UNIVERSITY OF CALIFORNIA
SANTA CRUZ

Weak-consistency group communication and membership

A dissertation submitted in partial satisfaction
of the requirements for the degree of
DOCTOR OF PHILOSOPHY
in
COMPUTER AND INFORMATION SCIENCES
by

Richard Andrew Golding

December 1992

The dissertation of Richard Andrew Golding is approved:

Prof. Darrell Long

Prof. Charles McDowell

Dr. Kim Taylor

Dr. John Wilkes

Dean of Graduate Studies and Research

Contents

List of Figures

List of Tables

Weak-consistency group communication and membership

Richard Andrew Golding

ABSTRACT

Many distributed systems for wide-area networks can be built conveniently, and operate efficiently and correctly, using a *weak consistency group communication* mechanism. This mechanism organizes a set of *principals* into a single logical entity, and provides methods to multicast messages to the members. A weak consistency distributed system allows the principals in the group to differ on the value of shared state at any given instant, as long as they will eventually converge to a single, consistent value. A group containing many principals and using weak consistency can provide the reliability, performance, and scalability necessary for wide-area systems.

I have developed a framework for constructing group communication systems, for classifying existing distributed system tools, and for constructing and reasoning about a particular group communication model. It has four components: message delivery, message ordering, group membership, and the application. Each component may have a different implementation, so that the group mechanism can be tailored to application requirements.

The framework supports a new message delivery protocol, called *timestamped anti-entropy,* which provides reliable, eventual message delivery; is efficient; and tolerates most transient processor and network failures. It can be combined with message ordering implementations that provide ordering guarantees ranging from unordered to total, causal delivery. A new group membership protocol completes the set, providing temporarily inconsistent membership views resilient to up to k simultaneous principal failures.

The Refdbms distributed bibliographic database system, which has been constructed using this framework, is used as an example. Refdbms databases can be replicated on many different sites, using the group communication system described here.

Acknowledgments

Several people have assisted in this research. Kim Taylor, at UC Santa Cruz, assisted with the proofs; she was supported in part by NSF grant CCR–9111132. Darrell Long assisted with some of the performance evaluation; he was supported in part by the National Science Foundation under Grant NSF CCR–9111220, by the Institute for Scientific Computing Research at Lawrence Livermore National Laboratory, and by the Office of Naval Research under grant N00014–92–J–1807. John Wilkes, at Hewlett-Packard Laboratories, provided additional suggestions and critique, particularly during the early exploration of the ideas.

The Refdbms 3.0 system was derived from the original *refdbms* system, written by John Wilkes in the Concurrent Systems Project at Hewlett-Packard Laboratories. Development of version 3 was aided by Eric Allman and the Mammoth Project at UC Berkeley under National Science Foundation Infrastructure Grant CDA–8722788. George Neville-Neil wrote the X11 user interface for version 3.0, and the alpha testers have provided important feedback in improving the system.

This research was supported by several different organizations. Early portions of this work were supported by the Concurrent Systems Project at Hewlett-Packard Laboratories, and by a University of California Seed Grant. The Santa Cruz Operation provided me with a one-year graduate fellowship.

Some simulation results were obtained with the aid of SIMSCRIPT II.5, a simulation language developed and supported by CACI Products Company of La Jolla, California.

Alan Emtage and Peter Deutsch, of Bunyip Information Systems and McGill University helped me understand the properties that wide-area information services need. Calton Pu, of Columbia University, encouraged me to work on the classification approach. Daniel Barbará, at the Matsushita Information Technology Laboratory, encouraged my investigation of weak consistency and information services.

The other graduate students in the Concurrent Systems Laboratory provided encouragement and support, most particularly Dean Long and William Osser.

My committee has been helpful in refining my ideas from a confused pudding of “things I’d like to look at” and “interesting directions” into a coherent whole. John Wilkes taught me how to write; Kim Taylor taught me how to prove correctness; Charlie McDowell kept me honest; and Darrell Long, my advisor, made me quantify my claims.

Three people have provided the love and support I needed to finish a work this size in altogether too short a time: Alan Emtage, Craig Cruz, and my partner, George Neville-Neil. These three have put up with my fears, grouchiness, and elation.

This dissertation is dedicated to the memory of my grandfather, Harry Lawrence Golding (1908–69). I think he would have liked it.

Chapter 1
Introduction

Most systems to date that operate across wide-area networks have been developed without a consistent set of tools for reasoning about or organizing their structure. As a result, wide-area systems are viewed as difficult to write, and ensuring their good performance is a black art.

My thesis is that many wide-area distributed systems can be built conveniently, and can operate efficiently and correctly, using the weak consistency group communication approach presented in this dissertation. In it, a set of principals are organized into a group, which applications can communicate with as if it were a single logical entity (Figure 1.1). The group communication mechanism is decomposed into a framework [Campbell92] of well-defined components and interfaces. The implementations of each component can be customized to meet application requirements. In particular, the state kept by the principals can be *weakly consistent,* meaning that the copies are allowed to diverge temporarily, as long as they will eventually come to agreement.

The framework provides a way to construct the group communication mechanism. The mechanism provides a *group multicast* service, allowing a principal to send a message to every group member, and a *group membership* service, allowing principals to join and leave the group. Each component can be implemented using one of many different protocols, providing different levels of service. For example, the component that sends and receives messages over the network will provide one of several message reliability guarantees. The framework approach allows code to be reused between applications, and ensures that the group communication semantics closely match application requirements.

This approach is useful for reasoning about distributed systems. The semantics of each component can be used to classify existing distributed systems. For example, systems can be classified

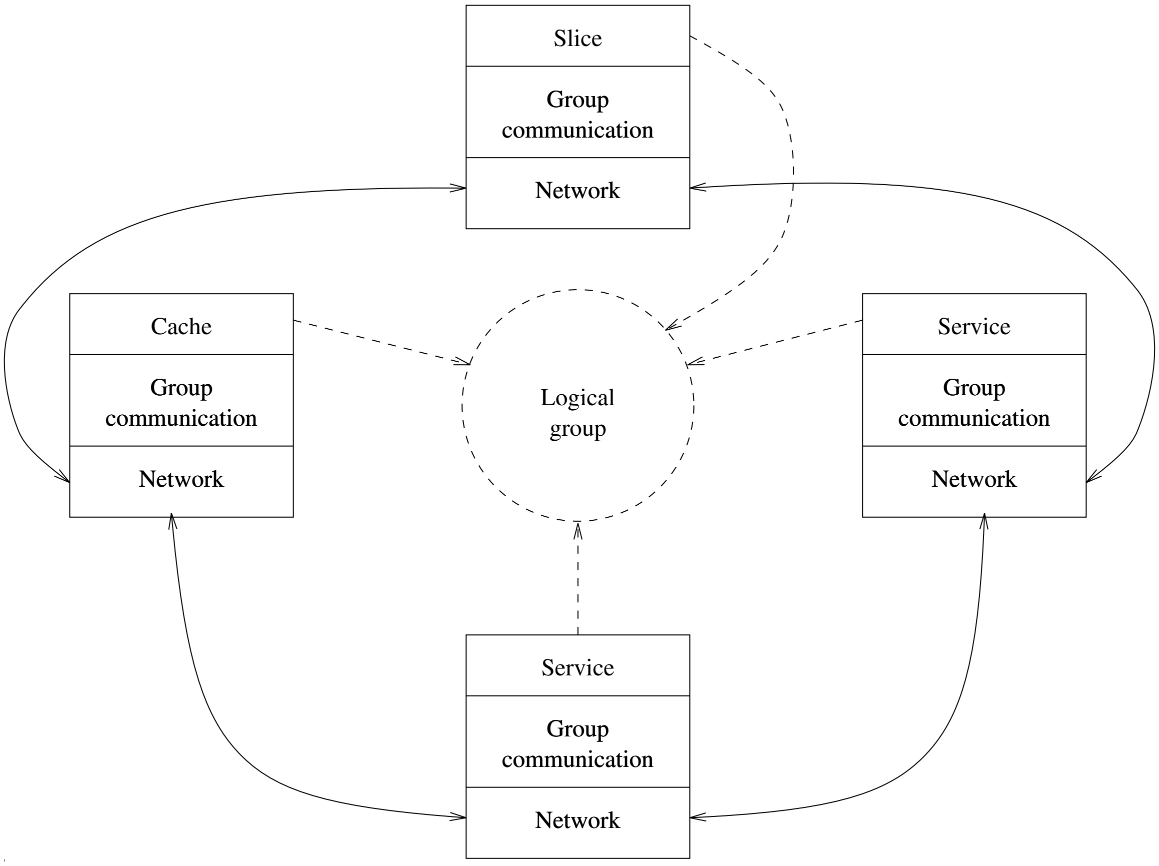


FIGURE 1.1: Overall system architecture. A wide-area system consists of a group of principals. Members and clients logically communicate with the group, and the group communication mechanism coordinates the communication with each member.

---

by the reliability of their message delivery mechanism. The correctness, performance, and fault tolerance of each component can be evaluated separately.

