5.0 Bandwidth (Kbyte/s)
Amoeba 4.0 RPC	1.1	814	993
Amoeba 5.0 RPC (FLIP)	2.1	1061	1061
Sprite RPC	2.0	820	884
Sun RPC	6.7	325	755
Peregrine RPC	0.6	1139	1001

Fig. 2.10. Performance numbers for different RPC implementations on Sun 3/60s. The Sprite numbers are measured from kernel to kernel. The others are measured from user to user. The fourth column gives the bandwidth for Amoeba 5.0 RPC using the data size for the system measured in each row.

For comparison, the delay of a 0-byte RPC in SunOS is 6.7 msec and the bandwidth for an 8-Kbyte RPC is 325 Kbyte/s (the maximum RPC size for SunOS is 8 Kbyte). This is due to the fact that SunOS copies each message several times before it is given to the network driver, due to its implementation on UDP/IP, and due to the higher cost for context switching. In Sprite, the delay is 2.0 msec and the maximum throughput is 821 Kbyte/s (these numbers are measured kernel to kernel). Although Sprite's kernel-to-kernel RPC does not do routing, the delay of the null RPC is almost the same as the delay for Amoeba 5.0, while Amoeba's delay is measured user-to-user. Sprite also uses a blast protocol for large messages, but its throughput is still less than the throughput achieved by Amoeba 5.0. This can be explained by the fact that Amoeba keeps its buffer contiguously in memory and that it has a much better context switching time [Douglis et al. 1991].

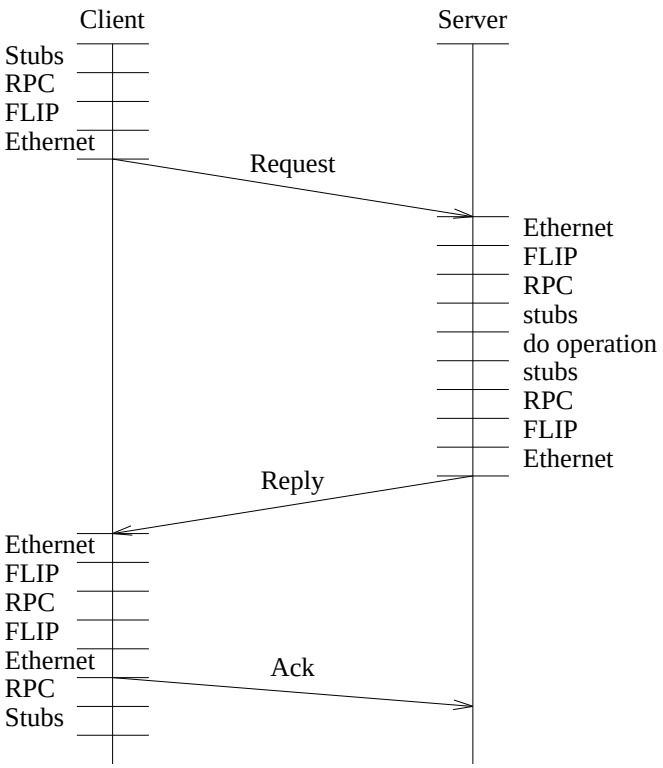
Compared to Peregrine's RPC [Johnson and Zwaenepoel 1991], Amoeba's delay for a 0-byte RPC is high and Amoeba's maximum throughput is low. Peregrine achieves on identical hardware a delay of 589 μsec and a bandwidth of 1139 Kbyte/s. Peregrine's performance for the null RPC is only 289 μsec above the minimum possible hardware latency. Peregrine achieves this performance by directly remapping the Ethernet receive buffer in the server machine to become the new thread's stack and by using preallocated and initialized message headers. Furthermore, Peregrine uses a two-message RPC protocol, while Amoeba is using a three-message RPC protocol, although the third message is only partly in the critical path. Peregrine achieves a high throughput by overlapping the copying of data from a packet with the transmission of the next packet. The last packet is, like the single-packet case, directly remapped, avoiding the copying of data. We believe that we can apply many of Peregrine's optimizations in Amoeba, which will probably result in a similar performance as Peregrine's. For more performance numbers on these and other RPC implementations see [Tanenbaum et al. 1990]. Thus, in addition to providing more functionality, FLIP makes it also possible to achieve very good performance.

To determine the overhead in FLIP, we measured the time spent in each layer during a null RPC (see Fig. 2.11). The overhead due to FLIP is 21% of the total delay for a null RPC. From the numbers given one can also compute what the costs are if the server and client were located on different networks. Each additional hop over another Ethernet increases the delay of a null RPC by 975 μsecs.

Detailed performance figures for group communication are given in the next chapter.

2.7. Discussion and Comparison

Many communication protocols have been introduced in the last decade. Some of them are accepted as official standards or are used by a large user community, such as X.25 [Zimmerman 1980] and IP; others are tailored to specific applications, such as the Express Transfer Protocol (XTP) [Saunders and Weaver 1990]. In a distributed



(a)

Layer	**Time (μsec)**
User	100
RPC	840
FLIP	450
Ethernet	750

(b)

Fig. 2.11. (a) The Amoeba RPC protocol and (b) the time spent in the critical path of each layer for a null RPC. The RPC protocol for a small RPC uses three messages: 1) a request message from the client to the server; 2) a reply message that acknowledges the request and unblocks the client; 3) an acknowledgement for the reply, so that the server can clean up its state. The acknowledgement is only for a small part in the critical path. The Ethernet time is the time spent on the wire plus the time spent in the driver and taking the interrupt.

system like Amoeba many entities are short-lived and send small messages. In such an environment, setting up connections would be a waste of time and resources. We therefore decided to make FLIP a connectionless protocol. In this section, we compare FLIP to other connectionless protocols and discuss the advantages and disadvantages of

FLIP over these other protocols. We will not compare FLIP further to connection-based protocols.

Requirements	FLIP support
Transparency	• FLIP addresses are location-independent • A FLIP box performs routing • Messages can be as large as $2^{32}-1$ bytes. • Routing tables change automatically.
Efficient at-most-once RPC	• FLIP is an unreliable message protocol • FLIP can use a blast protocol for large messages • A process uses a new FLIP address after it crashes
Group Communication	• A FLIP address may identify a number of processes • Routing tables change dynamically • FLIP uses data-link multicast, if possible • FLIP also works if multicast is not available
Security	• Addresses are hard to forge • FLIP takes advantage of trusted networks
Network Management	• Every machine is a router • Routing tables are dynamically updated
WAN	• Works for small WAN-based projects

Fig. 2.12. How FLIP meets distributed systems requirements discussed in section 2.1.

However, before comparing FLIP to other connectionless protocols, we first summarize the requirements that distributed computing imposes on the communication system and the support that FLIP offers to meet these requirements (see Fig. 2.12).

2.7.1. Discussion

The main property of FLIP that gives good support for distributed computing is a combination of dynamic routing and the fact that FLIP addresses identify logical entities (processes or groups) rather than machines. Dynamic routing is done in a way roughly similar to transparent bridges [Backes 1988]. Each FLIP box keeps a cache of hints that is dynamically updated by FLIP messages. To keep routing tables up-to-date with the network topology, FLIP headers have a type field and include hop counts. The combination of dynamic routing tables and communication between entities simplifies the implementation of higher-level protocols such as RPC and group communication

and gives enhanced support for process migration. Furthermore, little network management is required.

The only requirement for which FLIP does not have full support is wide-area networking. We think, however, that wide-area communication should not be done at the network layer, but in higher layers.

The costs for the functionality of FLIP can be divided roughly into 3 areas: limited scalability, costs for broadcast, and memory for routing tables. By using a flat name space we lose on scalability, but gain the ability to make addresses location-independent and on the ability to do routing on a per-entity basis. One could envision adding a domain identifier to the FLIP header, so that FLIP would scale to larger internetworks. Using a domain identifier, all the good properties of FLIP would exist in a single domain, but not between two domains.

A danger in our current implementation of FLIP is that addresses might clash. Two processes could accidentally register the same FLIP address†. In this case, messages sent to process A may end up at process B. However, as long as the same process is not talking to A and B at the same time and routes to A and B do not intersect, most of the messages will still be delivered correctly. In the current situation with a good random generator and seed, clashes of FLIP addresses do not occur. Of course, if the number of entities increases enormously, the chance of clashes increases.

By using locate messages we have the ability to reconfigure networks dynamically and move processes around. The costs are that FLIP will generate more broadcasts than a protocol like IP and that there is a startup cost involved in locating a destination. Furthermore, there is a danger that FLIP could cause a flood of broadcasts. To avoid this we have introduced a hop count in the header, kept state (1 Kbyte) in each kernel to break loops, and limited the number of broadcasts per second that a FLIP box can forward. The net result is that Amoeba in the environment depicted in Figure 2.8 (which contains loops) on average generates only 1.6 broadcasts per second to locate ports and 0.1 broadcasts per second to locate FLIP addresses (measured over a 60 hour time period: three working days and two nights). Given the fact that it takes approximately 500 μsec to process a broadcast, we are paying only 0.1% of the CPU cycles of a machine for dealing with broadcasts. We find this a good tradeoff.

We locate destinations by expanding the scope of each broadcast. This has the disadvantage that networks close by will receive more broadcasts than networks further away. Furthermore, it introduces a potentially longer delay for destinations far away or destinations that do not exist. Because the RPC implementation caches mappings of ports to FLIP addresses and the FLIP implementation caches the mapping of FLIP addresses to locations, very little locating takes place, so the number of broadcasts is low. Most of the broadcasts are due to attempts to locate services that no longer exist. In a large internetwork the number of broadcasts could be too high and the delay too

---

† As soon as we started running FLIP on all our machines, we came across this problem, because many of the pseudo-random generators were at that time fed with the same seed.

long. In such an environment, one could implement a scheme which caches unreachable ports and FLIP addresses to reduce the number of broadcasts for not existing services. This scheme is, however, not trivial to implement correctly.

By using routing tables in each kernel, we can do routing on a per-process basis. The cost for doing this is that each kernel must keep such a table. In our current environment, we are using tables that can store 100 FLIP addresses; this requires only 6 Kbyte of storage.

2.7.2. Comparison

The rest of this section discusses alternative solutions for communication in distributed systems. One of the most widely used internet protocols is IP [Postel 1981a; Comer 1992]. In IP, an address identifies a host. Thus, if a process is migrated from one host to another host, it must change its IP address and tell other processes that it did so. Because IP uses a hierarchical address space, machines cannot be disconnected from one network and connected to another network without changing their IP addresses, although a new extension to IP has been proposed to deal with mobile computers [Ioannidis et al. 1991]. FLIP's flat address space also has some disadvantages. Routing tables are larger. Instead of having one entry for a collection of addresses on one network, FLIP needs a separate entry for every address. With the flat address space, FLIP also scales less well to wide-area communication. Another fundamental difference between IP and FLIP is IP's limit to the size of a message (64 Kbyte). Higher-level protocols have to break messages in 64 Kbyte fragments and reassemble them at the other side. As a result, IP does not benefit from communication links that allow packets larger than 64 Kbyte. A final fundamental difference is that IP provides only limited support for secure communication. For example, the standard IP specification does not provide secure routing.

Besides these fundamental differences, there are also a number of differences that are dependent on the IP implementation and routing protocol used. The Internet Control Message Protocol improves end-to-end flow control and routing [Postel 1981b]. However, there are still many problems. For example, many IP implementations make a distinction between a router and a host. A router does routing, and a host runs processes and does not do routing. If the network topology changes, it often happens that machines have to be restarted or reconfigured manually. Furthermore, all ongoing communication between a machine that is about to be moved and other machines will have to be aborted. As most departments own a large number of machines and many networks, these changes need to be done more often than any system administrator cares for. FLIP eliminates almost all need for network management; system administrators can install, move, or remove machines without making changes to the configuration tables.

Another protocol that has been especially designed for distributed operating systems is the Versatile Message Transaction Protocol (VMTP) [Cheriton 1986, 1988a]. Like FLIP, VMTP provides a base to build higher-level protocols, and has been used

for the protocols in the V distributed system [Cheriton 1988b]. Unlike FLIP, VMTP is a transport protocol, which relies on an internet protocol for routing. Therefore VMTP may be implemented on top of FLIP, providing the VMTP abstraction with the advantages of FLIP.

Three types of addresses exist in VMTP. They differ in the time that they are usable. T-stable addresses, for example, are guaranteed not to be reused for at least T seconds after they become invalid. This allows a timer-based implementation of at-most-once Remote Procedure Call. If one were to run VMTP on FLIP, such timed addresses would not be needed, because the 56 bits of an address would almost certainly be unique and an entity can pick a new address at any time. VMTP is a reliable transport protocol, and uses a single mechanism for fragmentation and flow control on all network types. To be able to implement this protocol efficiently, the designers also put an artificial upper bound on the size of a network message. Due to this artificial upper bound, and the fact that networks differ greatly in their physical properties, VMTP may perform well on one network and less well on another.

The routing algorithm that FLIP uses for MULTIDATA packets is similar to the single-spanning-tree multicast routing algorithm discussed by Deering and Cheriton [Deering and Cheriton 1990]. In the same paper, the authors also discuss more sophisticated multicast routing algorithms. These algorithms could be implemented in FLIP using the *Variable Part* of the header.

2.8. Conclusion

In this chapter we have discussed protocol requirements for distributed systems and proposed a new protocol that meets them. Current internet protocols do not address various problems, leaving the solution to higher-level protocols. This leads to more complex protocols, that cannot perform well, because they cannot take advantage of hardware support. We presented the FLIP protocol that supports many of the requirements of distributed systems in an integrated way. FLIP addresses management of internetworks, efficient and secure communication, and transparency of location and migration.

FLIP is used in the Amoeba 5.0 distributed operating system to implement RPC and group communication over a collection of different networks. The advantages over Amoeba 4.0 include better scaling, easier management, and higher bandwidth. Round-trip delay is currently higher, but this can probably be improved by careful coding and tuning.

There is more work to be done. For example, we have no experience with large networks containing thousands of subnets. However, since Amoeba implements wide-area communication transparently in user space, using X.25 or TCP links between Amoeba sites, this is at least conceivable. Additionally, locating endpoints with location-independent addresses can be a problem, and we are currently considering a location service for a possibly large network of subnets that may or may not support hardware multicast.

Notes

The research presented in this chapter was done in collaboration with Robbert van Renesse and Hans van Staveren. Wiebren de Jonge suggested a clean and nice improvement to the one-way-function used in the host interface.

3

EFFICIENT RELIABLE GROUP COMMUNICATION

Most current distributed systems are based on RPC, but many applications need something else. RPC is inherently point-to-point communication and what is often needed is 1-to-*n* communication. This chapter discusses a protocol for sending messages from 1 source to *n* destinations. It makes the following research contributions:

- It identifies and discusses design issues in group communication.
- It introduces an improved algorithm for reliable totally-ordered group communication.
- It provides enough detail of the algorithm that it can be implemented in any distributed system.
- It gives detailed performance measurements of an implementation in an existing distributed system.

The outline of the chapter is as follows. In Section 3.1 we discuss the design issues in group communication. In Section 3.2 we describe the choices that we have made for each issue. In Section 3.3 we present the Amoeba kernel primitives for group communication. In Section 3.4 we give the algorithms for an efficient reliable broadcast protocol that provides total ordering. In Section 3.5 we give detailed performance measurements of the group communication. In Section 3.6 we compare our protocol with a number of other protocols and other systems that support group communication. In Section 3.7 we present our conclusions.

3.1. Design Issues in Group Communication

A few existing operating systems provide application programs with support for group communication [Liang et al. 1990]. To understand the differences between these existing systems, six design criteria are of interest: addressing, reliability, ordering, delivery semantics, response semantics, and group structure (see Fig. 3.1). We will discuss each one in turn.

Issue	Description
Addressing	Addressing method for a group (e.g., list of members)
Reliability	Reliable or unreliable communication?
Ordering	Order among messages (e.g., total ordering)
Delivery semantics	How many processes must receive the message successfully?
Response semantics	How to respond to a broadcast message?
Group structure	Semantics of a group (e.g., dynamic versus static)

Fig. 3.1. The main design issues for group communication.

At least four methods of *addressing* messages to a group exist. The simplest one is to require the sender to explicitly specify all the destinations to which the message should be delivered. A second method is to use a single address for the whole group. This method saves bandwidth and also allows a process to send a message without knowing which processes are members of the group [Frank et al. 1985]. Two less common addressing methods are *source addressing* [Gueth et al. 1985] and *functional addressing* [Hughes 1988]. Using source addressing, a process accepts a message if the source is a member of the group. Using functional addressing a process accepts a message if a user-defined function on the message evaluates to true. The disadvantage of the latter two methods is that they are hard to implement with current network interfaces.

The second design criterion, *reliability*, deals with recovering from communication failures, such as buffer overflows and garbled packets. Because reliability is more difficult to implement for group communication than for point-to-point communication, a number of existing operating systems provide *unreliable* group communication [Cheriton and Zwaenepoel 1985; Rozier et al. 1988], whereas almost all operating systems provide *reliable* point-to-point communication, for example, in the form of RPC.

Another important design decision in group communication is the *ordering* of messages sent to a group. Roughly speaking, there are 4 possible orderings: no ordering, FIFO ordering, causal ordering, and total ordering. No ordering is easy to under-

stand and implement, but unfortunately makes programming often harder. FIFO ordering guarantees that all messages from a member are delivered in the order in which they were sent. Causal ordering guarantees that all messages that are related are ordered [Birman et al. 1991]. More specifically: messages are in FIFO order and if a member after receiving message *A* sends a message *B*, it is guaranteed that all members will receive *A* before *B*. In the total ordering, each member receives all messages in the same order. The last ordering is stronger than any of the other orderings and makes programming easier, but it is harder to implement.

To illustrate the difference between causal and total ordering, consider a service that stores records for client processes. Assume that the service replicates the records on each server to increase availability and reliability and that it guarantees that all replicas are consistent. If a client may only update its own records, then it is sufficient that all messages from the same client will be ordered. Thus, in this case a causal ordering can be used. If a client may update any of the records, then a causal ordering is not sufficient. A total ordering on the updates, however, is sufficient to ensure consistency among the replicas. To see this, assume that two clients, C_1 and C_2, send an update for record *X* at the same time. As these two updates will be totally-ordered, all servers either (1) receive first the update from C_1 and then the update from C_2 or (2) receive first the update from C_2 and then the update from C_1. In either case, the replicas will stay consistent, because every server applies the updates in the same order. If in this case causal ordering had been used, it might have happened that the servers applied the updates in different orders, resulting in inconsistent replicas.

The fourth item in the table, *delivery semantics*, relates to when a message is considered successfully delivered to a group. There are three common choices: *k*-delivery, quorum delivery, and atomic delivery. With *k*-delivery, a broadcast is defined as being successful when *k* processes have received the message for some constant *k*. With quorum delivery, a broadcast is said to be successful when a majority of the current membership has received it. With atomic delivery either all surviving processes receive it or none do. Atomic delivery is the ideal semantics, but is harder to implement since processors can fail.

Item five, *response semantics* deals with what the sending process expects from the receiving processes [Hughes 1989]. There are four broad categories of what the sender can expect: no responses, a single response, many responses, and all responses. Operating systems that integrate group communication and RPC completely often support all four choices [Cheriton and Zwaenepoel 1985; Birman et al. 1990].

The last design decision specific to group communication is *group structure*. Groups can be either closed or open [Liang et al. 1990]. In a *closed* group, only members can send messages to the group. In an *open* group, nonmembers may also send messages to the group. In addition, groups can be either static or dynamic. In static groups processes cannot leave or join a group, but remain a member of the group for the lifetime of the process. Dynamic groups may have a varying number of members over time; processes can come and go.

If processes can be members of multiple groups, the semantics for *overlapping groups* must be defined. Suppose that two processes are members of both groups G_1 and G_2 and that each group guarantees a total ordering. A design decision has to be made about the ordering between the messages of G_1 and G_2. All choices discussed in this section (none, FIFO, causal, and total ordering) are possible.

To make these design decisions more concrete, we briefly discuss two systems that support group communication. Both systems support open dynamic groups, but differ in their semantics for reliability and ordering. In the V system [Cheriton and Zwaenepoel 1985], groups are identified with a group identifier. If two processes concurrently broadcast two messages, *A* and *B*, respectively, some of the members may receive *A* first and others may receive *B* first. No guarantees about ordering are given. Group communication in the V system is unreliable. Users can, however, build their own group communication primitives with the basic primitives. They could, for example, implement the protocols described in this chapter as a library package.

In the Isis system [Birman and Joseph 1987], messages are sent to a group identifier or to a list of addresses. When sending a message, a user specifies how many replies are expected [Birman et al. 1990]. Messages can be totally-ordered, even for groups that overlap. If, for example, processes P_1 and P_2 in Figure 3.2 simultaneously send a message, processes P_3 and P_4 will receive both messages in the same order. Reliability in Isis means that either *all* or *no* surviving members of a group will receive a message, even in the face of processor failures. Because these semantics are hard to implement efficiently, Isis also provides primitives that give weaker semantics, but better performance. It is up to the programmer to decide which primitive is required.

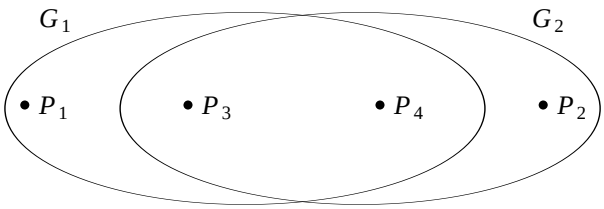


Fig. 3.2. Total ordering with overlapping groups. P_1 belongs to group G_1. P_2 belongs to group G_2. P_3 and P_4 are member of both groups.

3.2. Design Choices

Figure 3.3 lists the design issues and the choices we made. We will discuss each one in turn.

Addressing

Groups are addressed by a single address, called a *port*. A port is a large random number. By using a single address per group, a process can send a message to the group without knowing which and how many processes are members of the group.

Addressing groups with ports fits with Amoeba's client/server model. Services in Amoeba are also addressed by ports. When a service is started, it generates a new port and registers the port with the directory service. A client can look up the port using the directory service and asks its own kernel to send a message to the given port. The kernel maps the port onto a network address (a FLIP address). If multiple servers listen to the same port, only one (arbitrary) server will get the message. Thus, in Amoeba, processes and groups are addressed in a uniform way.

Issue	Choice
Addressing	Group identifier (port)
Reliability	Reliable communication; fault tolerance if specified
Ordering	Total ordering per group
Delivery semantics	All or none
Response semantics	None (RPC is available)
Group structure	Closed and dynamic

Fig. 3.3. Important design issues of Fig. 3.1 and the choices made in Amoeba.

Reliability

The group primitives provide by default reliable communication in the presence of communication failures. On the user's request, the group primitives can also recover from processor failures. We decided to make this an option, because providing these semantics is expensive and many applications do not need to recover from processor failures. Stronger semantics, like recovery from Byzantine failures (i.e., processors sending malicious or contradictory messages) and network partitions, are not supported by the group primitives. Applications requiring these semantics have to implement them explicitly. For example, the directory service discussed in Chapter 5 implements recovery from network partitions.

Although FLIP supports unreliable group communication, we decided to make only reliable group communication available to the programmer. This has the potential disadvantage that some users pay in performance for semantics that they do not need. It has the advantage, however, that the kernel only has to support one primitive, which

simplifies the implementation and makes higher level software more uniform. For the same reason Amoeba also supports only one primitive for point-to-point communication: RPC.

Ordering

The group primitives guarantee a total ordering per group. Many distributed applications are easy to implement with a total ordering and we have a simple and efficient protocol for doing reliable totally-ordered group communication (as we will see in the coming sections). There are three key ideas that make our approach feasible. First, to guarantee a total ordering the protocol uses a central machine per group, called the *sequencer*. (If the sequencer crashes, the remaining group members elect a new one.) Second, the protocol is based on a *negative acknowledgement* scheme. In a negative acknowledgement scheme, a process does not send an acknowledgement as soon as it receives a message. Instead, it sends a negative acknowledgement as soon as it discovers that it has missed a message. Third, acknowledgements are piggybacked on regular data messages to further reduce the number of protocol messages. These ideas are well known techniques. Chang and Maxemchuck, for example, discuss a protocol similar to ours that also combines these three ideas [Chang and Maxemchuk 1984].

Although at first sight it may seem strange to use a *centralized* sequencer in a *distributed system*, this decision is attractive. First, distributed protocols for total ordering are in general more complex and perform less well. Second, today's computers are very reliable and it is therefore unlikely that the sequencer will crash. The major disadvantage of having a sequencer is that the protocol does not scale to very large groups. In practice, however, this drawback is minor. The sequencer totally orders messages for a single group, not for the whole system. Furthermore, the sequencer performs a simple and computationally unintensive task and can therefore process many hundreds of messages per second.

There are two reasons for using a negative acknowledgement scheme. First, in a *positive acknowledgement scheme*, a process sends an acknowledgement back to the sender as soon as it receives the message. This works fine for point-to-point messages, but not for broadcast messages. If in a group of 256 processes, a process sends a broadcast message to the group, all 255 acknowledgements will be received by the sender at approximately the same time. As network interfaces can only buffer a fixed number of messages, a number of the acknowledgements will be lost, leading to unnecessary timeouts and retransmissions of the original message. Second, today's networks are very reliable and network packets are delivered with a very high probability. Thus not sending acknowledgements at all, but piggybacking them on regular data messages is feasible. Another alternative would be to use a positive acknowledgement scheme, but force the receivers to wait some ''random'' time before sending an acknowledgement [Danzig 1989]. This approach is attractive in unreliable networks, but it causes far more acknowledgements to be sent than with a negative acknowledgement scheme.

Delivery Semantics and Response Semantics

Per default, the group communication primitives deliver a message to all destinations, even in the face of communication failures. On the user request, the primitives can also guarantee ‘‘all-or-none’’ delivery in the face of processor failures. The protocols for providing these semantics are more expensive, and hence we decided to make it an option. User can trade performance for fault tolerance.

When a member receives a broadcast message, there is no group primitive available to send a reply. For the request/response type of communication RPC is available.

Group Structure

Unlike many other systems, we have chosen to use closed groups. A process that is not a member and that wishes to communicate with a group can perform an RPC with one of the members (or it can join the group). One reason for doing so is that a client need not be aware whether a service consists of multiple servers which perhaps broadcast messages to communicate with one another, or a single server. Also, a service should not have to know whether the client consists of a single process or a group of processes. This design decision is in the spirit of the client-server paradigm: a client knows what operations are allowed, but should not know how these operations are implemented by the service.

A second reason for closed groups is that it makes an efficient implementation of totally-ordered reliable broadcast possible. To implement the protocol, state is maintained for each member. If all processes can send messages to a group, they all have to keep state information about the groups that they are communicating with. Furthermore, the members also have to keep state for all the processes that are communicating with the groups. To make it possible to control the amount of state needed to implement the protocol, we decided on closed groups.

In Isis, this problem is solved in a different way. Isis presents the user with open groups, but implements it using closed groups. When a process wants to communicate with a group, the system either performs a join or an RPC with one of the members. In the latter case, the member broadcasts the message to the whole group. Thus, although the user has the illusion of open groups, the current implementation of Isis uses only closed groups.

A third reason for closed groups is that they are as useful as open groups. Just like in Isis, one can simulate an open group in Amoeba. A process performs an RPC with one of the members of the group. If a member receives an RPC, it broadcasts the request to the rest of the group. Compared to real open groups, the cost is that a request goes twice over the network instead of once.

Figure 3.4 shows a very small Amoeba system, with 12 processes and 3 groups, and how they interact. The parallel application replicates shared data using group communication to reduce access time. If the application wants to store an object with the directory service, it uses RPC to communicate with it. One of the directory servers will get the request. The directory server uses group communication to achieve fault toler-

ance and high availability. To store results on disk, a directory server communicates with the disk service, using RPC. The disk service may again use group communication internally for availability and fault tolerance (currently not done). Each application or service may be built out of one process or a group of processes which communicate with other services using RPC.

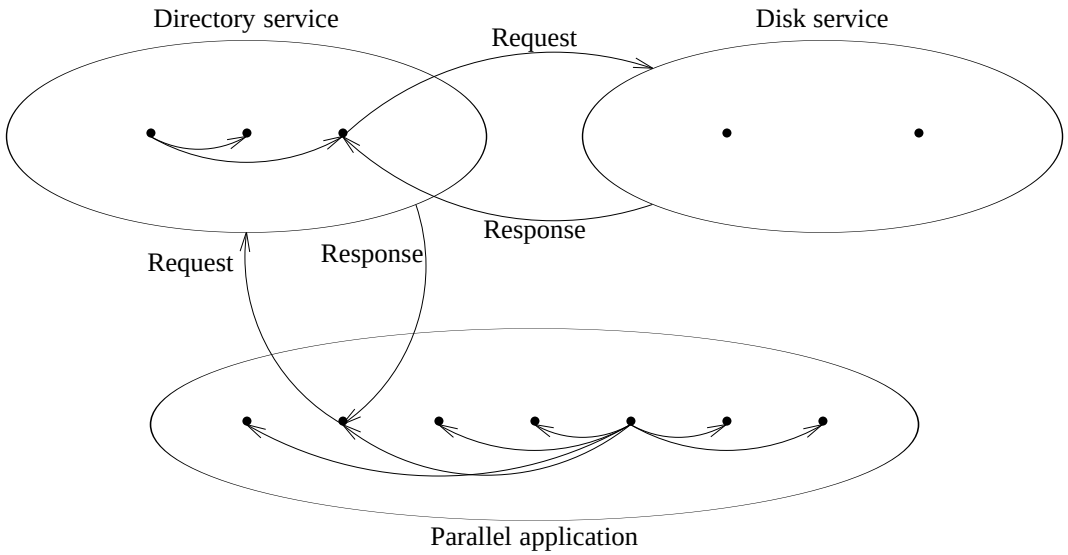


Fig. 3.4. An example Amoeba system with processes, groups, and their interaction. A request/response pair makes up one RPC. The request is sent to a port identifying a service and one of the servers will take the request; the corresponding response is sent back to the process doing the RPC.

3.3. Group Primitives in Amoeba

The primitives to manage groups and to communicate within a group are listed in Figure 3.5. We will discuss each primitive in turn.

A group is created by calling *CreateGroup*. The creator of the group is automatically member of the group. The first parameter is a port identifying the group. The second parameter is the number of member-crashes the group must be able to survive (0 if no fault tolerance is required). This is called the *resilience degree* of a group. The other parameters of *CreateGroup* specify information that simplify the implementation: the maximum number of members, the number of buffers that the protocol can use, and the maximum message size. Using this information, the kernel allocates memory for buffering messages and for member information. (Although these parameters could easily be replaced by default values, we decided against this for the sake of flexibility.) If not enough memory is available, *CreateGroup* fails. Otherwise, it succeeds and returns a small integer, called a group descriptor, *gd*, which is used to identify the group in subsequent group calls.

Once a group with port *p* has been created, other processes can become members

of it by calling *JoinGroup* with the port *p*. (The port is part of the Amoeba header *hdr*.) Only processes that know port *p* can join the group. When a message is sent to a group, only the group members receive the message. Like *CreateGroup*, *JoinGroup* returns a group descriptor for use in subsequent group calls. In addition to adding a process to a group, *JoinGroup* delivers a small message, *hdr*, to all other members. In this way, the other members are told that a new member has joined the group.

Function(parameters) → result	**Description**
CreateGroup(port, resilience, max_group, nr_buf, max_msg) → gd	Create a group. A process specifies how many member failures must be tolerated without loss of any message.
JoinGroup(hdr) → gd	Join a specified group.
LeaveGroup(gd, hdr)	Leave a group. The last member leaving causes the group to vanish.
SendToGroup(gd, hdr, buf, bufsize)	Atomically send a message to all the members of the group. All messages are totally-ordered.
ReceiveFromGroup(gd, &hdr, &buf, bufsize, &more) → size	Block until a message arrives. *More* tells if the system has buffered any other messages.
ResetGroup(gd, hdr, nr_members) → group_size	Recover from processor failure. If the newly reset group has at least *nr_member* members, it succeeds.
GetInfoGroup(gd, &state)	Return state information about the group, such as the number of group members and the caller's member id.
ForwardRequest(gd, member_id)	Forward a request for the group to another group member.

Fig. 3.5. Primitives to manage a group and to communicate within a group. A message consists of a header (a small message) and a buffer (a linear array of bytes). The header contains the port of a group. An output parameter is marked with "&".

Once a process is a member of a group, it can leave the group by calling *LeaveGroup*. Once a member has left the group, it does not receive subsequent broadcasts. In addition to causing the process to leave the group, *LeaveGroup* delivers *hdr* to all other members. In this way, the other members are told that a member has left. The member receives its own message, so that it can check whether it has processed all messages up to its leave message. The last member calling *LeaveGroup* automatically causes the group to vanish.

To broadcast a message, a process calls *SendToGroup*. This primitive guarantees that *hdr* and *buf* will be delivered to all members, even in the face of unreliable communication and finite buffers. Furthermore, when the *resilience degree* of the group is *r*, the protocol guarantees that even in the event of a simultaneous crash of up to *r* members, it will either deliver the message to all remaining members or to none. (This property does not contradict the result by Fischer, Lynch, and Paterson [Fischer et al. 1985], as our algorithms are based on timeouts.) Choosing a large value for *r* provides a high degree of fault tolerance, but this is paid for in performance. *SendToGroup* blocks until $r+1$ members have received the message. The tradeoff chosen is up to the user.

In addition to reliability, the protocol guarantees that messages are delivered in the same order to all members. Thus, if two members (on two different machines), simultaneously broadcast two messages, *A* and *B*, the protocol guarantees that either

1. All members receive *A* first and then *B*, or
2. All members receive *B* first and then *A*.

Random mixtures, where some members get *A* first and others get *B* first are guaranteed not to occur.

To receive a broadcast, a member calls *ReceiveFromGroup*; it blocks until the next message in the total order arrives. If a broadcast arrives and no such primitive is outstanding, the message is buffered. When the member finally does a *ReceiveFromGroup*, it will get the next one in sequence. How this is implemented will be described below. The *more* flag is used to indicate to the caller that one or more broadcasts have been buffered and can be fetched using *ReceiveFromGroup*. If a member never calls *ReceiveFromGroup*, the group may block (no more messages can be sent to the group), because it may run out of buffers. Messages are never discarded until received by all members.

ResetGroup allows recovery from member crashes. If one of the members (or its kernel) is unreachable, it is deemed to have crashed and the protocol enters a recovery mode. In this mode, it only accepts messages needed to run the recovery protocol and all outstanding *ReceiveFromGroup* calls return an error value that indicates a member crash. Any member can now call *ResetGroup* to transform the group into a new group that contains as many surviving members as possible. The second parameter is the

number of members that the new group must contain as a minimum. When *ResetGroup* succeeds, it returns the group size of the new group. In addition to recovering from crashes, *ResetGroup* delivers *hdr* to all new members. It may happen that multiple members initiate a recovery at the same moment. The new group is built only once, however, and consists of all the members that can communicate with each other. The *hdr* is also delivered only once.

The way recovery is done is based on the design principle that policy and mechanism should be separated. In many systems that deal with fault tolerance, recovery from processor crashes is completely invisible to the user application. We decided not to do this. A parallel application that multiplies two matrices, for example, may want to continue even if only one processor is left. A banking system may require, however, that at least half the group is alive. In our system, the user is able to decide on the policy. The group primitives provide only the mechanism.

GetInfoGroup allows a group member to obtain information about the group from its kernel. The call returns information such as the number of members in the group and the caller's member id. Each group member has a unique number.

The final primitive, *ForwardRequest*, integrates RPC with group communication. When a client does an RPC to a service, the client has no idea which server will get the request; it goes to one of the servers, effectively at random. If the server that gets the request is not able to serve the request (e.g., because it does not have the data requested), it can forward the request to another server in the group (*member_id* specifies the server). The forwarding occurs transparently to the client. The client cannot even tell that the service is provided by multiple servers.

To summarize, the group primitives provide an abstraction that enables programmers to design applications consisting of one or more processes running on different machines. It is a simple, but powerful, abstraction. All members of a single group see all events concerning this group in the same order. Even the events of a new member joining the group, a member leaving the group, and recovery from a crashed member are totally-ordered. If, for example, one process calls *JoinGroup* and a member calls *SendToGroup*, either all members first receive the join and then the broadcast or all members first receive the broadcast and then the join. In the first case the process that called *JoinGroup* will also receive the broadcast message. In the second case, it will not receive the broadcast message. A mixture of these two orderings is guaranteed not to happen. This property makes reasoning about a distributed application much easier. Furthermore, the group interface gives support for building fault-tolerant applications by choosing an appropriate resilience degree.

3.4. The Broadcast Protocol

The protocol to be described runs inside the kernel and is accessible through the primitives described in the previous section. It assumes that *unreliable* message passing between entities is possible; fragmentation, reassembly, and routing of messages are done at lower layers in the kernel (see Chapter 2). The protocol performs best on a net-

work that supports hardware multicast. Lower layers, however, treat multicast as an optimization of sending point-to-point messages; if multicast is not available, then point-to-point communication will be used. Even if only point-to-point communication is available, the protocol is in most cases still more efficient than performing n RPCs. (In a mesh interconnection network, for example, the routing protocol will ensure that the delay of sending n messages is only in the order of $\log_2 n$.)

Each kernel running a group member maintains information about the group (or groups) to which the member belongs. It stores, for example, the size of the group and information about the other members in the group. Any group member can, at any instant, decide to broadcast a message to its group. It is the job of the kernel and the protocol to achieve reliable broadcasting, even in the face of unreliable communication, lost packets, finite buffers, and node failures.

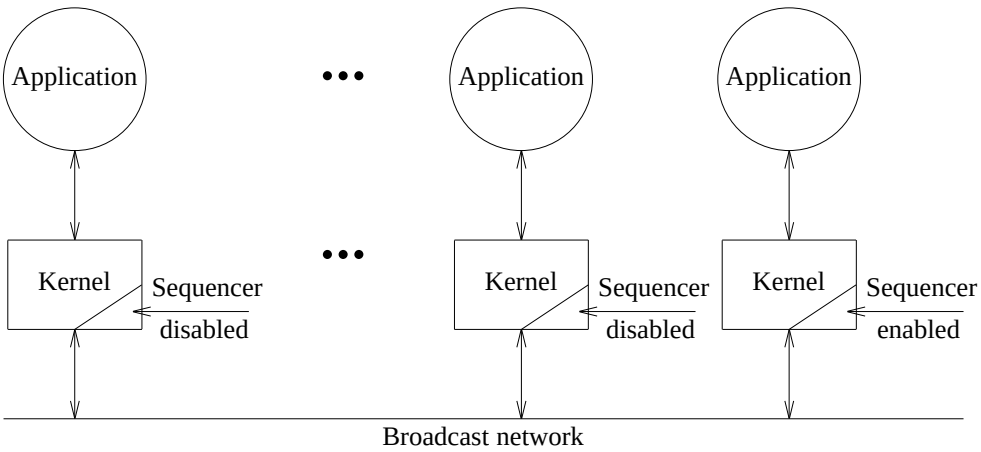


Fig. 3.6. System structure. Each node runs a kernel and a user application. Each kernel is capable of being sequencer, but, at any instant, only one of them functions as sequencer. If the sequencer crashes, the remaining nodes can elect a new one.

Without loss of generality, we assume for the rest of this section that the system contains one group, with each member running on a separate processor (see Fig. 3.6). When the application starts up, the machine on which the group is created is made the *sequencer*. If the sequencer machine subsequently crashes, the remaining members elect a new one (this procedure is described in Section 3.4.3). The sequencer machine is in no way special—it has the same hardware and runs the same kernel as all the other machines. The only difference is that it is currently performing the sequencer function in addition to its normal tasks.

3.4.1. Basic Protocol

A brief description of the protocol is as follows (a complete description is given in the next section). When a group member calls *SendToGroup* to send a message, *M*, it hands the message to its kernel and blocks. The kernel encapsulates *M* in an ordinary point-to-point message and sends it to the sequencer. When the sequencer receives *M*, it allocates the next sequence number, *s*, and broadcasts a message containing *M* and *s*. Thus all broadcasts are issued from the same node, the sequencer. Assuming that no messages are lost, it is easy to see that if two members concurrently want to broadcast, one of them will reach the sequencer first and its message will be broadcast first. Only when that broadcast has been completed will the other broadcast be started. Thus, the sequencer provides a total time ordering. In this way, we can easily guarantee the indivisibility of broadcasting per group.

When the kernel that sent *M*, receives the message from the network, it knows that its broadcast has been successful. It unblocks the member that called *SendToGroup*.

Although most modern networks are highly reliable, they are not perfect, so the protocol must deal with errors. Suppose some node misses a broadcast packet, either due to a communication failure or lack of buffer space when the packet arrived. When the following broadcast message eventually arrives, the kernel will immediately notice a gap in the sequence numbers. If it was expecting s next, and it receives $s + 1$ instead, it knows it has missed one.

The kernel then sends a special point-to-point message to the sequencer asking it for a copy of the missing message (or messages, if several have been missed). To be able to reply to such requests, the sequencer stores broadcast messages in the *history buffer*. The sequencer sends the missing messages to the process requesting them as point-to-point messages. The other kernels also keep a history buffer, to be able to recover from sequencer failures and to buffer messages when there is no outstanding *ReceiveFromGroup* call.

As a practical matter, a kernel has only a finite amount of space in its history buffer, so it cannot store broadcast messages indefinitely. However, if it could somehow discover that all members have received broadcasts up to and including *m*, it could then purge the broadcast messages up to *m* from the history buffer.

The protocol has several ways of letting a kernel discover this information. For one thing, each point-to-point message to the sequencer (e.g., a broadcast request), contains, in a header field, the sequence number of the last broadcast received by the sender of the message (i.e., a piggybacked acknowledgement). This information is also included in the message from the sequencer to the other kernels. In this way, a kernel can maintain a table, indexed by member number, showing that member *i* has received all broadcast messages up to T_i (and perhaps more). At any instant, a kernel can compute the lowest value in this table, and safely discard all broadcast messages up to and including that value. For example, if the values of this table are 8, 7, 9, 8, 6, and 8, the

kernel knows that everyone has received broadcasts 0 through 6, so they can be safely deleted from the history buffer.

If a node does not do any broadcasting for a while, the sequencer will not have an up-to-date idea of which broadcasts it has received. To provide this information, nodes that have been quiet for a certain interval send the sequencer a special message acknowledging all received broadcasts. The sequencer can also request this information when it runs out of space in its history buffer.

PB Method and BB Method

There is a subtle design point in the protocol; there are actually two ways to do a broadcast. In the method we have just described, the sender sends a point-to-point message to the sequencer, which then broadcasts it. We call this the *PB method* (Point-to-point followed by a Broadcast). In the *BB method*, the sender broadcasts the message. When the sequencer sees the broadcast, it broadcasts a special *accept* message containing the newly assigned sequence number. A broadcast message is only ''official'' when the *accept* message has been sent.

These methods are logically equivalent, but they have different performance characteristics. In the PB method, each message appears on the network twice: once to the sequencer and once from the sequencer. Thus a message of length n bytes consumes $2n$ bytes of network bandwidth. However, only the second message is broadcast, so each user machine is interrupted only once (for the second message).

In the BB method, the full message appears only once on the network, plus a very short *accept* message from the sequencer. Thus, only about n bytes of bandwidth are consumed. On the other hand, every machine is interrupted twice, once for the message and once for the *accept*. Thus the PB method wastes bandwidth to reduce the number of interrupts and the BB method minimizes bandwidth usage at the cost of more interrupts. The protocol switches dynamically between the PB method and BB method depending on the message size.

Processor Failures

The protocol described so far recovers from communication failures, but does not guarantee that all surviving members receive all messages that have been sent before a member crashed. For example, suppose a process sends a message to the sequencer, which broadcasts it. The sender receives the broadcast and delivers it to the application, which interacts with the external world. Now assume all other processes miss the broadcast, and the sender and sequencer both crash. Now, the effects of the message are visible but none of the other members will receive it. This is a dangerous situation that can lead to all kinds of disasters, because the ''all-or-none'' semantics have been violated.

To avoid this situation, *CreateGroup* has a parameter r, the *resilience degree* that specifies the resiliency. This means that the *SendToGroup* primitive does not return

control to the application until the kernel knows that at least r other kernels have received the message. To achieve this, a kernel sends the message to the sequencer point-to-point (PB method) or broadcasts the message to the group (BB method). The sequencer allocates the next sequence number, but does not officially accept the message yet. Instead, it buffers the message and broadcasts the message and sequence number as a request for broadcasting to the group. On receiving such a request with a sequence number, kernels buffer the message in their history and the r lowest-numbered send acknowledgement messages to the sequencer. (Any r members besides the sending kernel would be fine, but to simplify the implementation we pick the r lowest-numbered.) After receiving these acknowledgements, the sequencer broadcasts the *accept* message. Only after receiving the *accept* message can members other than the sequencer deliver the message to the application. That way, no matter which r machines crash, there will be at least one left containing the full history, so everyone else can be brought up-to-date during the recovery. Thus, an increase in fault tolerance is paid for by a decrease in performance. The tradeoff chosen is up to the user.

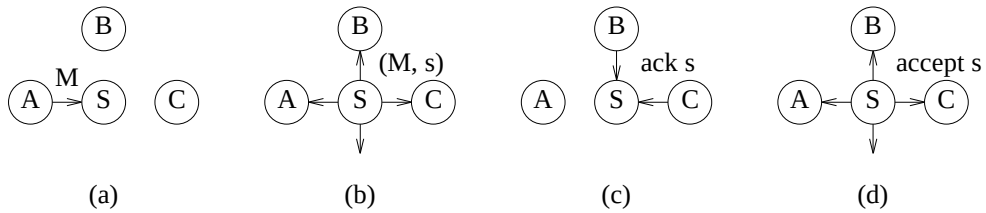


Fig. 3.7. PB protocol for $r = 2$.

The PB and BB method for $r = 2$ are illustrated in Figure 3.7 and in Figure 3.8. In Figure 3.7(a), machine A sends a message, *M*, to the sequencer, where it is assigned sequence number *s*. The message (containing the sequence number *s*) is now broadcast to all members and buffered (Fig. 3.7(b)). The r lowest-numbered kernels (say, machines *B* and *C* in Figure 3.7(c)) send an acknowledgement back to the sequencer to confirm that they have received and buffered the message with sequence number *s*. After receiving the r acknowledgements, the message with sequence number *s* is officially accepted and the sequencer broadcasts a short accept message with sequence number *s* (Fig. 3.7(d)). When a machine receives the accept message, it can deliver the message to the application.

The BB method for $r = 2$ is very similar to the PB method (see Fig. 3.8); only the events in (a) and (b) differ. Instead of sending a point-to-point message to the sequencer, machine *A* broadcasts the message to the whole group (Fig. 3.8(a)). When the sequencer receives this message, it allocates the next sequence number and broadcasts it (Fig. 3.8(b)). From then on the BB method is identical to the PB method. Thus, the important difference between the PB and BB method is that in the PB method the message *M* goes over the network twice, while in the BB method it goes only over the network once.

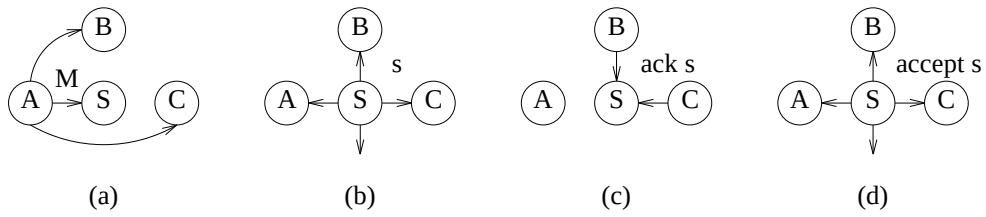


Fig. 3.8. BB method for $r = 2$.

At first sight, it may seem that a more efficient protocol can be used for $r > 0$. Namely, a kernel broadcasts the message to the group. On receiving a broadcast, r lowest-numbered kernels immediately buffer the message in their history and send acknowledgement messages to the sequencer, instead of waiting until the sequencer announces a sequence number for the broadcast request. After receiving the acknowledgements, the sequencer broadcasts the *accept* message. This protocol would save one broadcast message (the message from the sequencer announcing the sequence number for the broadcast request).

This protocol is, however, incorrect. Assume that r kernels have buffered a number of messages and have sent an acknowledgement for each of them and that all *accept* messages from the sequencer are lost. The following could now happen. The sequencer delivers the message to its application, and then the sequencer (and application) crashes. During recovery, the remaining members would have no way of deciding how the buffered messages should be ordered, violating the rule that all messages should be delivered in the same order. Even if they decide among themselves on some order, they could potentially deliver the messages in a different order than the sequencer did and still violate the rule. To avoid this situation, the sequencer announces the sequence number for a message before r kernels send an acknowledgement.

It is interesting to see how Isis deals with this situation. In Isis it may happen that after a processor failure, messages on the remaining processors are delivered in a different order than was done on the failed processor. If an application requires stronger semantics, it is up to the programmer to call a primitive that blocks the application until it is known that all other kernels will deliver the message in the same order [Birman et al. 1990].

In summary, there are two methods of sending a reliable totally-ordered broadcast, PB and BB. The PB method and the BB method are logically equivalent but have different performance characteristics. (In Section 3.5 we will give a detailed comparison between the PB method and the BB method.) For $r > 0$, additional messages are needed to guarantee that broadcasts are delivered in the same order, even in the face of processor failures.

3.4.2. Protocol during Normal Operation

In this section, we will describe in detail how the sender, sequencer, and receivers behave during normal operation (no member failures).

Data Structures

Figure 3.9 shows the data structures used by the protocol. Each kernel keeps state information for each of its members. The information stored for each member consists of general information, membership information, and history information. The general information includes the port of the group to which the member belongs, the network address for the group, on what message size to switch from the PB method to the BB method, the current state of the protocol (e.g., receiving, sending, etc.), *r*, and the current incarnation number of the group. The parameter *r*, *g_resilience*, specifies how many concurrent failures the group must tolerate without losing any messages. It is specified when the group is created. The incarnation number of a group, *g_incarnation*, is incremented after recovery from a member failure. Each message sent is stamped with the current incarnation number and is only processed if it is equal to *g_incarnation*; otherwise it is discarded. If no member failures happen, then *g_incarnation* stays at 0.

The membership information consists of the list of members, the total number of members, the current sequencer, and the current coordinator (only used during recovery from member failures). Furthermore, the kernel stores the member identifier, *g_index*, for this member and its rank, *g_memrank*. The member id does not change during the time the application is member of a group. The rank is used to decide if a kernel should send an acknowledgement back when a broadcast request arrives and the *resilience degree* is higher than 0. The rank of a member can change during its lifetime. If, for example, a group consists of three members, numbered 0, 1, and 2 respectively, the ranks for these members are initially equal to the member ids. If now, for example, member 1 leaves, then the rank of member 2 changes to 1. In this way, it is easy for each member to decide whether it belongs to the *r* lowest members. Since every member is guaranteed to receive all join and leave events in the same order, this information will always be consistent.

Each kernel keeps in the structure *struct member* for each member *m* the sequence number expected by *m*, *m_expect*, the last message number used by *m*, *m_messid*, and *m*'s network address. The sequencer uses the *m_expect* fields to determine which messages can be safely removed from the history buffer. The message number, *m_messid*, gives the last message number received from a member and is used to detect duplicates generated by timeouts. The *retry counter*, *m_retrial*, is used to determine when a member has failed. If a kernel is waiting for a reply from another kernel and it does not receive the message within a certain time frame, the counter is decremented. If it reaches zero, the kernel is considered to be down. The other fields are used only during recovery (see Section 3.4.3).

The history information consists of a circular buffer with a number of indices tel-

```
/* On each machine, a struct for each group is maintained. */
struct group {                          /* general info */
   port g_port;                         /* a port identifies a group */
   adr_t g_addr;                        /* group network address */
   int g_large;                         /* threshold between PB and BB */
   long g_flags;                        /* protocol state: FL_RECOVER, … */
   int g_resilience;                    /* resilience degree */
   short g_incarnation;                 /* incarnation number */

   /* Member information */
   int g_total;                         /* group size */
   struct member *g_member;             /* list of members */
   struct member *g_me;                 /* pointer to my member struct */
   struct member *g_seq;                /* pointer to sequencer struct */
   struct member *g_coord;              /* pointer to coordinator struct */
   int g_index;                         /* my index */
   int g_memrank;                       /* member rank */

   /* History information in circular buffer */
   hist_p g_history;                    /* history of bcast messages */
   int g_nhist;                         /* size of history */
   int g_nextseqno;                     /* next sequence number */
   int g_minhistory;                    /* lowest entry used */
   int g_nexthistory;                   /* next entry to store message */
};

/* On each machine, a struct for each member is maintained. */
struct member {
   adr_t m_member;                      /* member network address */
   int m_expect;                        /* seqno expected by member */
   int m_messid;                        /* next message id to use */
   int m_retrial;                       /* retry counter */
   int m_vote;                          /* vote for this member (recovery) */
   int m_replied;                       /* has the member replied? */
};

/* Broadcast protocol header */
struct bc_hdr {
   short b_type;                        /* type: BC_BCASTREQ, … */
   short b_incarnation;                 /* incarnation number */
   int b_seqno;                         /* global sequence number */
   int b_messnr;                        /* message identifier */
   int b_expect;                        /* seq number expected by application */
   int b_cpu;                           /* member identifier */
};
```

Fig. 3.9. Declarations.

ling how big the buffer is and which part of the buffer is filled. Each buffer in the history contains a complete message including the user-supplied data (the upper bound on the size of a message is passed as an argument when the group is created). The history buffer consists of three parts (see Fig. 3.10). The circular part between *g_nexthistory*

and *g_minhistory* consists of free buffers. The circular part between *g_minhistory* and *g_nextseqno* contains messages that have been sequenced, but are buffered to be delivered to the user application or so that the kernel can respond to retransmission requests. The circular part between *g_nextseqno* and *g_nexthistory* is used by the sequencer to buffer messages for which it does not know yet if all members have room to store them in their history buffers. After synchronizing with the other history buffers, the sequencer will take the buffered requests and process them (broadcast an official accept). The ordinary kernels use the third part to buffer messages that are received out-of-sequence until the missing messages are received. Buffering messages in the third part of the history buffer avoids unnecessary retransmission of those messages later on.

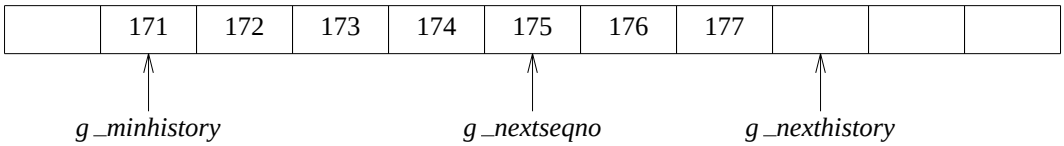


Fig. 3.10. The history buffer has three parts: 1) free buffers; 2) messages that have been sequenced but are buffered until they can be delivered to the user or are buffered for retransmissions; 3) messages that are buffered because the sequencer does not know if the other kernels have room in their history to store the message. Ordinary members use the third part to buffer messages that arrive out-of-sequence.

Each message sent by the protocol contains a fixed size protocol header, consisting of six fields. The *b_type* field indicates the kind of a message (see Fig. 3.11). The *b_incarnation* gives the incarnation of the member that is sending the message. The *b_seqno* field is used by the sequencer to sequence broadcasts. The *b_messnr* and *b_cpu* together uniquely identify a message. They are used to detect duplicates. The *b_expect* field is used to piggyback the acknowledgement for the last broadcast delivered so far. When receiving a message, a kernel updates *m_expect* with *b_expect*. If the sequencer knows that *m_expect* for each member is larger than *g_minhistory*, it can increase *g_minhistory* and thereby free history buffers.

Receiving a Message

Let us now look at the protocols for receiving and sending messages reliably. When a member wants to receive a broadcast, it invokes its local kernel by calling *ReceiveFromGroup* and passes it pointers to a header and a buffer (see Fig. 3.12). The kernel performs parameter checks to see if the call is legal. If not, it returns an error. If it is a legal call, the kernel checks the history to see if there are any messages buffered that can be delivered. If there are none, the thread calling *ReceiveFromGroup* is blocked until a message can be delivered. If a deliverable message is present, it is

Type	From	To	Description
BC_JOINREQ	Member	Sequencer	Request to join the group
BC_JOIN	Sequencer	Group	Accept join message
BC_LEAVEREQ	Member	Sequencer	Request to leave the group
BC_LEAVE	Sequencer	Group	Accept leave message
BC_BCASTREQ	Member	Sequencer or group	Request to broadcast
BC_BCAST	Sequencer	Group	Accept broadcast message
BC_ACK	Member	Sequencer	Message is received (if r > 0)
BC_RETRANS	Member	Sequencer	Request asking for missed message
BC_SYNC	Sequencer	Group	Synchronize histories
BC_STATE	Member	Sequencer	Tell next expected sequence number
BC_ALIVEREQ	Member	Member	Check if destination is alive
BC_ALIVE	Member	Member	Acknowledgement of BC_ALIVEREQ
BC_REFORMREQ	Coordinator	Group	Request for entering recovery mode
BC_VOTE	Member	Coordinator	Vote for new sequencer
BC_RESULT	Coordinator	Group	Result of the voting phase
BC_RESULTACK	Member	Coordinator	Ack for receiving the final vote.

Fig. 3.11. Possible values for *b_type* and their function.

copied from the history buffer into the buffer supplied by the application. The number of bytes received is returned to the application process.

Each time a broadcast comes in from the sequencer, the kernel checks to see if there is a thread waiting to receive a message. If so, it unblocks the thread and gives it the message.

Sending a Message

When a member wants to do a broadcast, it invokes its local kernel by calling *SendToGroup* and passes a header and data. The kernel then executes the algorithm given in Figure 3.13. It performs a number of parameter checks to see if the process making the call is a member of the group and if the message can be sent. If all checks succeed, it builds a message consisting of the protocol header, and the user-supplied

```
long ReceiveFromGroup(gd, hdr, buf, cnt, more)
   int gd;                                          /* group descriptor */
   struct header *hdr;                              /* pointer to Amoeba header buffer */
   char *buf;                                       /* pointer to empty data buffer */
   long cnt;                                        /* size of data buffer */
   int *more;                                       /* pointer to more flag */
{
   struct group *g;                                 /* pointer to group structure */
   struct hist *h;                                  /* pointer into the history */
   long rs;                                         /* size of the message to be received */

   if (gd < 0 || gd >= grp_maxgrp) return(BC_ILLARG);/* legal call? */
   g = groupindex[gd];                              /* set group pointer */
   if (!legal_port(&g–>g_port, &hdr–>h_port))       /* legal port? */
        return(BC_BADPORT);
   g–>g_flags |= FL_RECEIVING;                      /* start receiving */
   while(!HST_IN(g, g–>g_me–>m_expect)) {           /* is there a buffered message? */
        if (g–>g_flags & FL_RESET) {                /* don't block during recovery */
                g–>g_flags &= ~FL_RECEIVING;        /*  switch flag off */
                return(BC_ABORT);                   /* return failure */
        }
        block(g);                                   /* no, wait until one comes in */
   }
   g–>g_flags &= ~FL_RECEIVING;                     /* switch flag off */
   h = &g–>g_history[HST_MOD(g, g–>g_me–>m_expect)];/* get it */
   bcopy(h–>h_data, hdr, sizeof(struct header));    /* copy to user space */
   rs = MIN(cnt, h–>h_size – sizeof(struct header)); /* MIN(a,b) = (a < b ? a : b) */
   bcopy(h–>h_data + sizeof(struct header), buf, rs); /* copy to buf */
   g–>g_me–>m_expect++;                             /* message is now delivered */
   *more = g–>g_nextseqno – g–>g_me–>m_expect;       /* number of buffered messages */
   return(rs);                                      /* return number of bytes received */
}
```

Fig. 3.12. Algorithm used by kernel to receive a reliable broadcast.

header and data. The kernel sets *b_type* to BC_BCASTREQ, *b_cpu* to the member's id, *b_incarno* to the current incarnation, and *b_expect* to the value of the member's *m_expect*. Depending on the size of the data and *g_large*, the kernel sends the message point-to-point to the sequencer (PB method) or broadcasts the message to the group (BB method). The default value for *g_large* is equal to the maximum size of a network packet. Thus, small messages that fit in one network packet are sent using the PB method and larger messages are sent using the BB method. The programmer may override the default value. Once the message is sent, the kernel blocks the member until the message comes back from the sequencer or until it receives a timeout. If, after *n* retries, no message is received from the sequencer, the member assumes that the sequencer has crashed and enters recovery mode (see Section 3.4.3). During recovery it is determined if the send failed or succeeded.

```
int SendToGroup(gd, hdr, data, size)
  int gd;                                          /* group descriptor */
  struct header *hdr;                              /* pointer to header to be sent */
  char *data;                                      /* pointer to data to be sent */
  long size;                                       /* size of data */
{
  struct group *g;                                 /* pointer to group structure */
  long messid;                                     /* messid for the message to be sent */
  struct pkt *msg;                                 /* the message to be sent */

  if (gd < 0 || gd >= grp_maxgrp) return(BC_ILLARG);/* legal call? */
  g = groupindex[gd];                              /* set pointer */
  if (!legal_port(&g–>g_port, &hdr–>h_port))       /* legal port? */
       return(BC_BADPORT);
  if (g–>g_flags & FL_RESET) return(BC_ABORT);/* don't send during recovery */

  /* Checks done, start send. */
  g–>g_seq–>m_retrial = g–>g_maxretrial;           /* set maximum number of retries */
  messid = g–>g_me–>m_messid+1;                    /* set message identifier */
  g–>g_flags |= FL_SENDING;                        /* start sending message */
  do {
       setmsg(&msg, BC_BCASTREQ, −1, messid, g–>g_index, g–>g_me–>m_expect,
            g–>g_incarnation, hdr, (long) data, size);  /* build message */
       set_timer(g, msg, settimer);                /* set timer */
       if (g–>g_seq == g–>g_me) sendlocal(g, msg); /* am I the sequencer? */
       else if (size >= g–>g_large) multicast(&g–>g_addr, msg);/* use BB method? */
       else unicast(&g–>g_seq–>m_addr, msg);       /* use PB method */
       /* Block until broadcast succeeds, fails, or times out. */
       block(g);                                   /* suspend calling thread */
       if (g–>g_flags & FL_SENDING) {              /* timeout? */
            g–>g_seq–>m_retrial−−;                 /* decrease retry counter */
            if (g–>g_seq–>m_retrial <= 0) recover(g);/* did the sequencer crash? */
       }
  } while(g–>g_flags & FL_SENDING);                /* done? */
  return(g–>g_me–>messid ≥ messid ? BC_OK : BC_FAIL);/* return success or failure */
}
```

Fig. 3.13. Algorithm used by sending kernel to achieve reliable broadcast.

Protocol

Having looked at what the sender does to transmit a message to the sequencer for broadcast, let us now consider what a kernel does when the BC_BCASTREQ message comes in (see Fig. 3.14). If the message is sent using the BB method, then all members will receive the broadcast request (absent a network failure). If $b_seqno = 0$, the broadcast request is sent using the BB method and the ordinary members will buffer the request until the sequencer broadcasts an accept message. If $b_seqno > 0$, the sequencer has sequenced the message, but did not accept the message yet, because it is waiting for r acknowledgements. In this case, the members store the message in the third part of the history and send an acknowledgement (BC_ACK) if their rank is less

than or equal to *r*. This informs the sequencer that the message sent by *b_cpu* with message number *b_messnr* has been received. Once the sequencer has received *r* of these acknowledgements, it will accept the message officially and broadcast a short accept message (BC_BCAST without data).

When the sequencer receives the broadcast request, it updates its table with member information and it tries to free some buffers in its history using the piggybacked acknowledgements. Then, it checks if the message is a duplicate by examining *b_cpu* and *b_messnr*. If so, it informs the sender that the message already has been sent.

If the message is new and $r = 0$, the sequencer stores the message in its history and officially accepts the message (i.e., the sequencer changes the type of the message from BC_BCASTREQ to BC_BCAST). If $r > 0$, the sequencer stores the message in its history buffer, but it does not accept the message officially. Instead, it forwards the request with sequence number to the group and waits for *r* acknowledgements.

The history is processed each time a broadcast request is received, *r* acknowledgements for a buffered broadcast message have been received, or when the sequencer learns from a piggybacked acknowledgement that it can free buffers to accept a buffered request (a message stored in the third part of the history). *Processhist* accepts broadcast messages buffered in the history as long as the history does not fill up (see Fig. 3.15). It takes the next unprocessed message from the history buffer, broadcasts the message (PB method) or broadcasts an accept message (BB method) and increases *g_nextseqno*. At this point the message has officially been accepted.

If the message just accepted was sent by a member running on the sequencer kernel, the member can be unblocked and the *SendToGroup* returns successfully.

The protocol requires three algorithms to be executed. First, the sender must build a message and transmit it to the sequencer (PB) or broadcast it (BB). Second, the sequencer and members must process incoming BC_BCASTREQ messages. The sequencer broadcasts the messages with sequence number (PB) or broadcasts a short accept message (BB); the members buffer them until an official accept from the sequencer arrives (BB). Third and last, the members must handle arriving BC_BCAST messages from the sequencer. We have already described the first two steps; now let us look at the last one.

When a BC_BCAST arrives, the receiving kernel executes the procedure *broadcastreceive* (see Fig. 3.16). The kernel checks if the incoming message has user-supplied header and data. A BC_BCAST message with user-supplied header and data is a message sent following the PB method; otherwise the message is a short accept broadcast and the user-supplied header and data (received on a previous message) are buffered. If the sequence number is the one it expected, the message is stored in the history and processed by *processrec*. If the sequence number is not the expected one, the member has missed one or more broadcasts and asks the sequencer for retransmissions (BC_RETRANS). Out-of-sequence broadcasts are buffered in the history, but the

```
void bcastreq(g, bc, data, n)
  group_p g;                                        /* pointer to group structure */
  struct bc_hdr *bc;                                /* pointer to protocol header */
  char *data;                                       /* pointer do data in message */
  int n;                                            /* number of bytes in data */
{
  struct hist *hist;                                /* pointer in history */
  struct member *src = g->g_member + bc->b_cpu;     /* pointer to the sender */
  struct pkt *msg;                                  /* reply message */

  if (g->g_me != g->g_seq) {                        /* am I the sequencer? If no: */
      if(bc->b_seqno == 0) {
          /* A request without seqno has been received. This must be the BB method. */
          mem_buffer(g, src, bc, data, n);          /* buffer it */
      } else {
          /* A request with seqno; this must be for resilience > 0. Store the message
          * in the right place in the history and send an ack if my rank <= resilience. */
          store_in_history_and_send_ack(g, bc, data, n);
      }
      return;                                       /* done */
  }

  /* Yes, the sequencer. This must be the PB protocol. */
  if (src->m_expect < bc->b_expect)                 /* update member info? */
      src->m_expect = bc->b_expect;
  if (hst_free(g)) g->g_flags &= ~FL_SYNC;          /* is synchronization needed? */
  if (bc->b_messnr <= src->m_messid) {              /* old request? */
      /* Send sequence number back as an ack to the sender. */
      retrial(g, bc->b_cpu);
  } else if (HST_FULL(g)) {                         /* history full? */
      g->g_flags |= FL_SYNC;                        /* synchronize */
      synchronize(g);                               /* multicast a BC_SYNC messages */
  } else {                                          /* append message to history */
      src->m_messid = bc->b_messnr;                 /* remember messid */
      bc->b_seqno = g->g_nexthistory;               /* assign sequence number */
      if (g->g_resilience == 0) bc->b_type = BC_BCAST;/* accept request */
      /* Append to history and increase g_nexthistory. */
      hist = hst_append(g, bc, data, n);
      if (g->g_resilience > 0) {                    /* resilience degree > 0? */
          forward_msg_to_members(g, hist);          /* forward request with seqno */
      } else hist->h_accept = 1;                    /* message is accepted */
      processhist(g);                               /* accept the new broadcast */
  }
}
```

Fig. 3.14. Algorithm executed by all kernels (including sequencer) when a BC_BCASTREQ message arrives.

message is not processed, because the kernel is required to pass messages to the application in the correct order.

Processrec inspects the history buffer to see if the next expected message has

```
void processhist(g)
   struct group *g;                                  /* pointer to group structure */
{
   struct hist *hist;                                /* pointer into the history */
   struct member *src;                               /* pointer to the sender */
   struct *msg;                                      /* reply message */

   for(hist = &g–>g_history[HST_MOD(g, g–>g_nextseqno)];/* get first msg */
      g–>g_nextseqno < g–>g_nexthistory && hist->h_accept;/* process msg? */
      hist = &g–>g_history[HST_MOD(g, g–>g_nextseqno)]) {/* get next msg */
         if (!HST_SYNCHRONIZE(g)) {                  /* synchronize first? */
                  src = &g–>g_member[hist–>h_bc.b_cpu];  /* set the sender */
                  g–>g_nextseqno++;                  /* accept broadcast officially */
                  if (src != g–>g_seq && hist–>h_size >= g–>g_large) {/* BB method? */
                           /* Build accept message and multicast. */
                           buildmsg(&msg, 0, &hist–>h_bc, 0, 0);
                           multicast(msg, &g–>g_addr);
                  } else {                           /* PB method */
                           /* Build complete message and multicast it. */
                           buildmsg(&msg, 0, &hist–>h_bc, hist–>h_data,  hist–>h_size);
                           multicast(msg, &g–>g_addr);
                  }
                  if ((g–>g_flags & FL_SENDING) && hist–>h_bc.b_cpu == g–>g_index) {
                           /* Message was sent by a member running on the sequencer kernel. */
                           g–>g_flags &= ~FL_SENDING;    /* switch flag off */
                           unblock(g);                   /* unblock application */
                  }
         } else  {
                  g–>g_flags |= FL_SYNC;             /* synchronize */
                  synchronize(g);                    /* multicast BC_SYNC message */
                  break;                             /* stop processing */
         }
   }
}
```

Fig. 3.15. Algorithm executed by the sequencer to process the history.

been stored (see Fig. 3.17). If so, it increases *g_nextseqno* and updates its member information of the sender. If the sender of the message is a member at the receiving kernel, then the sending member is unblocked and the *SendToGroup* returns successfully.

The sequencer frees history buffers using the piggybacked acknowledgements contained in the messages from the members. The assumption is that a member will always send a message before the history fills up. This assumption need not be true, depending on the communication patterns of the applications. For example, a member may send a message that triggers another member to send messages. If this member misses the message, the system may very well become deadlocked. To prevent this from happening, each member kernel sends a BC_STATE message after receiving a certain number of messages without sending any. This is effectively a logical timer; a real timer could also be used, but this would be less efficient. The *b_expect* field in the

```
void broadcastreceive(g, bc, data, n)
   struct group *g;                                  /* pointer to group state */
   struct bc_hdr *bc;                                /* pointer to protocol header */
   char *data;                                       /* pointer to data */
   long n;                                           /* number of bytes in data */
{
   int received = 1;                                 /* is data received? */
   struct member *src;                               /* pointer to original sender */
   struct hist *hist;                                /* pointer into the history */

   src = g–>g_member + bc–>b_cpu;                    /* set source */
   /* If the PB method is used, the message contains the original data; otherwise
    * the message is only a short accept msg and the data should have been received
    * and is buffered. */
   if (n == 0) {                                     /* short accept msg? */
        /* Yes, the BB method or resilience degree > 0. */
        hist = &g–>g_history[HST_MOD(g, bc–>b_seqno)];
        if (g–>g_resilience > 0 && hist–>h_bc.m_messid == bc–>b_messnr) {
                /* The message is stored as BC_BCASTREQ in the history. */
                hist–>h_accept = 1;                  /* accept */
                return;                              /* done */
        }
        if (messbuffered(src, bc–>b_messnr))         /* is message buffered? */
                data = getmsg(src);                  /* yes, get it */
        else received = 0;                           /* not received the data yet */
   }
   if (g–>g_nextseqno == bc–>b_seqno && received) { /* accept it? */
        hist = hst_store(g, bc, data, n);            /* store new msg in history */
        hist->h_accept = 1;                          /* accept it */
        processrec(g);                               /* process history */
   } else if (g–>g_nextseqno < bc–>b_seqno ||
            (g–>g_nextseqno == bc–>b_seqno && !received)) {/* out of order? */
        if (received)  hst_store(g, bc, data, n);    /* yes, buffer it */
        ask_for_retransmission(g, g–>g_nextseqno);   /* ask for the missing msgs */
   }                                                 /* old message; discard */
}
```

Fig. 3.16. Algorithm for processing an incoming broadcast.

BC_STATE message informs the sequencer which messages have been delivered by the sender of the message.

If the sequencer runs out of history buffers and has not received enough BC_STATE messages to make the decision to free history buffers, it can explicitly ask members which messages they have received. It does so by sending a BC_SYNC message. Members respond to this message with a BC_STATE message.

```
void processrec(g)
   struct group *g;                                  /* pointer to group state */
{
   struct member *src;                               /* pointer to original sender */
   struct hist *hist;                                /* pointer into the history */

   for(hist = &g–>g_history[HST_MOD(g, g–>g_nextseqno)];/* set pointer into history */
      hist–>h_accept;                                /* is the message accepted? */
      hist = &g–>g_history[HST_MOD(g, g–>g_nextseqno)]) {/* set pointer to next entry */
         src = g–>g_member + hist–>h_bc.b_cpu;        /* set source */
         g–>g_nextseqno++;                            /* accept it */
         g–>g_nexthistory++;                          /* increase next history */
         src–>m_messid = hist–>h_bc.b_messnr;         /* remember last used message id */
         if (src != g–>g_me)                          /* did I send the message? */
                  src–>m_expect = MAX(src–>m_expect, hist–>h_bc.b_expect);
         if ((g–>g_flags & FL_SENDING) && hist–>h_bc.b_cpu == g–>g_index) {
                  /* I sent this message; unblock sending thread. */
                  g–>g_flags &= ~FL_SENDING;          /* switch flag off */
                  unblock(g);                         /* unblock sending thread */
         }
         if (IAMSILENT(g)) sendstate(g);              /* silent for a long time? */
   }
}
```

Fig. 3.17. Function *processrec* used in Fig. 3.16.

3.4.3. Protocol for Recovery

In the previous section, we assumed that none of the members ever failed. Now we will discuss the algorithms that are executed when a member or the sequencer fails.

Failures are detected by sending a BC_ALIVEREQ message to a kernel that has not been heard from in some time, and waiting for a reply. If, after a number of retries no BC_ALIVE message comes back, the enquiring kernel assumes that the destination has failed and initiates recovery. Picking the right number of retries is tricky. If the number is too low, a kernel may decide that another member has failed while in reality the other group member was just busy doing other things. If the number is too high, it can take a long time before a failure is detected.

Once a kernel has decided that another kernel has failed, it enters a recovery mode that causes subsequent calls to *ReceiveFromGroup* from local members to return an error status. Any surviving member may call *ResetGroup* to recover from a member failure. *ResetGroup* tries to re-form the group into a group that contains all the surviving members that can communicate with each other. If needed, it also elects a new sequencer. The second parameter of *ResetGroup* is the minimum number of members of the old group that are required for the new group to be valid. If *ResetGroup* succeeds, it returns the actual number of members in the new group. *ResetGroup* fails if it cannot form a group with enough members.

The protocol to recover from member crashes is based on the invitation protocol described by Garcia-Molina [Garcia-Molina 1982]. It runs in two phases. In the first

phase, the protocol establishes which members are alive and chooses one member as coordinator to handle the second phase. Every member that calls *ResetGroup* becomes a coordinator and invites other members to join the new group by sending a BC_REFORMREQ message. If a member is alive and it is not a coordinator, it responds with a BC_VOTE message containing the highest sequence number that it has seen (each member already keeps this number for running the protocol for communication failures as described above). If one coordinator invites another coordinator, the one with the highest sequence number becomes coordinator of both (if their sequence numbers are equal, the one with the lowest member id is chosen). When all members of the old group have been invited, there is one coordinator left. It knows which members are alive and which member has the highest sequence number.

In the second phase of the recovery, the group is restarted. If the coordinator has missed some messages, it asks the member with the highest sequence number for retransmissions (this is unlikely to happen, because the initiator of the recovery with the highest sequence number becomes coordinator). Once the coordinator is up-to-date, it checks to see if the complete history is replicated on all remaining members. If one of the members does not have the complete history, the coordinator sends it the missing messages. Once the new group is up-to-date, it builds a BC_RESULT message containing information about the new group: the size of the new group, the members in the new group, the new sequencer (itself), a new network address (FLIP address) for the group, and the new *incarnation* number of the group. The network address and incarnation number are included to make sure that messages directed to the old group will not be accepted by the new group. It stores the BC_RESULT message in its history and broadcasts it to all members. When a member receives the BC_RESULT message, it updates the group information, sends an acknowledgement (BC_RESULTACK) to the coordinator, and enters normal operation. The coordinator enters normal operation after it has received a BC_RESULTACK message from all members.

When back in normal operation, members never accept messages from a previous incarnation of the group. Thus, members that have been quiet for a long time, for example, due to a network partition, and did not take part in the recovery will still use an old incarnation number when sending a message to the new group. These messages will be ignored by the new group, treating the ex-member effectively as a dead member. The incarnation numbers make sure that no conflicts will arise when a member suddenly comes back to life after being quiet for a period of time.

If the coordinator (or one of the members) crashes during the recovery, the protocol starts again with phase 1. This continues until the recovery is successful or until there are not enough living members left to recover successfully.

In Figure 3.18 the recovery protocol is illustrated. In Figure 3.18(a) a possible start of phase 1 is depicted. Members 0, 1, and 2 simultaneously start the recovery and are coordinators. Member 3 has received more messages (it has seen the highest sequence number), but it did not call *ResetGroup*. Member 4, the sequencer, has crashed. In Figure 3.18(b), the end of phase 1 has been reached. Member 0 is the

coordinator and the other members are waiting for the *result* message (they check periodically if member 0 is still alive). In Figure 3.18(c), the end of phase 2 has been reached. Member 0 is the new sequencer. It has collected message 34 from member 3 and has stored the *result* message (number 35) in its history. The other members are also back in normal operation. They have collected missing messages from member 0 and have also received the BC_RESULT message.

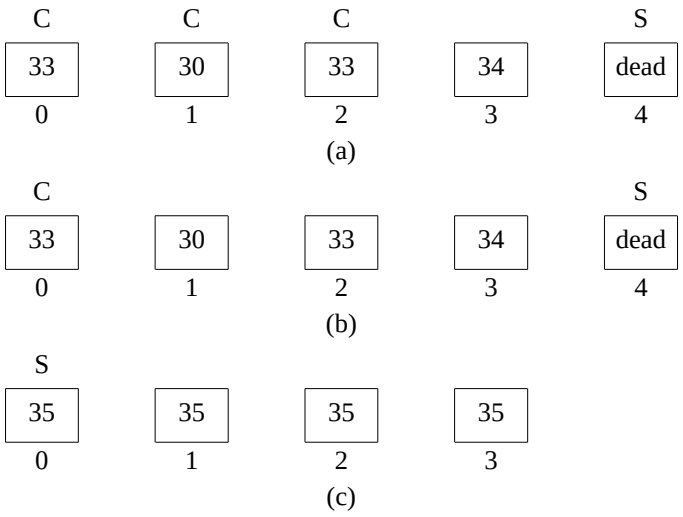


Fig. 3.18. A possible recovery for a group of size 5 after a crash of member 4. S is a sequencer. C is a coordinator. The number in the box is the sequence number of the last message received. The number below the box is the member id. (a) Shows a possible start of phase 1. (b) Shows the start of phase 2. In (c) the recovery has been completed.

3.5. Performance

The measurements were taken on a collection of 30 MC68030s (20 Mhz) connected by a 10 Mbits/s Ethernet. All processors were on the same Ethernet and were connected to the network by Lance chip interfaces (manufactured by Advanced Micro Devices). The machines used in the experiments were able to buffer 32 Ethernet packets before the Lance overflowed and dropped packets. Each measurement was done 10,000 times on an almost quiet network. The size of the history buffer was 128 messages. The experiments measured failure-free performance.

Most experiments have been executed with messages of size 0 byte, 1 Kbyte, 4 Kbyte, and 8,000 byte. The last size was chosen to reflect a fundamental problem in the implementation. In principle, the group communication protocols can handle messages of size 8 Kbyte or larger, but lower layers in the kernel prohibit measuring the communication costs for these sizes. Messages larger than a network packet size have to be fragmented in multiple packets. To prohibit a sender overrunning a receiver flow-control has to be performed on messages consisting of multiple packets. For point-to-point communication many flow-control algorithms exists [Tanenbaum 1989],

but it is not immediately clear how these should be extended to multicast communication. We are also not aware of new algorithms designed for multicast flow-control. The measurements in this section therefore do not include the time for flow-control and we have used a reasonable but arbitrary upper bound to the message size.

The first experiment measures the delay for the PB method with $r = 0$. In this experiment one process continuously broadcasts messages of size 0 byte, 1 Kbyte, 4 Kbyte, and 8,000 byte to a group of processes (the size of the message excludes the 116† bytes of protocol headers). All members continuously call *ReceiveFromGroup*. This experiment measures the delay seen from the sending process, between calling and returning from *SendToGroup*. The sending process runs on a different processor than the sequencer. Note that this is not the best possible case for our protocol, since only one processor sends messages to the sequencer (i.e., no acknowledgements can be piggybacked by other processors).

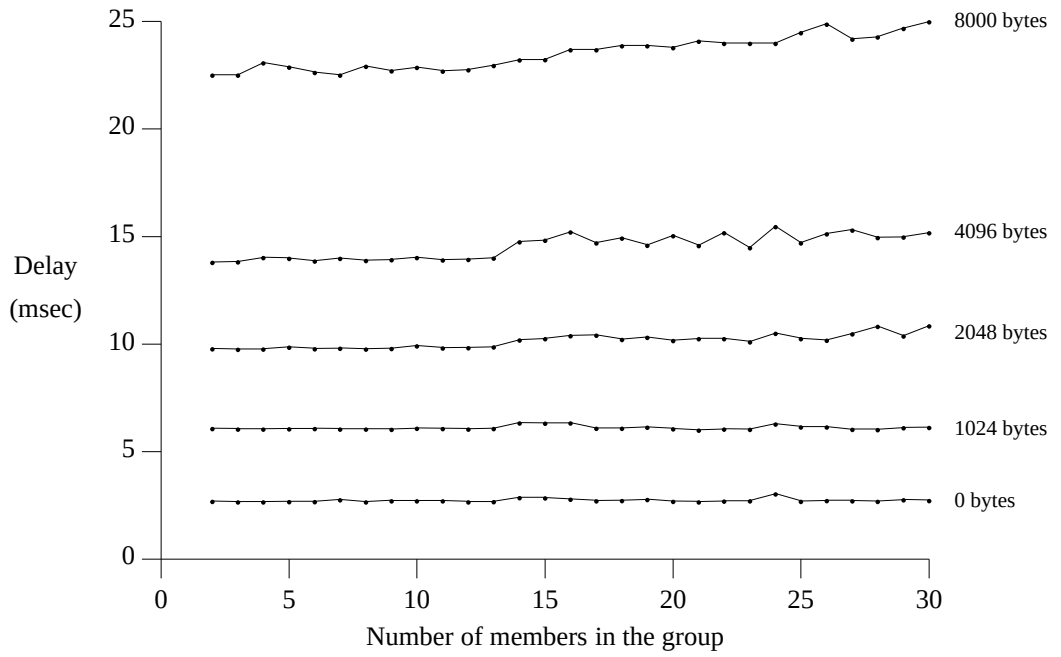


Fig. 3.19. Delay for 1 sender using PB method.

The results of the first experiment are depicted in Figure 3.19. For a group of two processes, the measured delay for a 0-byte message is 2.7 msec. Compared to the Amoeba RPC on the same architecture, the group communication is only 0.1 msec slower than the RPC. (The RPC numbers reported in Chapter 2 were measured on Sun 3/60s, which are faster for communication than the MC68030s used in the group experiments. Unfortunately, we did not have 30 Sun 3/60s available.) For a group of 30

† 116 is the number of header bytes: 14 bytes for the Ethernet header, 2 bytes flow control, 40 bytes for the FLIP header, 28 bytes for the group header, and 32 bytes for the Amoeba user header. The Amoeba user header is only sent once.

processes, the measured delay for a 0-byte message is 2.8 msec. From these numbers, one can estimate that each node adds 4 μsec to the delay for a broadcast to a group of 2 nodes. Extrapolating, the delay for a broadcast to a group of 100 nodes should be 3.2 msec. Sending an 8,000-byte message instead of a 0-byte message adds roughly 20 msec. Because the PB method is used in this experiment, this large increase can be attributed to the fact that the complete message goes over the network twice.

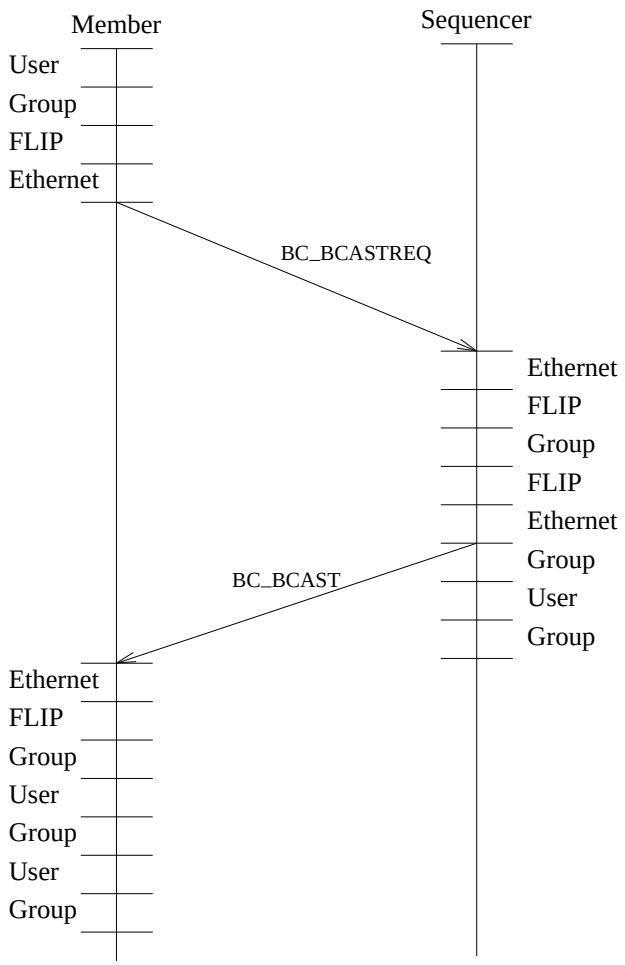
Figure 3.20 breaks down the cost for a single 0-byte *SendToGroup* to a group of size 2, using the PB method. Both members call ReceiveFromGroup to receive messages. To reflect the typical usage of the group primitives, ReceiveFromGroup is called by another thread than *SendToGroup*. Most of the time spent in user space is the context switch between the receiving and sending thread. The cost for the group protocol itself is 740 μsec.

The results of the same experiment but now using the BB method are depicted in Figure 3.21. The result for sending a 0-byte message is, as can be expected, similar. For larger messages the results are dramatically better, since in the BB method the complete message only goes over the network once. At first sight, it may look as if the BB method is always as good as or better than the PB protocol. However, this is not true. From the point of view of a single sender there is no difference in performance, but for the receivers other than the sequencer there is. In the PB protocol they are interrupted once, while in the BB protocol they are interrupted twice.

The next experiment measures the throughput of the group communication. In this experiment all members of a given group continuously call *SendToGroup*. We measure both for the PB method and the BB method how many messages per second the group can deal with. The results are depicted in Figure 3.22 and Figure 3.23. The maximum throughput is 815 0-byte messages per second. The number is limited by the time that the sequencer needs to process a message. This time is equal to the time spent taking the interrupt plus the time spent in the driver, FLIP protocol, and broadcast protocol. On the 20-MHz 68030, this is almost 800 μsec, which gives an upper bound of 1250 messages per second. This number is not achieved, because the member running on the sequencer must also be scheduled and allowed to process the messages.

The throughput decreases as the message size grows, because more data have to be copied. A receiver must copy each message twice: once from the Lance interface to the history buffer and once from the history buffer to user space. In the PB method, the sequencer must copy the message three times: one additional copy from the history buffer to the Lance interface to broadcast the message. (If our Lance interface could have sent directly from main memory, this last copy could have been avoided.) If Amoeba had support for sophisticated memory management primitives like Mach [Young et al. 1987] the second copy from the history buffer to user space could also have been avoided; in this case one could map the page containing the history buffer into the user's address space.

For messages of size 4 Kbyte and larger, the throughput drops more. (For some configurations we are not able to make meaningful measurements at all.) This comes



(a)

Layer	Time (μsec)
User	514
Group	740
FLIP	570
Ethernet	916

(b)

Fig. 3.20. (a) A break down of the cost in μsec of a single *SendToGroup* - *ReceiveFromGroup* pair. The group size is 2 and the PB method is used.
(b) The time spent in the critical path of each layer. The Ethernet time is the time spend on the wire plus the time spent in the driver and taking the interrupt.

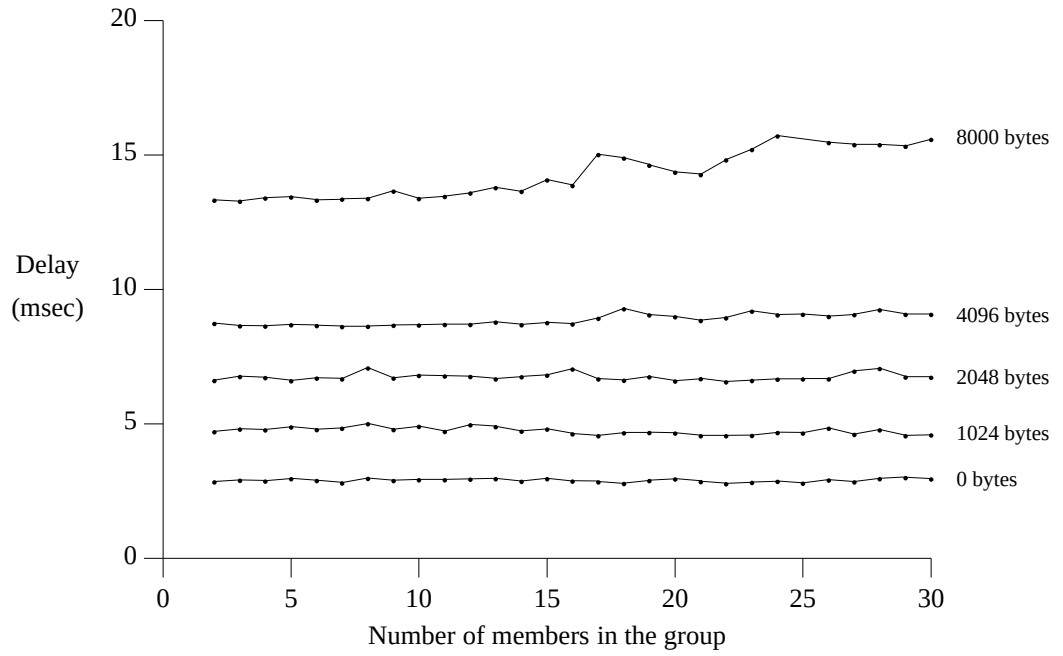


Fig. 3.21. Delay for 1 sender using the BB method.

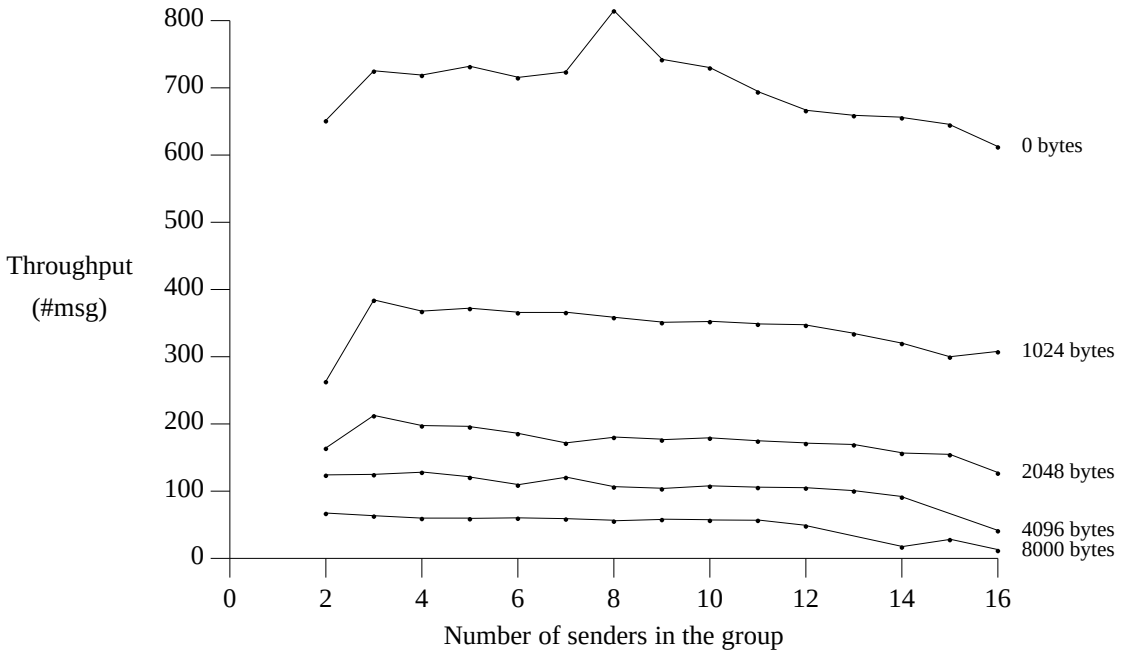


Fig. 3.22. Throughput for the PB Method. The group size is equal to the number of senders.

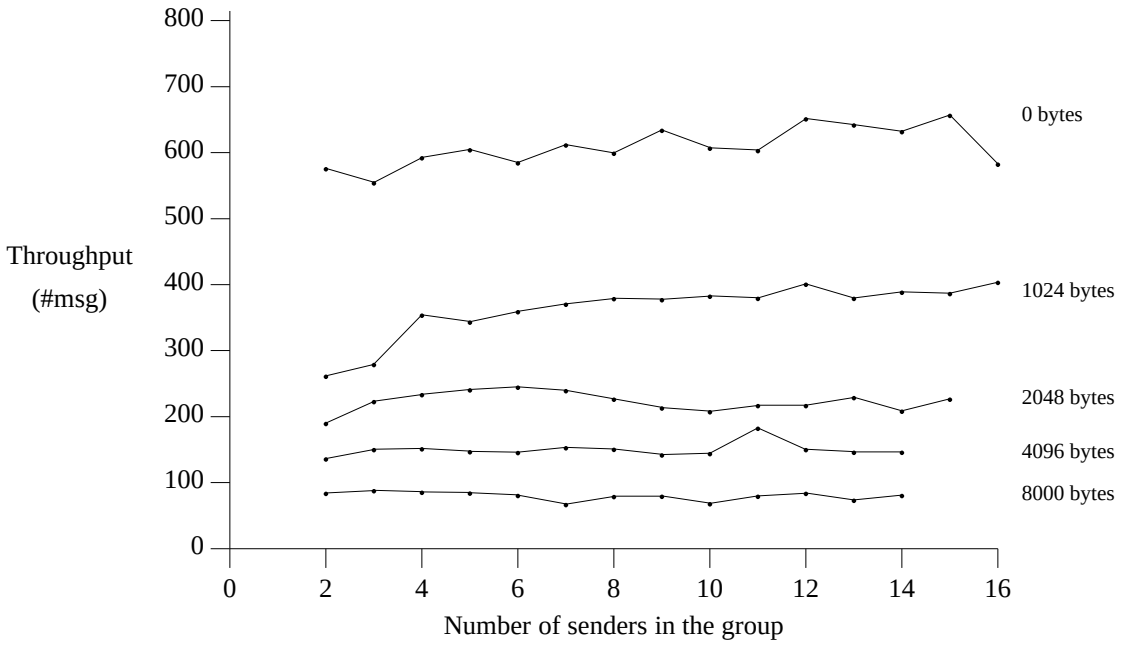


Fig. 3.23. Throughput for the BB Method. The group size is equal to the number of senders.

from the fact that our Lance configuration can buffer only 32 Ethernet packets, each with a maximum size of 1514 bytes. This means that the sequencer starts dropping packets when receiving 11 complete 4 Kbyte messages simultaneously. (If our system had been able to buffer more packets, the same problem would have appeared at some later point. The sequencer will need more time to process all the buffered packets, which will at some point result in timeouts at the sending kernel and in retransmissions.) The protocol continues working, but the performance drops, because the protocol waits until timers expire to send retransmissions. The same phenomenon also appears with groups larger than 16 members and 2-Kbyte messages.