Using this framework, I have developed protocols that provide weak consistency, and investigated their correctness and performance. Weak consistency is provided by delivering messages *reliably* and *eventually.* Reliable delivery ensures that every principal will eventually observe any group message, while eventual delivery allows messages to be delayed while systems are not functioning or are disconnected from the network. A group membership protocol complements the message delivery protocol.

The protocols have been used in the Refdbms 3.0 distributed bibliographic database system. This prototype system indicates that the framework is appropriate for building wide-area systems, and that the weak consistency protocols can be built simply.

1.1 Requirements

Wide-area systems must take network behavior and user expectations into account. These include the scale and reliability of the network, mobile computing systems, and application availability.

Two hosts on an Ethernet can exchange a pair of datagram packets in a few milliseconds, while hosts on different continents can require many hundreds of milliseconds. Packet loss rates of 40% are common, and can go much higher [Golding91b]. This argues for a system taking advantage of *locality:* using nearby hosts when possible, and avoiding long-distance communication.

Despite this environment, users expect a service to behave as if it were being provided on a local system. Several studies have shown that people work best if response time is under one second for queries presenting new information, and much less for queries that provide additional details [Schatz90].

Users also expect to be able to make use of the service as long as their local system is functioning. A widely-used information system should be unavailable to any user at most a few minutes each year, as long as the user's local system is functioning. A recent study of host reliability [Long91] shows that most hosts are available better than 90% of the time, and are continuously available for ten days on the average. My own research [Golding91b] has found that hosts within North America respond when polled about 90% of the time, indicating that long-term network failure is probably uncommon. This study also showed that communication between two hosts near each other was more reliable than between distant hosts.

There are many points in the Internet that can fail and partition the network; indeed, it is usually partitioned into several non-communicating subsets. The system therefore cannot assume that the principals that compose it will be able to communicate with each other all the time. The introduction of mobile computer systems exacerbates this problem, since they can be disconnected from the network for a long time, or may be "semi-connected" by an expensive low-bandwidth connection. Several researchers are investigating file systems that can tolerate disconnection [Kistler91, Heidemann92, Alonso90a].

The application architecture must also *scale* to the vast number of users that can access a widely available service. The Internet included more than a million hosts in July 1992 [Long91]; the potential user base was probably then in the several millions, and these numbers are increasing by about 30% every four to six months [Lottor92, Ganatra92]. Specialized services with limited audiences currently receive on the order of 10 000 queries per day (0.12 queries per second mean) [Emtage92b], while widely-used services such as library card catalogues can receive nearly 100 queries per second [Emtage92a].

1.2 Using replication

These requirements cannot be met without *replicating* parts of the system. A replicated system allows load to be shared by many replicas, improving availability and scalability. Clients use the service by contacting one replica. The service is available as long as the client can connect to at least one functioning replica. The replicas in turn communicate amongst themselves to coordinate the service.

Figure 1.2 shows how replicas might be placed in a simple internetwork. Portable systems include clients, and can possibly include replicas. When a portable system maintains a replica the service continues to be available even when the system has been disconnected from the network. The local replica does not receive updates made by other replicas until the system is reconnected to the network.

Clients can contact the nearest replica, improving communication locality. This reduces communication latency. It also decreases the load each communication imposes on the network by reducing the number of routers and communication links that must handle the messages. One approach is to place one replica in each geographic region or organization. Clients must be able to identify which replicas are nearby and maintain performance when nearby replicas fail; I have considered this problem elsewhere [Golding92b, Golding92c].

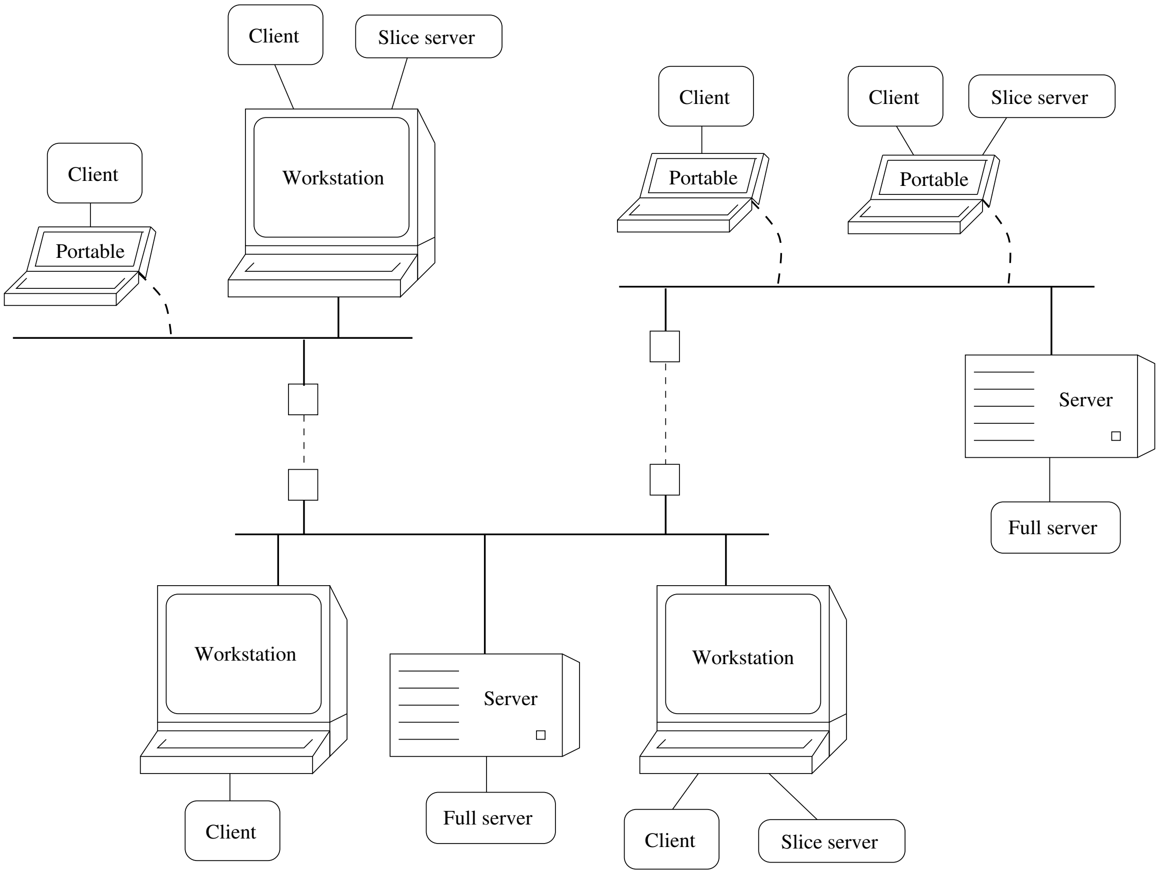


FIGURE 1.2: Placing replicas in an internetwork. Some local-area networks will have a nearby replica, while others must communicate with more distant replicas. Portable systems may include a "slice" replica that maintains a copy of part of the database. Properly-placed replicas that cache service information can improve performance.

1.3 Group communication mechanism

A *group communication mechanism* can be used to construct a replicated service. This mechanism (sometimes called a *distributed process group* mechanism) organizes a set of *principals* into a group [Birman87, Cheriton84]. The group acts as a single abstract entity.

The group supports two kinds of operations: *group multicast* to send messages to group members, and *group membership* to add or delete principals from the set of members. Applications apply group operations without concern for the principals that make up the group, and the group communication mechanism converts operations on the abstract group to communication with principals (Figure 1.3).

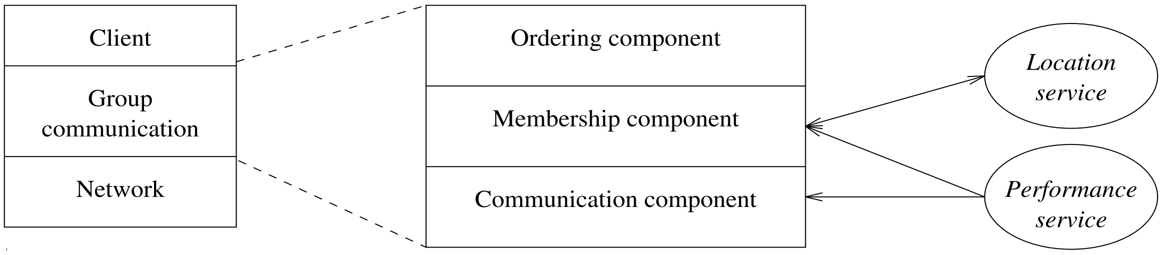


FIGURE 1.3: Components of a group communication mechanism. The group membership mechanism provides a group multicast protocol, which is implemented in the ordering and communication components, and a group membership protocol, which is implemented in the membership component. These mechanisms may make use of other outside services, including a location service or a performance prediction service.

The group multicast operation sends a message from one principal to every group member. The operation can be implemented using one of many different protocols. The implementation defines whether messages are delivered reliably or not, and how long delivery takes. Likewise, there are >many possible implementations of the group membership operations.

Both the multicast and membership implementations may rely on other network services. For example, the membership protocol might use a location service to learn what servers make up the group when a principal joins. The communication component might make use of a *performance prediction* service (Section 2.5) to improve performance.

1.4 Weak consistency

Each group member has a copy of *group state,* and uses the group communication mechanism to coordinate changes to it with the other members. The copies are *consistent* if they have the same value, while *inconsistent* copies can differ. The group multicast protocol determines the degree of consistency by providing guarantees on how reliably and quickly messages containing state changes will be delivered to group members.

Many existing group communication systems, among them Isis [Birman87, Birman91], Psync [Mishra89], Arjuna [Little90], and Lazy Replication [Ladin91], provide strong consistency guarantees, meaning that the system provides a multicast message service that ensures that every principal views every message in a strictly controlled order, and that no two principals can differ at any moment by more than a limited degree.

Other approaches provide intermediate guarantees. The *quasi-copy* approach allows specific bounds on the difference between copies [Alonso90b, Barbará90, Alonso90a], while epsilon serializability relaxes traditional serializability definitions to control the number of updates by which copies can differ [Pu91b, Pu91a].

In contrast, the *timestamped anti-entropy* group multicast protocols presented in this dissertation provide *eventual* or *weak consistency.* While multicast messages will be delivered to every group member, the time required is unbounded (though finite.) Thus, any two group members can hold different copies of group state at any instant, but eventually both members will receive the same set of state-change messages.

These weaker guarantees can have important performance and availability benefits as compared to strong consistency, particularly when considering wide-area and mobile systems. Strong consistency systems require expensive protocols, and perform poorly (or not at all) when communication is unreliable or when the network is partitioned. By contrast, weak consistency protocols can use fewer network packets, allow caching and delayed operation for mobile systems, and are not affected >by many forms of processor and network failure.

The timestamped anti-entropy protocol, like other weak-consistency protocols, achieves its fault-tolerance and efficiency by performing delayed communication between principals. Rather than multicasting a message right away, messages are placed in a queue and delivered later. Pairs of principals periodically contact each other to exchange the messages in their queues. This exchange is called an *anti-entropy session.* If a host is unavailable for some time, the principals that it hosts can perform exchanges when they begin functioning again. In this way the group communication mechanism hides host and network failures.

Many existing information systems, such as Usenet [Quarterman86] and the Xerox Clearinghouse system [Oppen81], use similar techniques. The work presented in this dissertation formalizes the weak consistency model and provides new mechanisms to make weak consistency group communication flexible, controllable, and robust.

1.5 The Refdbms 3.0 system

Refdbms 3.0 is a distributed bibliographic database system. Its implementation uses the weak-consistency protocols presented in this dissertation. I used its implementation to test many of the ideas presented here, and it will provide motivating examples through the next several chapters.

The Refdbms 3.0 system is based on the *refdbms* version 1 system that has been under development for several years at Hewlett-Packard Laboratories [Wilkes91]. That system emphasizes bibliographic information shared within a research group. Users can search databases by keywords, use references in TEX, locate copies of papers, and add, change, or delete references. I have extended it into a distributed, replicated database [Golding92a]. (Version 2 is an independent version, also based on the original.)

The extended system provides multiple databases distributed to widely dispersed sites. Databases can be specialized to particular topics, such as operating systems or an organization's technical reports. Each database can be replicated at several sites on the Internet, and users can create their own copy of interesting parts of the database. When a user enters a new reference in one copy, the reference is propagated to all other copies. The system also includes a simple mechanism for notifying users when interesting papers are entered into the database.

Refdbms stores references in a format similar to that used by refer [Lesk78, Tuthill83], as shown in Figure 1.4. Every reference has a *type,* such as TechReport or Article, and a unique, mnemonic *tag* like Lamport78a. Since these tags are determined by users and can potentially collide, the system internally uses a unique identifier consisting of a timestamp plus the address of the site that created the reference. References are stored in hashed and b-tree files using the BSD 4.4 libdb library, and are indexed both by tag and by keyword.

The weak-consistency framework in this dissertation was used to design and implement the new version of Refdbms. In general this has not affected the use of the system: users can do all of the same operations they could on the older centralized system. However, since replicas at different sites can have different contents while updates are propagating, users will occasionally see inconsistent information. This could be a problem when two authors at different sites are collaborating on a paper, or when one person tells another about an interesting reference they just found. Refdbms

```
%z Article (the type)
%K Lamport78a (the tag)
%A Leslie Lamport
%T Time, clocks, and the ordering of events in a distributed system
%J CACM.
%V 21
%N 7
%D 1978
%P 558 565
%x The concept of one event happening before another in a distributed
%x system is examined, and is shown to define a partial ordering of
%x the events. A distributed algorithm is given for synchronizing a
%x system of logical clocks which can be used to totally order the
%x events. The use of the total ordering is illustrated with a method
%x for solving synchronization problems. The algorithm is then
%x specialized for synchronizing physical clocks, and a bound is
%x derived on how far out of synchrony the clocks can become.
%k causal consistency, asynchrony, happens before
%k clock synchronization
```

FIGURE 1.4: An example reference.

resolves this problem by making potentially-inconsistent information available, but only if users ask for it. These problems are discussed further in Chapter 3.

1.6 Conventions in the text

When multiple citations are presented together, they are listed in order of decreasing importance or relevance. While this is not the usual practice, I have found it to be more useful than ordering them alphabetically. The references were maintained and formated using Refdbms.

Program fragments, user commands, and variable names are presented in a sans-serif face. Names of protocols are printed in a **bold** face. Most other names are presented in a standard Roman face.

1.7 Organization of the dissertation

In the next chapter I will define a number of terms and assumptions used in later chapters. In Chapter 3, I discuss group communication systems, and present a framework for constructing them and tailoring them to specific application requirements. Chapter 4 is a survey of existing group communication systems. I present the *timestamped anti-entropy* protocol in Chapter 5. That protocol guarantees reliable eventual message delivery, which is used in weak consistency group communication. Chapter 6 includes a group membership protocol that is a companion to timestamped anti-entropy. Chapter 7 investigates the performance of these protocols. Chapter 8 explores how the group communication framework can be extended to build sophisticated wide-area information systems. Finally, Chapters 9 and 10 present topics for future research and my conclusions on this work.

Chapter 2
Terms and definitions

In this chapter I will define a number of terms and assumptions used throughout this dissertation.

The term *protocol* is used throughout to mean a computational procedure that is performed by two or more separate principals and coordinated by messages passed over a network. This is distinct from an *algorithm,* which is more generally any computational procedure.

2.1 Consensus

The problem of reaching consistency between copies of group state is a form of *distributed consensus* [Turek92, Fischer85]. Consensus has been studied extensively, and it is well known that specific conditions on processors and the network are required for it to be possible. Some of these conditions are listed in Table 2.1.

The Internet and the hosts on it cannot formally achieve consensus because they cannot meet the necessary conditions. However, in practice the Internet closely approximates several of the conditions. These approximations will be presented briefly here, and discussed more completely in later sections of this chapter.

Hosts on the Internet approximate *synchronous processors:* there is some bound on the differences between the rates at which hosts operate. In practice this means there is a bound on the time required for any host to complete any protocol step.

The Internet provides at worst *unbounded communication latency.* In practice a bound can be established on the latency between two hosts when they are able to communicate, but network failures can delay messages for arbitrarily long periods.

TABLE 2.1: Conditions under which consensus is possible. Adapted from Turek and Shasha [Turek92].

		Point-to-point	Broadcast transmission		Point-to-point
		Unordered messages		Ordered messages	
Processors	Communication				
Asynchronous	Unbounded	No	No	Yes	No
Asynchronous	Bounded	No	No	Yes	No
Synchronous	Bounded	Yes	Yes	Yes	Yes
Synchronous	Unbounded	No	No	Yes	Yes

Finally, principals *do not fail.* A principal may appear to stop for some time while the host on which it runs is out of service. However, when the host has recovered the principal will recreate its state from stable storage and resume operation.

2.2 Principals

Principals are the entities that participate in group operations. Other terms such as site, replica, process, and server might seem appropriate, but are well-defined in other contexts and have inappropriate connotation.

Principals survive temporary failures and host crashes. They have some form of stable storage to record information that must survive failure. They also have volatile storage that is lost on failure. Both principals and hosts fail by stopping (also called crashing), so that spurious data are never transmitted on the network or written to stable storage. In practice a carefully implemented disk storage system can closely approximate this ideal [Gray86, Sullivan92, Seltzer90]. Many Unix network services, such as network file systems, name services, and mail routing behave in just this way: they are created afresh from data on disk every time a host recovers [Leffler89].

Principals have a mean time-to-failure (MTTF) much longer than the time required to perform certain protocols. Such principals can be constructed from less-reliable principals if stable storage is available. These assumptions eliminate pathological situations where principals recover, stay up for a very short time, then fail again. Studies of host reliability [Long91] indicate that most hosts function continuously for several days between crashes, while most protocols take at most a few minutes to complete.

Each principal has a unique identity, and principals that cease to exist do so for all time. It is not possible in the short term to distinguish between a slow principal and one that has exhibited a temporary failure and will soon recover. However, in the long term principals make progress at a bounded rate. When functioning, no principal is infinitely fast, and any temporary failure is recovered within a bounded interval.