Another interesting question is how many disjoint groups can run in parallel on the same Ethernet without influencing each other. To answer this question we ran an experiment in which a number of groups of the same size operated in parallel and each member of each group continuously called *SendToGroup*. We ran this experiment for group sizes of 2, 4, and 8 and measured the total number of 0-byte broadcasts per second (using the PB method). The experiment measures, for example, for two groups with 2 members the total number of messages per second that 4 members together succeeded in sending, with each member being member of one group and running on a separate processor. The results are depicted in Figure 3.24. The maximum throughput is 3175 broadcasts per second when 5 groups of size 2 are broadcasting at maximum 0-byte message throughput (this corresponds to at least 736,600 bytes per second;

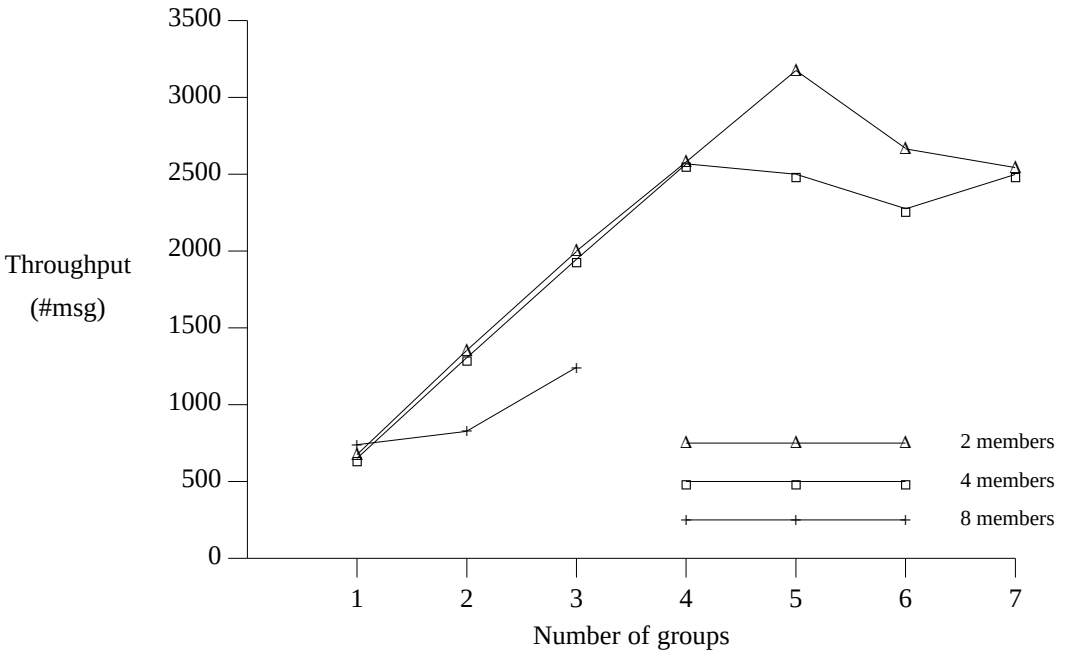


Fig. 3.24. Throughput for groups of 2, 4, and 8 members running in parallel and using the PB method. We did not have enough machines available to measure the throughput with more groups with 8 members.

3175 * 2 * 116 = 736,600). When another group is added the throughput starts dropping due to the number of collisions on the Ethernet. This is also the case for the poor performance of groups of size 8.

The final experiment measures the delay of sending a message with $r > 0$. Figure 3.25 and Figure 3.26 depict the delay for sending a message with *resilience degree*s from one to 15. As can be expected, sending a message with a higher r scales less well than sending with a degree of 0. In this case, the number of FLIP messages per reliable broadcast sent is equal to $3 + r$ (assuming no packet loss). Also, when using large messages and a high resilience degree, our hardware starts to miss packets. For these settings we are not able to make meaningful measurements.

The delay for sending a 0-byte message to a group of size two with a resilience degree of one is 4.2 msec. For a group of size 16 with a resilience degree of 15, the measured delay is 12.9 msec. This difference is due to the 14 additional acknowledgements that have to be sent. Each acknowledgement adds approximatelty 600 μsec.

3.6. Comparison with Related Work

In this section, we will compare our reliable broadcast protocol with other protocols and our system with other systems that provide broadcast communication. Figure 3.27 summarizes the results. In comparing protocols, several points are of interest. The first is the performance of the protocol. This has two aspects: the time before a

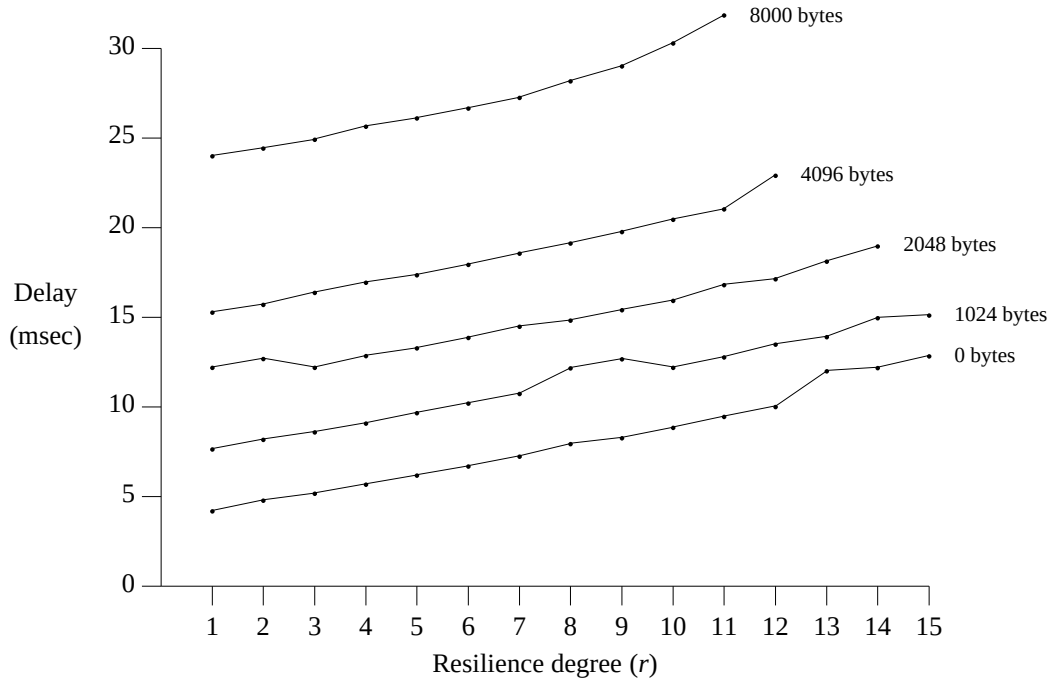


Fig. 3.25. Delay for 1 sender with different *r*s using PB method. Group size is equal to $r + 1$.

message can be delivered to the application and the number of protocol messages needed to broadcast the message. The second is the semantics of sending a broadcast message. There are three aspects: reliability, ordering, and fault tolerance. Although fault tolerance is an aspect of reliability, we list it as an separate aspect. The third is the hardware cost. The key aspect here is whether the protocol requires members to be equipped with additional hardware (e.g., a disk). Although more research has been done in broadcast communication than is listed in the table, this other research focuses on different aspects (e.g., multicast routing in a network consisting of point-to-point communication links) or requires synchronized clocks. For a bibliography of these papers we refer the reader to [Chanson et al. 1989].

Let us look at each protocol in turn. The protocols described by [Birman and Joseph 1987] are implemented in the Isis system. The Isis system is primarily intended for doing fault-tolerant computing. Thus, Isis tries to make broadcast as fast as possible in the context of possible processor failures. Our system is intended to do reliable ordered broadcast as fast as possible. If processor failures occur, some messages may be lost, in the $r = 0$ case. If, however, an application requires fault tolerance, our system can trade performance against fault tolerance. As reliable ordered broadcast in the event of processor failures is quite expensive, Isis includes primitives that provide a weaker ordering (e.g., a causal ordering).

Recently the protocols for Isis have been redesigned [Birman et al. 1991]. The

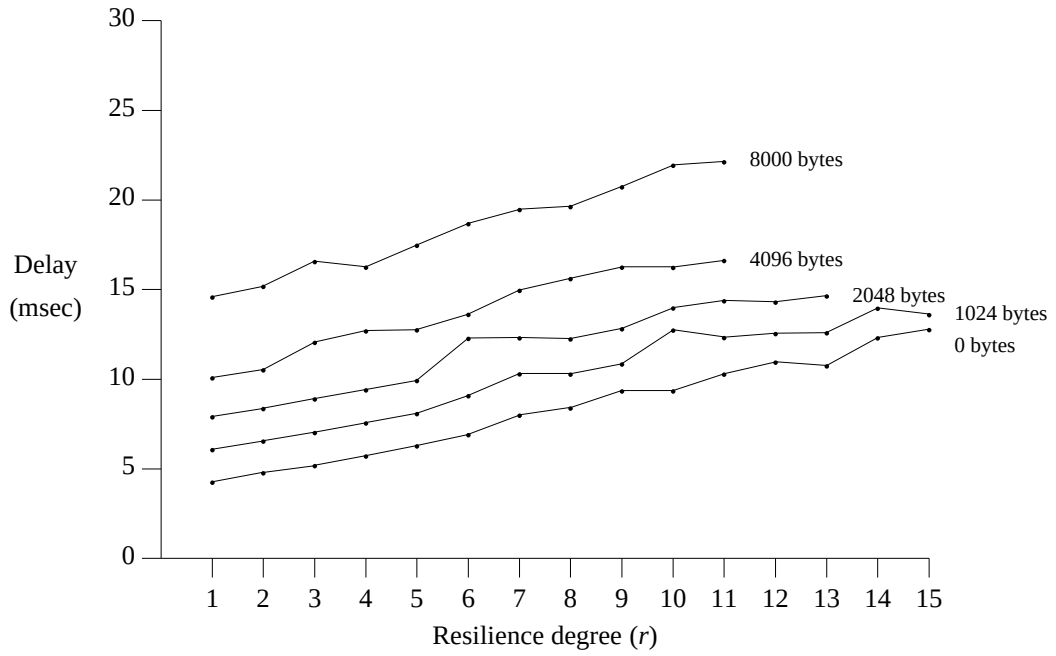


Fig. 3.26. Delay for 1 sender with different *r*s using BB method. Group size is equal to $r + 1$.

system is now completely based on a broadcast primitive that provides causal ordering. The implementation of this primitive uses reliable point-to-point communication. The protocol for totally-ordered broadcast is based on causal broadcast. As in our protocol, a sequencer (a token holder in Isis terminology) is used to totally order the causal messages. Unlike our protocol, the token holder can migrate. Depending if the sender holds the token, this scheme requires either one message or two messages, but each message possibly contains a sequence number for each member, while in our protocol the number of bytes for the protocol header is independent of the number of members. Thus in Isis, for a group of 1024 members, 4K bytes of data are possibly added to each message. Depending on the communication patterns, this data can be compressed, but in the worst case 4K is still needed. As an aside, the new version of Isis no longer supports a total ordering for overlapping groups.

Chang and Maxemchuk describe a family of protocols [Chang and Maxemchuk 1984]. These protocols differ mainly in the degree of fault tolerance that they provide. Our protocol for $r = 0$ resembles their protocol that is not fault-tolerant (i.e., it may lose messages if processors fail), but ours is optimized for the common case of no communication failures. Like our protocol, the CM protocol also depends on a central node, the token site, for ordering messages. However, on each acknowledgement another node takes over the role of token site. Depending on the system utilization, the transfer of the token site on each acknowledgement can take one extra control message. Thus

their protocol requires 2 to 3 messages per broadcast, whereas ours requires only 2 in the best case and only a fraction bigger than 2 in the normal case.

Protocol	Performance		Semantics			Additional Hardware
	Delay	#Pkts	Reliable	Ordering	Fault-tolerance	
BJ	2 Rounds	$2n$	Yes	Yes	$n-1$	No
BSS	2	$2n$	Yes	Yes	$n-1$	No
CM	$2 \ldots 2+n-1$	$2+\varepsilon$	Yes	Yes	$0 \ldots n-1$	No
CZ	2	$2 \ldots n$	No…Yes	No	No	No
EZ	2	2	Yes	Yes	$n-1$	No
LG	3 Phases	$1 \ldots 4n$	Yes	Yes	Yes	Yes
MMA	≥ 1	1	Yes	Yes	$n/2$	No
M	2	$2+\varepsilon$	Yes	Yes	Yes	Yes
NCN	2	$n+1$	Yes	No…Yes	$0 \ldots n-1$	No
PBS	≥ 1	1	Yes	Yes	$0 \ldots n-1$	No…Yes
TY	2	3	Yes	Yes	No…Yes	Yes
VRB	2 Rounds	$2n$	Yes	Yes	$n-1$	Yes
Ours	2	$2 \ldots 3+n-1$	Yes	Yes	$0 \ldots n-1$	No

Fig. 3.27. Comparison of different broadcast protocols. A protocol is identified by the first letters of the names of the authors. The group size is n. In a *round* each member sends a message. A *phase* is the time necessary to complete a state transition (sending messages, receiving messages, and local computation). For each protocol, we list the best performance. In some cases, the performance may be worse, for example, for higher degrees of fault tolerance.

Fault tolerance is achieved in the CM protocol by transferring the token. If a message is delivered after the token has been transferred L times, then L processor failures can be tolerated. This scheme introduces a very long delay before a message can be delivered, but uses fewer messages than ours. Finally, in the CM protocol all messages are broadcast, whereas our protocol uses point-to-point messages whenever possible, reducing interrupts and context switches at each node. This is important, because the efficiency of the protocol is not only determined by the transmission time,

but also (and mainly) by the processing time at the nodes. In their scheme, each broadcast causes at least $2(n-1)$ interrupts; in ours only n. The actual implementation of their protocol uses physical broadcast for all messages and is restricted to a single LAN.

The group communication for the V system, described in [Cheriton and Zwaenepoel 1985], integrates RPC communication with broadcast communication in a flexible way. If a client sends a request message to a process group, V tries to deliver the message at all members in the group. If any one of the members of the group sends a reply back, the RPC returns successfully. Additional replies from other members can be collected by the client by calling *GetReply*. Thus, the V system does not provide reliable, ordered broadcasting. However, this can be implemented by a client and a server (e.g., the protocol described by Navaratnam, Chanson, and Neufeld runs on top of V). In this case, a client needs to know how the service is implemented. We do not think this is a good approach. If an unreplicated file service, for example, is reimplemented as a replicated file service to improve performance and to increase availability, it would mean that all client programs have to be changed. With our primitives, no change is needed in the client code.

Elnozahy and Zwaenepoel describe a broadcast protocol (EZ in Fig. 3.27) especially designed for replicated process groups [Elnozahy and Zwaenepoel 1992b]. Like the CM protocols and like ours, it is based on a centralized site and negative acknowledgements. Unlike ours, it especially designed to provide a low delay in delivery of messages, while at the same time providing a high resilience degree. This goal is achieved by keeping an *antecedence graph* and adding to each message the incremental changes to this graph. By maintaining the antecedence graph this protocol does not need to send acknowledgements to confirm that the message is stored at r members. On the other hand, the application must potentially be rolled back when a processor fails.

The protocol described in [Luan and Gligor 1990] is one of the protocols that require additional hardware. In the LG protocol each member must be equipped with a disk. Using these disks the protocol can provide fault-tolerant ordered broadcasting, even if the network partitions. It uses a majority-consensus decision to agree on a unique ordering of broadcast messages that have been received and stored on disk. Under normal operation, the protocol requires $4n$ messages. However, under heavy load the number of messages goes down to 1. The delay before a message can be delivered is constant: the protocol needs three protocol phases before it can be delivered. In a system like Amoeba that consists of a large number of processors, equipping each machine with a disk would be far too expensive. Furthermore, the performance of the protocol is also much too low to be considered as a general protocol for reliable broadcasting.

A totally different approach to reliable broadcasting is described in [Melliar-Smith et al. 1990]. They describe a protocol that achieves reliable broadcast with a certain probability. If processor failures occur, it may happen that the protocol cannot

decide on the order in which messages must be delivered. They claim that the probability is high enough to assume that all messages are ordered totally, but nevertheless there is a certain chance that messages are not totally-ordered. The MMA protocol uses only one message, but a message cannot be delivered at an application until several other broadcast messages have been received. For a group of 10 nodes, a message can be delivered on average after receiving another 7.5 messages. With large groups, the delay is unacceptably large.

Montgomery [Montgomery 1978] coined the term *atomic broadcast* in an unpublished dissertation. The thesis describes the problem of reliable, totally-ordered multicast and proposes two solutions: one based on point-to-point communication and one based on broadcast. Both solutions are based on a centralized component that orders messages. To provide for fault tolerance the messages are always stored on stable storage. Another important difference between these two protocols and ours is that acknowledgements are not piggybacked. Instead, each node broadcasts once in a while a message saying which messages it has received, so that the central site can purge messages from its stable storage. No indication is given that the protocol was ever implemented, and no measurements are presented.

Navaratnam, Chanson, and Neufeld provide two primitives for reliable broadcasting [Navaratnam et al. 1988]. One orders messages; the other does not. Their protocol also uses a centralized scheme, but instead of transferring the token site on each acknowledgement, their central site waits until it has received acknowledgements from *each* node that runs a member before sending the next broadcast. In an implementation of the NCN protocol on the V-system, a reliable broadcast message costs 24.8 msec for a group of 4 nodes on comparable hardware. Our current implementation does this in less than 4.8 msec ($r = 3$).

In [Peterson et al. 1989] a communication mechanism is described called *Psync*. In Psync a group consists of a fixed number of processes and is closed. Messages are causally ordered. A library routine provides a primitive for a total ordering. This primitive is implemented using a single causal message, but members cannot deliver a message immediately when it arrives. Instead, a number of messages from other members (i.e., at most one from each member) must be received before the total order can be established.

Another protocol that requires hardware support for reliable broadcasting is described in [Tseung and Yu 1990]. The TY protocol requires that at least three components be added to the network: a Retransmission Computer, a Designated Recorder Computer, and one or more Playback Recorder Computers. The Playback Recorder Computers should be equipped with a disk (typically one Playback Recorder Computer is used per group). If fault tolerance is required, hot backup systems can be provided for the Retransmission Computer and the Designated Recorder Computer. The protocol works as follows. A user computer sends a point-to-point message to the Retransmission Computer. The Retransmission Computer plays a similar role as our sequencer. It adds some information to the message, such as a sequence number, and

broadcasts it. In the TY protocol, the Retransmission Computer is ready to broadcast the next message after the Designated Recorder Computer has sent an acknowledgement. The Designated Recorder stores messages for a short period, in case one of the Playback Recorder Computers has missed a message. The Playback Computers store the messages on disk for a long period of time to be able to send retransmissions to user computers if they have missed a message. This protocol requires more messages than our protocol (the acknowledgement from the Designated Recorder to the Retransmission Recorder is not needed in our protocol) and requires additional hardware. Furthermore, one computer (the Retransmission Computer) serves as the sequencer for all groups. If the sequencer becomes a bottleneck in one group, all other groups will suffer from this. Also, if the Retransmission Computer or the Designated Recorder crashes no group communication can take place in the whole system. For these reasons and the fact that groups are mostly unrelated, we order messages on a per group basis by having a separate sequencer for each group.

The last protocol that we consider which provides reliable broadcasting is described in [Verissimo et al. 1989]. The VRB protocol runs directly on top of the Medium Access Layer (MAC). Thus, the protocol is restricted to a single LAN, but on the other hand it allows for an efficient implementation. The protocol itself is based on the two phase commit protocol [Eswaran et al. 1976]. In the first phase, the message is broadcast. All receivers are required to send an acknowledgement indicating if they will accept the message. After the sender has received all acknowledgements, it broadcasts a message telling if the message can be delivered to the user application or not. The protocol assumes that the network orders packets and that there is a bounded transmission delay.

A somewhat related approach is Cooper's replicated RPC [Cooper 1985]. Although replicated RPC provides communication facilities for 1-to-n communication, it does not use group communication. Instead, it performs $n - 1$ RPCs. As it is not based on group communication, nor does it use multicast, we did not include it in the table. Replicated RPC, however, can be implemented using group communication [Wood 1992].

Another related approach is MultiRPC [Satyanarayanan and Siegel 1990]. From the programming point of view, MultiRPC behaves exactly the same as an ordinary RPC. However, instead of invoking one server stub, MultiRPC invokes multiple server stubs on different machines in parallel. Compared to performing $n - 1$ regular RPCs, MultiRPC is more efficient as the server stubs are executed in parallel. The authors also discuss preliminary results for sending the request message in a multicast packet to avoid the overhead of sending the requests n times in point-to-point packets. The replies on an RPC are sent using point-to-point communication and are processed sequentially by the client machine. There is no ordering between two MultiRPCs and MultiRPC does not provide reliable communication in case one of the servers crashes.

If messages are sent regularly and if messages may be lost when processor failures occur, our protocol is more efficient than any of the protocols listed in the

table. In our protocol, the number of messages used is determined by the size of the history buffer and the communication pattern of the application. In the normal case, two messages are used: a point-to-point message to the sequencer and a broadcast message. In the worst case, when one of the nodes is continuously broadcasting, $(n/HISTORY_SIZE) + 2$ messages are needed. For example, if the number of buffers in the history is equal to the number of processors, three messages per reliable broadcast are needed. In practice, with say 1 Mbyte of history buffers and 1 Kbyte-messages, there is room for 1024 messages. This means that the history buffer will rarely fill up and the protocol will actually average two messages per reliable broadcast. The delay for sending a message is equal to the time to send and receive a message from the sequencer. The delay before a message can be delivered to the application is optimal; as soon as a broadcast arrives, it can be delivered. Also, our protocol causes a low number of interrupts. Each node gets one interrupt for each reliable broadcast message (PB method).

If messages must be delivered in order and without error despite member crashes, the cost of the protocol increases. For resilience degree $r > 0$, each reliable broadcast takes $3 + r$ messages: one message for the point-to-point message to the sequencer, one broadcast message from the sequencer to all kernels announcing the sequence number, r short acknowledgements that are sent point-to-point to the sequencer, and one short accept message from the sequencer to all members. The delay increases. A message can only be accepted by the sequencer after receiving the message, broadcasting the message, and receiving r acknowledgements. However, the r acknowledgements will be received almost simultaneously. Thus, an increase in fault tolerance costs the application a decrease in performance. It is up to the application programmer to make the tradeoff.

Like some of the other protocols, our protocol uses a centralized node (the sequencer) to determine the order of the messages. Although in our protocol this centralized node does not do anything computationally intensive (it receives a message, adds the sequence number, and broadcasts it), it could conceivably become a bottleneck in a very large group. If $r > 0$, it is likely that the sequencer will become a bottleneck sooner due to the r acknowledgements that it has to process. Under heavy load, one could try to piggyback these acknowledgements onto other messages, to make the protocol scale better.

3.7. Conclusion

We have identified 6 criteria that are important design issues for group communication: addressing, reliability, ordering, delivery semantics, response semantics, and group structure. We have discussed each of these criteria and the choices that have been made for the Amoeba distributed system. The Amoeba interface for group communication is simple, powerful, and easy to understand. Its main properties are:

- Reliable communication.

- Messages are totally-ordered per group.
- Programmers can trade performance against fault tolerance.

Based on our experience with distributed programming we believe that these properties are essential in building efficient distributed applications. We will discuss this in the coming chapters.

We have described in detail the group communication interface and its implementation. In addition, we have provided extensive performance measurements on 30 processors. The delay for a null broadcast to a group of 30 processes running on 20-MHz MC68030s connected by 10 Mbit/s Ethernet is 2.8 msec. The maximum throughput per group is 815 broadcasts per group. With multiple groups, the maximum number of broadcasts per second has been measured at 3175.

Notes

Some of the research presented here contains material from the paper by Kaashoek and Tanenbaum published in the proceedings of the *Eleventh International Conference on Distributed Computing Systems* [Kaashoek and Tanenbaum 1991]. An early version of the PB protocol was published by Kaashoek, Tanenbaum, Flynn Hummel, and Bal in *Operating Systems Review* [Kaashoek et al. 1989]. A short description of the PB protocol and the BB protocol also appeared in [Tanenbaum et al. 1992]. The protocols presented were considerably improved due to discussions with Wiebren de Jonge.

4

PARALLEL PROGRAMMING USING BROADCASTING AND SHARED OBJECTS

As computers continue to get cheaper, there is increasing interest in harnessing multiple processors together to build large, powerful parallel systems. The goal of this work is to achieve high performance at low cost. This chapter first surveys the two major design approaches taken so far, multiprocessors and multicomputers, and points out their strengths and weaknesses. Then it introduces and discusses a hybrid form, called the *shared data-object model*. In the shared data-object model, processes can share variables of some abstract data type, while not requiring the presence of physical shared memory. The key issue is how to implement this model efficiently. Two implementations of the shared-data object model are described: one based on only group communication and one based on group communication and RPC. Both implementations use compile-time information at run-time to achieve high performance. To demonstrate the effectiveness of our approach, we describe some applications we have written for the shared data-object model, and give measurements of their performance.

A *parallel* system is one in which a user is attempting to utilize multiple processors in order to speed up the execution of a single program. The processors may be in a single rack and connected by a high-speed backplane bus, they may be distributed around a building and connected by a fast LAN, or some other topology may be used. What is important is that a large number of processors are cooperating to solve a single problem. In this chapter we will concentrate on parallel systems, using as a metric how much speedup can be achieved on a single problem by using n identical processors as compared to using only one processor.

A brief outline of the rest of this chapter follows. In Section 4.1, we will discuss the two major kinds of parallel computers in use, so we can contrast our design with

them later. In Section 4.2, we will examine so-called NUMA architectures, which are an attempt at combining the two types of parallel machines. In Section 4.3, we will present our model, the shared data-object model. The shared data-object model consists of three software layers. The top layer consists of parallel programs written in a new language called *Orca* [Bal 1990; Bal et al. 1992a]. Orca is a parallel programming language based on the shared data-object model. As Orca is one of the main topics in Bal's thesis [Bal 1990], we will discuss the top layer here only briefly in Section 4.4. The focus in this chapter is on the middle layer, the run-time systems for Orca. The first run-time system (RTS) is discussed in Section 4.5. The bottom layer provides the reliable group communication, discussed in the previous chapter. In Section 4.6, we discuss the design of the second RTS, an optimized version. The optimized version uses more compile-time information to achieve better performance at run-time. In Section 4.7, we discuss the compiler techniques and the new RTS mechanisms that are needed to achieve the optimization discussed in Section 4.6. In Section 4.8, we discuss a number of example programs and their performance on the unoptimized and optimized run-time system. In Section 4.9, we present a comparison with related work. In Section 4.10, we draw our conclusions.

4.1. Architectures for Parallel Programming

Parallel computers can be divided into two categories: those that contain physical shared memory, called *multiprocessors* and those that do not, called *multicomputers*. A simple taxonomy is given in Fig. 4.1. A more elaborate discussion of parallel architectures is, for example, given by Almasi and Gottlieb [Almasi and Gottlieb 1989].

4.1.1. Multiprocessors

In multiprocessors, there is a single, global, shared address space visible to all processors. Any processor can read or write any word in the address space by simply moving data from or to a memory address. The key property of this class of systems is *memory coherence*. Memory coherence is defined by Censier and Feautrier as follows: When any processor writes a value v to memory address m, any other processor that subsequently reads the word at memory address m, no matter how quickly after the write, will get the value v just written [Censier and Feautrier 1978].

In the context of multiprocessors, this definition is difficult to apply, as can be seen from the following example. At a certain moment, process A wants to write a word, and process B wants to read it. Although the two operations may take place a microsecond apart, the value read by B depends on who went first. The behavior is determined by the ordering of the read and write operations. Lamport, therefore, introduced another definition for the correct execution of multiprocessor programs, called *sequential consistency* [Lamport 1979]. In a multiprocessor, sequential consistency determines the allowable order of read and write operations:

A multiprocessor is sequentially consistent if the result of any execution is

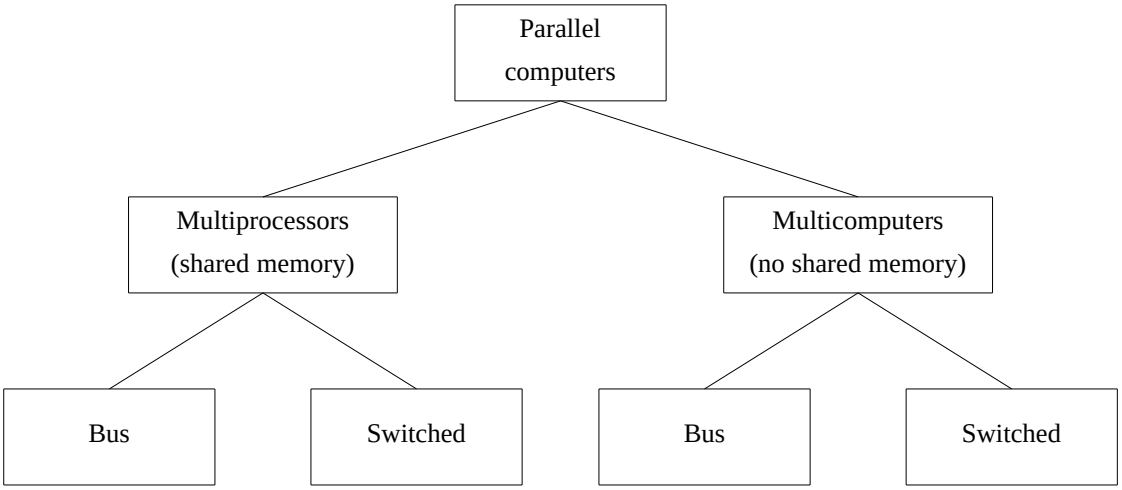


Fig. 4.1. A taxonomy of parallel computers.

the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [Lamport 1979].

Multiprocessor Hardware

There are two basic ways to build a multiprocessor, both of them expensive. The first way is to put all the processors on a single bus, along with a memory module. To read or write a word of data, a processor makes a normal memory request over the bus. Since there is only one memory module and there are no copies of memory words anywhere else, the memory is always coherent.

The problem with this scheme is that with as few as 4 to 8 processors, the bus can become completely saturated. To get around this problem, each processor is normally given a cache memory, as shown in Fig. 4.2. The caches introduce a new problem, however. If processors 1 and 2 both read the contents of memory address *m* into their caches, and one of them modifies it, when the other one next reads that address it will get a *stale* value. Memory is then no longer coherent.

The stale data problem can be solved in many ways, such as having all writes go through the cache to update the external memory, or having each cache constantly *snoop* on the bus (i.e., monitor it), taking some appropriate action whenever another processor tries to read or write a word that it has a local copy of, such as invalidating or updating its cache entry. Nevertheless, caching only delays the problem of bus saturation. Instead of saturating at 4 to 8 processors, a well-designed single-bus system might saturate at 32 to 64 processors. Building a single bus system with appreciably more than 64 processors is not feasible with current bus technology.

The second general approach to building a multiprocessor is using some kind of

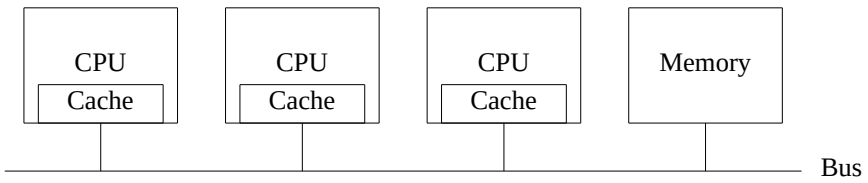


Fig. 4.2. A single-bus multiprocessor.

switching network, such as the *crossbar switch*, shown in Fig. 4.3(a). In this organization, each of the *n* processors can be connected to any one of the *n* memory banks via a matrix of little electronic switches. When switch *ij* is closed (by hardware), processor *i* is connected to memory bank *j* and can read or write data there. Since several switches may be closed simultaneously, multiple reads and writes can occur at the same time between disjoint processor-memory combinations.

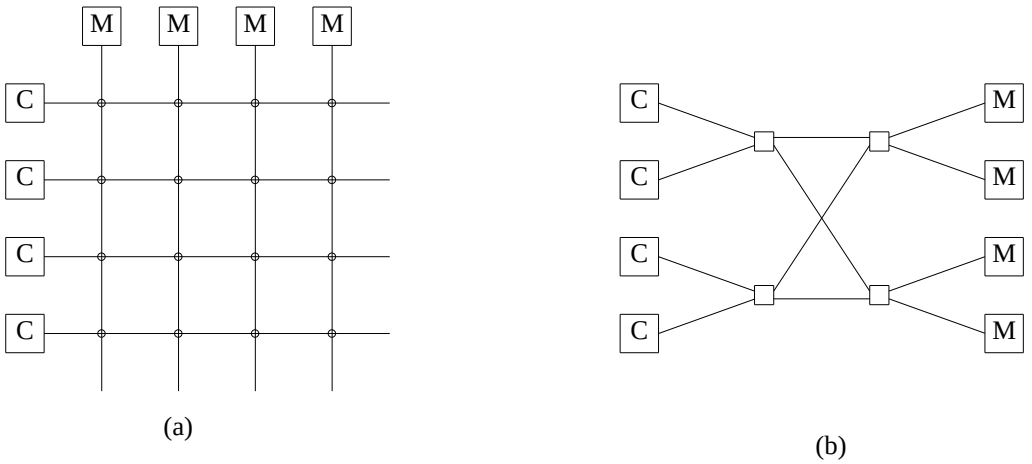


Fig. 4.3. (a) Crossbar switch. (b) Omega network. C is for CPU; M is for Memory.

The problem with the crossbar switch is that to connect *n* processors to *n* memory banks requires n^2 switches. As *n* becomes large, say 1024 processors and 1024 memories, the switch becomes prohibitively expensive and unmanageable.

An alternative switching scheme is to build a multiprocessor using the omega network shown in Fig. 4.3(b). In this figure, the CPUs are on the left and the memories are on the right. The omega network is a sophisticated packet switching network that connects them. To read a word of memory, a CPU sends a request packet to the appropriate memory via the switching network, which sends the reply back the other way.

Many variations of this basic design have been proposed, but they all have the problem that for a system with *n* processors and *n* memories, the number of switching

stages needed is on the order of $\log_2 n$, and the number of switching elements is in the order of $n \log_2 n$. As an example of what this means, consider a system of 1024 RISC processors running at 50 MHz. With $n = 1024$, 10 switches must be traversed from the CPU to the memory and 10 more on the way back. If this is to occur in 1 cycle (20 nsec), each switching step must take no longer than 1 nsec, and 10,240 switches are required. A machine with such a large number of very high speed packet switches is clearly going to be expensive and difficult to build and maintain.

Multiprocessor Software

In contrast to the multiprocessor hardware, which for large systems is complicated, difficult to build, and expensive, software for multiprocessors is relatively straightforward. Since all processes run within a single shared address space, they can easily share data structures and variables. When one process updates a variable and another one reads it immediately afterwards, the reader always gets the value just stored (memory coherence property).

To avoid chaos, co-operating processes must synchronize their activities. For example, while one process is updating a linked list, it is essential that no other process even attempt to read the list, let alone modify it. Many techniques for providing the necessary synchronization are well known. These include spin locks, semaphores, and monitors, and are discussed in any standard textbook on operating systems. The advantage of this organization is that sharing is easy and cheap and uses a methodology that has been around for years and is well understood.

4.1.2. Multicomputers

In contrast to the multiprocessors, which, by definition, share primary memory, *multicomputers* do not. Each CPU in a multicomputer has its own, private memory, which it alone can read and write. This difference leads to a significantly different architecture, both in hardware and in software.

Multicomputer Hardware

Just as there are bus and switched multiprocessors, there are bus and switched multicomputers. Fig. 4.4 shows a simple bus-based multicomputer. Note that each CPU has its own local memory, which is not accessible by remote CPUs. Since there is no shared memory in this system, communication occurs via message passing between CPUs. The ‘‘bus’’ in this example can either be a LAN or a high-speed backplane; conceptually, these two are the same, differing only in their performance. Since each CPU-memory pair is essentially independent of all the others, it is straightforward to build very large multicomputer systems.

Switched multicomputers do not have a single bus over which all traffic goes. Instead, they have a collection of point-to-point connections. In Fig. 4.5, we see two of the many designs that have been proposed and built: a grid and a hypercube. A grid is easy to understand and easy to lay out on a printed circuit board or chip. This architec-

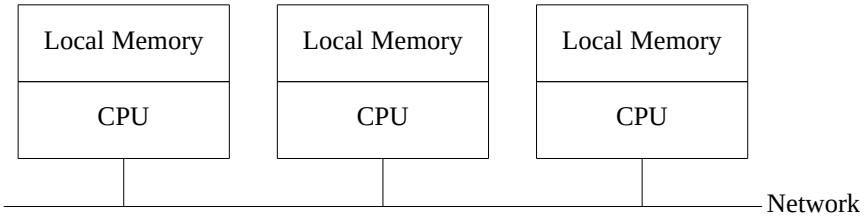


Fig. 4.4. A single-bus multicomputer.

ture is best suited to problems that are two dimensional in nature (graph theory, vision, etc.).

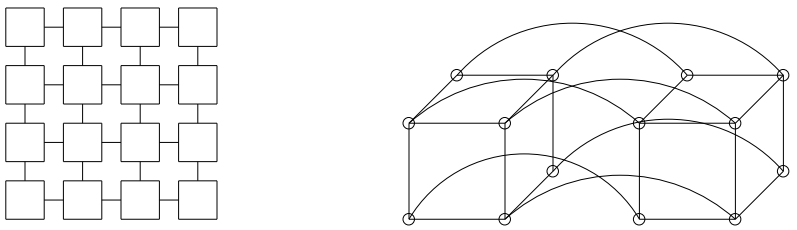


Fig. 4.5. Multicomputers. (a) Grid. (b) Hypercube.

Another design is a *hypercube*, which is an *n*-dimensional cube. One can imagine a 4-dimensional hypercube as a pair of ordinary cubes with the corresponding vertices connected, as shown in Fig. 4.5(b). Similarly, a 5-dimensional hypercube can be represented as two copies of Fig. 4.5(b), with the corresponding vertices connected, and so on. In general, an *n*-dimensional hypercube has 2^n vertices, each holding one CPU. Each CPU has a fan-out of *n*, so the interconnection complexity grows logarithmically with the number of CPUs.

Multicomputer Software

Since multicomputers do not contain shared memory (by definition), they must communicate by message passing. Various software paradigms have been devised to express message passing. The simplest one is to have two operating system primitives, SEND and RECEIVE. The SEND primitive typically has three parameters: the destination address, a pointer to the data buffer, and the number of bytes to be sent. In the simplest form, the RECEIVE primitive might just provide a buffer, accepting messages from any sender.

Many variations on this theme exist. To start with, the primitives can be *blocking* (synchronous), or *nonblocking* (asynchronous). With a blocking SEND, the sending process is suspended after the SEND until the message is actually accepted. With a

nonblocking SEND, the sender may continue immediately. The problem with allowing the sender to continue is that it may be in a loop, sending messages much faster than the underlying communication hardware can deliver them. Permitting this may result in lost messages.

A way to alleviate this problem somewhat is to have senders address messages not to processes, but to *mailboxes*. A mailbox is a special kind of buffer that can hold multiple messages. However, overflow is still a possibility.

A more fundamental problem with message passing is that, conceptually, it is really input/output. Many people believe that input/output should not be the central abstraction of a modern programming language. To hide the bare input/output Birrell and Nelson proposed RPC (see Section 1.3). Although this scheme hides the message passing to some extent, since the client and server each think of the other as an ordinary local procedure, it is difficult to make it entirely transparent. For example, passing pointers as parameters is difficult, and passing arrays is costly. Thus the programmer must usually be aware that the semantics of local procedure calls and remote ones are different.

In summary, although multiprocessors are easy to program, they are hard to build. Conversely, multicomputers are easy to build, but hard to program. What people would like is a system that is easy to program (i.e., shared memory) and that is also easy to build (i.e., no shared memory). How these two apparently contradictory demands can be reconciled is the subject of the rest of this chapter.

4.2. NUMA Machines

Various researchers have proposed intermediate designs that try to capture the desirable properties of both architectures. Most of these designs attempt to simulate shared memory on multicomputers. All of them have the property that a process executing on any machine can access data from its own memory without any delay, whereas access to data located on another machine entails considerable delay and overhead, as a request message must be sent there and a reply received.

Computers in which references to some memory addresses are cheap (i.e., local) and others are expensive (i.e., remote) have become known as *NUMA* (*NonUniform Memory Access*) machines. Our definition of NUMA machines is somewhat broader than what some other writers have used. We regard any machine which presents the programmer with the illusion of shared memory, but implements it by sending messages across a network to fetch chunks of data as NUMA. In this section we will describe three of the more interesting types of NUMA machines. They differ primarily in the granularity of access and the mechanism by which remote references are handled.

Word-Oriented NUMA Machines

One of the earliest machines was Cm*, built at Carnegie-Mellon University and consisting of a collection of LSI-11 minicomputers [Swan et al. 1977]. Each LSI-11 had a microprogrammed memory management unit (MMU) and a local memory. The MMU microcode could be downloaded when execution began, allowing part of the operating system to be run there. The LSI-11s were grouped together into clusters, the machines in each cluster being connected by an intracluster bus. The clusters were connected by intercluster buses.

When an LSI-11 referenced its own local memory, the MMU just read or wrote the required word directly. However, when an LSI-11 referenced a word in a remote memory, the microcode in the MMU built a request packet specifying which memory was being addressed, which word was needed, the opcode (READ or WRITE), and for WRITEs the value to be written. The packet was then sent out over the buses to the destination MMU via a store-and-forward network. At the destination, it was accepted, processed, and a reply generated, containing the requested word (for READ) or an acknowledgement (for WRITE).

The more remote the memory, the longer the operation, with the worst case being about 10 times the best case. It was possible for a program to run entirely out of remote memory, with a performance penalty of about a factor of 10. There was no caching and no automatic data movement. It was up to the programmer to place code and data appropriately for optimal performance. Nevertheless, to the programmer, this system had a single shared address space accessible to all processors, even though it was actually implemented by an underlying packet-switching network.

A more modern machine, which falls in the same class as Cm* is the BBN Butterfly [Dinning 1989]. The main difference between the Butterfly and Cm* is that the Butterfly has a more sophisticated interconnection network. The switching network implements among other things a collection of primitives for parallel processing.

Page-Oriented NUMA Machines

Cm* represents one end of the spectrum—sending requests for individual words from MMU to MMU in ‘‘hardware’’ (actually MMU microcode) over a set of tightly coupled backplane buses. At the other extreme are systems that implement virtual shared memory in software on a collection of workstations on a LAN. As in Cm*, the users are presented with a single shared virtual address space, but the implementation is quite different.

In its simplest form, the virtual memory is divided up into fixed size pages, with each page residing on exactly one processor. When a processor references a local page, the reference is done by the hardware in the usual way. However, when a remote page is referenced, a page fault occurs, and a trap to the operating system occurs. The operating system fetches the page, just as in a traditional virtual memory system, only now the page is fetched from another processor (which loses the page), instead of from the disk.

As in Cm*, pages are fetched by request and reply messages, only here they are generated and processed by the operating system, instead of by the MMU microcode. Since the overhead of the software approach is much higher, an entire page is transferred each time, in the hope that subsequent references will be to the same page. If two processors are actively writing the same page at the same time, the page will thrash back and forth wildly, degrading performance.

A significant improvement to the basic algorithm has been proposed, implemented, and analyzed by Li and Hudak [Li and Hudak 1989]. In their design, thrashing is reduced by permitting read-only pages to be replicated on all the machines that need them. When a read-only page is referenced, instead of sending the page, a copy is made, so the original owner may continue using it.

Li and Hudak present several algorithms for locating pages. The simplest is to have a centralized manager that keeps track of the location of all pages. All page requests are sent to it, and then forwarded to the processor holding the page. A variation of this scheme is to have multiple managers, with the leftmost *n* bits of the page number telling which manager is responsible for the page. This approach spreads the load over multiple managers. Even more decentralized page location schemes are possible, including the use of hashing or broadcasting.

It is worth pointing out that the basic page-oriented strategy could (almost) have been used with Cm*. For example, if the MMUs noticed that a particular page was being heavily referenced by a single remote processor and not at all being referenced by its own processor, the MMUs could have decided to ship the whole page to the place it was being used instead of constantly making expensive remote references. This approach was not taken because the message routing algorithm used the page number to locate the page. Modern machines, however, like Alewife [Chaiken et al. 1991], Dash [Lenoski et al. 1992], Encore GigaMax [Wilson 1987], KSR [Burkhardt et al. 1992], and Paradigm [Cheriton et al. 1991], basically follow this strategy, although some of them cache lines of words instead of pages.

NUMA systems have been designed where a page may be transported a maximum of only *k* times, to reduce thrashing [Ramachandran et al. 1989; Bolosky et al. 1989]. After that, it is wired down and all remote references to it are done as in Cm*. Alternatively, one can have a rule saying that once moved, a page may not be moved again for a certain time interval [Cox and Fowler 1989; Fleisch and Popek 1989; LaRowe Jr. et al. 1991].

Object-Based NUMA Machines

An inherent problem with page-oriented NUMA systems is the amount of data transferred on each fault is fixed at exactly one page. Usually this is too much. Sometimes it is too little. It is hardly ever exactly right. The next step in the evolution of NUMA machines is an object-based system, in which the shared memory contains well-defined objects, each one with certain operations defined on it [Cheriton 1985; Ahuja et al. 1986; Bisiani and Forin 1987; Black et al. 1987; Chase et al. 1989; Schwan

et al. 1989]. When a process invokes an operation on a local object, it is executed directly, but when it invokes an operation on a remote object, the object is shipped to the invoker. Alternatively, the operation name and parameters can be sent to the object, with the operation being carried out on the remote processor, and the result being sent back. Both of these schemes have the property that, in general, less unnecessary data are moved (unlike the page-oriented NUMA machines that move 1 Kbyte or 4 Kbyte or even 8 Kbyte, just to fetch a single byte).

Weak Ordering

A completely different approach to implement distributed shared memory is to weaken the ordering of read and write events, giving up the demand of coherence or sequential consistency [Dubois et al. 1988; Adve and Hill 1990]. For example, suppose three processes, *A*, *B*, and *C* all simultaneously update a word, followed by process *D* reading it. If coherence is required, then to carry out the three writes and the read, the word will probably have to be sent first to *A*, then to *B*, then to *C*, and finally to *D*.

However, from *D*'s point of view, it has no way of knowing which of the three writers went last (and thus whose value did not get overwritten). Bennett, Carter, and Zwaenepoel proposed for the Munin system that returning any one of the values written should be legal [Bennett et al. 1990; Carter et al. 1991]. The next step is to delay remote writes until a read is done that might observe the values written. It is even possible to perform an optimization and avoid some of the writes altogether. This strategy has been implemented in the Munin system, which uses hints from the programmer as to the access patterns of shared objects to make it easier to perform these and other optimizations.

A number of systems use a weak ordering in the implementation, but guarantee a sequentially-consistent memory model if programmers have synchronization points at the right places in their applications. Most of these systems are based on *release consistency* or on variants thereof [Gharachorloo et al. 1990; Carter et al. 1991; Bershad and Zekauskas 1991; Keleher et al. 1992; Lenoski et al. 1992]. By weakening the ordering of events in the implementation, these systems can achieve high performance by reducing the number of messages needed to implement the illusion of shared memory. The builders of these systems claim that adding the right synchronization points to the application programs is an easy task. Any carefully programmed parallel application already has them, as the cooperating tasks have to synchronize their activities anyway.

Other systems go even further in weakening the ordering [Ahamad et al. 1991; Delp et al. 1991]. Mether [Minnich and Farber 1990], for example, maintains a primary copy of each page and possibly one or more secondary copies. The primary copy is writable; the secondary copies are read-only. If a write is done to the primary copy, the secondary copies become obsolete and reads to them return stale data. Applications simply have to accept this.

Mether provides three ways for getting resynchronized: the owner of a secondary copy can ask for an up-to-date copy, the owner of the primary copy can issue an up-to-

date copy to the readers, or the owner of a secondary copy can just discard it, getting a fresh copy on the next read.

4.3. The Shared Data-Object Model

Weakening the semantics of the shared memory may improve the performance, but it has the potential disadvantage of breaking the ease of programming that the shared memory model was designed to provide. Mether, for example, offers only the syntax of shared variables, while forcing the programmer to deal with semantics even less convenient than message passing. On the other hand, approaches based on release consistency, which provide the programmer with a sequentially-consistent memory model on synchronization points, look like an attractive solution to the distributed shared memory problem.

In our research, we have devised an alternative model that preserves the convenience of (object-based) shared memory, yet has been implemented efficiently on multicomputers. The model, which we call *shared data-object model*, consists of four layers, as shown in Fig. 4.6.

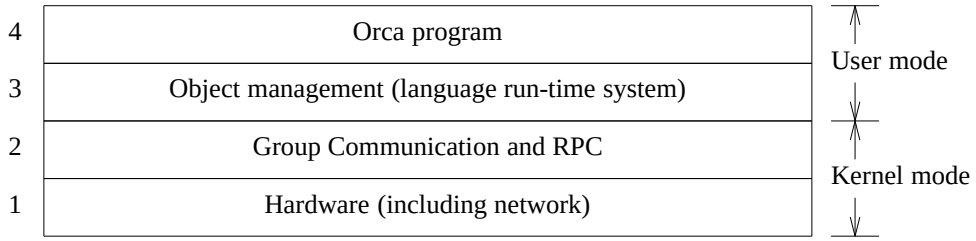


Fig. 4.6. Layers in the shared data-object model.

Layer 1 is the bare processor and networking hardware. Our scheme is most efficient on networks that support broadcasting (sending a message to all machines) or multicasting (sending a message to a selected group of machines) in hardware, but neither is required. Sending messages is assumed to be *unreliable*, that is, it is possible for messages to be lost.

Layer 2 is the software necessary to turn the *unreliable* communication offered by layer 1 into *reliable* communication. It implements two communication models: RPC for point-to-point communication and reliable group communication for 1-to-*n* communication. This is done using the protocols described in the previous two chapters. When layer 3 hands a message to layer 2 and asks for it to be reliably sent, layer 3 does not have to worry about how this is implemented or what happens if the hardware loses a packet. All of this is taken care of by layer 2.