2.3 Time and clocks

Throughout this dissertation, the word *time* refers to the time that might be measured by an external observer, as opposed to any internal or virtual time measure. When an event is said to happen *eventually* after time t, the probability that the event will not happen during the period $(t, t + \delta]$ goes to zero as $\delta \to \infty$.

Clocks provide monotonically increasing time-like measures within the system. Clocks progress at the same rate as real time in the long term; however, over short intervals clocks may advance at uneven rates.

Every principal p has access to some clock, denoted clock(p); the value of the clock at time t is clock(p, t). This clock allows every important event performed by the principal to be assigned a distinct timestamp. The clocks can be loosely synchronized, meaning that the clocks at any two principals differ by at most a constant ϵ :

$$(\forall t)(\forall p, q \in P)|\text{clock}(p, t) - \text{clock}(q, t)| < \epsilon.$$

This assumption is not required for most of the results in this dissertation. The text indicates any place where loose clock synchrony is assumed. Clock synchronization is a well-studied problem [Lamport78, Cristian89], and the NTP protocol currently provides this degree of synchrony on the Internet [Mills88].

In my experience most host clocks are within half an hour of the correct time, indicating a maximum ϵ of an hour. Hosts that synchronize using NTP are much more accurate, and an ϵ of about a minute appears sufficient.

2.4 Network

Hosts are connected by a network, and communicate using messages. In the short term, message transmission latency between two functioning hosts is bounded [Golding92b]. This assumption is required for standard Internet protocols such as TCP [Postel80, Comer88]. In the long term, temporary host and network failures, coupled with message retry, make message transmission latency finite but unbounded.

The network includes both the physical communication media and the low-level protocols that use it. For the Internet, this includes the long-distance backbone links, local-area networks, and the IP communication protocols.

The communication network does not always deliver messages in FIFO order, and it may lose or duplicate messages from time to time. It does not spontaneously create messages.[1]

Networks have both a physical topology and a logical topology. The physical topology is determined by connections between physical components, and many parts are often tree-like in structure where a single failure can disconnect, or partition, the network. The logical topology of the open Internet is a completely-connected graph, because the IP protocols hide the physical topology to allow every host to communicate with every other host. However, I make a weaker assumption: the network is connected, but it need not be complete. Since many organizations choose to protect their internal networks from the rest of the Internet, in practice the logical topology of the Internet is composed of a number of completely-connected subcomponents. A host has a set of *neighbors* in the logical network with which it can communicate.

No part of the network fails permanently, though temporary partitions can occur. The network need never be free of partitions, as long as any principal can eventually send a message to any other principal on a neighboring host if it continually tries to send until it receives an acknowledgment. This is a much weaker assumption than requiring periods when the network is free of partitions. My studies of message reliability on the Internet [Golding92b] suggest that the probability that the

[1] Spurious packets can occur on the Internet; however, it is unlikely that they would fall into an existing TCP conversation and have a valid checksum.

Internet is ever free of partitions is effectively zero, and the advent of portable computing systems ensures that there will ever be a time when all systems are connected and functioning.

Semi-partitions are possible, where only a low-bandwidth connection is available. For example, a mobile system could be connected through a low-bandwidth cellular modem or a noisy telephone line.

2.5 Network services

As mentioned earlier, clients must be able to identify the group members that are near them. This presumes the existence of two services: a *name* or *location service,* which identifies the principals in a group, and a *performance prediction service* that orders principals by locality.

The location service might, for example, map a service name into a set of server addresses. This service might be implemented using the current DNS, or by a more advanced system [Bowman90, Deutsch92]. Indeed, it can be implemented using weak consistency, as in the Xerox Clearinghouse system [Oppen81]. The service must always provide some way of locating at least one current group member, as long as the group still exists. I will discuss some related issues in Chapter 8.

The performance prediction service provides a way to select from the principals based on expected communication performance. Expected performance is based on a prediction of communication latency, failure, and bandwidth. If an operation requires that only a small amount of information be moved between sites, message and processing latency will dominate performance. If large amounts of information must be transferred, then bandwidth will dominate. The prediction should be biased by the probability that the client can communicate with the member. Concurrent with my work on group communication, I have begun investigating the problems of performance prediction [Golding91b, Golding92b] and of using these predictions in the **quorum multicast** protocol [Golding91a, Golding92d]. Preliminary results suggest that significant performance improvements can be achieved using simple prediction strategies.

Chapter 3
A framework for group communication systems

A wide-area system can include a large number of principals running at different sites in the network. In Chapter 1 I proposed using a group communication mechanism to coordinate the activities of these principals. The mechanism must be flexible, so that it can be adapted to the needs of an application. It should also provide a structure that can be used to reason about a system, and to re-use code between systems.

A *framework* is an object-oriented description of the components that make up a system, and the interfaces between them. It generalizes concepts such as layered design, often used in specifying network protocols [Tanenbaum81], and structured design [PageJones88]. It is related to the Object-Oriented Design methodology [Rumbaugh91]. The Choices object-oriented operating system provides frameworks for process management, virtual memory, storage, and other services [Campbell92, Islam92].

Each principal that is a member of a group will include an instance of the group communication framework. The framework defines components, which abstractly document the essential semantics of the system and can be viewed as abstract object classes. Concrete classes specialize these abstract classes by providing an implementation of the component. An instance of the framework consists of various objects instantiated from the concrete classes.

A framework is useful both as a tool to design components, and as a method for sharing design and coding effort between applications. In this chapter I will present a framework for constructing a group communication mechanism.

3.1 The framework

The group communication framework has four components, as shown in Figure 3.1: *application,* message *delivery,* message *ordering,* and *group membership* components. They communicate through three shared data structures: a message log, message summary information, and a group view. A principal includes one instance of each component and data structure.

The *message delivery component* implements a multicast communication service that exchanges messages with other principals. It decodes incoming messages and writes them to the message log, from which they will be delivered to the application or group membership component. It also maintains summary information of the messages sent and received that can be used by the message ordering component. The message delivery component determines whether the group communication system provides weak or strong consistency, by providing eventual or immediate message delivery.

The *group membership component* maintains a set of the principals that are in the group. The set is called the local *view* of the group. When the set changes, this component communicates with the group components at other principals according to a group membership protocol. The protocol ensures a degree of consistency between group views. The communication consists of messages sent through the message delivery component.

The network and the message delivery component can reorder messages arbitrarily. The *message ordering component* processes the stream of incoming messages to ensure they are presented to the application according to some ordering. This step may require delaying some messages until the ordering component can correctly establish the order. To ensure this is possible, the ordering component also processes outgoing messages so that the ordering components at other principals will have enough information to properly order messages, usually by adding a header to each message.

The *application* manages group state. It might receive requests from clients outside the group, and translate those requests into group messages. The message would be given to the message ordering component, which would add a header containing ordering information. The message would then be stored in the log until the message delivery component sent it to the other principals.

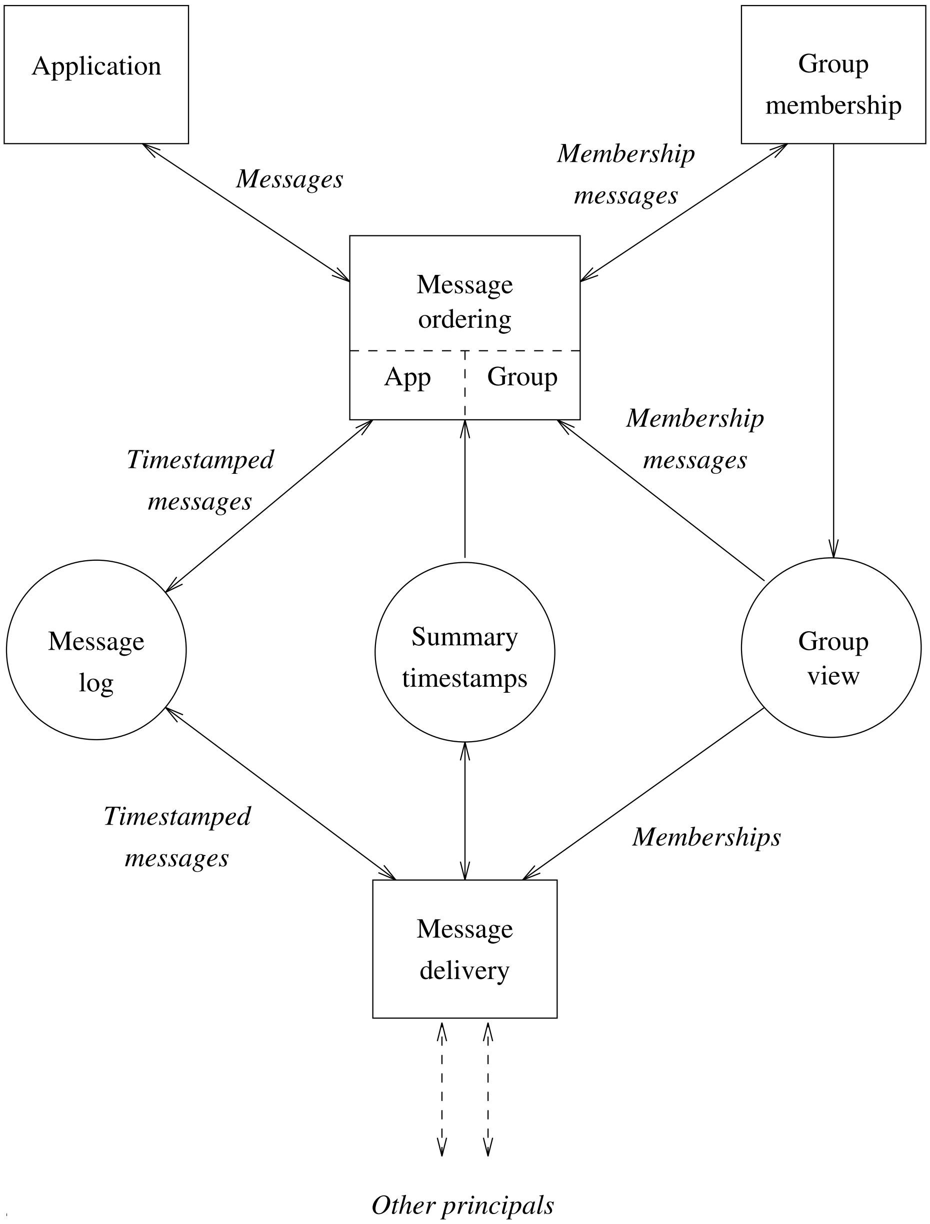


FIGURE 3.1: A framework for constructing a group communication system. Each principal in the group includes an instance of this framework, in the form of objects instantiated from concrete implementation classes.

At the other principal, the message would be received by the message delivery component and written to the message log. Some time later, enough information would be available so the ordering component could deliver the message to the application. The application component in this second principal would then act on the message and change its copy of the group state.

In the following sections I will discuss how this framework has been used to build two different wide-area services: the Refdbms bibliographic database [Golding92a] and the Tattler distributed reliability monitor [Long92]. I then detail each component and the implementations used by the two applications.

3.2 The application

The application component maintains the principal's copy of group state. The state has a logical *data model,* whether or not the principals actually store the data. The data model defines of the data to be shared, the operations to be performed on that data, and correctness constraints that must be maintained. The model determines what guarantees must be provided by the other components of the framework, and therefore what implementations can be used for them.

When a principal needs to perform an operation that effects a change to the group state, it encodes the operation in a message that is sent to the group. When it receives the message back, it performs the operation. When a principal is to execute an operation that does not change group state, it might be able to perform the operation using only local information, or it may need to send the operation to the group.

Some operations can tolerate inconsistent or out-of-date information. For example, updating a host address in a distributed name service does not require knowing the previous address, and it is not necessary for every replica in the service to observe the change immediately as long the change is propagated without too much delay. If every operation on the group state can tolerate inconsistency, then the message delivery component can be implemented with a protocol that provides weak consistency.

The operations allowed on the data can dictate a particular message ordering. If all operations are commutative, that is, if they can be applied in any order with the same net result, the message ordering component need not impose an order on the messages specifying the operations. It is more likely that operations will be order-dependent, in which case a total message order will ensure that every principal computes the same result for each operation.

If operations are order-dependent and messages are delivered eventually, the application will need to provide mechanisms for detecting and resolving conflicting messages. For example, one principal could send a message changing the state to one value, and another could concurrently send a message changing it to a different value. Local-area distributed systems can use locking mechanisms to avoid conflicts, but many wide-area applications cannot wait for a global locking operation before performing an update. Instead, principals make optimistic updates that must be checked before they are applied to the database. A message ordering implementation that delivers messages in a total order can provide a basis for consistent conflict detection.

Some applications require that the data contain unique identifiers. Unique identifiers are a common source of update collisions in weakly consistent systems, because different principals can use the same identifier in different ways. In some cases identifiers can be generated internally, but in other cases they must be provided by the user. Their presence can also determine whether two groups can merge their state.

The shared data may include explicit version or timestamp information. If they do, it may be possible to resolve update conflicts without requiring strict message orderings, and the ordering component may not need to append timestamp information to messages.

3.2.1 The Refdbms application

As discussed in Chapter 1, the Refdbms 3.0 system implements a distributed bibliographic database. A Refdbms database consists of a set of references, each with an internal *unique identifier* and a *tag* like `Smith91` that humans can use to name a reference. At all times the internal identifier is guaranteed to be unique. The tag *should* be unique, but this is not guaranteed for newly-added references until all sites holding a replica of the database can observe and resolve conflicting updates. The references are indexed by the tag and by an inverted index of content keywords.

Three operations can update the database: adding, changing, and deleting references. The update operations are neither commutative nor idempotent, meaning that every update operation must be performed exactly once, and in exactly the same order by every principal, if the databases

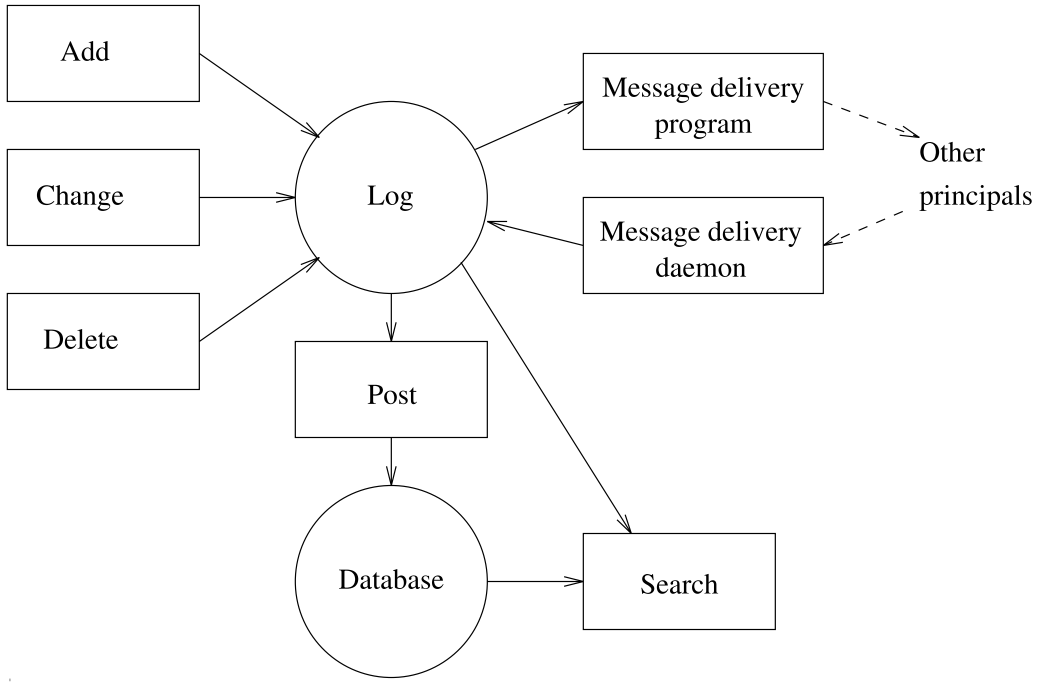


FIGURE 3.2: Structure of a Refdbms principal. The system uses reliable eventual delivery, implemented in the message delivery program and daemon, and total message ordering, implemented in the posting program.

are to reach agreement. This suggests that a message delivery component should deliver update messages in a total order, and that messages should be delivered reliably.

Users at different sites can submit conflicting updates. There are three sources of conflict: adding two different references with the same tag; changing one reference in two different ways; or deleting a reference then submitting another operation on it to a different principal. The basic mechanism for handling conflicts is to process update messages in the same order at every principal. I will discuss how conflicts are resolved in more detail in Section 3.2.3.

Users can also search for references. Searches need not return completely current information, as long as a search will eventually reflect any update. This implies that eventual message delivery is acceptable in the message delivery component.

Refdbms is implemented as a set of programs that communicate over the Internet using TCP (see Figure 3.2). Users can submit operations, which are written as messages to a log. From time to time the message delivery program propagates these messages to another replica by connecting to a daemon there, which in turn writes the update message to its log. Group membership changes are exchanged at the same time. The message delivery program and daemon together form the message delivery and group membership components. The message ordering component is contained in a posting program that periodically determines what updates can be delivered to the database.

3.2.2 The Tattler system

The Tattler system is a distributed availability monitor for the Internet [Long92], built by Long and Sriram. It monitors a set of Internet hosts, measuring how often they are rebooted and what fraction of the time they are available. The measurements are taken from several different network sites to minimize the effect of network failure on the results, and to make the sampling mechanism very reliable.

Each measurement site runs a *tattler,* which samples host uptimes and shares these measurements with other tattlers. Collectively the tattlers maintain a list of hosts to monitor and collect statistics on them. A record of the form $\langle$*host address, poll method, poll interval*$\rangle$ is kept for each host. The client interface allows hosts to be added or deleted from this list. The recorded statistics are stored in a database, which stores tuples of the form $\langle$*host address, boot time, sample time*$\rangle$.