Layer 3 is the language run-time system, which is usually a set of library procedures linked with the application program. Conceptually, programmers can have objects that are either PRIVATE or SHARED. The PRIVATE ones are not visible on other machines, so they can be accessed by direct memory reads and writes. In the simplest

RTS, the SHARED objects are replicated on all machines. Reads to them are local, the same as reads to PRIVATE objects. Writes are done by reliable broadcasting. An optimized version of the RTS uses both group communication and RPC. In the optimized RTS, SHARED objects are either replicated on all machines or stored on exactly one processor. Objects are entirely passive; they contain only data, not processes.

Layer 4 provides language support. Although it is possible for programmers to use the distributed shared memory by making calls directly on layer 3, it is much more convenient to have language support. At the Vrije Universiteit, we have designed a language, called *Orca*, which is especially designed to support parallel programming on distributed systems. In Orca, programmers can declare shared objects, each one containing a data structure for the object and a set of procedures that operate on it. Operations applied on single shared objects are indivisible, and are executed sequentially consistent. The next section discusses Orca in more detail.

4.4. Orca Programs

The goal of this section is to give just enough detail to make the rest of the chapter understandable. We start by describing the language. Then, we describe the interface between the compiler and the RTS.

4.4.1. The Language

Orca is a procedural language based on shared data-objects. Processes in Orca communicate through these objects, which are instances of abstract data types. Processes can share objects even if they run on different machines. The objects are accessed solely through the operations defined by the abstract data type.

An object type (or abstract data type) definition in Orca consists of a specification part and an implementation part. As a simple example, an object type *IntObject* encapsulating a single integer may be specified as follows:

```
object specification IntObject;
    operation Value(): integer;              # return value
    operation Assign(v: integer);            # assign new value
    operation AwaitValue(v: integer);        # wait until object has value v
end;
```

The specification part gives the operations that can be applied to objects of this type. An Operation is a set of guarded commands [Dijkstra 1975], which may block initially. The implementation part contains the data used to represent instances (variables) of the type, the code for implementing operations, and code for initializing variables of the type. The implementation part of type *IntObject* might be as shown in Fig. 4.7.

Objects are created by declaring variables of an object type. It is also possible to declare a dynamic array of objects, so the number of objects can vary at run time:

```
object implementation IntObject;
    x: integer;                                  # internal data

    operation Value(): integer;
    begin
        return x;                                # return current value
    end;

    operation Assign(v: integer);
    begin
        x := v;                                  # assign new value
    end;

    operation AwaitValue(v: integer);
    begin
        guard x = v do od;                       # wait
    end
begin
    x := 0;                                      # initialize object to zero
end;
```

Fig. 4.7. Implementation module for *IntObject*.

```
X: IntObject;                          # create (declare) an object
V: array[integer 1 .. N] of IntObject; # array of N objects
```

An operation is applied to an object using the following notation:

```
X$assign(3);                 # apply operation ''assign'' to object X
tmp := V[3]$value();         # apply operation ''value'' to object V[3]
```

An Orca program also defines one or more process types, which are similar to procedure declarations:

```
process p(n: integer; X: shared IntObject);
begin
    ...
end;
```

Processes of type *p* take two parameters: a value parameter *n* and a *shared* (similar to *call-by-reference*) parameter *X*. Only objects may be passed as shared parameters.

Processes are created dynamically, through a **fork** statement, which specifies the process type and actual parameters. Optionally, it can also specify the processor on which the new process is to be run. Processors are numbered sequentially, starting at 0. This notation is mainly used for assigning different processors to processes that must run in parallel. If no processor is specified, the new process runs on the same processor as its parent. The statement

fork p(23, X) **on**(3);

creates a new process on processor 3, passing 23 as value parameter and object *X* as a shared parameter.

Initially, an Orca program consists of one process, called *OrcaMain*, which runs on processor 0. This process may create objects, fork off child processes, and pass the objects as shared parameters to the children. Each child can pass these objects to its children, and so on. In this way, a hierarchy of processes is created, which communicate through shared objects.

The semantics of the model are straightforward:

1. All operations are applied to single objects.
2. All operations are executed sequentially consistent and indivisibly.

The indivisibility property simplifies programming, since users do not have to worry about what happens if two operations are applied simultaneously to the same object. In other words, mutual exclusion synchronization is done automatically.

The first property is a restriction, since it rules out atomic operations on collections of objects. This restriction, however, makes the model efficient to implement, because no two-phase update protocols [Eswaran et al. 1976] are needed. As it turns out, parallel applications seldom need atomic operations on multiple objects. If needed, however, they can be constructed in Orca, by building them as sequences of simple operations. In this case, the programmer must explicitly deal with synchronization (e.g., by defining a lock object).

Orca excludes any features that make a distributed implementation difficult, that make compile-time optimizations hard, or that potentially violate type security [Hoare 1981; Strom and Yemini 1986]. For example, it does not support global variables, pointers, and goto statements. Instead of pointers, it has a new type constructor, called a *graph* [Bal 1990].

4.4.2. The Interface between the Compiler and the Run-Time System

The application programs in layer 4 are translated by the Orca compiler into executable code for the target system†. The code produced by the compiler contains calls to RTS routines that manage processes, shared data-objects, and complex data structures (e.g., dynamic arrays, sets, and graphs). In this chapter, we will only discuss how operation invocations are compiled for a multicomputer environment.

As described above, it is very important to distinguish between *read* and *write* operations on objects. The compiler therefore analyses the implementation code of each operation and checks whether the operation modifies the object to which it is

† We assume the target system does not contain multiple types of processors. Although a heterogeneous implementation of Orca is conceivable, we do not address this issue here.

applied†. In most languages, this optimization would be difficult to implement. Consider, for example, a Pascal statement containing an indirect assignment through a pointer variable:

```
p^.f := 0;
```

It is hard to determine which data structure is affected by this statement. Orca does not have this problem, since the name of the data structure is given by the programmer. The Orca equivalent of the Pascal code given above would look like:

```
G[p].f := 0;
```

which explicitly specifies the name of the data structure that will be modified. So, in Orca the compiler can determine which operations modify which object's data structures and which do not.

The compiler stores information about operations in an *operation descriptor*. This descriptor specifies the type of the operation (read or write), the sizes and modes (input or output) of the parameters of the operation. If an Orca program applies an operation on a given object, the compiler generates a call to the RTS primitive *INVOKE*. This routine is called as follows:

```
INVOKE(object, operation-descriptor, parameters ...);
```

The first argument identifies the object to which the operation is to be applied. (It is a network-wide name for the object.) The second argument is the operation descriptor. The remaining arguments of *INVOKE* are the parameters of the operation. The implementation of this primitive is discussed below.

4.5. Object Management

Layer 3 implements the Orca RTS. As mentioned above, its primary job is to manage shared data-objects. In particular, it implements the *INVOKE* primitive. For efficiency, the RTS replicates objects so it can apply operations to local copies of objects whenever possible.

There are many different design choices to be made related to replication, such as where to replicate objects, how to synchronize write operations to replicated objects, and whether to update or invalidate copies after a write operation. We have looked at many alternative strategies [Bal et al. 1992b]. The RTS described in this section uses full replication of objects, updates replicas by applying write operations to all replicas, and implements mutual exclusion synchronization through a distributed update protocol.

The full replication scheme was chosen for its simplicity and good performance for many applications. This implementation is tailored to applications with a medium

---

† The actual implementation is somewhat more complicated, since an operation may have multiple guards (alternatives), some of which may be read-only.

grain of parallelism and a high ratio of read operations versus write operations. In the next section, we discuss a scheme that does either full replication or no replication on a per object basis, depending how an object is used by the Orca program. This implementation is more complicated, but performs well on a wider range of applications. Another alternative is to let the RTS decide dynamically where to store replicas. This strategy is employed in another implementation of Orca [Bal et al. 1992b].

We have chosen to use an update scheme rather than an invalidation scheme for two reasons. First, in many applications, objects contain large amounts of data (e.g., a 100K bit vector). Invalidating a copy of such an object is wasteful, since the next time the object is replicated its entire value must be transmitted. Second, in many cases, updating a copy will take no more CPU time and network bandwidth than sending invalidation messages.

The semantics of shared data-objects in our model define that simultaneous operations on the same object must conceptually be executed in some sequential order. The exact order in which they are to be executed is not defined, however. If, for example, a read operation and a write operation are applied to the same object simultaneously, the read operation may observe either the value before or after the write, but not an intermediate value. However, all processes having access to the object must see the events happen in the same order. The RTS described here solves the inconsistency problem by using a distributed update protocol that guarantees that all processes observe changes to shared objects *in the same order*. One way to achieve this would be to lock all copies of an object prior to changing the object. Unfortunately, distributed locking is quite expensive [Eswaran et al. 1976]. Our update protocol does not use locking. To avoid locking, the reliable and totally-ordered broadcast primitive, *SendToGroup*, described in the previous chapter, is used.

The RTS uses an *object manager* for each processor. The object manager is a lightweight process (thread) that takes care of updating the local copies of all objects stored on its processor. The object managers form a group when they start up. Objects (and replicas) are stored in an address space shared by the object manager and user processes. User processes can *read* local copies directly, without intervention by the object managers. Write operations on shared objects, on the other hand, are marshaled and then broadcast to all the object managers in the system. A user process that broadcasts a write operation suspends until the message has been completely handled by its local object manager. This is illustrated in Figure 4.8.

Each object manager maintains a queue of messages that have arrived but that have not yet been handled. As all processors receive all messages in the same order, the queues of all managers are the same, except that some managers may be ahead of others in handling the messages at the head of the queue.

The object manager of each processor handles the messages in its queue in strict FIFO order. A message may be handled as soon as it appears at the head of the queue. A message *GlobalOperation(obj, op, parameters)* is handled by removing it from the queue, unmarshaling it, locking the local copy of the object, applying the operation, and

```
INVOKE(obj, op, parameters)
    if op.ReadOnly then                          # check if it's a read operation
        set read-lock on local copy of obj;
        call op.code(obj, parameters);           # do operation locally
        unlock local copy of obj
    else
        SendToGroup GlobalOperation(obj, op, parameters);
        block current process;
    fi;
```

Fig. 4.8. The simplified code for the *INVOKE* run-time system primitive. This routine is called by user processes.

finally unlocking the copy. If the message was sent by a process on the same processor, the manager unblocks that process (see Fig. 4.9).

```
receive GlobalOperation(obj, op, parameters) from W →
    set write-lock on local copy of obj;
    call op.code(obj, parameters);               # apply operation to local copy
    unlock local copy of obj
    if W is a local process then
        unblock(W);
    fi;
```

Fig. 4.9. The simplified code to be executed by the object managers for handling *GlobalOperation* messages.

The full replication scheme does not provide memory coherence, because if machine *A* initiates a reliable broadcast to update a shared object, and machine *B* reads the (local copy of the) same object a microsecond later, *B* will get the old value. On the other hand, it does provide for indivisible update and sequential consistency, which is almost as good, as can easily be seen in the following example. Consider a multiprocessor with a coherent shared memory. At a certain moment, process *A* wants to write a word, and process *B* wants to read it. Although the two operations may take place a microsecond apart, the value read by *B* depends on who went first. Thus although the memory is coherent, the value read by *B* is determined by the detailed timing. The shared object model has a similar property. In both cases, programs whose correct functioning depends on who wins the race to memory are living dangerously, at best. Thus, although our memory model does not exhibit coherence, it does provide sequential consistency and this is sufficient.

4.6. Optimized Object Management

The Orca implementation described above is tailored to programs that exhibit a high read/write ratio or that transfer data from one process to all the other processes. Good examples are programs solving the Traveling Salesman Problem and the All-

Pairs Shortest Paths problem, which both obtain high speedups using this method (see Section 4.8).

Unfortunately, there are also cases where the approach is less efficient. As an example, consider an application like Successive Overrelaxation (SOR) [Stoer and Bulirsch 1983], which primarily requires point-to-point message passing. This form of communication is expressed in Orca by defining a message buffer object type with the following specification:

```
object specification Buffer;
        operation send(m: msg);          # asynchronous send
        operation receive(): msg;        # blocking receive
end;
```

If two processes (P_1 and P_2) need to communicate through message passing, they are both passed the same shared *Buffer* object to which they can send and receive messages.

With the unoptimized run-time system, these objects will be updated through broadcast messages. If process P_1 wants to send a message to P_2, it applies the *send* operation to the shared buffer. This operation will be sent using group communication, so not only P_2's machine receives this message, but so do all other processors. In other words, point-to-point communication does not really exist in the replicating system. All communication ultimately takes place through group communication. For applications like SOR, this uniform approach has a performance penalty.

As another example, consider a master/slave program in which the master generates work and stores it in a *job queue*. The slaves repeatedly take a job from the queue and execute it. The job queue can be implemented in Orca as a shared object with *put* and *get* operation, similar to the message buffer described above. Unlike a buffer, the job queue will be shared among many (or all) processes. Still, replicating the job queue is not really desirable, because all operations on the queue are write operations. Both the *put* and the *get* operations modify the queue's data, so the read/write ratio of this object will be zero. It would be more efficient not to replicate the object at all.

The conclusion is that the full replication scheme is efficient, except in some (important) cases. The goal for the optimized implementation is to let the compiler and RTS recognize these inefficient cases and disable replication for them. For each object, the system decides:

- Whether or not to replicate the object.
- For nonreplicated objects: *where* to store the object.

So, objects are either replicated everywhere or are stored on exactly one processor. As we will see, the two decisions may change dynamically, when new processes are created.

An important design issue is where to make these decisions, in the compiler or in the RTS. Although in many cases the compiler is able to make the right decision without any help from the RTS, there are two disadvantages of such an approach. First, the compiler will have to do complicated analysis, in particular when dynamic arrays of objects are used. Second, and more fundamental, the optimal replication strategy will in general depend on the underlying hardware architecture and communication protocols. The compiler would then become architecture dependent, which we want to avoid.

With the approach described here, the compiler still does the bulk of the work, but the RTS makes the actual decisions. For each type of process, the compiler analyzes how processes of this type access shared objects. If a process creates an object itself, the compiler generates a call to the run-time routine *Score*, which tells the RTS how the process is going to use the new object. In addition, the compiler generates a *process descriptor* for every process type, showing how processes of this type will access parameters that are shared objects.

As a trivial example, consider the declaration of a process type *AddOne* that takes two shared objects *X* and *Y* as parameters:

```
process AddOne(X, Y: shared IntObject);
begin
    Y$Assign(X$Value() + 1);      # X is read once, Y is written once
end;
```

The compiler will determine that processes of type *AddOne* will read their first parameter once and write their second parameter once. The compiler stores this information in the process descriptor for *AddOne*. After a fork statement of the form "**fork** AddOne(A, B);" the RTS uses this descriptor to determine that the new process will read *A* once and write *B* once.

Equipped with this information about new processes and objects, the RTS chooses a suitable replication strategy for each object. The implementation of the optimized RTS is described in Section 4.7.2. Since the RTS is distributed among multiple processors, some interprocess communication will be needed during the decision making. As we will see, however, the number of messages needed is small and the communication overhead is negligible.

Let us now describe in more detail the information that the optimizing compiler passes to the RTS and the method used to compute this information. For the optimization described in this chapter it would be sufficient just to estimate the *ratio* of read operations and write operations executed by each process. For more advanced optimizations, however, it is useful to have the access patterns themselves available. For some optimizations it makes a difference whether a process alternates read operations with write operations or first does *n* reads and then *n* writes. In both cases the read/write ratio will be 1, but the access patterns are different.

The optimizing compiler therefore computes, for each process, the actual access

pattern. It first generates a description of how the process reads and writes its shared objects, taking into account the control flow of that process. In the current implementation, the compiler computes based on this information two values for each object the process can access:

Nreads An estimate of the number of read operations.

Nwrites An estimate of the number of write operations.

In the future, we may want to pass the actual access patterns to the RTS, to be able to make better decisions at run-time.

4.7. The Optimizing Compiler and Run-Time System

In this section we describe the implementation of the optimizing compiler and the optimizing RTS.

4.7.1. The Optimizing Compiler

We will first describe the read-write patterns used by the new compiler, and then discuss how the patterns are generated and analyzed. The patterns give a static description of how each Orca process accesses its shared objects. The pattern does not contain information about nonshared objects and normal (local) variables. As an example, consider the Orca code fragment of Fig. 4.10.

```
function foo(X: shared IntObject);
        i: integer;
begin
        for i in 1 .. 100 do
                X$Assign(i);
        od;
end;

process bar(A, P: shared IntObject);
        tmp: integer;
begin
        P$Assign(10);
        if f(tmp) then tmp := A$Value();
        else foo(A);
        fi;
end;
```

Fig. 4.10. An Orca code fragment.

The read-write pattern for process *bar* is:

process bar: #2$W ; [#1$R | {#1$W}]

which specifies that the process will first write its second parameter (‘‘#2’’) and then either read its first parameter once or it will repeatedly write its first parameter. (Selection is indicated by square brackets and the vertical bar; for repetition, curly brackets are used.) The original *Value* and *Assign* operations defined by the programmer have

been replaced by an *R* (for Read) and a *W* (for Write) respectively. The function *foo* is absent in the pattern, although its effects (repeatedly writing *A*) are included. The patterns are essentially regular expressions. The advantage of regular expressions is their simplicity, making their analysis easy. A disadvantage is the loss of information that sometimes occurs. For example, the pattern for a for-statement does not show how many times the loop will be executed, even though this information can sometimes be determined statically.

The compiler computes the read-write patterns in two phases. It first performs a local analysis of each process and function, and then computes the final patterns by simulating inline substitution. Computing the patterns is straightforward, since features that make control flow and data flow analysis difficult, such as ''goto'' statements, global variables, and pointers have been intentionally omitted from Orca. Also, process types are part of the language syntax, so techniques such as separating control flow graphs are not needed [Jeremiassen and Eggers 1992]. The only difficulty is handling recursive functions. The current compiler stops simulating inline substitutions after a certain depth. A more advanced implementation should at least handle tail-recursion similarly to iteration.

The second major component of the optimizer is the pattern analyzer. It analyzes the pattern generated for each process type and determines the *Nreads* and *Nwrites* values for the shared objects the process can access. It considers each process type in turn. A process of this type can access any shared objects declared by itself or passed to it as a shared parameter. For each such object the analyzer determines how the object is used by the process. It computes an indication of how many times the process will read and write the object. It uses the heuristic that operations inside a loop will be executed more frequently than operations outside a loop. Also, operations inside a selection (**if** statement) are less likely to be executed than operations outside a selection.

4.7.2. The Optimizing Run-Time System

Using the information passed in a process descriptor, the RTS decides whether or not to replicate objects and where to store those objects that are not replicated. If a process invokes an operation on a nonreplicated shared object stored on a remote processor, the optimized RTS sends the operation and its parameters to this machine, asking it to execute the operation and return the results. For a nonreplicated shared object stored locally, the RTS simply executes the operation itself, without telling other processors about this event.

Which strategy is better depends on the underlying architecture and communication protocols. If the group communication primitives are using the Ethernet hardware multicast, updates are relatively cheap, making replication relatively attractive. For architectures that do not support physical multicast, the balance is likely to be different.

The RTS on each processor maintains state information for each shared object. The state of an object X includes the following information for *every* processor P:

1. The total number of reads for all processes on P.
2. The total number of writes for all processes on P.

Both numbers are estimates, based on the *Nreads* and *Nwrites* values generated by the compiler. Each processor keeps track of how every processor uses each object. This information is updated on every **fork** statement. Although a **fork** statement creates a process on only one processor, **fork** statements are always broadcast, using the same indivisible broadcast protocol as for operations. Hence, the state information on all processors is always consistent and all processors will make the same decisions.

As an example, assume processor 0 executes a statement

```
fork AddOne(A, B) on(2);     # AddOne was defined in Section 4.6.
```

The RTS on processor 0 broadcasts a message ''[FORK, AddOne, A, B, 2]'' to all processors. Each RTS receives this message and updates the state information for objects A and B. On each processor the *Nreads* value for object A is incremented by one, since A is read once by *AddOne*. Also, the *Nwrites* value of B is incremented by one, since B is written once. Based on this updated information, all processors now reconsider their decisions for A and B, using heuristics explained below. If A was not replicated before, the improved read/write ratio may now reverse this decision. Likewise, if B was replicated, the lower ratio may now cause it to drop the replicas. In the latter case, all processors also decide where to store the single copy of B. If A or B were stored on only one processor, the system may also decide to migrate these copies.

In any case, due to the globally-ordered reliable broadcast, all processors come to the same decisions. If, for example, they decide to no longer replicate B but store it only on processor 5, all processors except for processor 5 will delete their copies of B. The processor on which the new process is forked (processor 2 in our example) will also install copies of A and B (if required) and create a new process.

The heuristic for choosing the replication strategy is quite simple. Let R_i and W_i be the *Nreads* and *Nwrites* values for processor i for a given object X. First, if X is not replicated we store it on the processor that most frequently accesses it, which is the processor with the highest value for $R_i + W_i$. This processor is called the *owner* of X.

To decide whether or not to replicate object X, we compare the number of messages that will be generated with and without replication. With replication, one broadcast message will be generated for each write operation by any process. We compute the number of write operations by summing W_i for all i. Without replication, one Remote Procedure Call will be needed for every access by any processor except for the owner. We determine the processor that would become the owner (i.e., has the highest value for $R_i + W_i$) and sum $R_i + W_i$ for all i except this owner. Finally, we compute the total communication costs for both cases, using the average costs for RPC and broadcast messages. For the platform we use (see Section 5), an RPC costs about 2.5 msec and a broadcast 2.7 msec. We compare these costs and choose the strategy that has the least communication overhead.

The overhead of this decision-making process is quite small. The number of objects in Orca programs is usually small, so the state information requires little memory. The state is only updated after **fork** statements†. As Orca programs use course-grained parallelism, such statements occur infrequently. With replicated workers parallelism [Andrews 1991], for example, there is about one **fork** statement for every processor. The only disadvantage of the current implementation is that all **fork** statements must be broadcast.

Since the replication strategy may change dynamically, new mechanisms had to be implemented in the RTS. The optimized RTS must be able to drop the replicas of a replicated object, to replicate an unreplicated object, and to migrate an object from one processor to another. The latter case occurs if the RTS decides to change the owner of a nonreplicated object. The protocols implementing these mechanisms are relatively straightforward, as Orca has only passive objects and since simple communication primitives with clean semantics (RPC and indivisible broadcast) are used. In Emerald, for example, object migration is more complex, because objects may contain processes [Jul et al. 1988].

4.8. Example Applications and Their Performance

We will now look at the performance of Orca programs for three typical example applications. We will give performance figures for both the unoptimized and optimized RTS. The applications differ widely in their need for shared data. The first one, the Traveling Salesman Problem, greatly benefits from the support for shared data. The second application, the All-Pairs Shortest Paths Problem, benefits from the use of broadcast communication, even though it is hidden in the implementation of Orca. Finally, Successive Overrelaxation merely requires point-to-point communication, but still can be implemented in Orca by simulating message passing. The Orca code for the applications can be found in [Bal 1990; Bal et al. 1990].

For each program, we will first describe the access patterns produced by the pattern generator, as well as the decisions made by the RTS. Next, we give the speedups and absolute running times for the program, with and without the optimization. The measurements are taken on a collection of 20-Mhz MC68030s connected by a 10 Mbit/s Ethernet. The performance is measured by taking the time before the first statement of the program is executed and by taking the time after the last statement of the program is executed and subtracting. Thus, the running times include the time to fork processes. I/O, however, is not measured. Then, we will analyze the performance of the Orca program by looking at the communication overhead for the program executed on 16 processors and by comparing it with the equivalent C program.

As an aside, a RTS based on only RPC also has been implemented [Bal et al. 1992b]. For all the applications discussed in this chapter, the optimized RTS performs

† Strictly speaking, the state should also be updated after a process exits, but we did not implement this yet. For all applications mentioned in this paper, however, all processes continue to exist until the whole program terminates, so for these applications there would be no difference in performance.

as good or better. This comes from the fact that when a truly shared object has to be updated, the group RTSs only have to send a single multicast message, while the RPC RTS has to perform multiple RPCs. Although these RPCs are executed in parallel, the performance impact increases with the number of processors. We will not mention the RPC-based RTS further.

4.8.1. The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) requires to find the shortest route for a salesman to visit each city in a given set exactly once. The problem is solved in Orca using a master/slave type of program based on a branch-and-bound algorithm. The master generates partial routes and stores them in a job queue. Each slave repeatedly takes a job (route) from the queue and generates all possible full paths starting with the initial route and using the ''closest-city-next'' heuristic. All slaves keep track of the current shortest full route. As soon as a slave finds a better route, it gives the length of the route to all other processes. This value is used to prune part of the search tree. The Orca program uses three shared objects:

q	The job queue, containing all the jobs that have been generated by the master but that have not yet been handled by a slave.
min	The length of the best route found so far.
nworkers	The number of active worker processes. The master does not print the final result until this count drops to zero.

The patterns generated for the master (*OrcaMain*) and slaves are as follows.

```
process OrcaMain:                          # OrcaMain is the master
        min$W;                             # initialize global minimum
        nworkers$W;                        # initialize workers count
        {q$W};                             # generate the jobs
        nworkers$R;                        # wait until count drops to zero
        min$R;                             # get final value of minimum

process slave:
        {
                #1$W;                      # get job & delete it from q
                [#2$R; [#2$W | [{[#2$R; [#2$W|]|]}|]|]|] # execute job
        }
        #3$W;                              # decrement workers count
```

The second pattern for a slave process is somewhat complicated, because the slave process uses a recursive function. The pattern generator estimates the effect of this function by expanding it up to a certain depth. (Here we have used depth 2; in practice, a higher depth is used.) Also note that the notation ''[pattern |]'' represents an **if**-

statement (selection); the **then**-part is represented by the given pattern and the **else**-part does not contain any operation invocations.

First consider the job queue. Both the master and the slaves apply only write operations to this object, since adding and deleting jobs both change the queue's data structures. Therefore, object *q* of the master and the first parameter of the slaves will be assigned positive values for *Nwrites*. The *Nreads* value for these objects will be zero. The RTS will therefore decide not to replicate this object. The master process runs on the same processor as one of the slave processes (since the master is not computationally expensive). The RTS will store the queue object on this processor, since it contains two processes that write the object.

Now consider the second object, *min*, which contains the global minimum. The master reads and writes the minimum exactly once. The pattern for the slaves is complicated, but it is still easy to see that the average read/write ratio for this object (parameter #2) is higher than 1. (Within the outermost selection, the pattern contains a read operation, followed by a nested selection; the write operation in this nested selection can only be executed if the read operation has been executed first). In the slave processes, the *Nreads* value will therefore be higher than the *Nwrites* value. Consequently, the RTS will decide to fully replicate this object.

Finally, the third object, *nworkers*, will not be replicated, because it clearly is written more frequently than it is read. Although this is the right decision, it does not really improve performance, because the object is infrequently used.

The net effect of the compiler and run-time optimizations is that the job queue is no longer replicated but stored on the processor where the master (*OrcaMain*) runs. This decision reduces the communication overhead of the program, because *OrcaMain* can generate jobs locally (without any interprocess communication) and because a slave can now get a job without troubling other processors. In other words, the *get* operation on the job queue will be sent point-to-point instead of being broadcast. The global minimum is replicated, which is important since, in practice, it is read much more frequently than it is written [Bal et al. 1990].

The impact of this optimization on the speedup for a problem with 14 cities is shown in Figure 4.11. (For this problem, the master generates 1716 jobs, each containing a partial route with 4 cities.) The performance improvement is significant, especially for a large number of processors. The optimized RTS even achieves superlinear speedup for this specific set of input data. This is due to the fact that one processor finds quickly a close to optimal shortest path, which other processors use to prune parts of their search tree.

Most TSP problems we generated give superlinear speedup, which indicates that our single-processor algorithm (based on the "closest-city-next" heuristic) is not optimal. We certainly do not claim that the superlinear speedup is due solely to our optimizations. The point we do wish to make, however, is that the optimized system performs significantly better than the nonoptimized one.

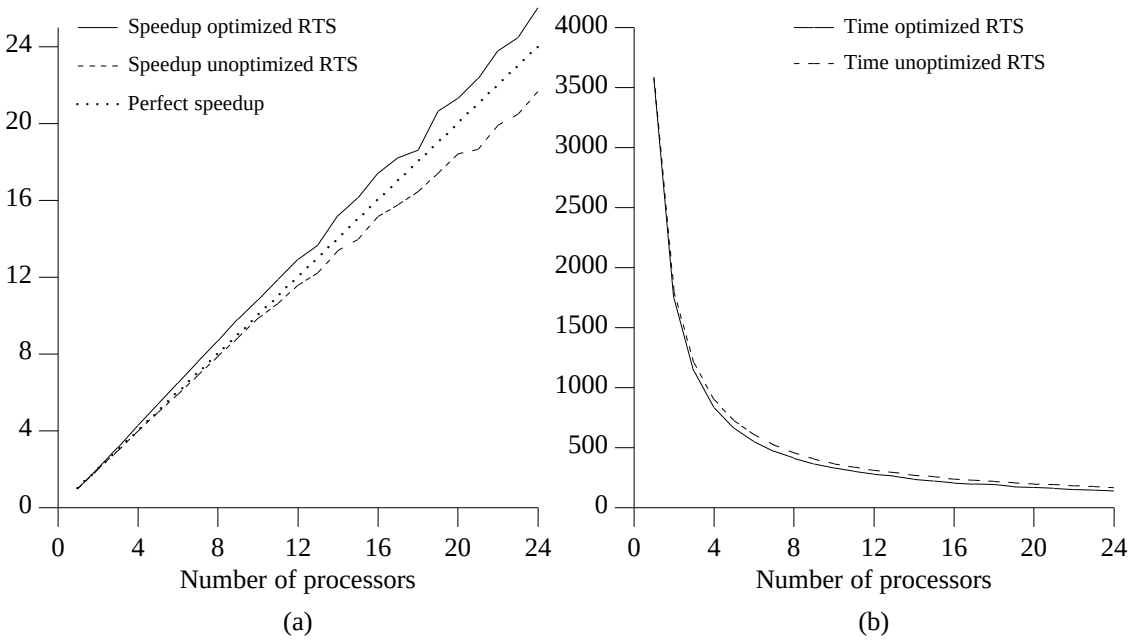


Fig. 4.11. Performance for the Traveling Salesman Problem using a 14-city problem. (a) Speedups for each implementation of the RTS. (b) Absolute running times in seconds for each implementation of the RTS.

To analyze the communication overhead for the TSP program, the RTS keeps statistics about the number of messages that are sent and their sizes. Figure 4.12 (a) shows the number of messages sent by the unoptimized RTS on processor 0, running both *OrcaMain* and a slave process and (b) shows the number of messages sent by the unoptimized RTS running only a slave process. Figures (c) and (d) show the numbers for the same processors, using the optimized RTS. The communication overhead is depicted per primitive: 2 rows for group communication (sending and receiving) and 4 rows for RPC (at the server side a request and a reply and at the client side a request and a reply). Rows containing only zeros are omitted from the table.

From the number of messages one can clearly see that the optimized RTS reduces the communication overhead. Consider the processor that is running *OrcaMain*. The unoptimized RTS sends 1809 group messages to implement the job queue, while the optimized RTS sends 1618 point-to-point RPCs. The slave processes in both RTSs send approximately the same number of messages (both RTSs have to send one message to get a job), but the optimized RTS receives fewer messages. Like the master process, the slave processes in the optimized RTS also incur less communication overhead.

Another interesting question is how the performance of this parallel Orca program compares with the performance of a program solving the same problem programmed in a standard sequential programming language. Figure 4.13 gives the performance figure for TSP written in C. To make a fair comparison, we ran the C pro-

Primitive	Message size (byte)						
	0	8-15	16-31	64-127	128-255	512-1023	1024-2047
Send group	1	2	0	93	1716	1	15
Receive group	16	17	9	1718	1716	1	15

(a)

Primitive	Message size (byte)						
	0	8-15	16-31	64-127	128-255	512-1023	1024-2047
Send group	0	1	0	152	0	0	0
Receive group	10	17	9	1718	1716	1	15

(b)

Primitive	Message size (byte)							
	0	8-15	16-31	32-63	64-127	128 - 255	512 - 1023	1024 - 2047
Receive request	0	0	0	0	15	1603	0	0
Send reply	0	0	0	15	1603	0	0	0
Send group	1	1	0	0	1	0	1	15
Receive group	16	16	9	0	1	0	1	15

(c)

Primitive	Message size (byte)							
	0	8-15	16-31	32-63	64-127	128 - 255	512 - 1023	1024 - 2047
Send request	0	0	0	0	1	123	0	0
Receive reply	0	0	0	1	123	0	0	0
Send group	0	1	0	0	0	0	0	0
Receive group	10	16	9	0	1	0	1	15

(d)

Fig. 4.12. Communication overhead TSP in terms of number of messages and their sizes. (a) The number of messages and their sizes for the processor running both *OrcaMain* and a slave process. (b) The number of messages and their sizes for a processor running only a slave process. Figures (c) and (d) show the equivalent numbers for the optimized RTS.

gram on the same hardware with the same operating system (Amoeba) as the parallel Orca program. The C compiler used is the ACK C compiler. (The Orca compiler is also made using the ACK compiler kit.) Furthermore, we measured the performance of the same C program on a state-of-the-art processor (SPARCstation1+), running a standard operating system (SunOS4.1.1), and compiled with a state-of-the-art C compiler (GNU C compiler 2.1).

From the numbers one can conclude that the Orca program running on a single processor is 6.6 times as slow as the same program on Amoeba written in C. There are two reasons for this difference in performance. First, the Orca program is written as a parallel program and uses shared objects, which have a considerable overhead. For example, to read the object *min*, a number of checks and procedure calls are performed, while in the C program the minimum can be read using one single instruction. Second, the Orca compiler is a prototype. Many of the optimizations that the compiler could do are not implemented yet. For example, the implementation of objects, arrays, and graphs incur considerable overhead. We are currently working on a redesign of the interface between the compiler and RTS, and the implementation of a new compiler. We expect that much of the additional overhead for the Orca language constructs will disappear.

Compiler	Operating System	Processor	Time (sec)
Orca	Amoeba	MC68030	3577
ACK cc	Amoeba	MC68030	546
GNU cc	SunOS4.1.1	SPARC	73

Fig. 4.13. Performance of TSP on a single processor.

4.8.2. The All-Pairs Shortest Paths Problem

The second application we consider is the All-pairs Shortest Paths problem (ASP). In this problem, it is desired to find the length of the shortest path from any node *i* to any other node *j* in a given graph with *N* nodes. The parallel algorithm we use is similar to the one given in [Jenq and Sahni 1987], which itself is a parallel version of Floyd's algorithm.

The distances between the nodes are represented in a matrix. Each processor contains a *worker* process that computes part of the result matrix. The parallel algorithm performs *N* iterations. Before each iteration, one of the workers sends a *pivot row* of the matrix to all the other workers. Since the pivot row contains *N* integers and is sent to all processors, this requires a nontrivial amount of communication.

The pattern for the worker processes is shown below (*OrcaMain* does not access any of the important shared objects):

```
process worker:
    {[#1$W | #1$R]}              # parameter 1 is the object Rowk
```

OrcaMain forks a number of workers, passing an object called *Rowk* as a shared parameter (parameter #1). This object is used for transferring the pivot rows. The process containing the pivot row stores this row into the shared *Rowk* object, where it can be read by all other processes.

The pattern for the workers clearly reflects this style of communication. A worker executes zero or more iterations, and during each iteration it either reads or writes the *Rowk* object. A worker writes the object if it contains the pivot row for the current iteration, else it waits until another worker has put the row in the object and then reads the row.

As far as the pattern analyzer can see, the expected read/write ratio of worker processes for the *Rowk* object is exactly 1, because it does not know which of the two alternatives will be executed most frequently. Using more aggressive optimization techniques (or perhaps even execution profiles), it might be possible to make a more accurate estimate. For *Rowk*, the compiler will therefore pass to the RTS a value for *Nread* that is equal to *Nwrite*. When in doubt, the RTS always adheres to the original replication strategy, which is to replicate objects everywhere. For ASP, replicating the object is essential, because it means that the pivot rows will be broadcast instead of being sent point-to-point. The performance of ASP will therefore be the same with the optimized and unoptimized compilers, as shown in Figure 4.14.

Figure 4.15 shows the communication overhead for ASP running on 16 processors. We only show the overhead for the unoptimized RTS, because both RTSs send and receive the same number of messages. We only show the overhead for one of the worker processes, because all processors execute the same worker process and *OrcaMain* sends a negligible number of messages. Almost all communication overhead can be attributed to adding pivot rows to the shared object *Rowk*. In each iteration one worker process adds a pivot row, which is used by all processes during that iteration. As the matrix is of size 500x500, each process sends 31 group messages of 2000 bytes. The other messages are due to forking of worker processes and due to synchronization before the end of the program.

Figure 4.16 shows the performance for ASP written in C. The Orca program on one processor is 3.5 times as slow as the C program on Amoeba. The main contributor is the difference between arrays in Orca and C. In Orca, arrays are dynamic and type secure, while in C they are not. Type security means that an illegal array access in Orca will result in a run-time error, while in C it might lead to an unexpected program crash. To retrieve an array element, the Orca program calls a RTS procedure to do the job. (The next Orca compiler will use in-line array-bound checking and will determine

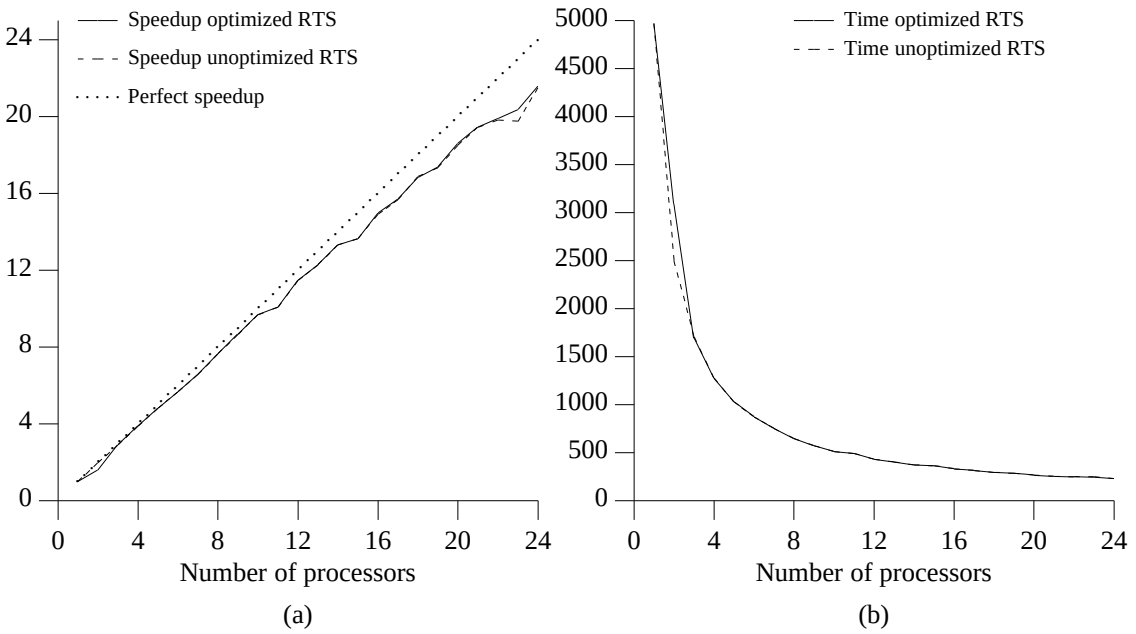


Fig. 4.14. Performance for the All-Pairs Shortest Paths Problem using a graph with 500 nodes. (a) Speedups for each implementation of the RTS. (b) Absolute running times in seconds for each implementation of the RTS.

Primitive	**Message size (byte)**			
	0	16-31	64-127	1024-2047
Send group	0	1	0	31
Receive group	9	17	1	516

Fig. 4.15. Communication overhead ASP for a processor running a worker process. Both the unoptimized and optimized RTS send and receive the same messages.

at compile time if array-bound checking is needed.) This procedure checks if the index is within the array bounds and if so, returns the value; otherwise it will generate a run-time error. The equivalent operation in C has the cost of only dereferencing a pointer.

Although type security incurs some run-time overhead, we think that it is a property worthwhile to have, especially in the arena of parallel programming. Consider a parallel program consisting of 100 processes. If this program is written in C (with library calls for synchronization and communication), one of the 100 processes may suddenly dump core without any explanation. In Orca, this cannot happen. The Orca

Compiler	Operating System	Processor	Time (sec)
Orca	Amoeba	MC68030	4968
ACK cc	Amoeba	MC68030	1425
GNU cc	SunOS4.1.1	SPARC	95

Fig. 4.16. Performance of ASP on a single processor.

programmer will get an exact error message telling which operation failed and on which line of the program the failure occurred.

4.8.3. Successive Overrelaxation

Successive overrelaxation (SOR) is an iterative method for solving discrete Laplace equations on a grid. During each iteration, the algorithm considers all non-boundary points of the grid. For each point, SOR first computes the average value of its four neighbors and then updates the point using this value. To avoid overwriting the old value of the matrix before it is used, the new matrix is computed on each iteration in two phases. The program treats the matrix as a checkerboard and alternately updates all black points and all red points. As each point has neighbors of the opposite color, a scratch matrix is not needed and each update phase can easily be parallelized. The implementation of SOR is based on descriptions published in a number of recent research papers [Butler et al. 1986; Chase et al. 1989; Carter et al. 1991].

SOR can be parallelized by partitioning the grid among a number of worker processes, one per processor. Each worker contains a vertical slice of the grid. The processes are organized in a row. At the beginning of an iteration, each worker needs to exchange edge values with its left and right neighbor. This can easily be implemented through shared buffer objects. We use two buffer objects for each pair of neighbors, for communication in each direction. Since each worker (except the first and last) has two neighbors, each worker accesses four shared objects.

In addition to these buffer objects, the program uses a single shared object (*finish*), which is used for terminating the program. After each iteration, all workers determine if they want to continue the SOR process or not. If all workers want to stop, the entire program terminates, else all workers continue with the next iteration. This is expressed using the object type *PollSequence*, which basically implements a barrier synchronization [Bal et al. 1990].

The specification of a worker process is as follows:

```
process worker(
    ToLeft, ToRight, FromLeft, FromRight: shared Buffer;
    finish: shared PollSequence;
    lb, ub: integer);
```

The main process (i.e., *OrcaMain*) creates all these shared objects and forks the worker processes, passing several objects to each process. A worker is assigned a portion of the columns of the grid; the index of the first and last column are passed as parameters. The code for the main process is shown below.

```
process OrcaMain();
    i, lb, ub: integer;
    Lbufs, Rbufs: array[integer 0 .. NCPUS] of Buffer;
    finish: PollSequence;
begin
    finish$init(NCPUS);     # Initialize PollSequence object
    for i in 0 .. NCPUS−1 do      # fork worker processes
        determine lower bound (lb) and upper bound (ub) for next worker
        fork worker(
                Lbufs[i], Rbufs[i+1],
                Rbufs[i], Lbufs[i+1],
                finish, lb, ub)  on(i);
    od;
end;
```

Here, *NCPUS* is the number of processors being used. The main process creates two arrays of buffer objects. Object *Lbufs*[*i*] is used for sending messages from worker *i* to worker *i–1* (i.e., its left neighbor). Worker *i* sends messages to its right neighbor(worker *i+1*) through object *Rbufs*[*i+1*].

The patterns for the main process and the workers are shown below:

```
process OrcaMain:
    finish$W;                                   # Initialize PollSequence object

process worker:
    {
        {[#1$W|]; [#2$W|]; [#3$W|]; [#4$W|]};# Communicate with neighbors
        #5$W;                                   # Bring out vote
        #5$R;                                   # Await decision
    }
```

OrcaMain merely initializes the *finish* object and does not use the shared objects otherwise. Each worker performs a number of iterations. At each iteration, it first communicates through its four buffer objects. (The first and last worker have only one neigh-

bor, which explains why the four write operations are conditional, and thus are surrounded by square brackets.) Next, the worker computes and then determines if it wants to continue or not. It writes this decision into the *finish* (parameter #5) object, waits until all workers have made up their mind, and then reads the overall decision from the *finish* object. (Waiting and reading the result are done in one operation, so they show as a single read operation in the pattern).

Let us first consider the *finish* object. Each worker has a read/write ratio of 1 for this object, so the analyzer will generate a value for *Nread* that is equal to *Nwrite* for this object in all worker processes. For the main process *Nwrite* is larger than *Nread*. Given the heuristics described in Section 4.7.2., the run-time system will decide to replicate this object everywhere.

Let us now discuss how the system handles the arrays of shared buffer objects. Each buffer object is passed as a shared parameter to exactly two worker processes. One of them sends messages to the buffer and the other accepts messages from it. The compiler will have assigned a positive *Nwrites* value to each parameter and a zero *Nreads* value, since the worker will apply only write operations to the objects.

The run-time system will therefore discover that each object is used on two processors and that it is written (and not read) by both of them. So, the run-time system will decide not to replicate these objects, and store each object on one of the processors that accesses it, the latter choice being arbitrary but consistent, so the objects get evenly distributed among the processors.

By performing these optimizations, each buffer object is now stored on only one processor, rather than being replicated everywhere. Communication between two neighboring processes is now implemented with point-to-point messages instead of broadcasting. This optimization improves the speedup of the program substantially, as is shown in Figure 4.17. With more than 20 processors, the performance of the unoptimized program decreases enormously, due to the large number of broadcast messages received by each processor. The optimized program does not have this problem.

Figure 4.18 shows the communication overhead of the 16-processor Orca program for both the optimized and unoptimized RTS. We only show the communication overhead for one of the 16 worker processes, as all workers (except the first and last) send and receive the same number of messages and as *OrcaMain* sends a negligible number of messages. The first and last worker send fewer messages, as these have only one neighbor.

The program makes 100 iterations. At each iteration, the boundary rows are exchanged with both neighbors and all worker processes synchronize to determine if another iteration is needed. Consider the unoptimized RTS. For each worker with two neighbors, exchanging the boundary rows results in $4 * 2 * 100 = 800$ broadcasts (4 operations per phase per iteration). 400 of the 800 are broadcasts with 640 bytes of data (80 double floating point numbers). The other 400 are broadcasts with no user data; they correspond to the *get* operation. The synchronization results in 100 broadcasts per worker. The 188 broadcasts are due to operations of which the guards first

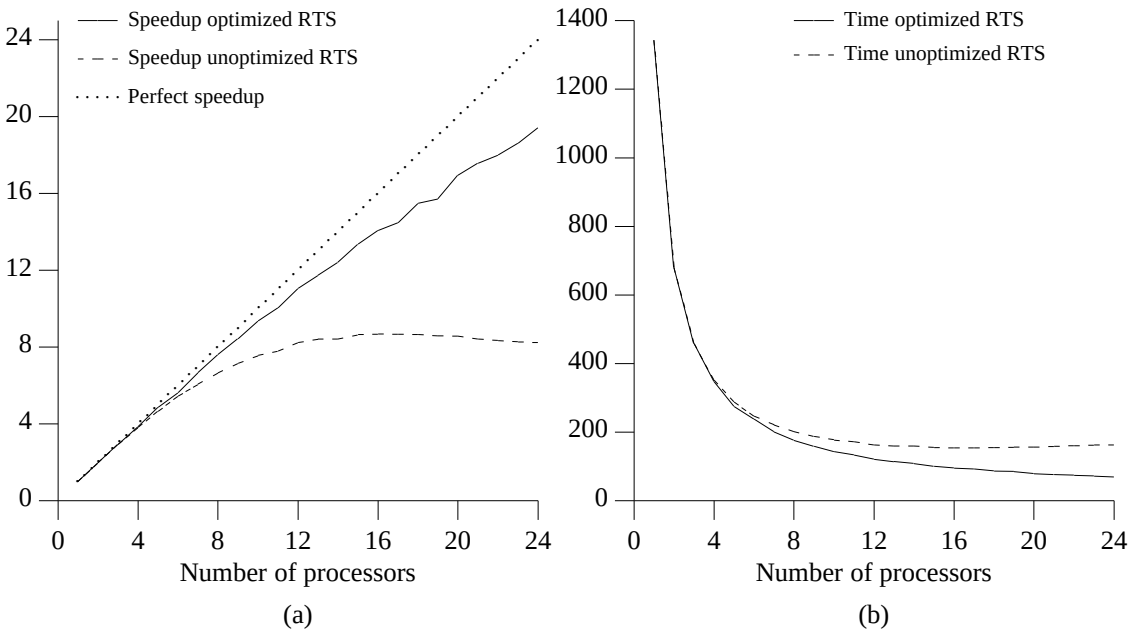


Fig. 4.17. Performance for Successive Overrelaxation for an 80 by 482 grid. (a) Speedups for each implementation of the RTS. (b) Absolute running times in seconds for each implementation of the RTS.

failed and then later, after a retrial, succeeded. The remaining broadcasts are due to forking worker processes and due to synchronization before the end of the program.