Only one operation updates a Tattler database: merging a set of samples. Each sample represents an interval when the host was known to be available. A sample that is being merged into a database will either be disjoint from every other sample recorded for the same host, or it will overlap with another sample. If it overlaps, the two samples are combined. Otherwise, the host has been rebooted and a new interval has begun.

Each time a tattler obtains a new sample, it logically multicasts the sample to other tattlers. Sample merging is commutative and idempotent, so message ordering is unimportant as long as messages are delivered reliably. However, unlike Refdbms, the Tattler does not explicitly implement a message log. The database contains all the information that would be maintained in the message log, so the implementations of the message ordering and delivery components can work directly from the database.

Each tattler is composed of four parts: a *client interface,* a *polling daemon,* a *data base daemon,* and a *tattler daemon.* Figure 3.3 shows this structure. The *polling daemon* produces sample

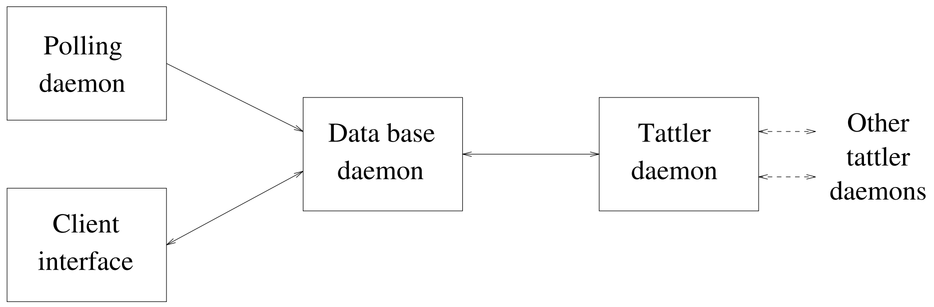


FIGURE 3.3: Structure of a Tattler.

observations. It takes samples at a specified rate, and can be requested to start or stop sampling using the *client interface.* The *data base daemon* provides stable storage for sample observations (from the polling daemon), and meta-data from the client interface and the tattler daemon. All of the group communication components are implemented in the *tattler daemon,* which exchanges samples, host lists, and membership information between tattler sites using a reliable, eventual delivery protocol.

3.2.3 Handling update collisions

Wide-area applications generally perform *optimistic* updates to group state that may conflict with other updates because pessimistic conflict-prevention mechanisms involve expensive, consistent coordination steps. In some applications, such as the Tattler, optimism is not a problem since all operations are commutative and cannot conflict. Other applications, including Refdbms, define operations that can conflict, so these applications must provide mechanisms to detect and resolve conflicting updates. These applications can also provide mechanisms to make conflicts unlikely even when they cannot be prevented.

As noted earlier, there are three kinds of conflict in Refdbms: between two add operations, between two change operations, and between a deletion and any other update. Different techniques are used to detect, resolve, and avoid each kind of conflict. All of the techniques make use of messages being delivered in the same order at every principal.

Two newly-added references conflict if they have the same *tag*. Recall that tags are assigned by users and are supposed to be unique within a database, but this cannot be guaranteed when users at different sites add references independently. This kind of conflict is detected when the second add message is delivered to the application at each principal. The first reference will already have been added to the database. When Refdbms finds that the tag has already been used, it computes a new tag for the reference by adjusting a suffix on the tag: Smith90 would become Smith90a, and Jones90b would be changed to Jones90c. There is a limit of up to ten suffix characters, but it is most unlikely that there will be more than 26^{10} references from one author in one year. The update message can then be re-processed using the new tag.

There is one problem with this scheme: users may have submitted change or delete operations for the modified reference. These operations should not be associated with the tag of that reference, since it could change when the add operation is performed and the change would then be applied to the wrong reference. Instead, each reference is given an internal identifier composed of a host name and timestamp that is guaranteed always to be unique and is never modified. Change operations can then be associated with the correct reference, even if its tag has been modified.

Conflicting change operations in Refdbms are more complex. They are not explicitly detected or resolved; instead, change operations are simply applied in the same order by every principal. However, change operation messages only carry the *difference* the change is supposed to apply to the reference. In this way if one user corrects the spelling of an author's name while another user at a different principal independently adds keywords, both changes will eventually appear in the reference. Fields can be grouped together, and a separate policy is used for each group of fields. For example, a change to any author-related field will result in all author-related fields being overwritten, while location lines can be inserted or deleted individually. This technique reduces the probability that two change operations will conflict, even if they apply to the same reference.

Finally, deletion cancels any other operations. Change or delete operations delivered after a reference has been deleted are simply ignored.

3.3 Message delivery

The message delivery component fills the same function as the transport layer in the ISO layered network model [Tanenbaum81], in that it exchanges messages with other principals without inter-

TABLE 3.1: Possible message delivery reliability guarantees, from strongest to weakest.

Kind	Guarantee
Atomic	Message is either delivered to every group member, or to none. Message is aborted if any group member fails.
Reliable	Delivered to every functioning group member or to none, but failed members need not receive the message. If the sender fails, delivery is not guaranteed but may occur.
Quorum	Delivered to at least some fraction of the membership. If the sender fails, delivery is not guaranteed.
Best effort	Delivery attempted to every member, but none are guaranteed to receive the message.

preting message contents. In my group communication framework, it retrieves messages entered into a message log by other components and transmits them to other principals.

The delivery component provides guarantees on message *reliability* and *latency.* The reliability guarantee determines which principals must receive a copy of the message, and latency determines how long delivery will take.

There are several possible message reliability levels, ranging from *atomic* to *best effort,* as listed in Table 3.1. Reliable mechanisms generally require extra state at each principal and induce more message traffic than unreliable ones. They require the sender to retain a copy of the message in its message log so the message can be retransmitted if necessary, and they require receivers to acknowledge incoming messages. Best effort mechanisms need not keep a copy of the message.

Reliable delivery was used for both Refdbms and the Tattler. Reliable delivery is essential for Refdbms, because even a single lost message can cause some principal to miss an update and permanently diverge from the proper value. Reliability is less essential for the Tattler, because that system can recover from a lost message the next time two databases are merged.

Message *latency* complements reliability: it determines how long principals may have to wait to receive a message if it is delivered to them. There are two aspects to latency: when the delivery process begins, and when it ends. The process can either begin immediately, or messages can be queued for later delivery. Once started, delivery can complete in either a bounded time, or

TABLE 3.2: Possible message delivery latency guarantees.

Kind	Guarantee
Synchronous	Delivery begins immediately, and completes within a bounded time.
Interactive	Delivery begins immediately, but may require a finite but unbounded time.
Bounded	Messages may be queued or delayed, but delivery will complete within a bounded time.
Eventual	Messages may be queued or delayed, and may require a finite but unbounded time to deliver.

eventually. The four combinations are listed in Table 3.2. Other guarantees can be used that fall between the ones listed.

Eventual delivery was used in both systems because synchronous or interactive delivery can severely limit fault tolerance. In particular it makes the system less tolerant of network partitions and site failures. If messages can be delayed, they can be delivered after the network or system failure has been repaired. The Internet is essentially never without partitions, and mobile computers may spend a substantial fraction of the time disconnected.

Eventual delivery also allows the system to delay messages until inexpensive communication is available. This might mean waiting to transmit messages until evening when the network is less loaded. Some mobile systems spend long periods "semi-connected" through a low-bandwidth wireless link, and it may be more effective to wait to transmit messages until the system is reconnected to a higher-speed link.

While interactive delivery is not necessary, both Refdbms and the Tattler are most convenient when updates propagate quickly. The Tattler takes steps to increase the propagation rate on observing changes to group membership or the list of monitored hosts. This propagates important changes quickly, while ordinary updates are propagated normally.

Reliable eventual delivery provides weak consistency. Every update to group state is encoded in a message, which is delivered to every principal. While the message is being sent, some principals will have received the message while others will not. This inconsistency between principals is removed when delivery completes.

I have developed the *timestamped anti-entropy* protocol as one implementation of the message delivery component. It provides reliable eventual message delivery in wide-area distributed systems. Chapter 5 discusses this protocol in detail. It maintains a summary of the messages and acknowledgments it has received, and periodically exchanges batches of messages between pairs of principals. The summaries make the exchange efficient by allowing each principal to send only the messages the other has not yet received. As long as every principal periodically performs these exchanges, every message will eventually be delivered to every principal, thus providing reliable eventual delivery. It masks transient failures by periodically retrying message exchanges, making it ideal for for the Internet and mobile computing.

3.3.1 Propagating messages versus state

There are two models for storing and transmitting messages. In the first model, each message is entered into a message log, sent to other principals, and later applied to the group state by each principal. Alternately, it can be immediately applied to the group state and its *effects* can be logged and transmitted to other principals. Refdbms uses a message log, while the Tattler operates from the sample database.

Message logs are simple. Every update operation produces one update message, which is then sent to every group member. After the message arrives at other principals, its operation can be applied to the group state. The messages can be tagged with timestamp information so that any ordering is possible. The group state need not include any extra information to ensure that messages are applied in the right order.

Propagating effects rather than updates is more complex, but it can be a more efficient solution when eventual delivery is allowable. If a part of the group state is updated very often, the results of several operations can be collapsed into a single result. That result can be sent to other principals, rather than one message for each operation.