Except for synchronizing at the end of each iteration, the optimized RTS sends almost no broadcasts. All the communication with the neighbors is performed using RPC. Per iteration, each worker process performs 8 operations to exchange rows (4 operations per phase). Of the 8 operations 4 can be performed locally without communicating, as a worker process owns one of the buffers used to exchange rows. The net result is that the communication overhead for the optimized RTS is much less than for the unoptimized RTS.

Figure 4.19 shows the performance for SOR written in C. The Orca program on one processor is only 2.2 times as slow as the program written in C. Again, the overhead is largely due to array accesses.

4.9. Comparison with Related Work

In this section, we compare our approach with several related languages and systems. In particular, we look at broadcasting for parallel computing, compiler optimizations for parallel programming, objects (as used in parallel object-based languages), Linda's Tuple Space, and Shared Virtual Memory.

Primitive	Message size (byte)							
	0	8-15	16-31	32-63	64-127	512 - 1023	1024 - 2047	2048 - 4095
Send group	0	1	100	588	0	400	0	0
Receive group	9	16	1600	8820	1	6000	1	15

(a)

Primitive	Message size (byte)						
	0	8-15	16-31	32-63	64-127	512-1023	1024-2047
Receive request	0	0	0	200	0	200	0
Send reply	200	0	0	0	0	200	0
Send request	0	0	0	200	0	200	0
Receive reply	200	0	0	0	0	200	0
Send group	0	1	100	0	0	0	0
Receive group	9	16	1600	0	1	0	1

(b)

Fig. 4.18. Communication overhead for one of the processors running a worker process. (a) Overhead for the unoptimized RTS. (b) Overhead for the optimized RTS.

Compiler	Operating System	Processor	Time (sec)
Orca	Amoeba	MC68030	1344
ACK cc	Amoeba	MC68030	619
GNU cc	SunOS4.1.1	SPARC	28

Fig. 4.19. Performance of SOR on a single processor.

Broadcasting for Parallel Computing

Essential to our model is the use of group communication. It is one of the main

keys to an efficient implementation of a large class of applications. Few systems have used broadcasting for parallel programming. The language SR provides a broadcast primitive, *co send* [Andrews et al. 1988], but it only implements unordered broadcast and (at present) does not use physical broadcast at the hardware level. This limits its usability to a smaller set of applications.

One of the implementations of Linda is also based on broadcasting [Carriero and Gelernter 1986; Ahuja et al. 1988]. This implementation relies on the hardware to provide reliable delivery. Using our reliable broadcast primitive one should be able to implement the tuple space in a similar way.

Stumm and Zhou describe a number of algorithms to implement distributed shared memory [Stumm and Zhou 1990]. They have developed a simulation model to compare these different algorithms. One of the algorithms, the full-replication algorithm, is also based on reliable, totally-ordered broadcast. Our results agree with their study. For applications with a high read/write ratio, the full replication algorithm out performs the other algorithms. However, for a low read/write ratio, it is better to use *partial replication*, in which objects are replicated on some (but not all) processors, or to store the shared data on a single processor.

Cheriton and Zwaenepoel describe a number of parallel applications in which they have used broadcasting [Cheriton and Zwaenepoel 1985]. The applications include distributed game playing, alpha-beta search, and the traveling salesman problem. The main difference between their work and ours, is that the programmer writes programs using group communication primitives, while we are providing a higher-level model to the programmer. Using the higher level model we are able to simplify the job of the programmer, while still maintaining a good performance.

Finally, Gehani describes a language based on broadcasting [Gehani 1984]. Broadcasting Sequential Processes (BSP) provides a number of different primitives for doing broadcasting, ranging from synchronous to asynchronous and from buffered to unbuffered. Although we strongly believe that broadcasting is fundamental for implementing parallel applications efficiently, we do not think that broadcast primitives are the right programming paradigm for writing parallel applications. We think that a higher level paradigm like the shared data-object model is needed to simplify the complicated task of writing parallel programs.

Compiler Optimizations for Parallel Programming

Most of the work on compiler optimizations for parallel programming takes place in the area of numerical applications, where shared arrays have to be decomposed and partitioned among the memories of different machines. In FORTRAN-D [Fox et al. 1990], for example, the programmer can specify a strategy for decomposing arrays into blocks. The compiler uses this information to distribute the array among the physical memories and automatically generates send/receive primitives when needed. Parallelism in such programs can be obtained by executing different iterations of a loop on different processors, or by performing higher-level operations (e.g., matrix additions)

in parallel. Such systems usually adhere to the Single Program Multiple Data (SPMD) style [Karp 1987]. Besides FORTRAN-D, many similar projects exist [Callahan et al. 1988; Koelbel et al. 1990; Rosing et al. 1991]. Also, some work on functional languages is related to this approach [Rogers and Pingali 1989; Chen et al. 1988].

Our work on Orca is much less focused on numerical applications and partitioned arrays. Parallelism in Orca is explicit (through a fork statement) and many forms of synchronization can be expressed, so Orca programs are not necessarily SPMD-like. Objects in Orca are not partitioned but replicated. Also, this replication is transparent to the user (except for performance). So, the goals of the optimizing Orca compiler and the techniques used are different from those of the languages mentioned above.

Objects

Objects are used in many object-based languages for parallel or distributed programming, such as Emerald [Jul et al. 1988], Amber [Chase et al. 1989], ALPS [Vishnubhotla 1988], and Mentat [Grimshaw 1990]. Objects in such languages typically have two parts:

1. Encapsulated data.
2. A *manager process* that controls access to the data.

The data are accessed by sending a message to the manager process, asking it to perform a certain operation on the data. As such objects contain a process as well as data, they are said to be *active*.

Although, in some sense, parallel object-based languages allow processes (objects) to share data (also objects), their semantics are closer to message passing than to shared variables. Access to the shared data is under full control of the manager process. In ALPS, for example, all operations on an object go through its manager process, which determines the order in which the operations are to be executed. Therefore, the only way to implement the model is to store an object on one specific processor, together with its manager process, and to translate all operations on the object into RPCs to the manager process.

Our model does not have such centralized control. Objects in Orca are purely passive: they contain data but no manager process. Access control to shared data-objects is distributed; it is basically determined by only two rules:

1. Operations must be executed indivisibly.
2. Operations are blocked while their guards are false.

Therefore, the model can be implemented by replicating data-objects on multiple processors, as we discussed in Section 4.4. Read operations can be applied to the local copy, without any message passing being involved. Moreover, processes located on different processors can apply read operations simultaneously, without losing any parallelism.

Linda's Tuple Space

Linda [Ahuja et al. 1986] is one of the first languages to recognize the disadvantages of central manager processes for guarding shared data. Linda supports so-called *distributed data structures*, which can be accessed simultaneously by multiple processes. In contrast, object-based languages typically serialize access to shared data structures. Linda uses the Tuple Space model for implementing distributed data structures.

In general, distributed data structures in Linda are built out of multiple tuples. Different tuples can be accessed independently from each other, so processes can manipulate different tuples of the same data structure simultaneously. Multiple **read** operations of the same tuple can also be executed simultaneously. Tuples are modified by taking them out of the Tuple Space first, so modifications of a given tuple are executed strictly sequentially.

Although the idea of distributed data structures is appealing, we think the support given by the Tuple Space for implementing such data structures has important disadvantages. For distributed data structures built out of single tuples, mutual exclusion synchronization is done automatically. Operations on complex data structures (built out of multiple tuples), however, have to be synchronized explicitly by the programmer. In essence, Tuple Space supports a fixed number of built-in operations that are executed indivisibly, but its support for building more complex indivisible operations is too low-level [Kaashoek et al. 1989].

In Orca, on the other hand, programmers can define operations of arbitrary complexity on shared data structures; all these operations are executed indivisibly, so mutual exclusion synchronization is always done automatically by the run-time system. This means it is the job of the implementation (the compiler and run-time system) to see which operations can be executed in parallel and which have to be executed sequentially. As discussed above, one way of doing this is by distinguishing between read and write operations and executing reads in parallel on local copies; more advanced implementations are also feasible.

Like Orca, Linda also uses extensive compile-time optimization [Carriero 1987], but this is mainly aimed at reducing the overhead of associative addressing of Tuple Space. Also, work has been done on a heuristic Linda kernel that determines during run-time where to store tuples, based on usage statistics [Lucco 1987]. As far as we know, however, no existing Linda system determines the optimal location for a tuple at compile time.

Shared Virtual Memory

Shared Virtual Memory (SVM) [Li and Hudak 1989] simulates physically shared memory on a distributed system. It partitions the global address space into fixed-sized pages, just as with virtual memory. Each processor contains some portion of the pages. If a process tries to access a nonlocal page, it incurs a page fault, and the operating system will then fetch the page from wherever it is located. Read-only pages may be

shared among multiple processors. Writable pages must reside on a single machine. They cannot be shared. If a processor needs to modify a page, it will first have to invalidate all copies of the page on other processors.

There are many important differences between the implementation of our model and SVM. SVM is (at least partly) implemented inside the operating system, so it can use the MMU. In Orca, everything except for the broadcast protocol is implemented in software outside the operating system. This difference gives SVM a potential performance advantage.

Shared data-objects are accessed through well-defined, high-level operations, whereas SVM is accessed through low-level read and write instructions. Consequently, we have a choice between invalidating objects after a write operation or updating them by applying the operation to all copies (or, alternatively, sending the new value). With SVM, there is no such choice; only invalidating pages is viable [Li and Hudak 1989]. In many cases invalidating copies will be far less efficient than updating them.

Several researchers have tried to solve this performance problem by relaxing the consistency constraints of the memory [Hutto and Ahamad 1990; Minnich and Farber 1990]. Although these weakly consistent memory models may have better performance, we fear that they also break the ease of programming for which DSM was designed in the first place. Since Orca is intended to simplify applications programming, Orca programmers should not have to worry about consistency. On the other hand, systems based on release consistency weaken the ordering of events only for the implementation [Carter et al. 1991; Lenoski et al. 1992]. The programmer still sees a sequentially consistent memory. In the future, we may investigate whether a compiler is able to relax the consistency transparently, much as is done in these systems.

Another important difference between Orca and SVM is the granularity of the shared data. In SVM, the granularity is the page size, which is fixed (e.g., 4 Kbyte). In Orca, the granularity is the object, which is determined by the user. So, with SVM, if only a single bit of a page is modified, the whole page has to be invalidated. This property leads to the well-known problem of *false sharing*. Suppose a process *P* repeatedly writes a variable *X* and process *Q* repeatedly writes *Y*. If *X* and *Y* happen to be on the same page (by accident), this page will continuously be moved between *P* and *Q*, resulting in thrashing. If *X* and *Y* are on different pages, thrashing will not occur. Since SVM is transparent, however, the programmer has no control over the allocation of variables to pages. In Orca, this problem does not occur, since *X* and *Y* would be separate objects and would be treated independently.

A more detailed comparison between our work and Shared Virtual Memory is given in [Levelt et al. 1992].

4.10. Conclusion

To summarize, in this chapter we have discussed multiprocessors (with shared memory) and multicomputers (without shared memory). The former are easy to program but hard to build, while the latter are easy to build but hard to program. We then introduced a new model that is easy to program, easy to build, and has an acceptable performance on problems with a moderate or large grain size. We have also designed and implemented a shared object layer and a language, Orca, that allows programmers to declare objects shared by multiple processes.

We have described two approaches to implementing shared data-objects in a distributed environment. In the first approach, objects are replicated on all the machines that need to access them. When the language run-time system needs to read a shared object, it just uses the local copy. When it needs to update a shared object, it reliably broadcasts the operation on the object. This scheme is simple and has a semantic model that is easy for programmers to understand. It is also efficient because the amount of data broadcast is exactly what is needed. It is not constrained to be 1 Kbyte or 8 Kbyte because the page size happens to be that, as in the work of Li and Hudak. Most of our broadcasts are much smaller.

The second approach is more complicated to implement, but gives better performance on applications that do not exhibit a high ratio of read operations versus write operations. This approach uses a combination of compile-time and run-time optimizations to determine which objects to replicate and where to store nonreplicated objects. We have applied the optimizer to three existing Orca applications. For all three applications, it made the right decisions. It significantly improved the speedup of two of the three programs.

Although our initial experiences with the second approach are promising, there still remain several problems to be solved. For example, for some applications, a partial replication scheme is better.

We believe that the approach of explicitly computing read-write patterns may serve as a basis for other optimizations. One important class of such optimizations is *adaptive caching*, proposed in the Munin project [Bennett et al. 1990]. The idea is to classify shared objects based on the way they are used. A *producer-consumer* object, for example, is written by one process and read by others. A *write-once* object is initialized once but not changed thereafter. Munin uses different coherence mechanisms for different classes of objects. In general, using weaker coherence rules means better performance.

Once the classification of objects is known, Munin is able to do many optimizations. Unfortunately, the current Munin implementation leaves the classification up to the programmer. With the read-write patterns generated by the Orca compiler, we hope to be able to do a similar classification automatically. This is another important area for future research.

In conclusion, we believe that the use of group communication combined with compile-time information is a good way to efficiently support shared objects. The

model is a promising approach to parallel programming, as it is simple to understand and efficient to implement. We believe that this paradigm offers a new and effective way to exploit parallelism in future computing systems.

Notes

All the research described in this chapter is based on a fruitful collaboration with Henri Bal and Andy Tanenbaum. This collaboration started around the summer of 1987 and has produced a number of papers, Bal's thesis, and this thesis. The description of Orca, the comparison with other parallel languages, and the design of the unoptimized Orca RTS appear in [Bal et al. 1989a; Bal 1990; Bal et al. 1992a; Bal and Tanenbaum 1991; Tanenbaum et al. 1992]. The performance measurements contain material from [Bal et al. 1990]. The joint research on the optimized RTS is new.

5

FAULT-TOLERANT PROGRAMMING USING BROADCASTING

Applications differ in their need for fault tolerance (see Fig. 5.1). In general, the greater the potential damage due to a failure the more the need for fault tolerance. If a nuclear reactor, for example, does not recover correctly from a failure, human lives might be lost. If a text formatting program does not recover from a failure, the cost of the failure is only the time to rerun the program. The cost and likelihood of a catastrophic failure determine how much users are willing to pay for fault tolerance. Users would rather have their text formatting program run as fast possible and rerun the whole program in the face of a failure than have the text formatting program run twice as slow and be able to recover from failures. For a nuclear reactor, the tradeoff is likely to be the other way around.

In this chapter, we will look at how useful group communication is for two application areas that have different needs for fault tolerance: parallel applications and distributed services. In the first part of the chapter, we describe a simple scheme for rendering parallel applications fault-tolerant. Our approach works for parallel Orca applications that are not interactive. The approach is based on making a globally-consistent checkpoint periodically and rolling back to the last checkpoint when a process or processor fails. Making a consistent checkpoint is easy in Orca, because its implementation is based on reliable group communication. The advantages of our approach are its simplicity, ease of implementation, low overhead, and transparency to the Orca programmer.

In the second part of this chapter, we will present an experimental comparison of two fault-tolerant implementations of Amoeba's directory service. One is based on RPC and the other is based on group communication. The purpose of the comparison is to show that the implementation using group communication is simpler and also more efficient than the implementation based on RPC. The directory service exemplifies distributed services that provide high reliability and availability by replicating data.

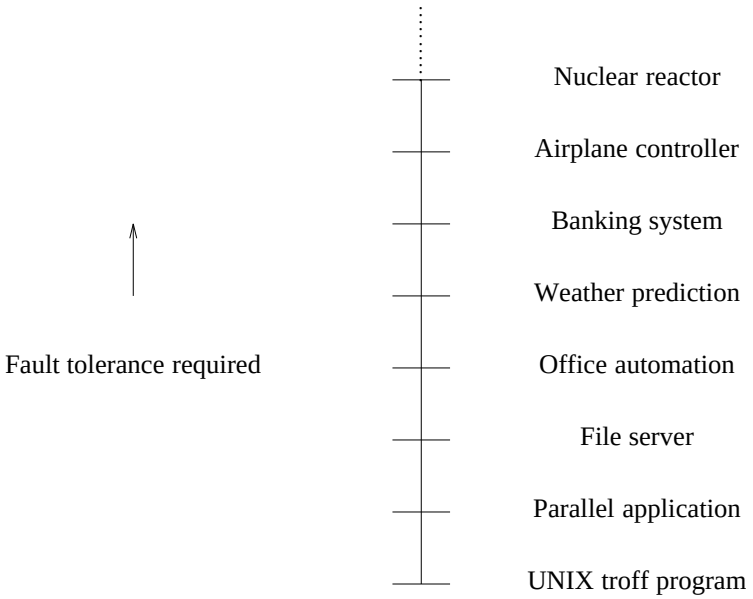


Fig. 5.1. Scale showing the amount of fault tolerance needed.

The chapter deviates in organization from previous chapters. It contains two related but independent subchapters, each with its own introduction, research contribution, and discussion. We chose this organization instead of two chapters, because both subchapters have one central theme: fault tolerance through group communication. The fault-tolerant Orca implementation uses a group with a resilience degree of zero and the fault-tolerant directory service uses a group with a resilience degree greater than zero. At the end of this chapter, we will relate the two subchapters.

5.1. Transparent Fault Tolerance in Parallel Orca Programs

Designers of parallel languages frequently ignore fault tolerance. If one of the processors on which a parallel program runs crashes, the entire program fails and must be run again. For a small-scale parallel system with few processors, this may be acceptable, since processor crashes are unlikely. The execution-time overhead of fault tolerance and its implementation costs may not be worth the extra convenience. After all, the goal of parallelizing a program is to reduce the execution time.

Consider, however, a parallel program that runs on a thousand CPUs for 12 hours to predict tomorrow's weather. Even if the mean-time-between-failure of a CPU is a few years, the chance of one of them crashing can not be neglected. Moreover, if a processor crashes when the computation is almost finished, the whole program has to be started all over again, thus doubling its execution time. In general, the larger the scale of the parallel system, the more important fault tolerance becomes. Future large-scale parallel systems will have to take failures into account. An obvious question is: Who will deal with processor crashes? There are two options. One is to have the programmer deal with them explicitly. Unfortunately, parallel programming is hard

enough as it is, and fault tolerance will certainly add more complexity. The alternative is to let the system (i.e., compiler, language run time system, and operating system) make programs fault-tolerant automatically, in a way transparent to programmers. The latter option is clearly preferable, but in general it is also hard to implement efficiently.

In this chapter, we will discuss how transparent fault tolerance has been implemented in the Orca parallel language described in the previous chapter. The failures that we consider are transient failures, such as hardware errors. The problem we have tried to solve is modest: to avoid having to restart long-running parallel Orca programs from scratch after each crash. The goal is to do so without bothering the programmer and without incurring significant overhead. We have not tried to solve the more general class of problems of making all distributed systems fault-tolerant. Instead, we consider only the class of parallel programs that take some input, compute for a long time, and then yield a result. Such programs do not interact with users and do not update files.

Our solution is simple: make global checkpoints of the global state [Chandy and Lamport 1985] of the parallel program using reliable group communication. If one of the processors crashes, the whole parallel program is restarted from the checkpoint rather than from the beginning. Users can specify the frequency of the checkpoints, but otherwise are relieved from any details in making their programs fault-tolerant.

The key issue is how to make a global checkpoint that is *consistent*, preferably without temporarily halting the entire program. It turns out that this problem can be solved in a surprisingly simple way in Orca.

The outline of the rest of this section is as follows. We first summarize how Orca applications are run on Amoeba. Next, Section 5.1.2 describes the design of a fault-tolerant Orca run time system. Section 5.1.3 gives the implementation details and the problems we encountered with Amoeba. In Section 5.1.4, we give some performance measurements. In particular, we show how much time it takes to make a checkpoint and to restart a program after a crash. In addition, we measure the overhead of checkpoints on example Orca programs. Section 5.1.5 compares our method with related approaches, such as explicit fault-tolerant parallel programming, message logging, and others. Finally, in Section 5.1.6, we present some conclusions and see how our work can be applied to other systems.

5.1.1. Running Orca Programs under Amoeba

Orca can be implemented efficiently on a distributed system using a RTS that replicates shared objects in the local memories of the processors. If a processor has a local copy of an object, it can do *read* operations locally, without doing any communication. *Write* operations are broadcast to all nodes containing a copy. All these nodes update their copies by applying the write operation to the copy. In this chapter, we will consider only the unoptimized RTS (the RTS based only on group communication). Extensions to our scheme are needed to make it work for the optimized version (the RTS based on group communication and RPC).

To understand how the checkpointing scheme works, it is important to know how Orca programs are run. Parallel Orca programs are run on the processor pool. An Orca application is started by the program *gax* (Group Amoeba eXecute). *Gax* asks the directory server for the capability of the program, and using the capability it fetches the Orca binary (Orca program linked with the Orca RTS) from the file server. Next, *gax* allocates the requested number of processors in the processor pool and starts the executable program on each processor (see Fig 5.2). These Orca processes together form one process group. A message sent to the group is received in the same order by all processes (including the sending process), even in the presence of communication failures, as described in Chapter 3.

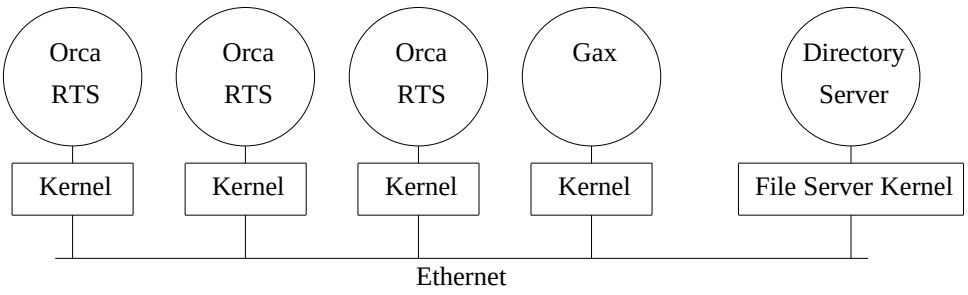


Fig. 5.2. An Amoeba system with an Orca application running on 3 processors. The Orca processes together form one process group. In the current configuration, the file server is for performance reasons linked with the kernel.

5.1.2. Designing a Fault-Tolerant RTS

There are many ways of making Orca programs fault-tolerant. One approach is to ask a special server to keep an eye on Orca applications and start them again if they fail. In Amoeba such a server is available and is called the *boot server*. Unfortunately, this does not gain much, since the application would have to start all over from the beginning. For applications that have to meet a deadline this is not appropriate.

Another approach is to use a message logging and playback scheme [Powell and Presotto 1983], such as optimistic recovery [Strom and Yemini 1985; Johnson 1989]. Message logging is designed for general distributed applications rather than just parallel applications, and may be too expensive for parallel applications. Optimistic recovery, for example, can deal with interactive programs. Programs using optimistic recovery will not ask for the same input twice or produce the same output twice. While this property is useful, it is not essential for most parallel programs, which frequently are not interactive. For those programs, the message logging solution is not necessary. A cheaper and simpler form of fault tolerance is adequate.

The method we use is to periodically make a globally-consistent checkpoint of all the Orca processes. After a crash, all processes continue from the last checkpoint. Since parts of the program may be executed multiple times, the method cannot be used for interactive programs or any programs that may have side effects.

The most important design issue is how to obtain a *globally-consistent* checkpoint. As an example, assume a process P1 makes a checkpoint at time T1 and then sends a message to process P2. (Although Orca programs do not send messages, their run-time systems do.) Assume that process P2 receives this message and then makes a checkpoint, at time T2. Obviously, the two checkpoints are not consistent. If both processes are rolled back to their checkpointed state, P1 would again send the message, but P2 would be in a state where it had already accepted the message. So, P2 would receive the message twice.

As explained in [Koo and Toueg 1987], checkpointing should be done consistently relative to communication. It is not necessary to have all processors make checkpoints at exactly the same time, but messages must not cross checkpoints. One could envision making a consistent checkpoint by first telling each processor to stop sending messages. When the whole system is quiet, each processor is instructed to make its local checkpoint. All these checkpoints will then be consistent, since no interprocess communication takes place during checkpointing. We will not go into the details of how such a design might work. Suffice it to say that this solution would not be very attractive. The reason is that it may take a lot of time and overhead to bring the whole (distributed) system to a halt. Also, the more processors there are, the more time is wasted.

In the unoptimized Orca run-time system based on reliable group communication it is almost trivial to make consistent checkpoints. As we explained in Chapter 4, the Orca run-time system communicates only through reliable, totally-ordered group communication. All processes receive all broadcast messages in the same order. Therefore, to make the global checkpoint consistent, all that is needed is to broadcast one *make checkpoint* message. This message will be inserted somewhere in the total order of broadcast messages. Assume that the *make checkpoint* message gets sequence number N in this total ordering. Then, at the time a process makes its local checkpoint, it will have received and handled messages 1 to $N-1$, but not messages $N+1$ and higher. This applies to *all* processes. Therefore the checkpoints are all consistent.

To recover from a processor crash all processes roll back to their last common checkpoint. No process will send a message before all processes have been rolled back. Thus, when a process sends a message, all processes will receive it with the same sequence number. Because all processes roll back and synchronize before they continue running, they will start in a consistent state.

For some applications checkpoints clearly are not going to be cheap. Each process must save its local data (or, at the very least, the difference with the previous checkpoint). For some processes, this may easily involve writing several hundred kilobytes to a remote file server. With the Amoeba file server, known as the *Bullet server* [Van Renesse et al. 1989], this may take a substantial fraction of a second. With multiple processes, things will get worse. The time to make a globally-consistent checkpoint will depend on the configuration of the distributed system and on the network. Clearly, having a single centralized Bullet server for a thousand pool processors is not a good

idea. However, the Bullet server can be (and is) replicated, so different pool processors can use different instantiations of this server.

The main advantage of our algorithm is that it is extremely simple and easy to implement. While it is not optimal it is still useful. In particular, for long-running noninteractive parallel applications, the overhead of making a checkpoint every few minutes is quite acceptable. Also, the frequency of the checkpointing can easily be changed. Using a lower frequency will decrease the overhead, but increase the average amount of re-execution due to crashes. We will get back to this performance issue in Section 5.1.4, where we give performance results.

To make our approach work for interactive programs, the system needs to make a globally-consistent checkpoint each time some interaction with the outside world happens. As making a globally-consistent checkpoint is reasonably expensive, we did not pursue this idea. For interactive parallel programs, the approach of Elnozahy and Zwaenepoel is more promising [Elnozahy and Zwaenepoel 1992a].

Another disadvantage of our approach is that it works only for the unoptimized version of the Orca RTS. The optimized version uses both RPC and group communication and it is therefore harder to make a consistent checkpoint. In this case, more complicated protocols such as Chandy's and Lamport's distributed snapshot protocol are needed [Chandy and Lamport 1985]. We will not pursue these ideas in this thesis.

5.1.3. Fault-Tolerant Implementation of Orca

In this section, we describe the problems encountered with implementing a fault-tolerant run-time system for Orca on Amoeba. To understand the implementation we have to take a closer look at how process management is done in Amoeba. Each Amoeba kernel contains a simple process server. When *gax* starts an Orca application on a processor, it sends a descriptor containing capabilities for the text, data, and stack segment to the processor's process server. The process server fetches the segments from the file server, builds a process from the segments, and starts the process. The capability for the new process, the *owner capability*, is returned to *gax*.

To checkpoint a single process, *gax* sends a *stun* signal to the process server that manages the process. The process server stops the signaled process when it is in a safe state, that is, when it is not in the middle of an RPC. (Interrupting a process while doing an RPC would break the at-most-once semantics.) When the process has stopped, the process server sends the process descriptor to the owner (*gax* in this case) and asks it what do to with the process. Using the capabilities in the processor descriptor, *gax* can copy the process state to the file server. After having copied the state, *gax* tells the process server to resume the process.

Making a globally-consistent checkpoint of the complete Orca application works in the following way. Every *s* seconds, a thread in the RTS of one of the machines broadcasts a *make checkpoint* message to all other processes in the application. When a process receives this totally-ordered message, it asks *gax* to make a checkpoint of it,

as described above. When all processes of the application have made a checkpoint, *gax* stores the capabilities for the checkpoints with the directory server.

To be able to rollback to a consistent checkpoint while making a new consistent checkpoint, *gax* marks the new checkpoint as *tentative*. When *gax* is sure that the new checkpoint is consistent, it marks the tentative checkpoint as consistent and deletes the previous consistent checkpoint.

Rolling back to a previous checkpoint works as follows. If a member of the group that runs the Orca application crashes, the group communication primitives return an error after some period of time. When a process sees such an error, it asks *gax* to roll the application back. *Gax* kills any surviving members and then starts the application again. Instead of using the original executable file, it uses the checkpoints of the processes.

The actual implementation is more complicated, due to a number of problems. The first problem is that by using *gax* we have introduced a single point of failure: if *gax* crashes, the Orca application cannot make any checkpoints or recover. To prevent this from happening, *gax* is registered with the boot service. The boot service checks at regular intervals whether *gax* is still there. If it is not there, the boot service restarts *gax*. (When *gax* starts running again, it kills the remaining processes and rolls the application back to the last checkpoint.) The boot service itself consists of multiple servers that check each other. As long as the number of failures at any point in time is smaller than the number of boot servers, the Orca application will continue running.

A second problem is that the checkpoints made by the process server do not contain all the state information about the parallel program (e.g., the messages buffered by the kernel). In particular, the kernel state information about process groups is not saved.

As an example, suppose the RTS initiates a global checkpoint by broadcasting a *make checkpoint* message. At about the same time, a user process executes a write operation on a shared object. As a result, its local RTS will send an *update* broadcast message and block the Orca process until this message has been handled. Assume that the broadcast protocol orders the *update* message just after the *make checkpoint* message. The broadcast protocol will buffer this message in all kernels and it will be delivered after the checkpoint is made. If after a crash a process has been rolled back to this checkpoint, all the kernel information about the group is gone, including the buffered message.

Fortunately, detecting that a message has been sent and not received by any process when the checkpoint was made is easy. After a thread has sent a broadcast message, it is blocked until the message is received and processed by another thread in the same process. If after recovery there are any threads blocked waiting for such events, they are unblocked and send the message again.

The problem that the kernel information about the group is not included in a checkpoint is harder. We have solved this problem by having *gax* maintain a state file, in which it keeps track of additional status information. This file contains the current

members of the process group, as well as the port names to which the restart messages (discussed below) are to be sent, the number of checkpoints made so far, and other miscellaneous information.

As a consequence of this approach, *gax* must read the status file during recovery and rebuild the process group. To rebuild the group, *gax* needs the help of the processes that are being rolled back. These processes must actively join the newly formed process group. Clearly, all this activity is only needed during recovery and not after making a checkpoint. The problem is that it is difficult for the processes to distinguish between these two cases (i.e., resuming after making a checkpoint and resuming after recovery). After all, the state of the parallel program after recovery is the same as the state of the latest checkpoint.

Our solution to this problem is as follows. After making a checkpoint, a checkpoint server does not continue the process immediately, but it first waits for a *continue* message from *gax*. If it receives this message, it simply continues. On the other hand, if a processor has crashed and the program has just recovered, *gax* sends a *restart* message instead of the usual *continue*. If the checkpoint server receives a *restart*, it first participates in rebuilding the group state, before continuing the application.

An implementation issue concerns the text segment of a process. It is not necessary to dump the text (code) segment of each process, since text segments do not change and can be obtained by reading the executable file containing the process's image. At the expense of writing some more code, our prototype implementation avoids saving the text segment of a process after the first checkpoint.

Another implementation issue is scalability. The cost of broadcasting the *make checkpoint* message is almost independent of the number of receivers (see Chapter 3) and therefore scales well. However, *gax* and the Bullet server are likely to become a bottleneck as the number of processors increases. This could be avoided by using a distributed algorithm for making checkpoints, for example, by sending the checkpoints to a neighbor instead of to the Bullet service.

Although the final implementation is more complicated than we expected, only minor modifications were required to existing software. No changes were made to the Amoeba kernel.

5.1.4. Performance

We have measured the performance of the implementation described in the previous section. The Orca programs run on a collection of 20-Mhz MC68030s. The directory server and the file server are duplicated. They run on Sun 3/60s and use WREN IV SCSI disks. All processors are connected by a 10 Mbit/s Ethernet. Given this environment, an Orca program running on n pool processors can tolerate 1 failure of the directory server or file server and $n-1$ concurrent pool-processor failures. If both directory or both file servers crash, the Orca program will block until one of each has recovered. If n concurrent pool-processor failures happen, the Orca program will be restarted from the latest checkpoint by the boot server. Because all processors are

connected by one network, the system will not operate if the network fails. If the system had contained multiple networks, it could have tolerated network failures as well, because the Amoeba network protocol, FLIP, is based on dynamic routing tables.

We have measured the overhead of checkpointing and recovery for a trivial Orca program called *pingpong*. *Pingpong* consists of processes running on different processors and sharing one integer object. Each process in turn increments the shared integer and blocks until it is its turn to increment it again. This program is interesting because it sends a large number of messages: for each increment of the shared integer the RTS sends one message. If this program were to run on a system based on message logging, it would experience a large overhead.

We ran *pingpong* with and without checkpointing and computed the average time for making one checkpoint, including the first checkpoint. The results are given in Figure 5.3. Each process has 7 segments: one text segment of 91 Kbytes, one data segment of 56 Kbytes, and 5 stack segments of 11 Kbytes each. The state file consists of 3670 bytes. The first checkpoint with text segment consists of 202 Kbytes and subsequent checkpoints (without text segment) are 111 Kbytes. If *pingpong* is running on 10 processors, taking a global checkpoint will involve writing 1.11 Mbytes to the file server. With 8 or more processors, the Bullet service becomes a bottleneck, because it is only duplicated.

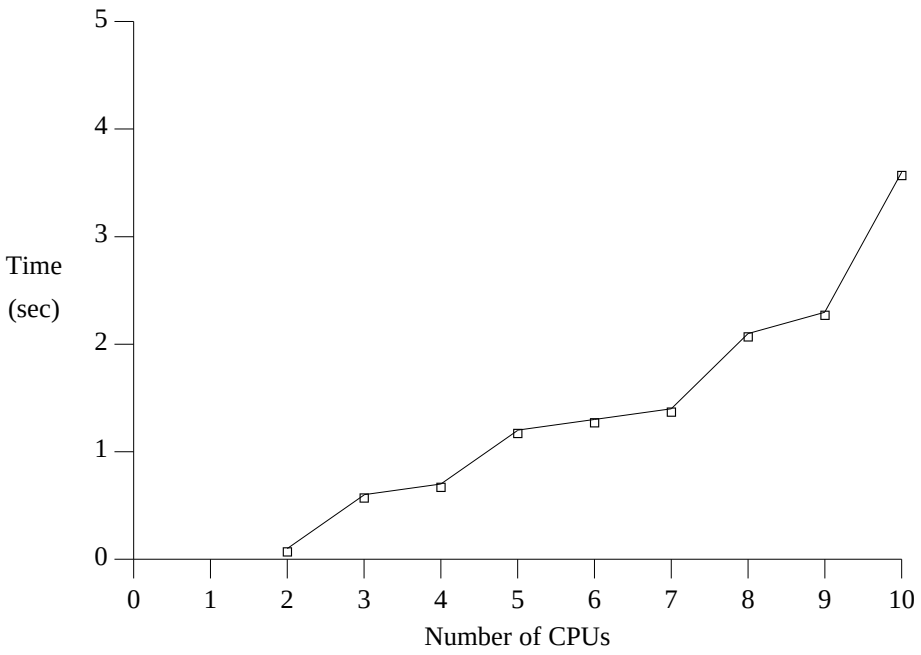


Fig. 5.3. The average cost of making a checkpoint.

The time to recover from a processor crash is equal to the time to detect the processor crash plus the time to start *n* processes. The time to detect a processor crash is tunable by the user. The user can specify how often the other members in the group

should be checked. The time to start a new process on Amoeba is 58 msec [Douglis et al. 1991]. Thus, most of the time for making a globally-consistent checkpoint is spent in transferring segments from each processor to the file server.

For any application the total overhead (the difference between execution time with and without checkpointing) introduced by our scheme depends on 3 factors:

1. The cost of making a checkpoint (dependent on the number of processors and the size of the process's image).
2. The cost of a roll back.
3. The mean time to failure (MTTF) of the system (hardware and software).

To minimize the average run-time of the application, given the numbers for these factors, one can compute the optimal computing interval (the time between two checkpoints such that the average run-time is minimized) and thus the overhead introduced by checkpointing (see Appendix B):

$$overhead = \frac{T_{cp}}{T_{comp}} + \frac{(T_{comp} + T_{cp}) / 2 + T_{rb}}{T_{MTTF}}$$

where T_{comp} is the computing interval, T_{cp} is the mean time to make a checkpoint, T_{rb} is the mean time to recover from a crash.

For example, if an application runs for 24 hours using 32 processors, the cost for a checkpoint is 15 seconds, the time to roll back is 115 seconds, and the MTTF is 24 hours, then the optimal checkpoint interval is 27 minutes. In this case the overhead is 1.6 percent. If the MTTF is 7 days, then the optimal checkpoint interval is 71 minutes and the overhead is 0.7 percent.

5.1.5. Comparison with Related Work

Although considerable research has been done both on parallel programming and on fault tolerance, few researchers have looked at fault-tolerant parallel programming. In this section, we will look at some of this work and also at more general techniques for fault tolerance.

Fault-Tolerant Parallel Programming

An alternative to our approach is to let programmers deal with processor crashes. Bal describes experiences with explicit fault-tolerant parallel programming [Bal 1992]. The language used for these experiments was Argus [Liskov and Scheifler 1983; Weihl and Liskov 1985; Liskov 1988]. Below we will first compare Bal's work on Argus and our work on Orca.

The Argus model is based on guardians, RPC, atomic transactions, and stable storage. Processes communicate through RPC. A guardian is a collection of processes and data located on the same processor. Programmers can define stable objects, which are manipulated in atomic transactions (consisting of multiple RPC calls, possibly

involving many different guardians). If a transaction commits, the new state of the stable objects changed by the transaction is saved on stable storage. After a guardian crashes, it is recovered (possibly on a different processor) by first restoring its stable objects and then executing a user-defined recovery procedure.

The parallel Argus programs use multiple guardians, typically one per processor. Each guardian does part of the work and all guardians run in parallel. Guardians occasionally checkpoint important status information, by storing this information in stable objects.

An important advantage of letting the programmer deal with fault tolerance is the increased efficiency. The Argus programs, for example, only save those bits of state information that are essential for recovering the program after a failure. They do not checkpoint data structures that the programmer knows will not change, nor do they save temporary (e.g., intermediate) results. In addition, it is frequently possible to recover only the guardian that failed, rather than all guardians (and processes) in the program. Finally, the programmer can control *when* checkpoints are made. For example, if a process has just computed important information that will be needed by other processes, it can write this information to stable storage immediately.

In Orca, programmers do not have these options. Programming in Orca is simpler, because fault tolerance is handled transparently by the system. For the parallel Argus programs, the extra programming effort varied significantly between applications. Some applications were easy to handle. For example, a program using replicated worker style parallelism [Carriero et al. 1986] merely needed to write the worker's jobs (tasks) to stable storage. If a worker crashed, the job it was working on was simply given to someone else, similar to the scheme described in [Bakken and Schlichting 1991]. Other applications, however, required much more effort. For parallel sorting, for example, a lot of coordination among the parallel processes was needed to obtain fault tolerance [Bal 1992].

Of course, there are many other language constructs that can be directly used for fault-tolerant parallel programming. Examples are: exception handlers, fault-tolerant Tuple Space [Xu 1988] and atomic broadcasting. Our group protocol, for example, can tolerate processor crashes, so it can be used for building fault-tolerant parallel applications [Kaashoek and Tanenbaum 1990]. Little experience in using these mechanisms for parallel programs is reported in the literature, however.

Other Mechanisms Providing Transparent Fault Tolerance

Several other systems provide fault tolerance in a transparent way [Birman and Joseph 87; Strom and Yemini 1985; Koo and Toueg 1987; Johnson 1989; Sistla and Welch 1989; Li et al. 1991; Elnozahy and Zwaenepoel 1992a]. Most of these schemes are based on message logging and playback, usually in combination with periodic checkpoints. They are more general than the method we described, in that they can also handle interactive distributed programs and sometimes can even give real-time guarantees about the system.

One of the message logging systems is done by Johnson [Johnson 1989] and runs on V, a system similar to Amoeba. Johnson's approach is much more general than our approach (it can deal with interactions with the outside world) but it is also more complicated and harder to implement. Johnson, for example, modified the V kernel [Cheriton 1988b]. Furthermore, it logs every message. This increases the cost for communication substantially (14 percent or higher). Using our scheme the overhead depends on the frequency of making checkpoints and is independent of the number of messages that an application sends. For parallel programs, this property is important, since such programs potentially communicate frequently.

An interesting system related to ours is that of Li, Naughton, and Plank [Li et al. 1990]. This system also makes periodic globally-consistent checkpoints. Unlike ours, however, it does not delay the processes until the checkpoint is finished. Rather, it makes clever use of the memory management unit. The idea is to make all pages that have to be dumped *read-only*. After this has been done, the application program is resumed and a *copier* process is started in parallel, which writes the pages to disk. Since all pages are read-only to the application, there is no danger of losing consistency. If the application wants to modify a page, it incurs a page-fault. The page-fault handler asks the copier process to first checkpoint this page. After this has been done, the page is made writable and the application is resumed. In this way, much of the checkpointing can overlap the application.

In principle, we could have used this idea for making a checkpoint of a single process, but it would require extensive modifications to the memory management code and the way checkpoints are made in Amoeba. As we wanted to keep our implementation as simple as possible, we were not prepared to implement this optimization.

Another related approach is that of Wu and Fuchs [Wu and Fuchs 1990]. They describe a method to make a page-based shared virtual memory fault-tolerant. Like our method, their method is transparent to the programmer and is intended for long running parallel computations. In their method, the owner process of a modified page takes a checkpoint before sending the page to the process that requests it. Therefore, unlike our method, the frequency of checkpointing is determined by the patterns of data sharing. Frequent checkpointing occurs if two processes alternately write the same page.

In a recent paper, Elnozahy, Johnson, and Zwaenepoel give performance figures on a fault-tolerant system using consistent checkpointing [Elnozahy et al. 1992]. The protocol to make a consistent checkpoint uses a server that coordinates the global checkpoint. During each run of the protocol, each process makes a tentative checkpoint, which is made permanent after the protocol has completed successfully. The protocol ensures that each set of permanent checkpoints forms a consistent checkpoint, which is identified with a monotonically increasing integer. This integer is tagged on every application message to enable the protocol to run in the presence of message reordering or loss. Besides using a different protocol for making a consistent checkpoint, the authors employ a number of optimizations, such as incremental checkpointing, copy-on-write checkpointing, and a log-based file server, to achieve very good perfor-

mance. From the measurements on a number of parallel applications the authors conclude that consistent checkpointing is an efficient way to provide fault tolerance for long running applications.

A final related approach is the Isis toolkit [Birman and Joseph 87; Birman et al. 1990]. The toolkit provides the programmer with a whole spectrum of tools for building fault-tolerant distributed applications. Some of the tools require explicit involvement of the programmer, others are almost completely transparent. Using Isis' transparent state transfer and logging facilities one can easily turn a not fault-tolerant service into a replicated fault-tolerant service. The toolkit is built using the Isis group communication primitives discussed in Chapter 3.

5.1.6. Conclusion

We have described a simple method for making parallel Orca programs fault-tolerant. The method is fully transparent, and works for parallel programs that take input, compute, and then yield a result. The method is not intended for interactive programs, nor for real-time applications.

Our method makes use of the fact that processes in the Orca implementation communicate through indivisible reliable group communication. In a system based on point-to-point message passing, it would be harder to make a consistent checkpoint of the global state of the program. One approach might be to simulate indivisible broadcasting, for example by using the algorithm described in [Bal and Tanenbaum 1991]. This algorithm includes time-stamp vectors in each message being sent, which are used in determining a consistent ordering. Another method would be to use Chandy's and Lamport's distributed snapshot algorithm [Chandy and Lamport 1985].

This section also described an actual implementation of our system, on top of the Amoeba distributed operating system. We have identified a number of problems with some Amoeba services, in particular the process server. We managed to get around these problems, but the implementation would have been much simpler if certain restrictions in Amoeba were removed (e.g., checkpoints do not include kernel state). Finally, we have given some performance results of our system, using a simple application.

5.2. A Fault-Tolerant Directory Service

In this section, we will look at another fault-tolerant application: the directory service. The directory service exemplifies the class of applications that provide a highly reliable and highly available service through replication of data. We compare two implementations of the directory service on the Amoeba distributed operating system: one based on RPC and one based on group communication. Both have been implemented in C, not in Orca. From the comparison we conclude that for this class of applications group communication is a more appropriate paradigm than RPC. The directory service based on group communication has a simpler design and has better performance.

The directory service is a vital service in the Amoeba distributed operating system [Van Renesse 1989]. It provides, among other things, a mapping from ASCII names to capabilities. In its simplest form a directory is basically a table with 2 columns: one storing ASCII strings and the other storing the corresponding capabilities. Capabilities in Amoeba identify and protect objects (e.g., files). The set of capabilities a user possesses determines which objects he can access and which not. The directory service allows the users to store these capabilities under ASCII names to make life easier for them.

The layout of an example directory with six entries is shown simplified in Figure 5.4. This directory has one row for each of the six file names stored in it. The directory also has three columns, each one representing a different protection domain. For example, the first column might store capabilities for the owner (with all the rights bits on), the second might store capabilities for members of the owner's group (with some of the rights bits turned off), and the third might store capabilities for everyone else (with only the read bit turned on). When the owner of a directory gives away a capability for it, the capability is really a capability for a single column, not for the directory as a whole. When giving a directory capability to an unrelated person, the owner could give a capability for the third column, which contains only the highly restricted capabilities. The recipient of this capability would have no access to the more powerful capabilities in the first two columns. In this manner, it is possible to implement the UNIX protection system, as well as devise other ones for specific needs.

	Column 1	Column 2	Column 3
File1	Capability	Capability	Capability
File2	Capability	Capability	Capability
File3	Capability	Capability	Capability
File4	Capability	Capability	Capability
File5	Capability	Capability	Capability
File6	Capability	Capability	Capability

Fig. 5.4. Layout of an example directory.

The directory service supports the operations shown in Figure 5.5. There are operations to manipulate directories, to manipulate a single row (i.e., a tuple consisting of a string and a capability) of a directory, and to manipulate a set of rows. One of the most important things to know about the directory service is the frequency of the read operations (e.g., list directory) and write operations (e.g., delete directory), because these numbers influence the design. Measurements over three weeks showed that 98%

of all directory operations are reads. Therefore, both the RPC directory service and the group directory service optimize read operations.

Operation	Description
Create dir	Create a new directory
Delete dir	Delete a directory
List dir	List a directory
Append row	Add a new row to a directory
Chmod row	Change protection
Delete row	Delete row of a directory
Lookup set	Lookup the capabilities in a set of rows
Replace set	Replace the capabilities in a set of rows

Fig. 5.5. Operations supported by the directory service.

In this chapter, we focus on the implementation of the directory service and not on the reasons why this interface was chosen. This has been discussed by Van Renesse [Van Renesse 1989]. For other papers discussing the design of a naming service we refer the reader to [Curtis and Wittie 1984; Schroeder et al. 1984; Lampson 1986; Cheriton and Mann 1989].

The directory service must be highly reliable and highly available. Users rely on the directory service to store capabilities without losing them and users must always be able to access their capabilities. To fulfill these demands the directory service replicates (name, capability) pairs on multiple machines, each with its own disk. If one of the machines is unavailable, one of the other machines will be able to reply to a user's request. If one of the disks becomes unreadable, one of the other disks can be used to reply to a user's request. The key problem is to keep the replicas of a name-capability pair consistent in an efficient way. An update to a directory must be performed quickly, because otherwise many applications will run less efficiently.

We require that the directory service maintains *one-copy serializability* [Bernstein et al. 1987]. The execution of operations on the directory service must be equivalent to a serial execution of the operations on a nonreplicated directory service. To achieve this goal each operation of the directory service is executed indivisibly. The directory service does not support indivisible execution of a set of operations, as this requires atomic transactions [Gray 1978; Lampson 1981]. It also does not support failure-free operations for clients, as this requires updating a log file on each operation.

We feel that these semantics are too expensive to support, and, moreover, are seldom necessary.

Both implementations of the directory service assume clean failures. A processor works or does not work (i.e., fail-stop failures), and it does not send malicious or contradictory messages (i.e., it does not exhibit Byzantine failures). The RPC implementation also assumes that network partitions will not happen; the group implementation, however, guarantees consistency even in the case of clean network partitions (e.g., any two processors in the same partition can communicate while any two processors in different partitions cannot communicate) [Davidson et al. 1985]. Stronger failure semantics could have been implemented using techniques as described in [Cristian 1991; Barrett et al. 1990; Hariri et al. 1992; Shrivastava et al. 1992]. Again, we feel that these stronger semantics are too expensive to support, and, moreover, are overkill for an application like the directory service.