Since there are no messages, the group state must include ordering or timestamp information. In the Tattler each sample contains a timestamp. When updates are propagated from one principal to another, samples are exchanged and merged into the other database. In the Tattler, the sample timestamp is used just as a message timestamp would be. A sample in the database may reflect the merging of several measurements, so there can be fewer samples sent between principals than if each measurement were logged individually. Some systems that use state exchange can also tolerate some lost "messages" because the value can be obtained from a different principal in a later update exchange.

Unfortunately, many applications cannot use state exchange. It is impossible to construct global orderings on updates before they are applied to the database because updates are always applied immediately. In some distributed systems, such as Refdbms, update conflicts cannot be resolved without global message orderings. Other applications simply cannot maintain the necessary information in their group state.

Deleting items from the group state requires special consideration when message logs are not used. Deletion should be a stable property: once an item has been deleted, it should remain so forever. The item should not spontaneously reappear, though of course a new item with the same value could be added by an application. A record of the deletion must be maintained until the deletion has been observed by all principals, so that no principal can miss the operation and re-introduce the item to other principals. In the Clearinghouse these records were called *death certificates* [Demers88], while the Bloch-Daniels-Spector distributed dictionary algorithm [Bloch87] places timestamps on the gaps between items as well as on the items themselves. The Tattler uses the death certificate approach to track hosts that should no longer be polled.

3.4 Message ordering

The message ordering component is responsible for ensuring that messages are delivered to the application in a well-defined order. This order may be different from the order in which messages are received. For example, an application should receive updates to a database record after the

TABLE 3.3: Some popular message ordering guarantees.

Kind	Guarantee
Total, causal	The strongest ordering. Messages are delivered in the same order at every principal, and that order respects potential causal relations between messages.
Total, noncausal	Messages are delivered in the same order at every principal, but that order may not always respect potential causal relations.
Causal	Messages are delivered in an order that respects potential causal relations. If two messages could be causally related they are delivered in the same order at every principal. If they are not, they may be delivered in different orders.
FIFO	Messages from each principal will be delivered in order, but the messages from different principals may be interleaved in any order.
Unordered	Messages are delivered without regard for order.

message creating the record. Even if the messages were sent in the right order, they may be rearranged in transit and arrive at their destination in a different order.

Table 3.3 lists some of the most common message orderings. Some of these ensure that every principal delivers messages in the same order. An application can use this property to ensure that updates occur in the same order everywhere. Total causal ordering, for example, is provided by the Isis **ABCAST** protocol [Birman90]. Other orderings respect potential causality [Lamport78]. If there is any possibility that the contents of one message depend on the effects of another message, the ordering component guarantees that the other message will be delivered first. The Isis **CBCAST** protocol provides this ordering.

Message ordering guarantees can be limited just to message senders, to the principal group, or among all principals anywhere in the network. The FIFO guarantee is limited to message senders, and can be useful when each principal is sending out an independent stream of updates. Limiting consistency to the group is more common, but it is insufficient when the group must interact with other systems. Ladin's Lazy Replication mechanism [Ladin91] provides ways to order messages by any potential causal relation that can be detected by a principal, even those caused by activities outside the group. This guarantee is sometimes called *external causal consistency.*

A message ordering mechanism can be evaluated by the amount of extra information that must be appended to messages, by the amount of state each principal must maintain, and by the delay it imposes between receipt and delivery. Some causally-consistent mechanisms require that messages be tagged with a number of timestamps or message identifiers [Mishra89]. Total orderings can be accomplished with a per-principal counter or timestamp, though the resulting order will not be causal unless the counter or timestamp respects the *happens-before* relation [Lamport78].

3.4.1 Using message ordering

The Tattler does not require a message order because the operation of merging a sample into the database is not order-dependent. A sample represents a range of times that a host was known to be continuously available. When a new sample is to be processed, it will either overlap an existing sample, in which case the two will be combined, or it represents a new range.

The operations on a Refdbms database, on the other hand, are order-dependent. The value of a reference is the value of the last update applied to it. For two principals to record the same value for a reference, they must apply the same updates in the same order. For Refdbms, each update message is tagged with a timestamp from its originator's clock. Messages are then applied to the database in timestamp order. Recall that every principal has access to a local clock that is loosely synchronized with other clocks, and that every event can be marked with a unique timestamp from that clock.

This simple ordering is total, but it is not necessarily causal. Consider two principals A and B that can communicate with latency λ, where this latency is much smaller the difference ϵ between their clocks. A sends a message to B, which then sends another message. The second message is causally dependent upon the first message. However, if the clock at A is ahead of the clock at B, the first message will receive a timestamp greater than that of the second message.

Furthermore, this scheme is biased so that messages from principals whose clocks lag behind others will always be applied before those with faster-running clocks. As long as clocks are loosely synchronized to within some ϵ and the mean time between updates to a reference is larger than ϵ this bias has little effect.

Message ordering can require delaying updates for extended periods. Users, on the other hand, may need to use the results of an update immediately. Refdbms resolves this by making recent database changes available in a *pending image* of a reference. If there are conflicting updates, the contents of the pending image are only an approximation of the final reference. The pending image is removed when there are no update operations pending for the reference. The pending image can be retrieved by providing a tag of the form Smith92.pending. This allows citations of pending references to be embedded in a LaTeX document or sent to another user by electronic mail.

My performance evaluation in Chapter 7 shows that the simple total ordering used in Refdbms does not substantially delay message delivery on average. Messages are delayed at most by the maximum difference between clocks, plus the delay between receiving a message and receiving a greater or equal timestamp from every other group member. The difference between clocks is bounded by ϵ. The performance evaluation of the timestamped anti-entropy protocol shows that the variance in delivery latency is small, so that a message with one timestamp will arrive at about the same time as messages with similar timestamps from other principals.

3.5 Group membership

This component is responsible for maintaining the *view* of what principals make up the group. The group components at different principals exchange messages among themselves separate from the normal application messages. In some systems these group operation messages are processed by the message ordering component so that group changes are consistent with application messages. For example, every member can observe a principal joining the group at the same point in the message sequence. In the Refdbms and Tattler systems, however, this sort of consistency is not important because none of the operations on group state depend on the membership. Therefore group messages are delivered independent of application update messages.

There are two fundamentally different models for group membership, depending on whether group membership is based on a join/leave protocol or whether it is a process of discovering group members. The first mechanism is used in many existing systems, including Isis, Arjuna, most replication protocols, Refdbms, and the Tattler. The second mechanism has been proposed by Cristian [Cristian91], and works by discovering what principals believe they are members. It generally requires global broadcast, which is infeasible in networks the size of the Internet. This mechanism is not considered further.

Four operations can be performed on the membership view: hosts can *join, leave, fail,* and *recover.* The membership component incrementally builds up group membership as principals execute protocols for each of the four operations. Some implementations will also provide a protocol for merging two groups. A principal is considered to be a member if it has successfully executed the **join** protocol, and it remains so until it executes the **leave** protocol. This implies that there is some notion of the existence of a group independent of the principals that make it up. It might even be possible for a group to exist without any members.

Group state management is an essential part of the **join** and **leave** protocols. When a principal finishes executing the **join** protocol, it must have received a copy of the group state. This copy will be derived from the state maintained by one or more principals that were already group members. The new member also must receive a copy of the message log, message summaries, and group view. It is important that this state transfer not violate the message reliability and ordering guarantees provided by the other components. For example, the message log should include any update message that has not yet been applied to the group state, but which has been received by the principals that supplied the state. If it were otherwise, an update message might never be delivered to a new member and its copy of the group state could permanently diverge from other copies. Group membership mechanisms that allow groups to merge must also provide a way to merge the state of both groups.

The group membership component must provide a guarantee on its fault tolerance, which is measured by resilience to member failure. Since a principal can only contact member principals in its view, the group membership mechanism will fail if the only principal to know about another fails. The "knows-about" graph is correct if the transitive closure of all views is equal to the group membership. This ensures that every group member can contact every other group member, and that no other principals are in a view at any principal. To ensure that the graph stays correct after as many as k failures, the minimum vertex-cut of the graph between any two principals must be $k + 1$ or greater.

The group mechanism can also be evaluated by the amount of state each principal must maintain. Existing mechanisms range from centralized registries to fully distributed systems where every principal is a peer. Few mechanisms require more than $O(n)$ state in the number of group members, and some require only $\Theta(\log n)$.

3.5.1 Using group membership

I have developed two group membership mechanisms, one that only allows principals to join and leave, the other allowing group merges (Chapter 6). Both implementations maintain a tuple $\langle \mathit{principal}, \mathit{status}, \mathit{timestamp} \rangle$ for each principal in a view, requiring $\Theta(n)$ state at every principal. These protocols ensure fault-tolerance by requiring new members to obtain at least $k + 1$ *sponsors* among the membership, ensuring that the minimum vertex-cut is never too low. As long as fewer than k member principals fail, the graph will remain connected.