In the rest of this section, we assume that the following basic requirements for a fault-tolerant directory service are met. Each directory server should be located on a separate electrical group (with its own fuse) and all the directory servers should be connected by multiple, redundant, networks. Because the Amoeba communication primitives are implemented on top of FLIP, the latter requirement can be fulfilled. Although it could run on multiple networks without a single software modification, the current implementation runs on a single network.

The outline of this section is as follows. In Section 5.2.1 we discuss an implementation of the directory service using RPC. In Section 5.2.2 we discuss an implementation using group communication. In Section 5.2.3 we will compare these two implementations and give performance measurements for both. In Section 5.2.4 we will draw some conclusions based on the comparison and describe related work.

5.2.1. Directory Service Based on RPC

The Amoeba directory service has a long history. It started out as a simple server with few operations running on a single processor. It had no support for fault tolerance. Robbert van Renesse designed and implemented the latest version, which supports more complex operations and which is highly reliable and available [Van Renesse 1989]. Here we will give a short overview of this design and implementation.

The RPC directory service is duplicated to achieve reliability. It uses two Bullet file servers and two disk servers (see Fig. 5.6). The Bullet servers and the disk servers share the same disk. The directory service uses the Bullet files to store the directories, one copy on each Bullet server. The directory service uses the disk servers to store its administrative data. The reason for storing the administrative data directly on disk is that Bullet files are immutable. Thus, if a Bullet file would have been used to store the administrative data, then each time a bit has to be changed in the administrative data, a complete new Bullet file has to be created, containing a copy of the previous contents plus the changed bit. To avoid this overhead, the directory service stores the administrative data directly on mutable media: the raw disk.

The actual implementation of the directory service is slightly more complicated: each server caches recently used directories in RAM. When a request is received, a directory server looks first in its cache to see if the directory is stored there. If so, it can reply immediately without going to the Bullet server. If not, the directory server performs an RPC with the Bullet server to get the requested directory and stores it in its cache for subsequent requests. The cache is there to avoid the overhead of making an additional RPC with the Bullet server.

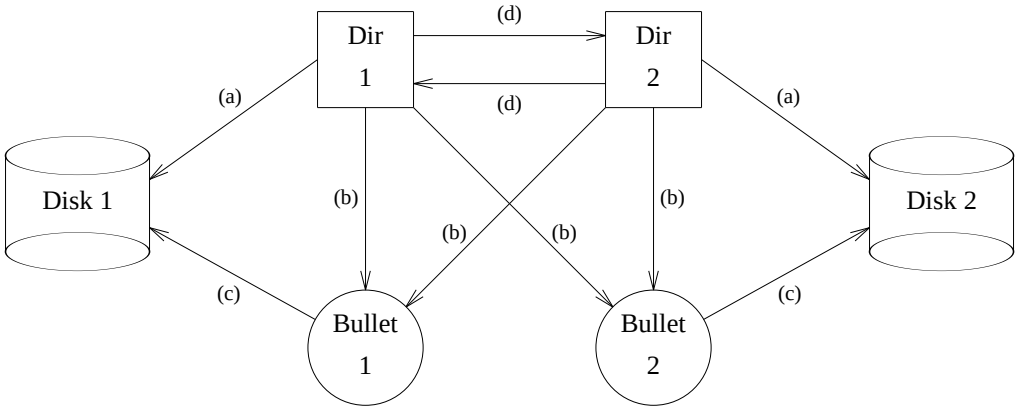


Fig. 5.6. The configuration. The boxes represent the directory servers, while the circles represent the Bullet file servers. The arrows imply the client/server relationship. (a) Administrative Data; (b) Directories; (c) Files; (d) Intentions and other internal messages.

The administrative data are stored on a raw disk partition of n fixed-length blocks. Blocks 1 to $n - 1$ contain an object table of (Capability 1, Capability 2) pairs. The number stored in the *object* field of a directory capability indexes into this table. The capabilities in an object table entry point to two identical Bullet files that store the directory and the *check* field for access protection. An update of the directory means that the capabilities have to be replaced by the capabilities for the new Bullet files. These capabilities are actually overwritten, which is why raw disks and not Bullet files are used.

Block 0 is the *commit block* (see Fig. 5.7). The *copy mode* indicates whether the server is functioning alone or together with another server. During normal operation the *copy mode* should be set to 2 (called *two-copy mode*), but if this server knows for sure that the other server is unavailable, the *copy mode* will be set to 1 (called *one-copy mode*). In one-copy mode, the server can perform operations for users without the consent of the other server. If the server does not know for sure that the other server is unavailable (e.g., because the network is partitioned), new updates cannot proceed to prevent inconsistencies. Lookup and list operations can still be executed, however. When the other server recovers, the *copy mode* is set to 2 again.

Each time an update operation is performed, the *sequence number* is increased.

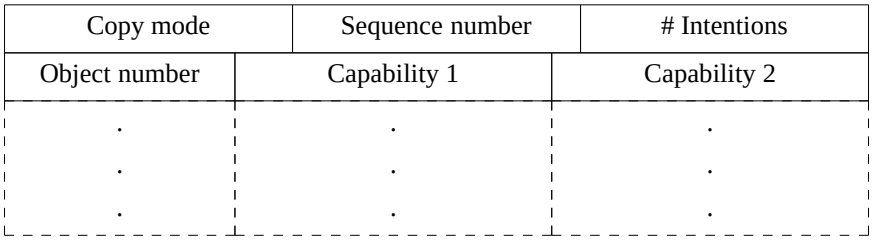


Fig. 5.7. Layout of the commit block.

The *sequence number* is sometimes required in recovery operations to tell which of the two servers has the most recent version of the administration file. The rest of the commit block is an *intentions list* of updates that may or may not have been performed yet. Each entry in the list specifies the object number of the directory to be replaced, and the capabilities of the Bullet files that store the new directory. This information makes updates to the administrative data crash resistant.

Default Operation

Let us now look at how the directory service keeps the copies of the directories consistent. It uses the *duplication protocol*, which is depicted in Figure 5.8. The protocol combines replication and concurrency control. It is only suited for a directory service that is duplicated; a more complex protocol (two-phase commit) is needed for, for example, a triplicated service.

The initiator is the directory server that got a user request involving the update of one or more directories. Read operations can be handled by either directory server without the need for communication between the servers. For write operations, the initiator records that it is busy, and creates the new versions of the directories. It creates, in its own internal RAM memory, a new version of the commit block with the capabilities of the new directories in the intentions list and an increased sequence number.

If the server is in two-copy mode, it will have to get the consent of the other server. It sends the new commit block to the other server and waits for a reply. If the request is rejected, this means that the other server also wants to do an update. The initiator undoes all operations and tries again later. To prevent a second clash, one of the servers will always wait a random time before retrying.

If no reply arrives within a certain time, the initiator makes sure that the other server has really crashed. A reliable method is to reset the processor that runs the server, thus killing the server. When the initiator knows for sure that the other server is unavailable, it sets the copy mode in the RAM copy of the commit block to 1.

Now, in either case, it writes the commit block to disk. At this point the operation has officially succeeded, and a reply is sent to the client. The initiator and the other server, if alive, still have some administrative affairs to deal with. The other server has to write its commit block to disk. Both have to update their object tables and

```
Initiator:
        lock(busy);                                 /* set busy variable */
        create new directories on Bullet files;     /* the other copies are made later */
        SequenceNo += 1;
        put Bullet capabilities in intentions list; /* build list in RAM */
        if (CopyMode == 2) {                        /* two-copy mode? */
                send intention to other server;     /* perform RPC */
                if (RpcFailure) {                   /* did RPC fail? */
                        reset other server;         /* force the other server to fail */
                        CopyMode = 1;               /* go to one-copy mode */
                }
                if ( reply == rejected) {           /* other server rejects request ? */
                        remove newly created Bullet files;
                        SequenceNo –= 1;
                        if (me == 2) wait random time;
                        try again;                  /* start over again */
                }
        }
        write commit block to disk;                 /* commit intentions */
        update object table;                        /* replace capability */
        update directory cache;                     /* update RAM cache */
        send reply to client;
        if (me == 1 || CopyMode == 1)               /* only one server must delete old dirs */
                remove old Bullet files;
        if (full intentions list) {                 /* once in while clean intentions */
                write changes to object table to disk;
                write commit block with empy intentions list to disk;
        }
        unlock(busy);                               /* done */

Other server:
        if (!trylock(busy)) reply rejected;
        else {                                      /* got the lock */
                update object table;
                invalidate cache;
                reply OK;
                add intentions to commit block on disk;/* commit */
                if (me == 1 || CopyMode == 1)       /* only one server must delete old dirs */
                        remove old Bullet files;
                if (full intentions list) {         /* once in while clean intentions */
                        write changes to object table to disk;
                        write commit block with empy intentions list to disk;
                }
                unlock(busy);
        }
```

Fig. 5.8. The duplication protocol.

caches, and periodically have to clean up the intentions lists stored in the commit blocks on disk. The cleaning up is not done on every operation to avoid additional disk accesses.

When the client's RPC returns successfully, the user knows that one new copy of

the directory is stored on disk (as a Bullet file) and that the other server has received the request and has stored the request in its intentions list on disk or will do so shortly. It also knows that the other copy of the directory will be generated lazily (in the background) when the directory service is not busy.

Let us briefly analyze the cost of the duplication protocol in terms of communication cost and disk operations. In case of a read request, no communication is needed and no disk accesses are needed (if the directory is in the cache). In case of a write request, one Bullet file per directory has to be created (the other Bullet file is created later in the background), one RPC needs to be done, and one disk write for the commit block needs to be done. In the current implementation, the intentions list is cleaned by a watchdog thread running in the background or when it fills up.

Recovery From Failures

The directory service can recover from one processor failure. As the directory service is only duplicated, it cannot recover from network partitions, since in this case the two copies could become inconsistent.

The protocol for recovery works as follows (see Fig. 5.9). When a directory server comes back again, it reads its commit block from disk. If it is in one-copy mode, it can continue immediately with its usual operation. If it is in two-copy mode, it checks to see if the other server is running. If not, it waits until the other server is restarted. If the other server is running, it asks for a copy of the object table and the commit block (if the other side has a higher sequence number). After the server is up-to-date, it tells the other server to enter two-copy mode. If the RPC with the other server succeeds, the server enters normal operation.

```
Recovery:
    if (CopyMode == 2) {
        alive = true if other server is alive;
        if (!alive) try again;                  /* recovery requires both servers to be up */
        copy objects and commit block;          /* if other side has higher sequence number */
        tell other server to enter two-copy mode;
        if (!OK) try again;                     /* did server fail during recovery ? */
    }
    continue operation as usual;                /* recovery is done */
```

Fig. 5.9. The recovery protocol.

A server that crashed in two-copy mode cannot recover without the other server being up. The reason is that the other server is likely to be in one-copy mode and therefore has the most recent versions of the directories. A server that crashed in one-copy mode can enter normal operation immediately, because only one server can have crashed in one-copy mode. The other server must be in two-copy mode and is thus still down or waiting for the one-copy mode server to come up.

5.2.2. Directory Service Based on Group Communication

Having described the RPC design and implementation, we will now look at the group design and implementation. Unlike the RPC directory service, the group implementation is triplicated (though four or more replicas are also possible without changing the protocol) and uses active replication. The implementation that we will describe tolerates one server failure, but works even in the face of network partitions, while the RPC directory service cannot tolerate network partitions. If one assumes that network partitions do not happen, then the current group implementation (with a very few minor modifications) can tolerate 2 server failures. To keep the copies consistent, it uses a modified version of the read-one write-all policy, called *accessible copies* [El Abbadi et al. 1985]. Recovery is based on the protocol described by Skeen [Skeen 1985].

The organization of the group directory service is depicted in Figure 5.10. The directory service is currently built out of three directory servers, three Bullet servers, and three disk servers. Like the RPC implementation, each directory server and its associated Bullet server share a single disk. Unlike the RPC implementation, each directory server only uses one Bullet server to store only one copy of each directory in a separate Bullet file.

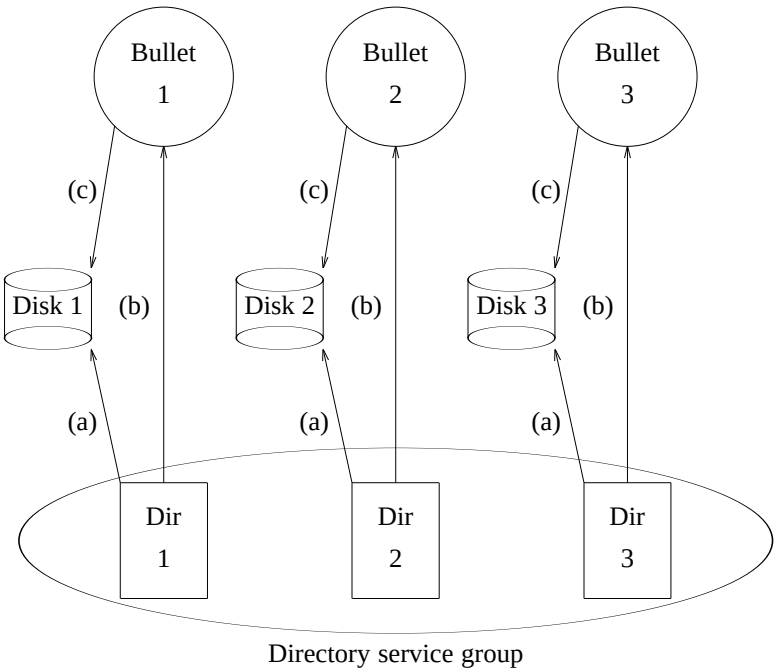


Fig. 5.10. Organization of the directory service based on group communication. (a) Administrative data; (b) Directories; (c) Files.

The directory servers initially form a group with a resilience degree, *r*, of 2. This means that if *SendToGroup* returns successfully, it is guaranteed that all three have received the message and thus that, even if two processors fail, the message will still be processed by the third one. Furthermore, it is guaranteed that even in the presence of

communication and processor failures, each server will receive all messages in the same order. The strong semantics of *SendToGroup* make the implementation of the group directory service simple.

The group service stores almost the same administrative data as the RPC service. The administrative data are stored on a raw disk partition of n fixed-length blocks. Blocks 1 to $n-1$ contain an object table of capabilities, indexed by object number. Each capability in the object table points to a Bullet file that stores the directory, a random number for access protection, and the sequence number of the last change. The group implementation stores only a single capability, because each directory server stores its own copy of the directory. Another difference is that the sequence number for the last change is also stored with the directory. In the group implementation, if a directory is changed only the file has to be written, while in the RPC implementation both a Bullet file and the commit block have to be written. In this way, the group implementation saves one disk operation (writing the commit block), though the RPC implementation could have used the same optimization.

Block 0, the commit block for the group directory service is shown in Figure 5.11. There are three main differences with the RPC commit block: 1) there is no *intentions list*; 2) instead of the *copy mode*, there is a *configuration vector*; 3) there is an additional field to keep track whether a server is recovering. (The actual implementation needs an intentions list for the directory operation *replace set*, but to simplify our presentation we ignore this operation.) The configuration vector is a bit vector, indexed by server number. If server 2, for example, is down, bit 2 in the vector is set to 0. It describes the last configuration with a majority of which the server was a member.

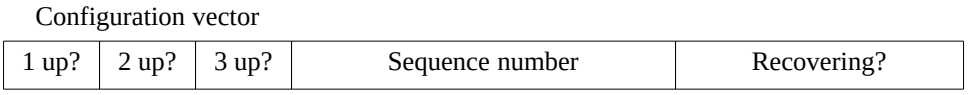


Fig. 5.11. Layout of the commit block.

During recovery, the sequence number is computed by taking the maximum of all the sequence numbers stored with the directory files and the sequence number stored in the commit block. At first sight it may seem strange that a sequence number is also stored in the commit block, but this is needed for the following case. When a directory is deleted, the reference to the Bullet file containing the directory and the sequence number is deleted, but the server must record somewhere that it performed an update. The sequence number in the commit block is used for this case. It is only updated when a directory is deleted.

The *recovering* field is needed to keep track whether a server crashed during recovery. If this field is set, the server knows that it crashed during recovery. In this case, it sets the sequence number to zero, because its state is inconsistent. It may have recent versions of some directories and old versions of other directories. The sequence

number is set to zero to ensure that other servers will not try to update their directories from a server whose state is inconsistent.

Default Operation

The protocol to keep the directories consistent and to perform concurrency control is given in Figure 5.12. A server in the group directory service consists of several threads: the server threads and one group thread. The server threads are waiting for a request from a client. The group thread is waiting for an internal message sent to the group. At each server there can be multiple server threads, but there is only one group thread. A server thread that receives a request and initiates a directory operation is called the *initiator*.

The initiator first checks if the current group has a majority (i.e., at least two of the three servers must be up). If not, the request is refused; otherwise the request is processed. The reason why even a read request requires a majority is that the network might become partitioned. Consider the following situation. Two servers and a client are on one side of the network partition and the client deletes the directory *foo*. This update will be performed, because the two servers have a majority. Now assume that the two servers crash and that the network partition is repaired. If the client asks the remaining server to list the directory *foo*, it would get the contents of a directory that it successfully deleted earlier. Therefore, read requests are refused if the group of servers does not have a majority. (There is an escape for system administrators in case two servers lose their data forever due to, for example, a head crash.)

As in the RPC implementation, read operations can be handled by any server without the need for communication between the servers. When a read request is received, the initiator checks if the kernel has any messages buffered using *GetInfoGroup*. If so, it blocks to give the group thread a chance to process the buffered messages; before performing a read operation, the initiator has to be sure that it has performed all preceding write operations. If a client, for example, deletes a directory and then tries to read it back, it has to receive an error, even if the client requests were processed at different directory servers. As messages are sent using $r = 2$, it is sufficient to see if there are any messages buffered on arrival of the read request. Once these buffered messages are processed, the initiator can perform the read request.

Write operations require communication among the servers. First, the initiator generates a new *check* field, because all the servers must use the same *check* field when creating a new directory. Otherwise, some servers may consider a directory capability invalid, whereas others consider it valid. The initiator broadcasts the request to the group using the primitive *SendToGroup* and blocks until the group thread received and executed the request. Once it is unblocked, it sends the result of the request back to the client.

The group thread is continuously waiting for a message sent to the group (i.e., it is blocked in *ReceiveFromGroup*). If *ReceiveFromGroup* returns, the group thread first checks if the call to *ReceiveFromGroup* returned successfully. If not, one of the

```
Initiator:
        if(!majority()) return failure;                 /* majority? */
        if(read_operation(request)) {                   /* read request? */
                GetInfoGroup(&group_state);             /* check for buffered messages */
                buffered_seqno = buffered(&group_state);
                wait until seqno = buffered_seqno;
        } else {                                        /* write request */
                generate check-field;                   /* for new directory */
                SendToGroup(request, check-field, me);
                wait until group thread has received and executed the request;
        }
        send reply to client;

Group thread:
        if(group failure) {                             /* did a server fail? */
                rebuild majority of group;              /* call ResetGroup */
                if(group rebuild failed) enter recovery;
                GetInfoGroup(&group_state);             /* get info about rebuilt group */
                write commit block;                     /* update configuration vector */
                try again;                              /* start receiving again */
        } else {
                create directory on Bullet file;        /* use supplied check-field */
                update cache;
                update object table;
                write changed  object table to disk;    /* commit */
                increase_and_wakeup(seqno);
                if(sender == me) wakeup initiator;
                remove old Bullet files;
        }
```

Fig. 5.12. Protocol to ensure consistency of the copies of a directory.

servers must have crashed. In this case, it rebuilds the group by calling *ResetGroup*, updates its commit block, and calls *ReceiveFromGroup* again. If it does not succeed in building a group with a majority of the members of the original group, the server enters recovery mode.

If *ReceiveFromGroup* returns successfully, the server creates the new directories on its Bullet server, updates its cache, updates its object table, and writes the changed entry in the object table to its disk. As soon as one server has written the new entry to disk, the operation is committed. If no server fails, each server will receive all requests and service all requests in the same order and therefore all the copies of the directories stay consistent. There might be a small delay, but eventually each server will receive all messages.

When the client's RPC returns successfully, the user knows that one new copy of the directory is stored on disk and that at least two other servers have received the request and stored the new directory on disk, too, or will do so shortly. If one server fails, the client can still access his directories.

Let us again analyze the cost of a directory operation in terms of communication

cost and disk operations. As in the RPC implementation, read operations do not involve communication or disk operations (if the requested directory is in the cache). Write operations require one group message sent with $r = 2$, a Bullet operation to store the new directory, and one disk operation to store the changed entry in the object table. Compared to the RPC implementation, the number of disk operations is smaller. The RPC implementation requires an additional disk operation to store the intentions list. The number of messages in the group service, however, is higher. A *SendToGroup* with $r = 2$ requires 5 messages, whereas an RPC only requires 3 messages. The cost of sending a message, however, is an order of magnitude less than the cost of performing a disk operation. Thus, roughly, the performance of the group implementation is better than the performance of the RPC implementation, while providing more fault tolerance and a higher availability.

This analysis is not completely fair, however. If the RPC implementation, like the group implementation, had stored the sequence number with the directory files and thereby avoided one disk write for the commit block, the performance of the RPC implementation would have been better, as it would send fewer messages. On the other hand, if the RPC service had been triplicated, it would have been slower than the group service, because then it would have sent more messages.

Recovery Protocol

A server starts executing the recovery protocol when it is a member of a group that forms a minority or when it comes up after having been down. The protocol for recovery of the group service is more complicated than the protocol for the RPC service, because more servers are involved. Consider the following sequence of events in a group of three servers that is up and running. Server 3 crashes. Servers 1 and 2 rebuild the group, so their *configuration vector*s have the value 110 (1 and 2 are up; 3 is down). Now, both 1 and 2 also crash. When server 1 comes up again, its vector reads 110, but on its own it cannot form a group. To execute a client update request, a majority of the servers must be up and form one group; otherwise, copies of a directory could become inconsistent, for example, in the case of a network partition.

If server 3 also comes up, its vector reads 111. At first sight, it may appear that 1 and 3 can form a group, as together they form a majority. However, this is not sufficient. Server 2, who is still down, may have performed the latest update. To see this, consider the following sequence of events just before 1 and 2 crashed. A client update request is received by server 1, it successfully broadcasts it to server 1 and 2. Now both 1 and 2 have the message buffered. It can happen that 1 crashes before processing the message, while 2 crashes after processing the message. In this case, server 2 has the latest version of the directories and thus 1 and 3 cannot form a new group and start accepting requests.

Now assume that server 2 comes up instead of server 3. The *configuration vector* of both servers 1 and 2 read 110. From this information they can conclude that 3 crashed before 1 and 2 did. Furthermore, no update can have been performed after 1 or

2 crashed, because there was no majority. Servers 1 and 2 together are therefore sure that one of them has the latest version of the directories. Thus, they can recover without server 3 and use the *sequence number* to determine who actually has the latest version.

In general, two conditions have to be met to recover:

1. The new group must have a majority to avoid inconsistencies during network partitions.
2. The new group must contain the set of servers that possibly performed the latest update.

It is the latter requirement that makes recovery of the group service complicated. During recovery the servers need an algorithm to determine which are the servers that failed last.

Such an algorithm exists; it is due to Skeen [Skeen 1985], and it works as follows. Each server keeps a *mourned set* of servers that crashed before it. When a server starts recovering, it sets the new group to only itself. Then, it exchanges with all other alive servers its mourned set. Each time it receives a new mourned set, it adds the servers in the received *mourned set* to its own *mourned set*. Furthermore, it puts the server with whom it exchanged the mourned set in the new group. The algorithm terminates when all servers minus the *mourned* set are a subset of the new group.

Figure 5.13 gives the complete recovery protocol. When a server enters recovery mode, it first tries to join the group. If this fails, it assumes that the group is not created yet and it creates the group. If, after a certain waiting period, an insufficient number of members have joined the group, the server leaves the group and starts all over again. It may have happened that two servers have created a new group (e.g., one server on each side of a network partition) and that they both cannot acquire a majority of the members.

Once a server has created or joined a group that contains a majority of all directory servers, it executes Skeen's algorithm to determine the set of servers that crashed last, the *last set*. If this set is not a subset of the new group, the server starts all over again, waiting for servers from the *last set* to join the group. If the *last set* is a subset of the new group, the new group has the most recent version of the directories. The server determines who in the group has them and gets them. Once it is up-to-date, it writes the new configuration to disk and enters normal operation.

The recovery protocol can be improved. Skeen's algorithm assumes that network partitions do not occur. To make his algorithm work under our assumption that network partitions can happen, we forced the servers that form a group with a minority of the number of servers to fail. Now the recovery protocol will fail in certain cases in which it is actually possible to recover. Consider the following sequence of events. Server 1, 2, and 3 are up; server 3 crashes; server 1 and 2 form a new group; server 2 crashes. Now as we want to tolerate network partitions correctly, we forced server 1 to fail. However, this is too strict. If server 1 stays alive and server 3 is restarted, server

```
Recovery:
        re-join server group or create it;
        while (minority && !timeout) {              /* wait for some time */
                GetInfoGroup(&group_state);
        }
        if(minority) try again;                     /* start over again */
        newgroup[me] = 1;                           /* initialize new group vector */
        SequenceNo[me] = Sequence number;           /* initialize sequence number vector *.
        initialize mourned vector from configuration vector;
        for (all members in group) {                /* exchange info with each server */
                exchange info with server s;        /* exchange mourned set and sequence no */
                if (success) {                      /* RPC succeeded? */
                        newgroup[s] = 1;            /* add server to new group */
                        SequenceNo[s] = Sequence number;
                        mourned set += received mourned set;/* take union */
                }
        }
        last = all servers – mourned set;           /* the servers that performed the last update */
        if(last is not subset of new group) try again;
        s = HighestNumber( SequenceNo );            /* determine server with highest sequence no */
        get copies of latest version of directories from s;
        if (!success) try again;                    /* succeeded in getting copies? */
        write commit block;                         /* store configuration vector */
        enter normal operation;                     /* recovery is done */
```

Fig. 5.13. Recovery protocol for group directory service.

1 and 3 can form a new group, because server 1 must have available all the updates that server 2 could have performed. The rule in general is that two servers can recover, if the server that did not fail has a higher sequence number, as in this case it is certain that the new member has not formed a group with the (now) unavailable member in the meantime.

5.2.3. Experimental Comparison

Both the RPC and group service are operational. The RPC service has been in daily use for over two years. The group directory service has been used in an experimental environment for several months and will shortly replace the RPC version. Both directory services run on the same hardware: machines comparable to a Sun 3/60 connected by 10 Mbit/s Ethernet. The Bullet servers run on Sun 3/60s and are equipped with Wren IV SCSI disks.

Performance Experiments with Single Client

We have measured the failure-free performance of three kinds of operations on an almost quiet network. The results are shown in Figure 5.14. The first experiment measures the time to append a new (name, capability) pair to a directory and delete it subsequently (e.g., appending and deleting a name for a temporary file). The second experiment measures the time to create a 4-byte file, register its capability with the

directory service, look up the name, read the file back from the file service, and delete the name from the directory service. This corresponds to the use of a temporary file that is the output of the first phase of a compiler and then is used as an input file for the second phase. Thus, the first experiment measures only the directory service, while the second experiment measures both the directory and file service. The third experiment measures the performance of the directory server for one lookup operations.

Operation (# copies)	**Group (3)**	**RPC (2)**	**Sun NFS (1)**	**Group +NVRAM (3)**
Append-delete	184	192	87	27
Tmp file	215	277	111	52
Directory lookup	5	5	6	5

Fig. 5.14. Performance of three kinds of directory operations for three different Amoeba implementations and for one UNIX implementation. All times are in msec.

For the ‘‘append-delete’’ test and for the ‘‘tmp file’’ test, the implementation using group communication is slightly more efficient than the one using RPC. Thus, although the group directory service is triplicated and the RPC implementation is only duplicated, the group directory service is more efficient. The reason is that the RPC implementation uses one additional disk operation to store intentions and the new sequence number. For read operations, the performance of all implementations is the same. Read operations do not involve any disk operations, as all implementations cache recently used directories in RAM, and involve only one server.

For comparison reasons, we ran the same experiments using Sun NFS; the results are listed in the third column. The measurements were run on SunOS 4.1.1 and the file used was located in */usr/tmp/*. NFS does not provide any fault tolerance or consistency (e.g., if another client has cached the directory, this copy will not be updated consistently when the original is changed). Compared to NFS, providing high reliability and availability costs a factor of 2.1 in performance for the ‘‘append-delete’’ test and 1.9 in performance for the ‘‘tmp file’’ test.

The dominant cost in providing a fault-tolerant directory service is the cost for doing the disk operations. Therefore, we have implemented a third version of the directory service, which does not perform any disk operations in the critical path. Instead of directly storing modified directories on disk, this implementation stores the modifications to a directory in a 24 Kbyte NonVolatile RAM (NVRAM). When the server is idle or the NVRAM is full, it applies the modifications logged in NVRAM to the directories stored on disk. Because NVRAM is a reliable medium, this implemen-

tation provides the same degree of fault tolerance as the other implementations, while the performance is much better. A similar optimization has been used in [Daniels et al. 1987; Liskov et al. 1991; Hariri et al. 1992].

Using NVRAM, some sequences of directory operations do not require any disk operations at all. Consider the use of */tmp*. A file written in */tmp* is often deleted shortly after it is used. If the append operation is still logged in NVRAM when the delete is performed, then both the append and the delete modifications to */tmp* can be removed from NVRAM without executing any disk operations at all.

Using group communication and NVRAM, the performance improvements for the experiments are enormous (see the fourth column in Fig. 5.14.). This implementation is 6.8 and 4.3 times more efficient than the pure group implementation. The implementation based on NVRAM is even faster than Sun NFS, which provides less fault tolerance and has a lower availability. If the RPC service had been implemented with NVRAM, one could expect similar performance improvements.

Performance Experiments with Multiple Clients

To determine the performance of the directory services for multiple clients we ran three additional experiments. The first experiment measures the throughput for lookup operations; its results are depicted in Figure 5.15. The graph shows the total number of directory lookup operations that were processed by a directory service for a varying number of clients. A rough estimate of the maximum number of lookup operations that, in principle, can be processed per second can be easily computed. The time needed by a server to process a read operation is roughly equal to 3 msec (the time for a lookup operation minus the time to perform an RPC with the server). The maximum number of read operations per server is therefore 333 per second. Thus, the upper bound on read operations for the group service using 3 servers is 1000 per second and for the duplicated RPC implementation it is 666 per second.

Neither service achieves its upper bound, because the client requests are not evenly distributed among the servers. To understand this, recall from Chapter 2 how a server is located. The first time a client performs an RPC with some service, its kernel locates the service by calling *flip_broadcast* with as argument a message containing the port *p* for the requested service. Every server that listens to the port *p* answers with an RPC HEREIS message. The client's kernel stores the FLIP address for each server that answers in its port cache and sends the request to the first server that replied. If at some point one of the servers is busy and is not listening to *p* when a request comes in, its kernel sends a NOTHERE back to the client's kernel. The client's kernel removes the server's FLIP address from its port cache and selects another server from its port cache or locates the service again if its port cache does not contain an alternative.

This heuristic for choosing a server is not optimal. Some clients may pick the same server, while another server is idle. From the graph one can see this happen. This possibility was also reflected in our measurements. The numbers depicted are averages over a large number of runs, but the standard deviation is high. In some runs,

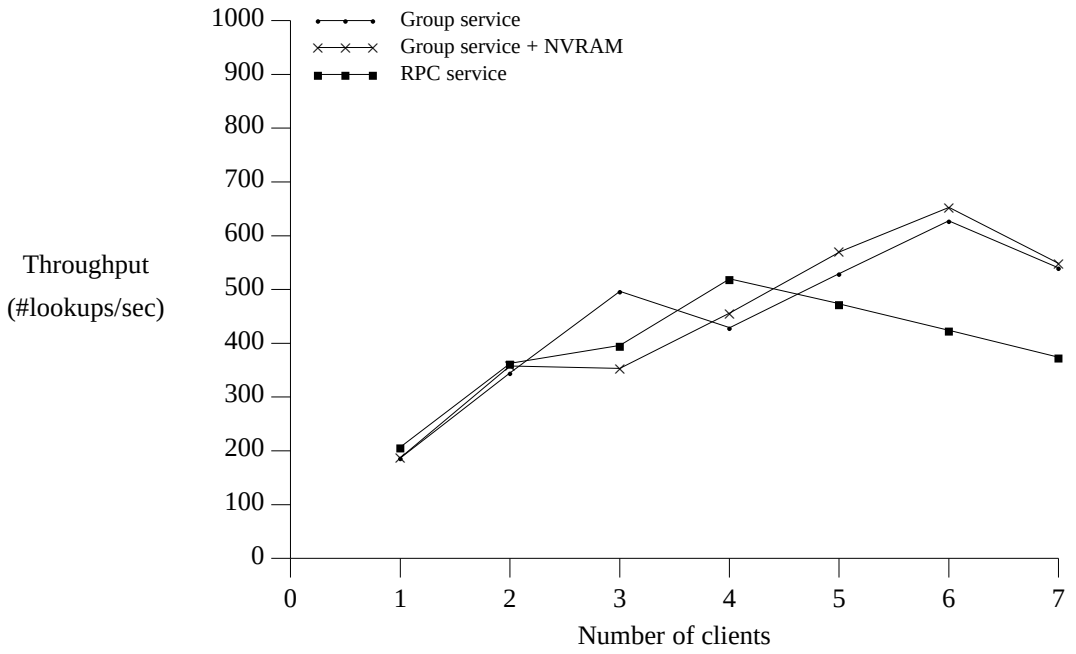


Fig. 5.15. Throughput for read operations

the standard deviation was almost 100 operations per second. The heuristic, however, is good enough that with higher load the clients requests are evenly distributed among the servers. One can also conclude from the graph that the RPC directory service can support fewer clients than the group service. The RPC directory service gets overloaded with 520 requests per second, whereas the group service gets overloaded with 652 requests per second.

Figure 5.16 shows the throughput for the ‘‘append-delete’’ test. This experiment measures the maximum number of pairs of append-delete operations that the service can support per second. Again, an upper bound can easily be estimated for each service. Processing a pair of append-delete operations takes roughly 22 msec in the group NVRAM service, 179 msec for the group service, and 187 for the RPC service. As write operations cannot be performed in parallel, the upper bounds per service are 45, 5, and 5. All three implementations reach the upper bound.

5.2.4. Discussion and Comparison

Making a fair comparison between the group directory service and the RPC directory service is hardly possible, as both services assume different failure modes. The RPC service is duplicated and does not provide consistency in the face of network partitions, whereas the group service is triplicated and does provide consistency in the face of network partitions. Furthermore, the RPC implementation employs lazy replication, whereas the group implementation employs active replication, resulting in a higher degree of reliability and availability for the group directory service. After the

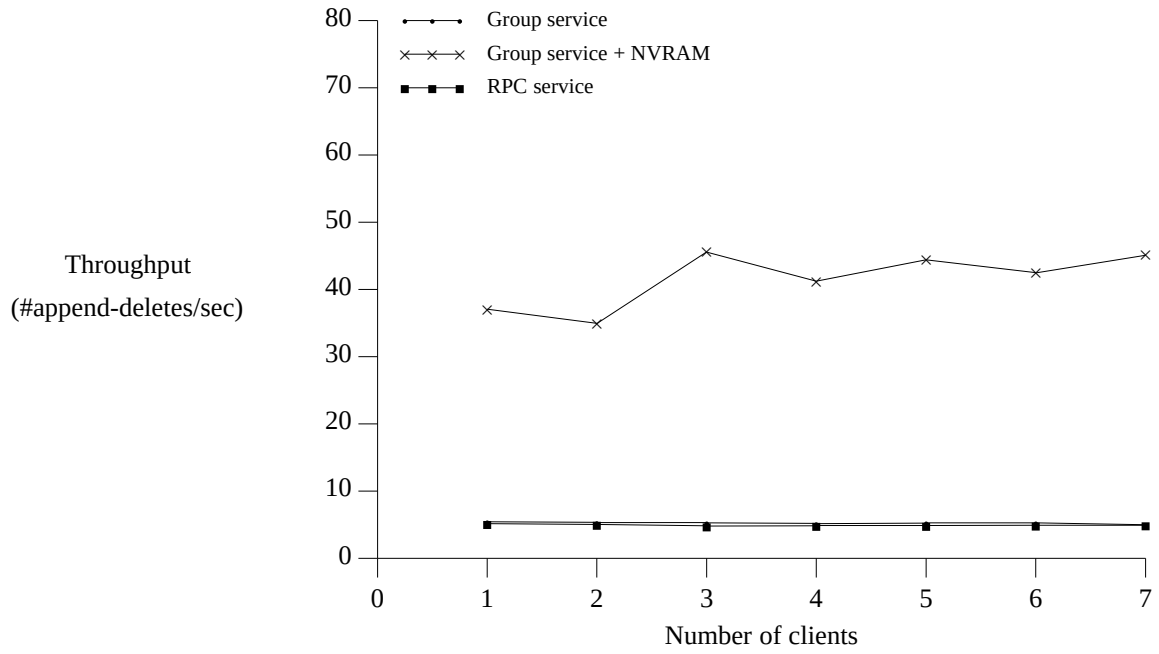


Fig. 5.16. Throughput for pairs of append-delete operations. As append and delete operations are both write operations, the actual write throughput is twice as high.

RPC directory service has performed an update on a directory, the new directory is directly stored on only one Bullet file. If the Bullet server storing this file crashes before the second replica is generated, the directory will become unavailable. In the group directory service this cannot happen, because the service creates all replicas at about the same time before the client is told that the update has succeeded.

Although the group service provides a higher reliability and availability, its protocols for normal operation (no failures) are as simple as the RPC protocols. The group recovery protocols are more complex, but this is due to the fact that the group service is built out of three servers instead of out of two. A three server implementation of the RPC service would require a similar protocol for recovery as used in the group service.

The performance of the group directory service is better than the RPC directory service. This is, however, mainly due to the fact that the group implementation avoids one disk write. The RPC directory service could have been implemented in such a way that it also avoids the additional disk write. Such an implementation is likely to have the same performance as the group implementation. On the other hand, if the RPC directory service had been triplicated, it would have had to implement two-phase locking [Eswaran et al. 1976], resulting in higher communication overhead compared with the group implementation.

Summarizing, the group directory service is hard to compare with the RPC directory service due to differences in the failure semantics and differences in the imple-

mentations that are not related to using group communication or using RPC. However, we believe that the comparison gives enough insight in all the design choices and implementation issues that an RPC directory service with the same specification as the group directory service will be more complicated and has no better performance than a group implementation.

There is an extensive literature on designing and building fault-tolerant systems, both covering practical and theoretical research. It is outside the scope of this thesis, however, to review all this work. Instead we focus on a number of actual implementations of fault-tolerant file and directory services in systems similar to Amoeba.

Marzullo and Schmuck describe a fault-tolerant implementation of Sun's Network File System (NFS) using the Isis toolkit [Marzullo and Schmuck 1988]. As the authors did not want to change the client side nor the server side of NFS, they introduced an extra level of indirection. Client processes do not talk directly with the file service, but go through an intermediate process, called an *agent*. The agents hide from the clients that the file service is replicated and use internally one of Isis's broadcast primitives to keep their state consistent. The agents update the replicas of a file using regular Sun RPC, because to employ broadcast would have meant changing the file servers to use Isis.

Harp is another approach to increase the fault tolerance of NFS [Liskov et al. 1991]. Unlike Marzullo and Schmuck, the authors of Harp decided to change the file server to avoid an extra level of indirection. Harp is based on a primary copy protocol [Alsberg and Day 1976]. Clients communicate with one designated server, called the *primary*, to perform operations. The other servers are termed *secondaries*. On a write operation, the primary first sends the results to secondaries before sending a reply to the client. All servers store the result in NVRAM and copy the result lazily to disk to improve performance. If the primary crashes, the secondaries elect a new primary.

Another fault-tolerant file system is Coda [Satyanarayanan 1990]. Coda replicates files at the server side and also caches files at the client side. The clients cache whole files, so even if all servers fail, the clients are able to continue working with the cached files. If client and servers are connected, callbacks are used to keep the caches of the clients and servers consistent. The servers themselves use active replication and an optimistic variant of the read-one write-all policy to keep replicas consistent. The implementation is based on a parallel RPC mechanism that exploits the multicast capability of a network [Satyanarayanan and Siegel 1990].

Another approach to a fault-tolerant distributed file system is Echo [Hisgen et al. 1990]. Like Harp, Echo uses a primary copy scheme. Unlike Harp, it does not perform replication at the file level, but at the level of an array of disk blocks. One of the reasons for doing so it that Echo uses *multiported* disks, which can be accessed by multiple servers. The multiported disks and replication at the level of disk blocks allow a primary to continue working even if all secondaries have failed. The primary can directly write to all disks without having to go through the secondaries.

A fault-tolerant directory service is described by Mishra, Peterson, and Schlicht-

ing [Mishra et al. 1989]. This directory service also uses active replication, but it is based on the assumption that operations are idempotent. It uses Psync's protocols to enforce an ordering on the messages sent by the servers [Peterson et al. 1989]. To enhance concurrency of the directory service operations, the directory service uses the partial ordering and the property that some operations are commutative (e.g., list directory and lookup entry). To be able to recover, the servers checkpoint their state to non-volatile storage. Using the checkpoint and the partial order among messages, the service can reconstruct the state before a failure.

Daniels and Spector describe an algorithm designed for replicated directories [Daniels and Spector 1983; Bloch et al. 1987]. Their algorithm is based on Gifford's weighted voting [Gifford 1987]. The algorithm exploits the observation that many operations on a single directory entry can be performed in parallel if the operations access different entries. Simulations done by the authors show that the additional cost for their algorithm is low, while it provides better performance.

5.2.5. Conclusion

We have tried to support the claim that a distributed system should not only support RPC, but group communication as well. Group communication allows simpler and more efficient implementations of a large class of distributed applications. As an example to demonstrate the claim we looked in detail at the design and implementation of a fault-tolerant directory service. The directory service using group communication is not only easier to implement, but also more efficient.

5.3. Summary and Discussion

In this chapter we have discussed two different approaches to making applications fault-tolerant for two different application areas. We considered long running noninteractive parallel applications and distributed services that need to be highly available and highly reliable. Both approaches are based on group communication, but in a different way. The fault-tolerant Orca RTS uses group communication with a resilience degree of zero. The directory service uses group communication with a higher resilience degree. Thus, the directory service pays for the fault tolerance each time a message is sent to the group, whereas the RTS only pays for fault tolerance when a checkpoint is made. On the other hand, in the approach used by the directory service recovery is faster than in the approach used by the RTS. If a failure occurs in the RTS, the RTS starts from the latest consistent checkpoint and recomputes all the work done since the latest checkpoint. The directory service can continue serving requests as soon as the group is rebuilt.

Another important difference is that the RTS approach is less general than the approach used by the directory service. The RTS relies on a fault-tolerant file system to store checkpoints and the state file, whereas the directory service does not rely on other fault-tolerant services. Furthermore, the approach used by the RTS works only

for noninteractive programs that are willing to tolerate slow recovery. The approach used by directory service can be used by any other distributed service.

The most important conclusion that one can draw from this chapter is that group communication is a suitable tool to build fault-tolerant applications with. Due to reliable totally-ordered group communication, the fault-tolerant RTS is easy to implement, while still providing good performance. Other approaches not based on group communication require more complicated protocols, but may provide better performance. By using group communication in the directory service, the service is not only easy to implement, but also gives a better performance than the same service implemented using RPC.

Notes

The section describing a fault-tolerant implementation of Orca contains material from the paper by Kaashoek, Michiels, Tanenbaum, and Bal published in the proceedings of the *Third Symposium on Experiences with Distributed and Multiprocessor Systems* [Kaashoek et al. 1992a]. The program *gax* was written by Raymond Michiels. A short position paper by Kaashoek, Tanenbaum, and Verstoep published at the *Fifth ACM SIGOPS European Workshop* contains a preliminary version of the comparison of a directory service based on RPC and group communication [Kaashoek et al. 1992b]. Most of the material in this thesis on the comparison is new. The description of the RPC directory service borrows from the work described in the thesis of Van Renesse [Van Renesse 1989]. Kees Verstoep has turned the prototype group directory service into a production quality service.

6

SUMMARY

Most distributed systems provide only RPC to build parallel and distributed applications. However, RPC is inherently point-to-point communication and many applications need one-to-many communication. Using one-to-many communication, processes can send a message to n destinations. The abstraction providing one-to-many communication is called group communication. This dissertation describes the design, implementation, performance, and usage of a system providing group communication.

A straightforward method of providing group communication is to use RPC. If a process sends a message to a group of n processes, it performs $n - 1$ RPCs. This implementation, however, is far from optimal. It uses $2(n - 1)$ packets and is therefore slow and wasteful of network bandwidth. Another problem with this implementation is that it does not provide any ordering between messages from different senders. If two processes, *A* and *B*, simultaneously perform $n - 1$ RPCs, some of the destinations may first get the RPC from *A* and then the RPC from *B*, whereas other destinations may get them in the reverse order. This interleaving of RPCs complicates the implementation of many applications.

This thesis proposed a different way of doing group communication. Our system can best be described using a three layer model (see Fig. 6.1). The bottom layer provides unreliable group communication. The middle layer turns the unreliable group communication into reliable totally-ordered group communication. The top layer uses group communication to implement applications. We will summarize the goals for each layer and how they were achieved.

6.1. Bottom Layer: FLIP

The goal in the bottom layer was to make hardware broadcast and hardware multicast available to higher layers. To achieve this in a clean and network-independent way, a routing protocol is needed. Instead of extending existing protocols, we decided to design a new protocol. The reason for doing so was that existing routing protocols

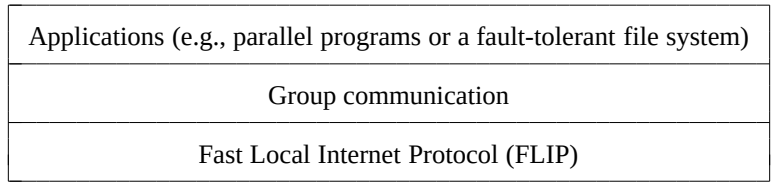


Fig. 6.1. Structure of the system.

were not designed for distributed systems and do not fulfill all the demands that a distributed system imposes on a routing protocol.

The new protocol, called FLIP, was designed, implemented, tested, and measured. FLIP meets the following distributed system requirements:

- Transparency

  Unlike most other routing protocols, FLIP addresses are location-independent. They do not identify a host, but an entity or a group of entities. Location-independent addresses are implemented by letting each host perform dynamic routing. A FLIP routing table is a cache of hints that map FLIP addresses to data-link addresses. FLIP updates the routing table each time a FLIP message arrives and uses special messages (LOCATE, HEREIS, and NOTHERE) to locate addresses that do not appear in the routing table or for which the routing table contains out-of-date information.

- Efficient at-most-once RPC

  FLIP simplifies an efficient implementation of at-most-once RPC. One of the problems in achieving at-most-once semantics is deciding whether an incoming request has been executed or not. With FLIP, this problem is easily solved. Each time a server is started, the server chooses a new FLIP address. Thus, all the requests sent to the previous incarnation of the server will fail automatically. FLIP also allows for an efficient implementation of RPCs by being a datagram protocol and by employing a blast protocol for large messages. This ensures that both small and large RPCs can be performed efficiently.

- Group communication

  FLIP implements unreliable group communication by letting FLIP addresses identify entities and by using a special data message (MULTIDATA). Because addresses identify entities, they can be used to identify a group of processes. By sending data in a MULTIDATA message, the FLIP router knows that it should forward the message to all locations for which it has an entry in its routing table. Furthermore, the router can ask the network drivers to use a single multicast address for all locations that are identified by the FLIP address.

- Security

  FLIP itself does not provide security, but it provides support for building efficient secure distributed systems. A system administrator can designate a network as either ‘‘trusted’’ or ‘‘untrusted.’’ A network in a machine room, for example, can be marked as trusted, while a public telephone line should be marked as untrusted. FLIP’s routing protocol will ensure that messages that are requested to be sent securely traverse only trusted networks. In case there is no secure route the packet has to be encrypted. By making a distinction between trusted and untrusted networks, FLIP can send messages over trusted networks without performing computation-intensive encryption.

  In addition to trusted and untrusted networks, FLIP supports public and private addresses to make the forgery of source addresses difficult. Processes send messages to a process’s public address and receive messages on their secret private address. This makes it hard, for example, for a process to impersonate the file server, because it needs to know the file server’s secret private address. If no authentication is used, the only way for a malicious user to impersonate the file server is to modify an Amoeba kernel that allows processes to receive messages on a public address.

- Network Management

  Current networks are complicated. They are built out of many different networks connecting many hosts of different architectures. FLIP requires almost no manual help from a system administrator to manage such a complex environment. New hosts and networks may be added, existing hosts may be moved, and networks may be reconfigured without informing FLIP about these events. FLIP will automatically detect these changes and will re-route messages, if needed. FLIP can do this, because addresses are location-independent and routing tables are dynamically updated.

- WAN

  A weak point of FLIP is WAN communication. FLIP was especially designed for local-area internetworks and not for an internetwork consisting of thousands of networks spread all over the world. FLIP does not provide any special support for WAN communication. On the contrary, FLIP employs mechanisms that are likely not to scale to large internetworks. Some early experience, however, suggests that FLIP can be used for small scale WAN internetworks.

6.2. Middle Layer: Group Communication

The middle layer turns the unreliable group communication of the FLIP layer into reliable and totally-ordered group communication, which can used by programmers to build applications. The main primitives it provides are: *CreateGroup, JoinGroup, LeaveGroup, SendToGroup, ReceiveFromGroup,* and *ResetGroup*. These primitives provide an abstraction that enables programmers to design applications consisting of one or more processes running on different machines. Furthermore, the primitives guarantee that all members of a single group see all events in the same total order. The events of a new member joining the group, a member leaving the group, a member sending a message to the group, and recovery from a failed member are all totally ordered. This property simplifies distributed programming.

The group communication primitives also allow users to trade performance against fault tolerance. A user can ask the system to guarantee that messages are delivered even in the presence of processor failures. If a user specifies, for example, a resilience degree of one, the system will guarantee that all messages will be delivered in the same order even if one of the group members fails. This property simplifies building fault-tolerant distributed programs.

The protocol for implementing reliable totally-ordered group communication is based on a negative acknowledgement scheme and a sequencer. In a positive acknowledgement scheme, a member sends an acknowledgement to the sender as soon as it receives the sender's message. For group communication this scheme results in bursty traffic and potentially in the loss of acknowledgements. Consider a group of 64 members of which 63 are sending an acknowledgement roughly at the same time. They need to compete for network access. Furthermore, the acknowledgements will arrive at the sender almost simultaneous. If the sender's network interface does not have enough buffer space, it is likely that some of the acknowledgements will be dropped, defeating the whole purpose of sending it and resulting in unnecessary retransmissions of the original message. Furthermore, current networks are very reliable. Thus, sending $n - 1$ acknowledgements for each message is overkill. For these reasons the group communication protocol is based on a negative acknowledgement scheme.

Total ordering is enforced by having one sequencer per group. The sequencer is not fundamentally different from the other members in the group. It just has a flag set telling it to process messages differently. Any member is capable of becoming a sequencer. If the sequencer fails, the remaining members elect a new one. The main advantage of having a sequencer is that the protocols for totally-ordered broadcast become simple and have a good performance. Sending a message to a group of 30 processes takes only 2.8 msec. The obvious disadvantage is that the sequencer is a potential performance bottleneck in a large system. The number of messages per second per group is limited by the number of messages that the sequencer can process per second. This has been measured at 815 messages per second. We believe that this

number is high enough to support small to medium-large groups. Furthermore, comparable decentralized protocols have not achieved better performance.

6.3. Top Layer: Applications

The goal in the top layer was to demonstrate that group communication simplifies building efficient distributed applications. We have looked at two application areas: parallel applications and fault-tolerant applications.

Parallel applications

The model for parallel computation that we have used is the shared data-object model. This model supports parallelism explicitly through a *fork* statement, but communication is completely transparent to the programmer. Orca processes running on different processors can share variables of some abstract data type, even if there is no physically shared memory present. By giving the programmer the illusion of shared memory the difficult task of writing a parallel program is considerable easier. They key issue in the shared data-object model is to implement the shared objects efficiently in a distributed environment by avoiding sending messages, as the transmission and processing of messages is factors more expensive than accessing local RAM.

We have shown that the RTS for the shared data-object model can be implemented easily and efficiently using group communication. We discussed two implementations: an unoptimized RTS and an optimized RTS. The unoptimized RTS replicates shared objects in each processor's RAM. Read operations on shared objects can be performed directly from local RAM. Write operations are broadcast to all processors involved in the parallel computation. As these broadcasts are reliable and totally-ordered, the copies of the shared data-objects stay consistent. This RTS optimizes read operations, because for many parallel applications the read/write ratio is high.

Although replicating shared objects is a good strategy, there are certain cases where it is better to avoid replication. The optimized RTS uses compile-time information to classify shared objects in two categories: objects that should be replicated (read/write ratio ≥ 1) and objects that should not be replicated (read/write ratio < 1). Furthermore, the optimized RTS tries to store nonreplicated shared objects on the processor that performs most operations. The Orca processes running on that processor can perform all operations without communicating with any other processor. Processors that do not have a copy access data remotely via RPC.

Using three example applications, each having different communication patterns, we have demonstrated that both the unoptimized and optimized RTS achieve good performance. The unoptimized RTS achieved good performance for applications that use shared objects with a high read/write ratio or use shared objects as a broadcast channel. The optimized RTS also achieves good performance for applications that use shared objects as a point-to-point communication channel.

Fault-Tolerant Parallel Applications

In parallel applications, fault tolerance is often ignored. The main goal in a parallel application is to achieve better performance by using multiple processors and to avoid wasting computing cycles on anything else. However, by using multiple processors the chance that the computation aborts due to a processor failure increases. Furthermore, parallel applications often run for a long time. Thus, restarting a parallel application from scratch after a processor failure is not an attractive solution.

We have shown that the unoptimized RTS for the shared data-object model can be made fault-tolerant in an easy and inexpensive way. The solution is to periodically make a global consistent checkpoint and to roll back to the latest consistent checkpoint after a processor failure. This solution works for noninteractive applications that read input, compute, and produce a result.

As the unoptimized RTS only uses reliable totally-ordered broadcast for communication, it is easy to make a consistent checkpoint without freezing the complete application. Periodically, a thread broadcasts a *make checkpoint* message. All processes will receive this message in the same total order and can therefore make a checkpoint immediately on receiving the message, and continue computing immediately after making the checkpoint. Thus, the overhead for fault tolerance is equal to the cost of making a global consistent checkpoint.

Another attractive property of the solution is that the overhead for fault tolerance is completely controlled by the user. The user specifies the period between two checkpoints. By specifying a long period, the overhead for fault tolerance decreases, but the cost for recovery increases. If a user, for example, specifies a period of infinity, the solution is the same as starting from scratch after each failure. The cost for fault tolerance is then zero, but the cost for recovery is equal to the time needed to recompute everything that was computed before the failure occurred.

Fault-Tolerant Distributed Service

To demonstrate that group communication is also useful for ‘‘normal’’ fault-tolerant applications, we designed, implemented, and measured a fault-tolerant directory service. The directory service exemplifies services that achieve fault tolerance by active replication of data. It uses group communication with a high resilience degree to keep the replicated directories consistent.

The main cost in providing fault tolerance for this service is the overhead of performing disk operations. To avoid the cost for disk operations in the critical path, the directory service was modified to use NonVolatile RAM (NVRAM). This implementation can perform 90 write operations per second, a factor of 9 more than without NVRAM. The maximum number of read operations is the same for both implementations: 652 operations per second.

The directory service using group communication was based on an implementation using RPC. We compared the group version with the RPC version. The group ser-

vice achieves better performance and its design is simpler. The comparison, however, is not completely fair, as the RPC implementation is duplicated, whereas the group implementation is triplicated. The RPC service can perform 10 write operations per second (without using NVRAM) and 531 read operations per second.

6.4. Conclusion

The goal in this thesis was to demonstrate that group communication is suitable for distributed systems, as it simplifies distributed programming and give enormous performance benefits if a network supports broadcast or multicast. In attempting to achieve this goal, we made the following research contributions:

- A clean, simple, and integrated three-layered model for group communication.
- A new network protocol, FLIP, that meets the communication requirements of a distributed system.
- An efficient protocol for reliable and totally-ordered group communication.
- A new model and implementation for DSM based on compiler optimizations, group communication, and shared data-objects.
- An efficient scheme to make a broad class of parallel applications fault-tolerant in a transparent way.
- A fault-tolerant directory service based on group communication, demonstrating that fault-tolerant services based on group communication are easier to implement and more efficient than the same service implemented using RPC.

By showing that reliable totally-ordered group communication can be done as efficiently as an RPC and by using it in a number of parallel and fault-tolerant applications, we demonstrated the utility and desirability of group communication in distributed systems.

Appendix A
The FLIP Protocol

The FLIP protocol makes it possible that routing tables automatically adapt to changes in the network topology. The protocol is based on 6 message types. We will discuss in detail the actions undertaken by the packet switch when receiving a FLIP fragment.

LOCATE

```
/* Remember that source can be reached through network ntw on location loc. */
UpdateRoutingTable(pkt→source, ntw, loc, pkt→act_hop, pkt→flags & UNSAFE);
if (pkt→act_hop == pkt→max_hop and
        lookup(pkt→destination, &dsthop, ntw, pkt→flags & SECURITY)) {
   /* Destination is known; send HEREIS message back. */
   pkt→type = HEREIS;
   pkt→max_hop = pkt→act_hop + dsthop;
   pkt→act_hop −= Networkweight[ntw];
   pkt_send(pkt, ntw, loc);
} else { /* destination is unknown or incorrect */
   /* Forget all routes to destination, except those on ntw */
   RemoveFromRoutingTable(pkt→destination, ALLNTW, ALLLOC, ntw);
   /* Forward pkt on all other networks, if the hop count and security allow it */
   pkt_broadcast(pkt, ntw, pkt→max_hop − pkt→act_hop, pkt→flags & SECURITY);
}
```

Fig. 7.1. The protocol for LOCATE messages. A LOCATE message is broadcast, while the route to the source address is remembered.

If a fragment arrives over an untrusted network, the *Unsafe* bit in the *Flags* field is set. This also happens when a fragment is sent over an untrusted network. Furthermore, FLIP refuses to send a fragment with the *Security* bit set over an untrusted network. We have omitted these details from the protocol description below to make it easier to understand.

The LOCATE message is to find the network location of a NSAP (see Fig. 7.1). It

is broadcast to all FLIP boxes. If a FLIP box receives a LOCATE message, it stores the tuple (Source Address, Network, Location, Actual Hop Count, Flags & UNSAFE) in its routing table, so that a reply to the LOCATE message can find its way back. If the *Actual Hop Count* in the LOCATE message is equal to the *Maximum Hop Count*, the *Destination Address* is in the routing table, the destination network is safe (if necessary), and the destination network is not equal to the source network, the LOCATE message is turned into an HEREIS message and sent back to the *Source Address*. The *Maximum Hop Count* of the HEREIS message is set to the *Actual Hop Count* of the LOCATE message plus the hop count in the routing table. If the *Actual Hop Count* in the LOCATE message is less than the *Maximum Hop Count*, the entries for *Destination Address* in the routing table are removed, except for the entries that route the address to the network on which the LOCATE arrived, and the message is broadcast on the other networks.

HEREIS

```
hops = pkt→max_hop − pkt→act_hop;
AddToRoutingTable(pkt→destination, ntw, loc, hops, pkt→flags & UNSAFE);
if (route(pkt→source, &dstntw, &dstloc, ntw, pkt→act_hop, pkt→flags & SECURITY)) {
   /* A network is found that is different from the network on which
    * pkt arrived and on which the destination is reachable. */
   pkt→act_hop −= Networkweight[dstntw];
   pkt_send(pkt, dstntw, dstloc);
} else discard(pkt);  /* Source is unknown, too far away, or unsafe. */
```

Fig. 7.2. The protocol for HEREIS messages. HEREIS messages are returned to the source in response to LOCATE messages, while the route to the destination (of the LOCATE message) is remembered.

It is important that the packet switch only sends an HEREIS message back if the *Actual Hop Count* of the LOCATE message is *equal* to the *Maximum Hop Count*. By using a large *Maximum Hop Count* the sender of the LOCATE message can force an interface module to respond instead of a packet switch and at the same time invalidate any old routing information for the address that is located. If the address to be located is registered at an interface on a distance smaller than *Maximum Hop Count*, this scheme works correctly, because an interface always sends an HEREIS back for an address that is registered with it.

An HEREIS message is sent as a reply to a LOCATE message (see Fig. 7.2). If an HEREIS message arrives, the tuple (Destination Address, Network, Location, Actual Hop Count, Flags & UNSAFE) is added to the routing table. If the *Source Address* is in the routing table and the network on which the source can be reached is not equal to the network on which the message arrived and the incremented *Actual Hop Count* does not exceed the *Maximum Hop Count*, the message is forwarded. Otherwise, the message is discarded. If the destination network is equal to the source network, *route()* will return false; the message is discarded.

UNIDATA

```
UpdateRoutingTable(pkt→source, ntw, loc, pkt→act_hop, pkt→flags & UNSAFE);
hops = pkt→max_hop − pkt→act_hop;
switch (route(pkt→destination, &dstntw, &dstloc, ntw, hops, pkt→flags & SECURITY)) {
   case OK:    /* forward message */
      pkt→act_hop += Networkweight[dstntw];
      pkt_send(pkt, dstntw, dstloc);
      break;
   case TooFarAway:    /* send pkt back to source. */
      pkt→type = NOTHERE;
      pkt→acthop −= Networkweight[ntw];
      pkt_send(pkt, ntw, loc);
      break;
   case Unsafe:    /* send pkt back to source. */
      pkt→type = UNTRUSTED;
      pkt→flags |= UNREACHABLE;
      pkt→acthop −= Networkweight[ntw];
      pkt_send(pkt, ntw, loc);
      break;
   }
}
```

Fig. 7.3. The protocol for UNIDATA messages. If the destination is known and, if necessary, safe, the message is forwarded. If the destination is unknown, the message is returned as a NOTHERE message. If the message can only be transferred over trusted networks, and the destination network is untrusted, the message is returned as an UNTRUSTED message.

UNIDATA messages are used to transfer fragments of a message between two NSAPs. When such a message arrives, the tuple (Source Address, Network, Location, Actual Hop Count, Flags & UNSAFE) is stored in the routing table (see Fig. 7.3). If the Destination Address is in the routing table, the destination network is not equal to the source network, the incremented *Actual Hop Count* does not exceed the *Maximum Hop Count*, and the destination network is safe, the message is fragmented (if needed), and each fragment is forwarded. If there are multiple choices in the routing table, one is chosen, based on an implementation-defined heuristic, such as the safety or the minimum number of hops. The null destination address maps to all networks and locations.

If the *Destination Address* of a UNIDATA message is not in the routing table, or the destination network is unsafe, the message is transformed into a NOTHERE message by setting the *Type* to NOTHERE, and is returned to the *Source Address*. The data in the message is *not* discarded, unless the decremented *Actual Hop Count* was zero.

If the *Maximum Hop Count* minus the *Actual Hop Count* of a UNIDATA message is less than the Hop Count stored in the routing table, the implementor can decide to send a NOTHERE message back to the sender. Chances are that the message would not have reached its destination. A new locate of the *Destination Address* will re-establish

NOTHERE

```
RemoveFromRoutingTable(pkt→destination, ntw, loc, NONTW);
hops = pkt→max_hop − pkt→act_hop;
if (route(pkt→destination, &dstntw, &dstloc, ntw, hops, pkt→flags & SECURITY) {
   /* There is another route to destination; use it. */
   pkt→type = UNIDATA;
   pkt→act_hop += Networkweight[dstntw];
   pkt_send(pkt, dstntw, dstloc);
} else if (route(pkt→source, &dstntw, &dstloc, ntw, hops, pkt→flags & SECURITY)) {
   /* Forward to original source. */
   pkt→act_hop −= Networkweight[dstntw];
   pkt_send(pkt, dstntw, dstloc);
} else discard(pkt);   /* Source is unknown, too far away, or untrusted. */
```

Fig. 7.4. The protocol for NOTHERE messages. If there is an alternative route, try that one. Otherwise forward back to the original source.

the route, and update the routing tables. If the destination network is untrusted, then the *Unreachable* bit is set, and the message is returned as an UNTRUSTED message.

If a NOTHERE message arrives at a FLIP box, the corresponding entry in the routing table is invalidated (see Fig.7.4). If another route is present in the routing table, the *Type* field is set back to UNIDATA. Now operation continues as if a UNIDATA message arrived, except that the routing table operation is skipped. This way an alternate route, if available, will be tried automatically. If not, the NOTHERE is forwarded to its source (if still safe).

UNTRUSTED

```
UpdateRoutingTable(pkt→destination, ntw, loc, pkt→act_hop, UNSAFE);
hops = pkt→max_hop − pkt→act_hop;
if (route(pkt→destination, &dstntw, &dstloc, ntw, hops, SECURE) {
   /* There is another safe route to destination; use it. */
   pkt→type = UNIDATA;
   pkt→act_hop += Networkweight[dstntw];
   pkt_send(pkt, dstntw, dstloc);
} else if (route(pkt→source, &dstntw, &dstloc, ntw, hops, SECURE)) {
   /* Return to source (if still safe). */
   pkt→act_hop −= Networkweight[dstntw];
   pkt_send(pkt, dstntw, dstloc);
} else discard(pkt);   /* Source is unknown, too far away, or untrusted. */
```

Fig. 7.5. The protocol for an UNTRUSTED message. If there is an alternative safe route, try that one. Otherwise return to its original source.

If an UNTRUSTED message arrives at a FLIP box (see Fig. 7.5), the route in the routing table is updated, and a new safe route, if present, is tried. If there is no such route, the message is forwarded back to the original source (but only if there exists a route back that is safe).

MULTIDATA

```
UpdateRoutingTable(pkt→source, ntw, loc, pkt→act_hop, pkt→flags & UNSAFE);
/* See if there are any known (and safe) destinations. */
if (list = lookup(dstaddr, &dsthop, ntw, pkt→flags & SECURITY)) {
   /* Send message to all locations on list, if the hop count allows it. */
   pkt_multicast(list, pkt→max_hop – pkt→act_hop);
} else discard(pkt);
```

Fig. 7.6. The protocol for MULTIDATA messages. Forward to all known destinations.

A MULTIDATA message is transferred like an UNIDATA message (see Fig. 7.6). However, if there are multiple entries of the *Destination Address* in the routing table, the message is forwarded to all destinations instead of just one. If there is no entry for the *Destination Address*, or the destination network is unsafe, the message is discarded and not returned as a NOTHERE message. FLIP does not assume that a network has support for multicast. If a network has such a capability, FLIP will try to take advantage of it. If not, the message is sent point-to-point to all destinations on the network.

Appendix B
Analysis of the Cost of a Checkpoint

In this section we will deduce the formula to compute the overhead of checkpointing and the optimal computing interval given the time to make a checkpoint, the time to recover from a crash, and the MTTF of the system.

For the derivation we introduce the following variables:

T_{tot} is the time needed to run the application without checkpointing;
T_{comp} is the time between two checkpoints;
T_{cp} is the mean time to make a global consistent checkpoint;
T_{rb} is the mean time to roll back;
T_{MTTF} is the mean time to failure.

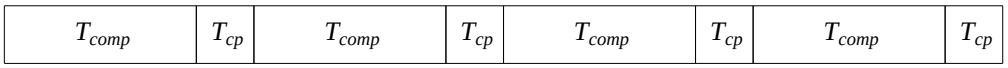


Fig. 8.1. An Orca execution without failures.

An execution of an Orca application without any failures consists of a sequence of alternating computing intervals of length T_{comp} and checkpoint intervals of length T_{cp} (see Fig. 8.1). If a failure happens during a computing interval or a checkpoint interval, an extra interval with duration equal to the recover time plus the time wasted before the crash is inserted (see Fig. 8.2). The mean time wasted can be estimated by $(T_{comp} + T_{cp}) / 2$.

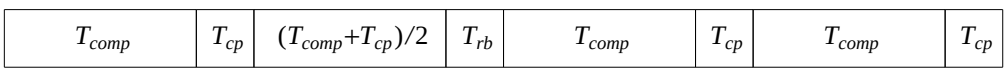


Fig. 8.2. An Orca execution with a failure.

Let T_{tot} be the total computation time needed to finish the application. Then, the number of computing intervals is equal to T_{tot} / T_{comp} and the number of failures during the total run-time is approximately T_{tot} / T_{MTTF} (assuming $T_{cp} << T_{tot}$ and $T_{rb} << T_{tot}$). Thus the average run-time is:

$$\frac{T_{tot}}{T_{comp}} (T_{comp} + T_{cp}) + \frac{T_{tot}}{T_{MTTF}} (\frac{T_{comp} + T_{cp}}{2} + T_{rb}).$$

The overhead for checkpointing is the average run-time divided by T_{tot} minus 1:

$$overhead = \frac{T_{cp}}{T_{comp}} + \frac{\frac{T_{comp} + T_{cp}}{2} + T_{rb}}{T_{MTTF}}.$$

Minimizing the overhead function gives:

$$optimal\ computing\ interval = \sqrt{2T_{cp}T_{MTTF}}$$

References

Adve, S. V. and Hill, M. D., "Weak Ordering - A new Definition," *Proc. Seventeenth Annual International Symposium on Computer Architecture*, pp. 2-14, Seattle, WA, May 1990. Cited on page 96.

Ahamad, M. and Bernstein, A. J., "An Application of Name Based Addressing to Low Level Distributed Algorithms," *IEEE Trans. on Soft. Eng.*, Vol. 11, No. 1, pp. 59-67, Jan. 1985. Cited on page 4.

Ahamad, M., Hutto, P. W., and John, R., "Implementing and Programming Causal Distributed Shared Memory," *Proc. Eleventh International Conference on Distributed Computing Systems*, pp. 274-281, Arlington, TX, May 1991. Cited on page 96.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D., "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, MA, 1974. Cited on page 7.

Ahuja, S., Carriero, N. J., and Gelernter, D. H., "Linda and Friends," *IEEE Computer*, Vol. 19, No. 8, pp. 26-34, Aug. 1986. Cited on pages 96, 124.

Ahuja, S., Carriero, N. J., Gelernter, D. H., and Krishnaswamy, V., "Matching Language and Hardware for Parallel Computation in the Linda Machine," *IEEE Trans. on Computers*, Vol. 37, No. 8, pp. 921-929, Aug. 1988. Cited on page 122.

Almasi, G. S. and Gottlieb, A., "Highly Parallel Computing," Benjamin/Cummings, Redwood City, CA, 1989. Cited on page 88.

Alsberg, P. and Day, J., "A Principle for Resilient Sharing of Distributed Resources," *Proc. Second International Conference on Software Engineering*, pp. 627-644, Oct. 1976. Cited on page 159.

Andrews, G. R., "Paradigms for Process Interaction in Distributed Programs," *ACM Computing Surveys*, Vol. 23, No. 1, pp. 49-90, Mar. 1991. Cited on page 109.

Andrews, G. R., Olsson, R. A., Coffin, M., Elshoff, I., Nilsen, K., Purdin, T., and Townsend, G., "An Overview of the SR Language and Implementation," *ACM Trans. Prog. Lang. Syst.*, Vol. 10, No. 1, pp. 51-86, Jan. 1988. Cited on page 122.

Athas, W. C. and Seitz , C. L., "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer*, Vol. 21, No. 8, pp. 9-24, Aug. 1988. Cited on page 1.

Backes, F., "Transparent Bridges for Interconnection of IEEE 802 LANs," *IEEE Network*, Vol. 2, No. 1, pp. 5-9, Jan. 1988. Cited on page 39.

Bakken, D. E. and Schlichting, R. D., "Tolerating Failures in the Bag-of-Tasks Programming Paradigm," *Proc. 21st International Symposium on Fault-Tolerant Computing*, pp. 248-255, Montreal, Canada, June 1991. Cited on page 138.

Bal, H. E., "Programming Distributed Systems," Silicon Press, Summit, NJ, 1990. Cited on pages 7, 88, 100, 109, 127.

Bal, H. E., "Fault-Tolerant Parallel Programming in Argus," *Concurrency−Practice and Experience*, Vol. 4, No. 1, pp. 37-55, Feb. 1992. Cited on pages 137, 138.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "A Distributed Implementation of the Shared Data-Object Model," *Proc. First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1-19, Ft. Lauderdale, FL, Oct. 1989a. Cited on page 127.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "Experience with Distributed Programming in Orca," *Proc. 1990 International Conference on Computer Languages*, pp. 79-89, New Orleans, LA, Mar. 1990. Cited on pages 109, 111, 117, 127.

Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S., "Orca: A language for Parallel Programming of Distributed Systems," *IEEE Trans. on Soft. Eng.*, Vol. 18, No. 3, pp. 190-205, Mar. 1992a. Cited on pages 7, 88, 127.

Bal, H. E., Kaashoek, M. F., Tanenbaum, A. S., and Jansen, J., "Replication Techniques for Speeding up Parallel Applications on Distributed Systems," *Concurrency−Practice and Experience*, Vol. 4, No. 5, pp. 337-355, Aug. 1992b. Cited on pages 101, 102, 109.

Bal, H. E., Steiner, J. G., and Tanenbaum, A. S., "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, Vol. 21, No. 3, pp. 261-322, Sep. 1989b. Cited on pages 3, 15.

Bal, H. E. and Tanenbaum, A. S., "Distributed Programming with Shared Data," *Computer Languages*, Vol. 16, No. 2, pp. 129-146, 1991. Cited on pages 127, 140.

Barrett, P. A., Hilborne, A. M., Verissimo, P., Rodrigues, L., Bond, P. G., Seaton, D. T., and Speirs, N. A., "The Delta-4 Extra Performance Architecture (XPA)," *Proc. 20th International Symposium on Fault-Tolerant Computing*, pp. 481-488, Newcastle, UK, June 1990. Cited on page 143.

Bennett, J. K., Carter, J. B., and Zwaenepoel, W., "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. Second Symposium on Principles and Practice of Parallel Programming*, pp. 168-176, Seattle, WA, Mar. 1990. Cited on pages 96, 126.

Bernstein, P. A., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," Addison-Wesley, Reading, MA, 1987. Cited on page 142.

Bershad, B. N. and Zekauskas, M. J., "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors," CMU-CS-91-170, Carnegie Mellon University, Pittsburgh, PA, Sep. 1991. Cited on page 96.

Birman, K. P., Cooper, R., Jospeh, T. A., Kane, K. P., Schmuck, F., and Wood, M., "Isis - A Distributed Programming Environment," Cornell University, Ithaca, NY, June 1990. Cited on pages 46, 47, 59, 140.

Birman, K. P. and Joseph, T. A., "Exploiting Virtual Synchrony in Distributed Systems," *Proc. Eleventh Symposium on Operating Systems Principles*, pp. 123-138, Austin, TX, Nov. 87. Cited on pages 9, 138, 140.

Birman, K. P. and Joseph, T. A., "Reliable Communication in the Presence of Failures," *ACM Trans. Comp. Syst.*, Vol. 5, No. 1, pp. 47-76, Feb. 1987. Cited on pages 47, 79.

Birman, K. P., Schiper, A., and Stephenson, P., "Lightweight Causal and Atomic Group Multicast," *ACM Trans. Comp. Syst.*, Vol. 9, No. 3, pp. 272-314, Aug. 1991. Cited on pages 46, 79.

Birrell, A. D., "Secure Communication Using Remote Procedure Calls," *ACM Trans. Comp. Syst.*, Vol. 3, No. 1, pp. 1-14, Feb. 1985. Cited on page 34.

Birrell, A. D. and Nelson, B. J., "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, Vol. 2, No. 1, pp. 39-59, Feb. 1984. Cited on pages 4, 18, 33.

Bisiani, R. and Forin, A., "Architectural Support for Multilanguage Parallel Programming on Heterogeneous Systems," *Proc. Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 21-30 , Palo Alto, CA, Oct. 1987. Cited on page 96.

Black, A., Hutchinson, N., Jul, E., Levy, H. M., and Carter, L., "Distribution and Abstract Types in Emerald," *IEEE Trans. on Soft. Eng.*, Vol. 13, No. 1, pp. 65-76, Jan. 1987. Cited on page 96.

Bloch, J. J., Daniels, D. S., and Spector, A. Z., "A Weighted Voting Algorithm for Replicated Directories," *Journal of the ACM*, Vol. 34, No. 4, pp. 859-909, Oct. 1987. Cited on page 160.

Bolosky, W. J., Fitzgerald, R. P., and Scott, M. L., "Simple But Effective Techniques for NUMA Memory Management," *Proc. Twelfth Symposium on Operating Systems Principles*, pp. 19-31, Litchfield Park, AZ, Dec. 1989. Cited on page 95.

Burkhardt, H., Frank, S., Knobe, B., and Rothnie, J., "Overview of the KSR1 Computer System," KSR-TR-9202001, Kendall Square Research, Boston, MA, Feb. 1992. Cited on page 95.

Butler, R., Lusk, E., McCune, W., and Overbeek, R., "Parallel Logic Programming for Numeric Applications," *Proc. Third International Conference on Logic Programming*, pp. 375-388, London, July 1986. Cited on page 117.

Callahan, C. D., Cooper, K. D., Hood, R. T., Kennedy, K., and Torczon, L., "ParaScope: a Parallel Programming Environment," *International Journal of Supercomputer Applications*, Vol. 2, No. 4, pp. 84-99, Winter 1988. Cited on page 123.

Carriero, N. J., "Implementation of Tuple Space Machines," RR-567, Dept. of Computer Science, Yale University, New Haven, CT, Dec. 1987. Cited on page 124.

Carriero, N. J. and Gelernter, D. H., "The S/Net's Linda Kernel," *ACM Trans. Comp. Syst.*, Vol. 4, No. 2, pp. 110-129, May 1986. Cited on page 122.

Carriero, N. J., Gelernter, D. H., and Leichter, J., "Distributed Data Structures in Linda," *Proc. Thirteenth Symposium on Principles of Programming Languages*, pp. 236-242, St. Petersburg, FL, Jan. 1986. Cited on page 138.

Carter, J. B., Bennett, J. K., and Zwaenepoel, W., "Implementation and Performance of Munin," *Proc. Thirteenth Symposium on Operating System Principles*, pp. 137-151, Pacific Grove, CA, Oct. 1991. Cited on pages 96, 117, 125.

Censier, L. M. and Feautrier, P., "A New Solution to Cache Coherence Problems in Multicache Systems," *IEEE Trans. on Computers*, pp. 1112-1118, Dec. 1978. Cited on page 88.

Chaiken, D., Kubiatowicz, J., and Agrawal, A., "LimitLess Directories: a Scalable Cache Coherence Scheme," *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 224-234, Santa Clara, CA, Apr. 1991. Cited on page 95.

Chandy, K. M. and Lamport, L., "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Comp. Syst.*, Vol. 3, No. 1, pp. 63-75, Feb. 1985. Cited on pages 130, 133, 140.

Chang, J., "Simplifying Distributed Database Design by Using a Broadcast Network," *Proc. ACM SIGMOD 1984 Annual Conference*, pp. 223-233, Boston, MA, June 1984. Cited on page 4.

Chang, J. and Maxemchuk, N. F., "Reliable Broadcast Protocols," *ACM Trans. Comp. Syst.*, Vol. 2, No. 3, pp. 251-273, Aug. 1984. Cited on pages 49, 80.

Chanson, S. T., Neufeld, G. W., and Liang, L., "A Bibliography on Multicast and Group Communication," *Operating Systems Review*, Vol. 23, No. 4, pp. 20-25, Oct. 1989. Cited on page 79.

Chase, J. S., Amador, F. G., Lazowska, E. D., Levy, H. M., and Littlefield, R. J., "The Amber System: Parallel Programming on a Network of Multiprocessors," *Proc. Twelfth Symposium on Operating Systems Principles*, pp. 147-158, Litchfield Park, AZ, Dec. 1989. Cited on pages 96, 117, 123.

Chen, M., Choo, Y., and Li, J., "Compiling Parallel Programs by Optimizing Performance," *Journal of Supercomputing*, Vol. 1, No. 2, pp. 171-207, July 1988. Cited on page 123.

Cheriton, D. R., "Preliminary Thoughts on Problem-Oriented Shared Memory," *Operating Systems Review*, Vol. 19, No. 4, pp. 26-33, Oct. 1985. Cited on page 96.

Cheriton, D. R., "VMTP: a Transport Protocol for the Next Generation of Communication Systems," *Proc. SIGCOMM 86*, pp. 406-415, Stowe, VT, Aug. 1986. Cited on pages 7, 41.

Cheriton, D. R., "VMTP: Versatile Message Transaction Protocol," RFC 1045, SRI Network Information Center, Feb. 1988a. Cited on page 41.

Cheriton, D. R., "The V Distributed System," *Commun. ACM*, Vol. 31, No. 3, pp. 314-333, Mar. 1988b. Cited on pages 9, 18, 42, 139.

Cheriton, D. R., Goosen, H. A., and Boyle, P. D., "Paradigm: a Highly Scalable Shared Memory Multicomputer," *IEEE Computer*, Vol. 24, No. 2, pp. 33-49, Feb. 1991. Cited on page 95.

Cheriton, D. R. and Mann, T. P., "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance," *ACM Trans. Comp. Syst.*, Vol. 7, No. 2, pp. 147-183, May 1989. Cited on page 142.

Cheriton, D. R. and Zwaenepoel, W., "Distributed Process Groups in the V kernel," *ACM Trans. Comp. Syst.*, Vol. 3, No. 2, pp. 77-107, May 1985. Cited on pages 4, 9, 45−47, 82, 122.

Cohen, D., "On Holy Wars and a Plea for Peace," *IEEE Computer*, Vol. 14, No. 10, pp. 48-54, Oct. 1981. Cited on page 25.

Comer, D. E., "Internetworking with TCP/IP 2nd ed.," Prentice-Hall, Englewood Cliffs, NJ, 1992. Cited on pages 6, 41.

Cooper, E. C., "Replicated Distributed Programs," *Proc. Tenth Symposium on Operating Systems Principles*, pp. 63-78, Orcas Islands, WA, Dec. 1985. Cited on page 84.

Cox, A. L. and Fowler, R. J., "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," *Proc. Twelfth Symposium on Operating Systems Principles*, pp. 32-44, Litchfield Park, AZ, Dec. 1989. Cited on page 95.

Cristian, F., "Understanding Fault-Tolerant Distributed Systems," *Commun. ACM*, Vol. 34, No. 2, pp. 56-78, Feb. 1991. Cited on pages 4, 143.

Curtis, R. and Wittie, L., "Global Naming in Distributed Systems," *IEEE Software*, Vol. 1, No. 4, pp. 76-80, July 1984. Cited on page 142.

Daniels, D. S. and Spector, A. Z., "An Algorithm for Replicated Directories," *Proc. Second Annual Symposium on Principles of Distributed Computing*, pp. 104-113, Montreal, Canada, Aug. 1983. Cited on page 160.

Daniels, D. S., Spector, A. Z., and Thompson, D. S., "Distributed Logging for Transaction Processing," *Proc. ACM SIGMOD 1987 Annual Conference*, pp. 82-96, San Francisco, CA, May 1987. Cited on page 156.

Danzig, P. B., "Finite Buffers and Fast Multicast," *Perf. Eval. Rev.*, Vol. 17, No. 1, pp. 79-88, May 1989. Cited on page 49.

Dasgupta, P., Leblanc, R. J., Ahamad, M., and Ramachandran, U., "The Clouds Distributed Operating System," *IEEE Computer*, Vol. 24, No. 11, pp. 34-44, Nov. 1991. Cited on page 18.

Davidson, S. B., Garcia-Molina, H., and Skeen, D., "Consistency in Partitioned Networks," *ACM Computing Surveys*, Vol. 17, No. 3, pp. 341-370, Sep. 1985. Cited on page 143.

Dechter, R. and Kleinrock, L., "Broadcast Communication and Distributed Algorithms," *IEEE Trans. on Computers*, Vol. 35, No. 3, pp. 210-219, Mar. 1986. Cited on page 4.

Deering, S. E., "Host Extensions for IP Multicasting," RFC 1112, SRI Network Information Center, Aug. 1988. Cited on page 6.

Deering, S. E. and Cheriton, D. R., "Multicast Routing in Datagram Internetworks and Extended LANs," *ACM Trans. Comp. Syst.*, Vol. 8, No. 2, pp. 85-110, May 1990. Cited on page 42.

Delp, G. S., Farber, D. J., Minnich, R. G., Smith, J. M., and Tam, M.-C., "Memory as a Network Abstraction," *IEEE Network*, Vol. 5, No. 4, pp. 34-41, July 1991. Cited on page 96.

Dijkstra, E. W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," *Commun. ACM*, Vol. 18, No. 8, pp. 453-457, Aug. 1975. Cited on page 98.

Dinning, A., "A Survey of Synchronization Methods for Parallel Computers," *IEEE Computer*, Vol. 22, No. 7, pp. 66-77, July 1989. Cited on page 94.

Douglis, F., Kaashoek, M. F., Tanenbaum, A. S., and Ousterhout, J. K., "A Comparison of Two Distributed Systems: Amoeba and Sprite," *Computing Systems*, Vol. 4, No. 4, pp. 353-384, 1991. Cited on pages 37, 137.

Dubois, M., Scheurich, C., and Briggs, F. A., "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, Vol. 21, No. 2, pp. 9-21, Feb. 1988. Cited on page 96.

Elnozahy, E. N., Johnson, D. B., and Zwaenepoel, W., "The Performance of Consistent Checkpointing," *Proc. Eleventh Symposium on Reliable Distributed Systems*, Houston, TX, Oct. 1992. Cited on pages 10, 139.

Elnozahy, E. N. and Zwaenepoel, W., "Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit," *IEEE Trans. on Computers*, Vol. 41, No. 5, pp. 526-531, May 1992a. Cited on pages 133, 138.

Elnozahy, E. N. and Zwaenepoel, W., "Replicated Distributed Processes in Manetho," *Proc. 22nd International Symposium on Fault-Tolerant Computing*, pp. 18-27, Boston, MA, July 1992b. Cited on page 82.

El Abbadi, A., Skeen, D., and Cristian, F., "An Efficient, Fault-Tolerant Algorithm for Replicated Data Management," *Proc. Fifth Symposium on Principles of Database Systems*, pp. 215-229, Portland, OR, Mar. 1985. Cited on page 148.

Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L., "The Notion of Consistency and Predicate Locks in a Database System," *Commun. ACM*, Vol. 19, No. 11, pp. 624-633, Nov. 1976. Cited on pages 84, 100, 102, 158.

Evans, A., Kantrowitz, W., and Weiss, E., "A User Authentication Scheme Not Requiring Secrecy in the Computer," *Commun. ACM*, Vol. 17, No. 8, pp. 437-442, Aug. 1974. Cited on page 14.

Finlayson, R., Mann, T., Mogul, J., and Theimer, M., "A Reverse Address Resolution Protocol," RFC 903, SRI Network Information Center, June 1984. Cited on page 6.

Fischer, M. J., Lynch, N. A., and Paterson, M. S., "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, Vol. 32, No. 2, pp. 374-382, Apr. 1985. Cited on page 53.

Fleisch, B. D. and Popek, G. J., "Mirage: a Coherent Distributed Shared Memory Design," *Proc. Twelfth Symposium on Operating Systems Principles*, pp. 211-223, Litchfield Park, AZ, Dec. 1989. Cited on page 95.

Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C-W., and Wu, M-Y., "FORTRAN-D Language Specification," TR90-141, Rice University, Dec. 1990. Cited on page 122.

Frank, A. J., Wittie, L. D., and Bernstein, A. J., "Multicast Communication on Network Computers," *IEEE Software*, Vol. 2, No. 3, pp. 49-61, May 1985. Cited on page 45.

Garcia-Molina, H., "Elections in a Distributed Computing System," *IEEE Trans. on Computers*, Vol. 31, No. 1, pp. 48-59, Jan. 1982. Cited on page 70.

Gehani, N. H., "Broadcasting Sequential Processes," *IEEE Trans. on Soft. Eng.*, Vol. 10, No. 4, pp. 343-351, July 1984. Cited on pages 4, 122.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. Seventeenth Annual International Symposium on Computer Architecture*, pp. 15-26, Seattle, WA, May 1990. Cited on page 96.

Gifford, D. K., "Weighted Voting for Replicated Data," *Proc. Seventh Symposium on Operating System Principles*, pp. 150-159, Pacific Grove, CA, Dec. 1987. Cited on page 160.

Gray, J. N., "Notes on Database Operating Systems," Vol. 60, pp. 393-481, in *Operating Systems: an Advanced Course, Lecture Notes in Computer Science*, Springer Verlag, New York, 1978. Cited on page 142.

Grimshaw, A. S., "The Mentat Run-Time System: Support for Medium Grain Parallel Computing," *Proc. Fifth Distributed Memory Computing Conference*, pp. 1064-1073, Charleston, SC, Apr., 1990. Cited on page 123.

Gueth, R., Kriz, J., and Zueger, S., "Broadcasting Source-Addressed Messages," *Proc. Fifth International Conference on Distributed Computing Systems*, pp. 108-115, Denver, CO, 1985. Cited on page 45.

Hariri, S., Choudhary, A., and Sarikaya, B., "Architectural Support for Designing Fault-Tolerant Open Distributed Systems," *IEEE Computer*, Vol. 25, No. 6, pp. 50-61, June 1992. Cited on pages 143, 156.

Hisgen, A., Birrell, A. D., Jerian, C., Mann, T., Schroeder, M., and Swart, C., "Granularity and Semantic Level of Replication in the Echo Distributed File System," *IEEE TCOS Newsletter*, Vol. 4, No. 3, pp. 30-32, 1990. Cited on page 159.

Hoare, C. A. R., "The Emperor's Old Clothes," *Commun. ACM*, Vol. 24, No. 2, pp. 75-83, Feb. 1981. Cited on page 100.

Hughes, L., "A Multicast Interface for UNIX 4.3," *Software−Practice and Experience*, Vol. 18, No. 1, pp. 15-27, Jan. 1988. Cited on page 45.

Hughes, L., "Multicast Response Handling Taxonomy," *Computer Communications*, Vol. 12, No. 1, pp. 39-46, Feb. 1989. Cited on page 46.

Hutto, P. W. and Ahamad, M., "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories," *Proc. Tenth International Conference on Distributed Computing Systems*, pp. 302-309, Paris, May 1990. Cited on page 125.

Ioannidis, J., Duchamp, D., and Maguire Jr., G. Q., "IP-based Protocols for Mobile Internetworking," *Proc. SIGCOMM 91 Conference on Communications Architectures and Protocols*, pp. 235-245, Zürich, Switserland, Sep. 1991. Cited on page 41.

Jenq, J.-F. and Sahni, S., "All Pairs Shortest Paths on a Hypercube Multiprocessor," *Proc. 1987 International Conference on Parallel Processing*, pp. 713-716, St. Charles, IL, Aug. 1987. Cited on page 114.

Jeremiassen, T. E. and Eggers, S. J., "Computing Per-Process Summary Side-Effect Information," *Fifth Workshop on Languages and Compilers for Parallel Computing*, pp. 115-122, New Haven, CT, Aug. 1992. Cited on page 107.

Johnson, D. B., "Distributed System Fault Tolerance Using Message Logging and Checkpointing," TR89-101 (Ph.D. thesis), Rice University, Dec. 1989. Cited on pages 10, 131, 138, 139.

Johnson, D. B. and Zwaenepoel, W., "The Peregrine High-Performance RPC System," TR91-151, Rice University, Mar. 1991. Cited on page 37.

Joseph, T. A. and Birman, K. P., "Low Cost Management of Replicated Data in Fault-Tolerant Systems," *ACM Trans. Comp. Syst.*, Vol. 4, No. 1, pp. 54-70, Feb. 1986. Cited on page 4.

Jul, E., Levy, H. M., Hutchinson, N., and Black, A., "Fine-Grained Mobility in the Emerald System," *ACM Trans. Comp. Syst.*, Vol. 6, No. 1, pp. 109-133, Feb. 1988. Cited on pages 109, 123.

Kaashoek, M. F., Bal, H. E., and Tanenbaum, A. S., "Experience with the Distributed Data Structure Paradigm in Linda," *Proc. First USENIX/SERC Workshop on Experience with Building Distributed and Multiprocessor Systems*, pp. 171-191, Ft. Lauderdale, FL, Oct. 1989. Cited on page 124.

Kaashoek, M. F., Michiels, R., Bal, H. E., and Tanenbaum, A. S., "Transparent Fault-Tolerance in Parallel Orca Programs," *Proc. Symposium on Experiences with Distributed and Multiprocessor Systems III*, pp. 297-312, Newport Beach, CA, Mar. 1992a. Cited on page 161.

Kaashoek, M. F. and Tanenbaum, A. S., "Fault Tolerance Using Group Communication," *Proc. Fourth ACM SIGOPS European Workshop*, Bologna, Italy, Sep. 1990. Cited on page 138. (Also published in Operating Systems Review, Vol. 25, No. 2)

Kaashoek, M. F. and Tanenbaum, A. S., "Group Communication in the Amoeba Distributed Operating System," *Proc. Eleventh International Conference on Distributed Computing Systems*, pp. 222-230, Arlington, TX, May 1991. Cited on page 86.

Kaashoek, M. F., Tanenbaum, A. S., Flynn Hummel, S., and Bal, H. E., "An Efficient Reliable Broadcast Protocol," *Operating Systems Review*, Vol. 23, No. 4, pp. 5-20, Oct. 1989. Cited on page 86.

Kaashoek, M. F., Tanenbaum, A. S., and Verstoep, K., "An Experimental Comparison of Remote Procedure Call and Group Communication," *Proc. Fifth ACM SIGOPS European Workshop* , Le Mont Saint-Michel, France, Sep. 1992b. Cited on page 161.

Karp, A. H., "Programming for Parallelism," *IEEE Computer*, Vol. 20, No. 5, pp. 43-57, May 1987. Cited on page 123.

Keleher, P., Cox, A. L., and Zwaenepoel, W., "Lazy Release Consistency for Software Distributed Shared Memory," *Proc. Nineteenth Annual Symposium on Computer Architecture*, pp. 13-21, Cold Coast, Australia, May 1992. Cited on page 96.

Knuth, D. E. and Moore, R. W., "An Analysis of Alpha-Beta Pruning," *Artificial Intelligence*, Vol. 6, pp. 293-326, 1975. Cited on page 7.

Koelbel, C., Mehrotra, P., and Van Rosendale, J., "Supporting Shared Data Structures on Distributed Memory Architectures," *Proc. Second Symposium on Principles and Practice of Parallel Programming*, pp. 177-186, Seattle, WA, Mar. 1990. Cited on page 123.

Koo, R. and Toueg, S., "Checkpointing and Roll-Back Recovery for Distributed Systems," *IEEE Trans. on Soft. Eng.*, Vol. 13, No. 1, pp. 23-31, Jan. 1987. Cited on pages 132, 138.

Kung, H. T., "Gigabit Local Area Networks: a Systems Perspective," *IEEE Communications Magazine*, Vol. 30, No. 4, pp. 79-89, Apr. 1992. Cited on page 5.

LaRowe Jr., R. P., Ellis, C. S., and Kaplan, L. S., "The Robustness of NUMA Memory Management," *Proc. Thirteenth Symposium on Operating System Principles*, pp. 137-151, Pacific Grove, CA, Oct. 1991. Cited on page 95.

Lamport, L., "How To Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. on Computers*, Vol. 28, No. 9, pp. 690-691, Sep. 1979. Cited on pages 88, 89.

Lampson, B. W., "Atomic Transactions," pp. 246-265, in *Distributed Systems - Architecture and Implementation, Lecture and Notes in Computer Science*, Springer Verlag, Berlin, 1981. Cited on page 142.

Lampson, B. W., "Designing a Global Name Service," *Proc. Fifth Annual Symposium on Principles of Distributed Computing*, pp. 1-10, Calgary, Canada, Aug. 1986. Cited on page 142.

Lawler, E. L. and Wood, D. E., "Branch-and-Bound Methods: a Survey," *Operations Research*, Vol. 14, No. 4, pp. 699-719, July 1966. Cited on page 7.

Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. S., "The Stanford Dash Multiprocessor," *IEEE Computer*, Vol. 25, No. 3, pp. 63-79, Mar. 1992. Cited on pages 95, 96, 125.

Levelt, W. G., Kaashoek, M. F., Bal, H. E., and Tanenbaum, A. S., "A Comparison of Two Paradigms for Distributed Shared Memory," *Software−Practice and Experience*, 1992. Cited on pages 7, 125. (accepted for publication)

Li, K. and Hudak, P., "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Comp. Syst.*, Vol. 7, No. 4, pp. 321-359, Nov. 1989. Cited on pages 9, 95, 124, 125.

Li, K., Naughton, J. F., and Plank, J. S., "Real-Time, Concurrent Checkpoint for Parallel Programs," *Proc. Second Symposium on Principles and Practice of Parallel Programming*, pp. 79-88, Seattle, WA, Mar. 1990. Cited on page 139.

Li, K., Naughton, J. F., and Plank, J. S., "Checkpointing Multicomputer Applications," *Proc. Tenth Symposium on Reliable Distributed Systems*, pp. 1-10, Bologna, Italy, Oct. 1991. Cited on page 138.

Liang, L., Chanson, S. T., and Neufeld, G. W., "Process Groups and Group Communication: Classification and Requirements," *IEEE Computer*, Vol. 23, No. 2, pp. 56-68, Feb. 1990. Cited on pages 4, 45, 46.

Liskov, B., "Distributed Programming in Argus," *Commun. ACM*, Vol. 31, No. 3, pp. 300-312, Mar. 1988. Cited on pages 10, 137.

Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., and Williams, M., "Replication in the Harp File System," *Proc. Thirteenth Symposium on Operating System Principles*, pp. 226-238, Pacific Grove, CA, Oct. 1991. Cited on pages 156, 159.

Liskov, B. and Scheifler, R. W., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Trans. Prog. Lang. Syst.*, Vol. 5, No. 3, pp. 381-404, July 1983. Cited on page 137.

Luan, S. W. and Gligor, V. D., "A Fault-Tolerant Protocol for Atomic Broadcast," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 3, pp. 271-285, July 1990. Cited on page 82.

Lucco, S. E., "A Heuristic Linda Kernel for Hypercube Multiprocessors," *Conf. on Hypercube Multiprocessors*, pp. 32-38, 1987. Cited on page 124.

Marzullo, K. and Schmuck, F., "Supplying High Availability with a Standard Network File System," *Proc. Eighth International Conference on Distributed Computing Systems*, pp. 447-453, San Jose, CA, June 1988. Cited on page 159.

Melliar-Smith, P. M., Moser, L. E., and Agrawala, V., "Broadcast Protocols for Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 1, pp. 17-25, Jan. 1990. Cited on page 82.

Minnich, R. G. and Farber, D. J., "Reducing Host Load, Network Contention, and Latency in a Distributed Shared Memory System," *Proc. Tenth International Conference on Distributed Computing Systems*, pp. 468-475, Paris, May 1990. Cited on pages 96, 125.

Mishra, S., Peterson, L. L., and Schlichting, R. D., "Implementing Fault-Tolerant Replicated Objects Using Psync," *Proc. Eighth Symposium on Reliable Distributed Systems*, pp. 42-52, Seattle, WA, Oct. 1989. Cited on page 160.

Montgomery, W. A., "Robust Concurrency Control for a Distributed Information System," MIT/LCS/TR-207 (Ph.D. thesis), M.I.T., Cambridge, MA, Dec. 1978. Cited on page 83.

Mullender, S. J., "Principles of Distributed Operating System Design," Ph.D. Thesis, Vrije Universiteit, Amsterdam, 1985. Cited on page 10.

Mullender, S. J., Van Rossum, G., Tanenbaum, A. S., Van Renesse, R., and Van Staveren, H., "Amoeba: a Distributed Operating System for the 1990s," *IEEE Computer*, Vol. 23, No. 5, pp. 44-53, May 1990. Cited on pages 10, 18.

National Bureau of Standards, "Data Encryption Standard," Fed. Inf. Process. Stand. Publ. 46, Jan. 1977. Cited on page 23.

Navaratnam, S., Chanson, S., and Neufeld, G., "Reliable Group Communication in Distributed Systems," *Proc. Eighth International Conference on Distributed Computing Systems*, pp. 439-446, San Jose, CA, June 1988. Cited on page 83.

Nitzberg, B. and Lo, V., "Distributed Shared Memory: a Survey of Issues and Algorithms," *IEEE Computer*, Vol. 24, No. 8, pp. 52-60, Aug. 1991. Cited on page 9.

Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B., "The Sprite Network Operating System," *IEEE Computer*, Vol. 21, No. 2, pp. 23-36, Feb. 1988. Cited on page 18.

Peterson, L. L., Buchholtz, N. C., and Schlichting, R. D., "Preserving and Using Context Information in IPC," *ACM Trans. Comp. Syst.*, Vol. 7, No. 3, pp. 217-246, Aug. 1989. Cited on pages 9, 83, 160.

Plummer, D. C., "An Ethernet Address Resolution Protocol," RFC 826, SRI Network Information Center, Nov. 1982. Cited on page 6.

Postel, J., "Internet Protocol," RFC 791, SRI Network Information Center, Sep. 1981a. Cited on pages 6, 41.

Postel, J., "Internet Control Message Protocol," RFC 792, SRI Network Information Center, Sep. 1981b. Cited on pages 6, 41.

Powell, M. L. and Presotto, D. L., "Publishing: a Reliable Broadcast Communication Mechanism," *Proc. Ninth Symposium on Operating System Principles*, pp. 100-109, Bretton Woods, NH, Oct. 1983. Cited on page 131.

Ramachandran, U., Ahamad, M., and Khalidi, M. Y., "Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer," *Proc. 1989 International Conference on Parallel Processing*, Vol. 2, pp. 160-169, St. Charles, IL, Aug. 1989. Cited on page 95.

Rogers, A. and Pingali, K., "Process Decomposition Through Locality of Reference," *Proc. SIGPLAN 89 Conf. on Programming Language Design and Implementation*, pp. 69-80, Portland, OR, July 1989. Cited on page 123.

Rosing, M., Schnabel, R., and Weaver, R., "The DINO parallel programming language," *Journal of Parallel and Distributed Computing*, Vol. 13, pp. 30-42, 1991. Cited on page 123.

Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., and Neuhauser, W., "Chorus Distributed Operating System," *Computing Systems*, Vol. 1, No. 4, pp. 305-370, 1988. Cited on pages 18, 45.

Saltzer, J. H., Reed, D. P., and Clark, D. D., "End-to-End Arguments in System Design," *ACM Trans. Comp. Syst.*, Vol. 2, No. 4, pp. 277-288, Nov. 1984. Cited on page 24.

Satyanarayanan, M., "Scalable, Secure, and Highly Available Distributed File Access," *IEEE Computer*, Vol. 23, No. 5, pp. 9-22, May 1990. Cited on page 159.

Satyanarayanan, M. and Siegel, E. H., "Parallel Communication in a Large Distributed Environment," *IEEE Trans. on Computers*, Vol. 39, No. 3, pp. 328-348, Mar. 1990. Cited on pages 84, 159.

Saunders, R. M. and Weaver, A. C., "The Xpress Transfer Protocol (XTP) - A Tutorial," *Computer Communication Review*, Vol. 20, No. 5, pp. 67-80, Oct. 1990. Cited on page 37.

Schroeder, M. D., Birrell, A. D., and Needham, R. M., "Experience with Grapevine: The Growth of a Distributed System," *ACM Trans. Comp. Syst.*, Vol. 2, No. 1, pp. 3-23, Feb. 1984. Cited on page 142.

Schwan, K., Blake, B., Bo, W., and Gawkowski, J., "Global Data and Control in Multicomputers: Operating System Primitives and Experimentation with a Parallel Branch-and-Bound Algorithm," *Concurrency−Practice and Experience*, Vol. 1, No. 2, pp. 191-218, Dec. 1989. Cited on page 96.

Shrivastava, S. K., Ezhilchelvan, P. D., Speirs, N. A., Tao, S., and Tully, A., "Principal Features of the VOLTAN Family of Reliable Node Architectures for Distributed Systems," *IEEE Trans. on Computers*, Vol. 41, No. 5, pp. 542-549, May 1992. Cited on page 143.

Sistla, A. P. and Welch, J. L., "Efficient Distributed Recovery Using Message Logging," *Proc. Eighth Annual Symposium on Principles of Distributed Computing*, pp. 223-238, Edmonton, Alberta, Aug. 1989. Cited on page 138.

Skeen, D., "Determining the Last Process to Fail," *ACM Trans. Comp. Syst.*, Vol. 3, No. 1, pp. 15-30, Feb. 1985. Cited on pages 148, 153.

Spafford, E. H., "The Internet Worm: Crisis and Aftermath," *Commun. ACM*, Vol. 32, No. 6, pp. 678-688, June 1989. Cited on page 35.

Spector, A. Z., Pausch, R., and Bruell, G., "Camelot: a Flexible, Distributed Transaction Processing System," *Proc. CompCon 88*, pp. 432-439, San Francisco, CA, Feb. 1988. Cited on page 10.

Stoer, J. and Bulirsch, R., "Introduction to Numerical Analysis," Springer-Verlag, New York, NY, 1983. Cited on pages 7, 104.

Strom, R. E. and Yemini, S., "Optimistic Recovery in Distributed Systems," *ACM Trans. Comp. Syst.*, Vol. 3, No. 3, pp. 204-226, Aug. 1985. Cited on pages 10, 131, 138.

Strom, R. E. and Yemini, S., "Typestate: a Programming Language Concept for Enhancing Software Reliability," *IEEE Trans. on Soft. Eng.*, Vol. 12, No. 1, pp. 157-171, Jan. 1986. Cited on page 100.

Stumm, M. and Zhou, S., "Algorithms for Implementing Distributed Shared Memory," *IEEE Computer*, Vol. 23, No. 5, pp. 54-64, May 1990. Cited on page 122.

Swan, R. J., Fuller, S. H., and Siewiorek, D. P., "Cm* - A Modular, Multimicroprocessor System," *Proc. Nat. Comp. Conf.*, pp. 645-655, 1977. Cited on page 94.

Tanenbaum, A. S., "Computer Networks 2nd ed.," Prentice-Hall, Englewood Cliffs, NJ, 1989. Cited on pages 5, 72.

Tanenbaum, A. S., "Modern Operating Systems," Prentice-Hall, Englewood Cliffs, NJ, 1992. Cited on pages 2, 15.

Tanenbaum, A. S., Kaashoek, M. F., and Bal, H. E., "Parallel Programming Using Shared Objects and Broadcasting," *IEEE Computer*, Vol. 25, No. 8, pp. 10-19, Aug. 1992. Cited on pages 86, 127.

Tanenbaum, A. S., Kaashoek, M. F., Van Renesse, R., and Bal, H. E., "The Amoeba Distributed Operating System - a Status Report," *Computer Communications*, Vol. 14, No. 6, pp. 324-335, Aug. 1991. Cited on page 15.

Tanenbaum, A. S., Mullender, S. J., and Van Renesse, R., "Using Sparse Capabilities in a Distributed Operating System," *Proc. Sixth International Conference on Distributed Computing Systems*, pp. 558-563, Cambridge, MA, May 1986. Cited on page 13.

Tanenbaum, A. S., Van Renesse, R., Van Staveren, H., Sharp, G., Mullender, S. J., Jansen, A., and Van Rossum, G., "Experiences with the Amoeba Distributed Operating System," *Commun. ACM*, Vol. 33, No. 12, pp. 46-63, Dec. 1990. Cited on pages 10, 18, 37.

Tseung, L. C. N. and Yu, K-C., "The implementation of Guaranteed, Reliable, Secure Broadcast Networks," *1990 ACM Eighteenth Annual Computer Science Conference*, pp. 259-266, Washington D.C., Feb. 1990. Cited on page 83.

Van Renesse, R., "The Functional Processing Model," Ph.D. Thesis, Vrije Universiteit, Amsterdam, 1989. Cited on pages 10, 15, 141−143, 161.

Van Renesse, R., Tanenbaum, A. S., Van Staveren, H., and Hall, J., "Connecting RPC-based Distributed Systems Using Wide-area Networks," *Proc. Seventh International Conference on Distributed Computing Systems*, pp. 28-34, Berlin, Sep. 1987. Cited on page 35.

Van Renesse, R., Tanenbaum, A. S., and Wilschut, A., "The Design of a High-Performance File Server," *Proc. Ninth International Conference on Distributed Computing Systems*, pp. 22-27, Newport Beach, CA, June 1989. Cited on page 132.

Verissimo, P., Rodrigues, L., and Baptista, M., "AMp: a Highly Parallel Atomic Multicast Protocol," *Proc. SIGCOMM 89*, pp. 83-93, Austin, TX, Sep. 1989. Cited on page 84.

Vishnubhotla, P., "Synchronization and Scheduling in the ALPS Objects," *Proc. Eighth International Conference on Distributed Computing Systems*, pp. 256-264, San Jose, CA, June 1988. Cited on page 123.

Weihl, W. and Liskov, B., "Implementation of Resilient, Atomic Data Types," *ACM Trans. Prog. Lang. Syst.*, Vol. 7, No. 2, pp. 244-269, Apr. 1985. Cited on page 137.

Wilson, A. W., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Proc. Fourteenth Annual International Symposium on Computer Architecture*, pp. 244-252, Pittsburgh, PA, June 1987. Cited on page 95.

Wood, M. D., "Replicated RPC Using Amoeba Closed Group Communication," IR-306, Vrije Universiteit, Amsterdam, Oct. 1992. Cited on page 84.

Wu, K-L. and Fuchs, W. K., "Recoverable Distributed Shared Virtual Memory," *IEEE Trans. on Computers*, Vol. 39, No. 4, pp. 460-469, Apr. 1990. Cited on page 139.

Xu, A. S., "A Fault-tolerant Network Kernel for Linda," TR-424, M.I.T., Cambridge, MA, Aug. 1988. Cited on page 138.

Young, M., Tevenian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., and Baron, R., "Duality of Memory and Communication in the Inplementation of a Multiprocessor Operating System," *Proc. Eleventh on Operating Systems Principles*, pp. 63-67, Austin, TX, Nov. 1987. Cited on page 74.

Zimmerman, H., "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. on Communications*, Vol. 28, No. 4, pp. 425-432, Apr. 1980. Cited on pages 16, 37.

Zwaenepoel, W., "Protocols for Large Data Transfers over Local Networks," *Proc. Ninth Data Communications Symposium*, pp. 22-32, Whistler Mountain, Canada, Sep. 1985. Cited on page 36.

Index

A

Accessible copies, 148
ALPS, 123
Amber, 123
Amoeba, 10
Amoeba process descriptor, 133
ARP, 6
ASP, 114
At-most-once remote procedure call, 32

B

BB method, 57
Bcastreq, 65
Blocking send, 92
Boot server, 131
Broadcasting, 5, 6
Broadcasting Sequential Processes, 122
Broadcastreceive, 66
Bullet server, 132
Bus-based multicomputer, 91
Bus-based multiprocessor, 89

C

Capability, 13, 32
Causal ordering, 46
Check field, 13, 144
Checkpoint, 132
Closed group, 46, 50
Cm*, 94
Coda, 159
Commit block, 144, 149
Configuration vector, 149
Copy mode, 144
CreateGroup, 51
Crossbar switch, 90

D

Delivery semantics, 46
DES, 23
Directory, 141
Directory service, 32, 141, 167
Distributed operating system, 2
Distributed shared memory, 9, 93, 122
Distributed shared memory (DSM), 7
Distributed system, 1
Distributed system requirements, 17
Duplication protocol, 145
Dynamic group, 46

E

Echo, 159
Emerald, 123
EP-identifier, 23

F

Fault tolerance, 3
Fault-tolerant application, 10, 167
FIFO ordering, 46
FLIP, 6, 16, 163
FLIP address, 20
FLIP box, 21
FLIP header, 24

FLIP interface, 22
FLIP packet switch, 21
FLIP routing table, 26
ForwardRequest, 54
Fragment, 20
Functional specialization, 3

G

Gax, 131, 133
Geographically distributed application, 4
GetInfoGroup, 54
Globally-consistent checkpoint, 132, 167
Group addressing, 45
Group communication, 6, 18, 163, 165
Group descriptor, 51
Group primitives, 51
Group structure, 46

H

Harp, 159
Hopcnt, 23, 24
Hypercube, 1, 92

I

ICMP, 6
IGMP, 6
Initiator, 150
Intentions list, 145
Internetwork, 16
Invitation protocol, 70
IP, 6, 16, 41
Isis, 9, 47, 79, 140

J

JoinGroup, 52

L

LeaveGroup, 53
Linda, 122, 124

M

Mailbox, 93
Memory coherence, 88, 89
Mentat, 123
Mether, 96
Microkernel, 12
Mourned set, 153
Multicasting, 5, 6
Multicomputers, 91
Multiprocessors, 88
Munin, 96

N

Negative acknowledgement scheme, 49, 165
Network management, 19, 34, 164
Network operating system, 2
Network service access point (NSAP), 20
Network weight, 26
Nonblocking send, 92
NonUniform Memory Access Machines (NUMA), 93
NonVolatile RAM (NVRAM), 155

O

Object field, 13, 144
Object manager, 102
Object-based NUMA, 95
Omega network, 90
One-copy mode, 144
One-copy serializability, 142
One-way function, 13, 23
Open group, 46
Operation descriptor, 101
Optimistic recovery, 131
Optimum computing interval, 137
Orca, 88, 98
Order, 5
Ordering, 45
OSI model, 16
Overlapping groups, 47
Owner capability, 13, 133

P

Packet, 20
Page-oriented NUMA, 94
Parallel application, 3, 166
Partial replication, 122
PB method, 57
Pingpong program, 136
Port, 13, 32, 48
Port cache, 32, 156
Positive acknowledgement scheme, 49, 165
Primary copy protocol, 159
Primary server, 159
Private address, 22
Process descriptor, 105
Processhist, 66
Processor pool, 10
Processrec, 68
Psync, 9, 160
Public address, 23

R

RARP, 6
ReceiveFromGroup, 53, 62
Recovery protocol, 70
Release consistency, 96
Reliability, 45
Remote Procedure Call (RPC), 4, 18, 32, 93, 163
ResetGroup, 53
Resilience degree, 53
Response semantics, 46
Roll back, 132, 167

S

Score, 105
Secondary server, 159
Security, 19, 34, 164
SendToGroup, 53, 63
Sequencer, 49, 55, 165
Sequentially consistent, 88
Shared data-object model, 7, 97, 166
Shared Virtual Memory, 124, 125
Snoopy cache, 89
SOR, 117
SR, 122
Stale value, 89
Static group, 46
Stun signal, 133
Synchronization, 91

T

TCP, 16
Total ordering, 46, 53
Transparency, 18, 163
TSP, 110
Tuple Space, 124
Two-copy mode, 144

U

UDP, 16

V

V, 9, 47, 82
VMTP, 7, 41

W

Weak ordering, 96, 125
Wide-area network, 1, 19, 35, 164
Word-oriented NUMA, 94

Curriculum Vitae

Name:	M. Frans Kaashoek
Date of birth:	April 4, 1965
Place of birth:	Leiden, The Netherlands
Nationality:	Dutch
Email:	kaashoek@cs.vu.nl

Education

Secondary school	Sept ’77 - Aug ’83	Eerste Christelijk Lyceum, Haarlem, The Netherlands.
University	Sept ’83 - Dec ’88	Vrije Universiteit, Amsterdam, The Netherlands. Master degree (with honors).
	Dec ’88 - Dec ’92	Vrije Universiteit, Amsterdam, The Netherlands. Ph.D. student in Computer Systems. Supervisor: A.S. Tanenbaum.

Work experience

Fall semester ’85 - Dec ’88: Teaching assistant for the courses ‘‘Introduction to Programming,’’ ‘‘Data Structures,’’ and ‘‘Compiler Construction.’’

Fall semester ’87 - Dec ’88: Research assistant on the Amsterdam Compiler Kit.

Dec ’88 - Dec ’92 : Teaching assistant for the courses ‘‘Compiler Construction,’’ ‘‘Operating Systems,’’ ‘‘Computer Networks,’’ and ‘‘Distributed Operating Systems.’’

Publications (refereed)

1. Kaashoek, M.F., van Renesse, R., van Staveren, H., and Tanenbaum, A.S., ‘‘FLIP: an Internetwork Protocol for Supporting Distributed Systems,’’ *ACM Trans. Comp. Syst.* (accepted for publication)

2. Levelt, W.G., Kaashoek, M.F., Bal, H.E., and Tanenbaum, A.S., ‘‘A Comparison of Two Paradigms for Distributed Shared Memory,’’ *Software−Practice and Experience.* (accepted for publication)

3. Tanenbaum, A.S., and Kaashoek, M.F., ‘‘The Amoeba Microkernel’’ in *Distributed Open Systems*, Brazier and Johansen (eds).

4. Kaashoek, M.F., Tanenbaum, A.S., and Verstoep, K., ‘‘Group Communication in Amoeba and its Applications,’’ *Proc. OpenForum 92*, Utrecht, The Netherlands, Nov. 1992

5. Kaashoek, M.F., Tanenbaum, A.S., and Verstoep, K., ‘‘An Experimental Comparison of Remote Procedure Call and Group Communication,’’ *Proc. Fifth ACM SIGOPS European Workshop*, Le Mont Saint-Michel, France, Sept. 1992

6. Tanenbaum, A.S., Kaashoek, M.F., and Bal, H.E., ‘‘Parallel Programming Using Shared Objects and Broadcasting,’’ *IEEE Computer*, Vol. 25, No. 8, pp. 10-19, Aug. 1992.

7. Bal, H.E., Kaashoek, M.F., Tanenbaum, A.S., and Jansen, J., ‘‘Replication Techniques for Speeding up Parallel Applications on Distributed Systems,’’ *Concurrency−Practice and Experience*, Vol. 4, No. 5, pp. 337-355, Aug. 1992.

8. Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S., ‘‘Orca: A language for Parallel Programming of Distributed Systems,’’ *IEEE Trans. on Soft. Eng.*, Vol. 18, No. 3, pp. 190-205, Mar. 1992.

9. Kaashoek, M.F., Michiels, R., Bal, H.E., and Tanenbaum, A.S., ‘‘Transparent Fault-tolerance in Parallel Orca Programs,’’ *Proc. Symposium on Experiences with Distributed and Multiprocessor Systems III*, pp. 297-312, Newport Beach, CA, Mar. 1992.

10. Douglis, F., Kaashoek, M.F., Tanenbaum, A.S., and Ousterhout, J.K., ‘‘A Comparison of Two Distributed Systems: Amoeba and Sprite,’’ *Computing Systems*, Vol. 4, No. 3, pp. 353-384, Dec. 1991.

11. Tanenbaum, A.S., Kaashoek, M.F., van Renesse, R., and Bal, H.E., ‘‘The Amoeba

Distributed Operating System - a Status Report,'' *Computer Communications*, Vol. 14, No. 6, pp. 324-335, Aug. 1991.

12. Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S., ''Orca: a Language Based on Shared Data-objects,'' *Proc. EPCC Workshop on Linda-like Systems and Their Implementation*, Edinburgh Parallel Computing Centre, University of Edinburgh, pp. 5-13, Edinburgh, June 1991.

13. Kaashoek, M.F., and Tanenbaum, A.S., ''Group Communication in the Amoeba Distributed Operating System,'' *Proc. Eleventh Internatinal Conference on Distributed Computer Systems*, IEEE Computer Society, pp. 222-230, Arlington, TX, May 1991.

14. Flynn Hummel, S., Tanenbaum, A.S., and Kaashoek, M.F., ''A Scalable Object-based Architecture,'' *Second Workshop on Scalable Shared-Memory Multiprocessors*, Toronto, May 1991.

15. Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S., ''Parallel Programming on Amoeba,'' *Proc. Sixth International Workshop on the Use of Supercomputers in Theoretical Sciences*, Plenum Press, pp. 89-105, Antwerpen, Belgium, Jan. 1991.

16. Kaashoek, M.F., and Tanenbaum, A.S., ''Fault Tolerance Using Group Communication,'' *Proc. Fourth ACM SIGOPS European Workshop*, Bologna, Italy, Sept. 1990.

17. Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S., ''Experience with Distributed Programming in Orca,'' *Proc. 1990 International Conference on Computer Languages*, pp. 79-89, New Orleans, LA, Mar. 1990.

18. Kaashoek, M.F., Bal, H.E., and Tanenbaum, A.S., ''Experience with the Distributed Data Structure Paradigm in Linda,'' *Proc. First USENIX/SERC Workshop on Experience with Building Distributed and Multiprocessor Systems*, pp. 171-191, Ft. Lauderdale, FL, Oct. 1989.

19. Bal, H.E., Kaashoek, M.F., and Tanenbaum, A.S., ''A Distributed Implementation of the Shared Data-Object Model,'' *Proc. First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1-19, Ft. Lauderdale, FL, Oct. 1989.

Publications (submitted for publication)

20. Kaashoek, M.F., Tanenbaum, A.S., ''Efficient Reliable Group Communication for Distributed Systems,'' IR-295, Vrije Universiteit, Amsterdam, June 1992.

21. Bal, H.E., and Kaashoek, M.F., ''Optimizing Object Distribution in Orca by

Integrating Compile-Time and Run-Time Techniques,'' IR-254 (revised), Vrije Universiteit, Amsterdam, Sept. 1991.

22. Kaashoek, M.F., Tanenbaum, A.S., and Verstoep, K., Using Group Communication to Implement a Fault-Tolerant Directory Service,'' IR-305, Vrije Universiteit, Amsterdam, Oct. 1992.

Publications (unrefereed)

23. Bal, H.E., Tanenbaum, A.S., and Kaashoek, M.F., ''Orca: A Language for Distributed Programming,'' *SIGPLAN Notices*, Vol. 25, No. 5, pp. 17-25, May 1990.

24. Moergestel, L., Bal, H., Kaashoek, F., van Renesse, R., Sharp, G., van Staveren, H., Tanenbaum, A.S., ''Amoeba on a Multiprocessor,'' IR-206, Vrije Universiteit, Amsterdam, Dec. 1989.

25. Tanenbaum, A.S., Kaashoek, M.F., Langendoen, K.G., and Jacobs, C.J.H., ''The Design of Very Fast Portable Compilers,'' *SIGPLAN Notices*, Vol. 24, No. 11, pp. 125-132, Nov. 1989.

26. Kaashoek, M.F., Tanenbaum, A.S., Flynn Hummel, S., and Bal, H.E., ''An Efficient Reliable Broadcast Protocol,'' *Operating Systems Review*, Vol. 23, No. 4, pp. 5-20, Oct. 1989.

27. Kaashoek, F., and Langendoen, K., ''The Code Expander Generator,'' IM-9, Vrije Universiteit, Amsterdam, May 1988.

This page intentionally left blank.

This page intentionally left blank.

Contents

Acknowledgements iv

Samenvatting vi

1 INTRODUCTION 1

1.1 Applications for Distributed Computer Systems 3

1.2 Why Group Communication? 4

1.3 Problems and Solutions 6

1.4 Related Work 8

1.5 Experimental Environment: Amoeba 10

1.6 Outline of the Thesis 14

2 FAST LOCAL INTERNET PROTOCOL 16

2.1 Distributed System Requirements 17

2.2 Flip Service Definition 20

2.3 The Host Interface 22

2.4 The Flip Protocol 24

2.4.1 The FLIP Fragment Format 24

2.4.2 The FLIP Routing Protocol 26

2.5 Using Flip under Amoeba 30

2.6 Performance of FLIP 36

2.7 Discussion and Comparison 37

2.7.1 Discussion 39

2.7.2 Comparison 41

2.8 Conclusion 42

3 EFFICIENT RELIABLE GROUP COMMUNICATION 44

3.1 Design Issues in Group Communication 45
3.2 Design Choices 47
3.3 Group Primitives in Amoeba 51
3.4 The Broadcast Protocol 54
3.4.1 Basic Protocol 56
3.4.2 Protocol during Normal Operation 60
3.4.3 Protocol for Recovery 70
3.5 Performance 72
3.6 Comparison with Related Work 78
3.7 Conclusion 85

4 PARALLEL PROGRAMMING USING BROADCASTING AND SHARED OBJECTS 87

4.1 Architectures for Parallel Programming 88
4.1.1 Multiprocessors 88
4.1.2 Multicomputers 91
4.2 NUMA Machines 93
4.3 The Shared Data-Object Model 97
4.4 Orca Programs 98
4.4.1 The Language 98
4.4.2 The Interface between the Compiler and the Run-Time System 100
4.5 Object Management 101
4.6 Optimized Object Management 103
4.7 The Optimizing Compiler and Run-Time System 106
4.7.1 The Optimizing Compiler 106
4.7.2 The Optimizing Run-Time System 107
4.8 Example Applications and Their Performance 109
4.8.1 The Traveling Salesman Problem 110
4.8.2 The All-Pairs Shortest Paths Problem 114
4.8.3 Successive Overrelaxation 117
4.9 Comparison with Related Work 120
4.10 Conclusion 126

5 FAULT-TOLERANT PROGRAMMING USING BROADCASTING 128

5.1 Transparent Fault Tolerance in Parallel Orca Programs 129
5.1.1 Running Orca Programs under Amoeba 130
5.1.2 Designing a Fault-Tolerant RTS 131
5.1.3 Fault-Tolerant Implementation of Orca 133
5.1.4 Performance 135
5.1.5 Comparison with Related Work 137
5.1.6 Conclusion 140

5.2 A Fault-Tolerant Directory Service 140
5.2.1 Directory Service Based on RPC 143
5.2.2 Directory Service Based on Group Communication 148
5.2.3 Experimental Comparison 154
5.2.4 Discussion and Comparison 157
5.2.5 Conclusion 160

5.3 Summary and Discussion 160

6 SUMMARY 162

6.1 Bottom Layer: FLIP 162
6.2 Middle Layer: Group Communication 165
6.3 Top Layer: Applications 166
6.4 Conclusion 168

Appendix A: the FLIP Protocol 169

Appendix B: Analysis of the Cost of a Checkpoint 174

References 176

Index 192

Curriculum Vitae 195

Acknowledgements

The classical paradigm of a student working single-handedly on his dissertation in an ivory tower is not applicable to the way this thesis was done. Many people contributed in various important ways. Without their help the present dissertation would have never been written. To reflect the style in which the research was done, each chapter ends with a short note giving the history of the results presented.

I am especially indebted to Andy Tanenbaum, my supervisor, who taught me my first steps in Science. He created a very stimulating environment to work in, he taught me how to write, how to discuss, and the importance of grant proposals and papers. He gave me complete freedom in the search of a topic, but followed every single step carefully and provided me with suggestions and critical comments. I hope he considers me a grown-up now.

I wish to thank my co-supervisor, Henri Bal. In addition to being a very fine person to work with, Henri also undertook the unpleasant task of making sure that the thesis writing was on schedule and of reading the very first draft of each chapter. His eye for detail caught many errors and his suggestions for the organization of each chapter have helped me in structuring the material presented.

I also wish to thank the external referee, Willy Zwaenepoel. By the time Willy received a draft of the dissertation, many reviewers and people had commented on it. Nevertheless, after he had read a chapter a long email message showed up in my mailbox with comments on all aspects of my writing and the arguments that I had used. I also wish to thank him for the advice that he has given me in the last couple of years on the various problems that a Ph.D student encounters.

I am grateful to the many co-authors of various papers: Henri Bal, Fred Douglis, Susan Flynn Hummel, Ceriel Jacobs, Jack Jansen, Koen Langendoen, Willem Levelt, Raymond Michiels, John Ousterhout, Robbert van Renesse, Hans van Staveren, Andy Tanenbaum, and Kees Verstoep. Most of the results in this thesis can be traced back to one of these collaborative efforts. It was a pleasure to have the opportunity to work with them.

Another group of excellent people who contributed in many ways to the results presented in this dissertation is the VU Amoeba team. Although the delivery of the Amoeba distribution and Greg's TODO lists made life sometimes less pleasant (especially for Greg), it was great fun to be member of this team. I am very grateful for the patience and the help that they provided while I installed and debugged the next version of the kernel or FLIP driver. On the other hand, given the coffee that most members of

the team drink, I am not surprised that they can stomach occasional kernel crashes. It usually knocked me out for two weeks.

I wish to thank John Carter, Leendert van Doorn, Fred Douglis, Elmootazbellah Nabil Elnozahy, Dick Grune, Philip Homburg, Ceriel Jacobs, Wiebren de Jonge, Koen Langendoen, Robbert van Renesse, John Romein, Loek Schoenmaker, Greg Sharp, Kees Verstoep, and Mark Wood for reading drafts of this dissertation. Their careful reading and suggestions considerably improved it.

I also wish to thank the Department of Mathematics and Computer Science of the Vrije Universiteit for providing support for carrying out the research. I am grateful for their generous funding of the many trips, books, and computers. Also, special thanks to the system administrators for doing an excellent job in maintaining a fantastic computer infrastructure, even during evenings and weekends.

Thanks to my parents for their ever continuing guidance and financial support. I am deeply indebted to them for everything they taught me.

Most of all I would like to thank Mathilde for all her strong and loving support.

Samenvatting

Een gespreid computersysteem is een verzameling computers, verbonden door één of meer datanetwerken. De computers in een gespreid systeem kunnen met elkaar communiceren door via deze netwerken boodschappen te versturen. Een bedrijfssysteem voor een gespreid computersysteem heeft tot taak ervoor te zorgen dat gebruikers er gemakkelijk op kunnen werken.

De meeste gespreide bedrijfssystemen bieden de programmeur communicatieprimitieven voor het sturen van een boodschap van één proces naar precies *één* ander proces. Veel programmatuur echter heeft baat bij communicatieprimitieven waarmee men een boodschap van één proces naar *meerdere* processen tegelijkertijd kan sturen. Dergelijke communicatieprimitieven worden *groepscommunicatie*-primitieven genoemd. Deze dissertatie beschrijft het ontwerp, de implementatie, het gebruik en de effectiviteit van een gespreid bedrijfssysteem dat gebaseerd is op groepscommunicatie. Het onderzoek is praktisch van aard: alle gerapporteerde ideeën zijn geïmplementeerd als onderdeel van het gespreide systeem Amoeba en de meeste resultaten worden dagelijks gebruikt.

Deze groepscommunicatie kan men eenvoudig realiseren door een primitieve te maken die meerdere keren de bestaande communicatieprimitieve aanroept om een boodschap van één proces naar één ander proces te sturen, maar deze methode is duur: om een boodschap betrouwbaar naar n processen te sturen zijn $2(n-1)$ netwerkpakketten nodig. Een ander probleem met deze aanpak is dat boodschappen van verschillende zenders niet geordend worden: als twee processen, A en B, $n-1$ boodschappen betrouwbaar versturen, dan kunnen sommige ontvangers eerst de boodschap van A ontvangen en dan de boodschap van B, terwijl andere ontvangers de boodschappen in omgekeerde volgorde kunnen ontvangen. Het ongeordend zijn van de boodschappen bemoeilijkt het schrijven van gespreide programmatuur.

In deze dissertatie wordt een andere methode voor groepscommunicatie voorgesteld. Het hier beschreven systeem kan het best als een gelaagd model beschouwd worden (zie figuur 1). De onderste laag biedt onbetrouwbare groepscommunicatie, de middelste laag gebruikt de onbetrouwbare groepscommunicatie van de onderste laag om betrouwbare en totaal geordende groepscommunicatie te implementeren en de bovenste laag gebruikt de betrouwbare en totaal geordende groepscommunicatie van de middelste laag om er gespreide programmatuur mee te implementeren. In deze samenvatting wordt de functionaliteit van elke laag kort samengevat en wordt aangegeven hoe de geleverde functionaliteit bereikt wordt.

3	Programmatuur (bv. parallelle programma's of een foutbestendig opslagsysteem)
2	groepscommunicatie
1	Fast Local Internet Protocol (FLIP)

Figuur 1. Structuur van het systeem.

Onderste laag: FLIP

Bij veel datanetwerken kan men een boodschap onbetrouwbaar en ongeordend van één naar meerdere computers sturen. De onderste laag stelt deze fysieke broadcast- en multicast-faciliteiten van het netwerk beschikbaar voor hogere lagen. Om dit op een nette en netwerk-onafhankelijke wijze te doen is een routeringsprotocol nodig. In plaats van bestaande protocollen aan te passen presenteren we in hoofdstuk 2 een nieuw routeringsprotocol, genaamd Fast Local Internetwork Protocol (FLIP). De reden voor deze keuze is, dat bestaande routeringsprotocollen niet voor gespreide systemen ontworpen zijn en dus niet aan alle eisen voldoen die door een gespreid computersysteem aan een routeringsprotocol gesteld worden.

In hoofdstuk 2 worden de volgende eisen geïdentificeerd: transparantie, efficiënte communicatie van één proces naar één ander proces in de vorm van Remote Procedure Call (RPC), groepscommunicatie, data-beveiliging, beheer van datanetwerken en communicatie over wereldwijde datanetwerken. FLIP is een routeringsprotocol dat aan alle genoemde eisen voldoet of het mogelijk maakt voor een gespreid bedrijfssysteem aan deze eisen te voldoen.

Als voorbeeld zullen we kort samenvatten hoe FLIP transparantie biedt. In tegenstelling tot de adressen in de meeste andere routeringsprotocollen zijn FLIP-adressen *plaats-onafhankelijk*: een FLIP-adres identificeert een proces of een groep van processen en niet de computer, met het gevolg dat, als een proces van plaats verhuist, het zijn eigen FLIP-adres kan behouden.

Men kan adressen in routeringsprotocollen goed vergelijken met telefoonnummers. Als iemand verhuist naar een andere woonplaats, dan krijgt hij of zij een nieuw telefoonnummer. Dit is echter buitengewoon onhandig: het zou wenselijk zijn dat iemand zijn hele leven te bereiken is onder hetzelfde telefoonnummer, zodat iedereen hem kan bereiken waar hij zich op dat gegeven moment ook bevindt. De meeste routeringsprotocollen werken met adressen, zoals de PTT met telefoonnummers werkt: ze zijn plaats-afhankelijk. FLIP-adressen daarentegen gedragen zich zoals telefoonnummers in de ideale situatie: ze zijn volkomen plaats-onafhankelijk.

De plaats-onafhankelijke adressen in FLIP worden geïmplementeerd door iedere computer een algoritme voor dynamische routering te laten uitvoeren. Iedere computer heeft een routetabel die bij elk FLIP-adres een verwijzing bevat naar een lokatie van een computer. De verwijzingen zijn *hints*: ze zijn in de meeste gevallen correct, maar er bestaat de mogelijkheid dat ze achterhaald en daardoor onbruikbaar zijn. FLIP ontdekt automatisch of een hint achterhaald is, zoekt uit wat de juiste lokatie voor het

desbetreffende adres is en slaat deze nieuwe gegevens op in de routetabel. Zolang een proces niet verhuist en het netwerkenstelsel niet van configuratie verandert, zijn de hints correct. Bij elke verandering van plaats of verandering in de configuratie kan een hint onbruikbaar worden, maar over het algemeen zullen dergelijke veranderingen niet zo vaak voorkomen.

Middelste laag: groepscommunicatie

De middelste laag zet de onbetrouwbare multicast-communicatie van FLIP om in betrouwbare en totaal geordende groepscommunicatie. De primitieven voor groepscommunicatie, beschreven in hoofdstuk 3, garanderen dat alle leden van één groep alle gebeurtenissen betreffende die groep in dezelfde volgorde waarnemen. De toetreding van een nieuw lid tot de groep, het ontvangen van een boodschap gericht aan de groep en het falen van één van de leden worden door alle nog goed functionerende leden in dezelfde volgorde waargenomen. De geleverde abstractie maakt het eenvoudig voor programmeurs om gespreide programmatuur te schrijven.

Met deze primitieven voor groepscommunicatie kan de programmeur ook naar believen efficiëntie inruilen voor foutbestendigheid. Een programmeur kan het systeem vragen om te garanderen dat een boodschap correct afgeleverd wordt, zelfs wanneer één of meer computers falen (d.w.z. onbruikbaar worden door een fout in de software of hardware). Als de programmeur bijvoorbeeld een foutbestendigheidsgraad van 1 specificeert, zal het systeem garanderen dat alle goed-functionerende leden alle boodschappen in dezelfde volgorde ontvangen, zelfs als één van de leden van de groep faalt. Deze eigenschap van het systeem vereenvoudigt het schrijven van foutbestendige programmatuur.

Het protocol dat onbetrouwbare communicatie omzet in betrouwbare en totaal geordende communicatie, is gebaseerd op een *sequencer* en een *negative-acknowledgement*-schema. In een communicatieprotocol stuurt de ontvanger van een boodschap vaak direct een ‘‘acknowledgement’’, een boodschap om te bevestigen dat de eerste boodschap correct gearriveerd is. In een negative-acknowledgementschema worden er geen acknowledgementpakketjes gestuurd, maar vraagt het protocol om opnieuw een boodschap te sturen, zodra het ontdekt dat een computer een boodschap gemist heeft. Aangezien de tegenwoordige datanetwerken zeer goed functioneren, zal het niet vaak voorkomen dat een boodschap verloren gaat en levert een negative-acknowledgementschema een besparing op in het aantal acknowledgementpakketten.

De totale ordening wordt gegarandeerd door één sequencer per groep te hebben. Deze sequencer is niet fundamenteel verschillend van de andere leden in de groep. Elk lid kan sequencer worden, maar op elk moment is er maar één sequencer; als de sequencer faalt, kiezen de overige leden een nieuwe sequencer. In dit opzicht is de sequencer te vergelijken met de voorzitter van een commissie: elke commissie heeft één voorzitter en als deze niet aanwezig kan zijn, wordt er een nieuwe voorzitter gekozen.

Met behulp van een sequencer kan eenvoudig een totale ordening gegarandeerd

worden, en wel als volgt. Wanneer een lid een boodschap naar de groep wil sturen, dan verstuurt hij deze naar de sequencer. De sequencer kent er het volgende ongebruikte *sequence nummer* aan toe en stuurt de boodschap met sequence nummer naar de groep. Met behulp van het nummer kunnen alle overige leden er altijd voor zorgen dat ze de boodschappen die aan de groep gestuurd worden, in de totale ordening afleveren. Het complete protocol is gecompliceerder, omdat er rekening gehouden moet worden met het verlies van boodschappen en het falen van computers.

Het nadeel van een protocol met een sequencer zou kunnen zijn dat de sequencer mogelijkerwijs de prestatie van het protocol beperkt, aangezien alle boodschappen via de sequencer moeten gaan. In de praktijk blijkt dit mee te vallen. Ten eerste is er één sequencer per groep en niet één voor het hele systeem en dus hoeven alleen de boodschappen van één groep behandeld te worden door een gegeven sequencer. Ten tweede verricht de sequencer weinig werk: hij ontvangt de boodschap en stuurt de boodschap meteen weer weg, voorzien van een sequence nummer. Metingen laten dan ook zien dat de sequencer 815 boodschappen per seconde kan verwerken op standaard hardware (een collectie van 20-Mhz MC68030's verbonden door een 10 Mbit/s Ethernet). We verwachten dat dit ruim voldoende is voor het ondersteunen van middelgrote tot grote groepen. Het voordeel van een sequencer is dat het protocol voor totale ordening er eenvoudig en efficiënt door wordt. Het betrouwbaar en totaal geordend versturen van een boodschap naar een groep van 30 processen kost slechts 2.8 msec. De prestatie van het protocol is daarmee beter dan die van enig ander soortgelijk protocol.

De bovenste laag: programmatuur

Het doel van de bovenste laag was om aan te tonen dat groepscommunicatie het ontwikkelen van efficiënte gespreide programmatuur vereenvoudigt. In deze dissertatie zijn twee toepassingsgebieden bestudeerd: parallelle programmatuur en foutbestendige programmatuur. We zullen beide hier kort behandelen.

Parallelle programmatuur Het doel van parallelle programmatuur is prestatieverbetering door meerdere computers tegelijkertijd te gebruiken. Er zijn grofweg twee soorten computerarchitectuur waarop men parallelle programmatuur kan draaien: multiprocessors en gespreide systemen. Een *multiprocessor* is een verzameling computers die een gemeenschappelijk geheugen delen; ze zijn gemakkelijk te programmeren, maar moeilijk zo te bouwen dat ze voldoende efficiënt zijn. Gespreide systemen zijn moeilijk te programmeren maar gemakkelijk te bouwen. In hoofdstuk 4 beschrijven we hoe groepscommunicatie gebruikt kan worden om een model te implementeren dat de voordelen van beide combineert. Dit model, het *shared-data-object-model*, geeft de programmeur de illusie dat hij of zij op een systeem met gemeenschappelijk geheugen werkt, terwijl er in werkelijkheid geen fysiek gemeenschappelijk geheugen is. Het kernprobleem bij het implementeren van het shared-data-object-model is het efficiënt nabootsen van gemeenschappelijk geheugen door het versturen van zo weinig mogelijk

boodschappen, aangezien deze laatste duur zijn vergeleken met het lezen van een waarde uit het lokale geheugen.

In hoofdstuk 4 tonen we twee implementaties van het shared-data-object-model. Bij de eenvoudigste implementatie wordt een gemeenschappelijk object in het lokale geheugen van alle computers gerepliceerd. Leesoperaties op een gemeenschappelijk object kunnen nu eenvoudig uitgevoerd worden door direct het lokale geheugen te lezen, terwijl schrijfoperaties met behulp van groepscommunicatie naar alle computers gestuurd worden die het parallelle programma uitvoeren. Aangezien de boodschappen naar de groep betrouwbaar en totaal geordend verstuurd worden, blijven de copieën van een gemeenschappelijk object consistent. Deze implementatie bevoordeelt leesoperaties boven schrijfoperaties, wat nuttig is omdat vele parallelle programma's een hoge lees/schrijf-ratio hebben.

Repliceren van gemeenschappelijke objecten is in het algemeen maar niet altijd een goede strategie. De andere implementatie van het shared-data-object-model deelt gemeenschappelijke objecten in in twee klassen: objecten die gerepliceerd moeten worden (lees/schrijf ratio ≥ 1) en objecten die niet gerepliceerd behoeven te worden (lees/schrijf ratio < 1) en probeert objecten die niet gerepliceerd hoeven te worden op die computer te plaatsen die de meeste operaties op dat object uitvoert. De processen die op die computer draaien, kunnen de operaties nu uitvoeren door rechtstreeks het lokale geheugen te gebruiken in plaats van door het versturen van boodschappen. Processen die geen copie hebben, moeten een boodschap sturen naar de computer die het object beheert.

Met behulp van drie parallelle programma's, elk met een andere structuur, hebben we aangetoond dat beide implementaties goede prestaties leveren. De eerste implementatie is het meest efficiënt voor programmatuur die een gemeenschappelijk object vaker leest dan schrijft of die een gemeenschappelijk object als een broadcast-communicatiekanaal gebruikt, terwijl de tweede implementatie qua efficiëntie de voorkeer verdient voor programmatuur die een gemeenschappelijk object als punt-naar-punt communicatiekanaal gebruikt.

Foutbestendige programmatuur Hoofdstuk 5 richt zich op twee soorten foutbestendige programmatuur: parallelle programmatuur en ‘‘normale’’ foutbestendige programmatuur. We bespreken elke soort apart.

In parallelle programmatuur wordt foutbestendigheid vaak genegeerd. Echter, het belangrijkste doel van een parallel programma is het verkrijgen van hogere efficiëntie door het gebruiken van meerdere computers tegelijkertijd. Maar door meerdere computers te gebruiken, stijgt ook de kans dat het programma niet succesvol eindigt doordat één van de computers faalt. Opnieuw starten na zo'n fout is geen bevredigende oplossing, aangezien parallelle programma's vaak lang draaien.

Het eerste deel van hoofdstuk 5 laat zien dat de eerste implementatie van het shared-data-object-model gemakkelijk en goedkoop foutbestendig gemaakt kan worden. Dit gebeurt door periodiek de totale toestand van het programma op schijf op te

slaan en na een fout terug te gaan naar de laatst opgeslagen toestand. Deze aanpak werkt goed voor programma's die invoer lezen, onafgebroken rekenen en dan uitvoer produceren, maar werkt niet goed voor programma's die tussendoor invoer lezen of uitvoer produceren, omdat het programma teruggerold kan worden, maar de invoer en uitvoer niet.

De sleutel tot het foutbestendig maken van parallelle programmatuur is groepscommunicatie. Aangezien de eerste implementatie van het shared-data-object-model alleen totaal geordende groepscommunicatie gebruikt, is het mogelijk om de toestand van het programma, die immers opgebouwd is uit de toestand van meerdere processen, op te slaan zonder alle processen in het programma eerst stop te zetten. Periodiek wordt er een boodschap naar de groep gestuurd die aangeeft dat de toestand opgeslagen moet worden, bijvoorbeeld door deze naar schijf weg te schrijven. Aangezien alle processen deze boodschap in dezelfde volgorde ontvangen, kunnen ze na ontvangst meteen de toestand wegschrijven en vervolgens doorgaan met rekenen. De kosten van foutbestendigheid zijn dus gelijk aan de kosten van het sturen van een boodschap en het wegschrijven van de toestand.

Een aantrekkelijke eigenschap van deze aanpak is dat de lengte van het interval tussen het opslaan van toestanden wordt aangegeven door de programmeur, waardoor deze de kosten van de foutbestendigheid zelf kan bepalen. Een kort interval maakt foutbestendigheid relatief duur, maar de kosten voor het herstarten na een fout zijn laag (er hoeft maar weinig werk opnieuw gedaan te worden). Door een lang interval te kiezen, kunnen de kosten voor foutbestendigheid laag gehouden worden, maar als er een fout optreedt, moet er veel werk opnieuw uitgevoerd worden.

In ''normale'' foutbestendige programmatuur kan groepscommunicatie ook heel nuttig zijn. Om dit aan te tonen bespreken we in het tweede deel van hoofdstuk 5 twee implementaties van een foutbestendig gegevensopslagsysteem: de éne werkt met punt-naar-punt communicatie en de andere met groepscommunicatie. Het gekozen programma is een voorbeeld van programmatuur die foutbestendigheid bereikt door gegevens te repliceren op meerdere plaatsen.

Bij het vergelijken van de genoemde implementaties zijn twee aspecten van belang: hoe efficiënt het programma is en hoe ingewikkeld de implementatie is. De implementatie met behulp van groepscommunicatie is eenvoudiger en efficiënter dan de implementatie met behulp van punt-naar-punt communicatie, terwijl de implementatie gebaseerd op groepscommunicatie zelfs foutbestendiger is.

Conclusie

Door te laten zien dat betrouwbare en totaal geordende groepscommunicatie efficiënt geïmplementeerd kan worden en door te laten zien dat groepscommunicatie het programmeren van gespreide programmatuur vereenvoudigt, is een essentieel deel van het bewijs geleverd voor de stelling dat gespreide systemen groepscommunicatie behoren te verschaffen.