Refdbms uses the join-leave implementation because there is neither any need nor any sensible way to merge two databases. In Refdbms, a partitioned membership view graph will usually cause some updates never to be propagated from one partition to another, because the update will disappear once it has propagated everywhere in the partition. I balanced the expense of obtaining multiple sponsors against these problems, and decided that principals should obtain two sponsors when they join the group. This ensures that the view graph will always be resilient to at least one member failure.

The Tattler uses the implementation that allows group merges because its sampling operation is based on merging sample results. It allows principals to obtain just a single sponsor when joining because the effects of partitioning are not very severe. Tattlers can merge their sample databases after a partition has healed and no information will be lost. The only negative effect is that some principals in a membership view or hosts in a polling list that had been been deleted in one partition will reappear when the two are reconnected. This occurs because the record of deletion is maintained only until every principal in the partition has observed it.

3.6 Summary

The Refdbms and Tattler applications have been built and are running on the Internet. These represent two of the many kinds of wide-area applications that are likely to become available in the next several years. Both applications were constructed as a collection of principals organized into a weak-consistency principal group.

Weak consistency mechanisms provide fault tolerance and communication efficiency. The applications can tolerate extended host failure and can continue to operate when a principal becomes disconnected from others in the group. Messages can be delayed and batched to reduce the load the applications impose on the Internet. In particular I have found that the timestamped anti-entropy protocol provides a convenient message delivery mechanism that is flexible enough to support both applications.

I have developed a framework for constructing group communication mechanisms. The framework consists of an application, which defines the semantics of the state shared among the group; a message delivery component, which communicates messages from one member to another; a message ordering component, which assembles the incoming stream of messages into a coherent order and delivers them to the application; and a group membership component, which maintains a view of the membership. Each component can be implemented in many different ways, in order to match the semantics required by the application.

Eventually I expect this work to lead to a general-purpose toolkit, but even now it provides a structure for reasoning about and designing applications, and it is a valuable alternative to ad hoc application construction. Some modular architecture of this sort is necessary if wide-area distributed applications are to become common, efficient, and easy to construct.

Building programming language translators was once an expensive process, requiring many years of programmer effort; the separation of compilation into a distinct set of phases and the introduction of interoperable tools for each phase has made compiler-writing a subject for one-semester undergraduate courses. I believe that this approach to structuring wide-area applications will yield similar results for wide-area applications.

Chapter 4
Existing group communication systems

Several group communication systems have been proposed or built. Many other mechanisms have been developed that provide similar functions under a different name. In this chapter I will survey some existing approaches to constructing group communication mechanisms, discussing how each can be built and the guarantees they provide. These approaches are classified according to the component guarantees presented in Chapter 3 so that they can be compared with each other and with the weak-consistency implementations I have developed.

Each section in this chapter concentrates on one particular approach. The first two sections of this chapter cover two general approaches: centralization and consistent replication. While there are many variations on each, they all provide essentially the same guarantees. Since neither approach is well suited to large-scale wide-area systems, I only discuss them briefly. The remaining sections present different protocols or systems, each of which provides group communication in a different way.

The systems can be classified by the message reliability, latency, and ordering guarantees they provide. Table 4.1 summarizes the guarantees provided by each of the systems surveyed. The systems are organized vertically by increasing strength of the message ordering guarantee. Columns show the latency and reliability guarantees.

The sections that follow are organized roughly from strongest guarantee to weakest. The approaches in the first sections do not work well for large-scale groups, while the later sections discuss systems explicitly built for the wide area.

TABLE 4.1: The group communication systems surveyed. Listed in roughly increasing strength of ordering guarantee.

Message ordering	Reliable delivery: Interactive	Reliable delivery: Eventual	Unreliable delivery
Unordered	Reliable multicast	Anti-entropy Tattler OSCAR	Direct mail Rumor mongery
Causal	Lazy Replication ISIS CBCAST Psync	Lazy Replication	
Total, noncausal	ϵ-serializability	Refdbms OSCAR	
Total, causal	ISIS ABCAST Centralized systems Orca RTS Consistent replication		

4.1 Centralized protocols

The simplest way to build a wide-area service is to implement a server and allow clients everywhere to connect to it. This is the *centralized* approach. A centralized group communication system requires that all group members communicate with the central group server to send and receive every message. Many current wide-area services have taken this approach, including the WAIS text-retrieval system [Kahle89, Kahle91], the World Wide Web distributed hypertext system [Berners-Lee92], and the Archie FTP location service [Emtage92b]. A central server is easy to implement and uncomplicated to communicate with. Unfortunately, it is only as available as the host it runs on and the network between it and its clients. If the service becomes popular, a centralized server has no mechanism for spreading load to other systems – which was a problem for the Archie system within a year of its introduction.

The fault tolerance and scalability of a centralized server can be improved by providing additional servers using a *primary copy* or *master-slave* approach. Application requests are sent to the primary copy, which synchronously sends the request to all secondary copies. The primary copy is responsible for sequencing operations. When a replica recovers from a failure, or when the primary fails, an election is held to determine which replica becomes the primary. The Echo file system [Mann89, Hisgen90], for example, combined primary-copy replication with client caching. The Sun Network Interface Service [Sun88] (commonly called the “Yellow Pages” service) also uses primary and secondary servers.

The Domain Name Service [Mockapetris87] and the Clearinghouse name service [Oppen81] both combine centralization and replication. In both systems, the name space database is divided into a set of domains, and each domain must be stored at one or more servers. A server may store more than one domain. Some domains have only one server, while other domains are replicated to several servers. The Clearinghouse used epidemic replication (Section 4.10) to maintain multiple copies of a domain.

The *quasi-copy* technique [Alonso90b, Barbará90, Alonso90a] provides a way to specify bounds on the inconsistency allowed between master and slave copies of data. A user can specify that the value of a copy should differ from the “real” value by no more than some constant, or that it should not be out-of-date by more than some period or number of versions.

4.2 Consistent replication protocols

A replication protocol defines operations on a replicated data object. One principal is a *client,* and one or more principals are *servers* or *replicas*. The protocols provide client-to-server operations to read and write data, and server-to-server operations for adding and removing replicas and to handle replica failure. A principal can act both as a client and as a replica. Every read and write operation is atomic and consistent, the protocol aborts any operation that cannot observe an up-to-date value.

There are three broad classes of replication protocol: available copy, voting, and hybrids. The *available copy* protocols [Bernstein84, Bernstein87] are sometimes called the “read-one-write-all” protocols. Update operations must be applied at all available replicas, while a client can read from any replica. Correct execution of these protocols require that the network never partition, and that messages be delivered reliably. In the *voting* protocols, each replica is assigned one or more votes. Each operation collects votes from replicas, and can proceed when it has collected a quorum of votes. Majority Consensus Voting protocols [Thomas79, Gifford79] assign each replica one vote, and require each operation to collect a majority of votes. Other voting protocols change vote assignments when replicas fail [Barbará86], or change quorum requirements [Davčev85, Jajodia87, Long88]. The Virtual Partition protocol [El-Abbadi86] is a *hybrid* between available copy and voting.

Most of these protocols cannot scale to large numbers of replicas and require excessive communication overhead on wide-area networks. The voting protocols generally require communication with several replicas for every operation. All consistent replication protocols block a replica from providing service when it is partitioned from the rest of the network. The protocols rely on interactive message delivery, so they are sensitive to transient network and host failure. I have considered how several replication protocols can be implemented using a *group multicast* message delivery protocol [Golding92d, Golding91b].

4.3 Orca RTS

The Orca programming language [Bal89, Bal90] provides language constructs for distributed programming. Distributed applications are written in terms of shared data objects that can be accessed by any cooperating process. The shared objects are similar to a distributed shared memory, except that each object is a structured encapsulation of data values and operations. All update operations on distributed shared objects are *serializable;* that is, their effects are the same as if they had been applied in a serial order on a single central copy of the object.

The Orca compiler generates code that uses a run time system to manipulate shared objects. The run time system includes a group communication system and a process manager. Bal discusses three distributed run time systems, as well as one for a shared-memory multiprocessor [Bal89, Chapter 6]. One of the distributed implementations relied on reliable multicast, one on unreliable multicast, while the third used Amoeba RPC [Mullender90, Mullender86].

The first implementation is based upon a reliable, interactive multicast protocol that delivers update messages in a total order. Every process is multithreaded, and contains an object manager thread plus some application threads. When a thread issues an update operation, it multicasts an update message to every process and then blocks. The object manager receives these messages and executes the operations in the order received. When the update message has been executed at the process that issued the update, the blocked thread is awakened and proceeds. This implementation is simple because the underlying multicast protocol provides semantics that match the requirements of the application data model.

The second implementation, based on an unreliable interactive multicast protocol, is more complex. Processes maintain a count of the messages they have sent, along with a vector of the message counts of other processes. A process appends its message count vector to every outgoing message. When a process receives a message that was multicast by another process, it increments its view of the message count for the other process, then compares the vector on the message with its own. If they do not match, the process has missed some messages and can contact the process for which counts to not match to obtain the missing ones. Since missing messages are detected only when other messages are received, the run time system periodically generates dummy messages to ensure missing messages are detected in a timely fashion. This results in reliable, interactive message delivery.

The message count vector is similar to the message summary maintained by the timestamped anti-entropy protocol. Both mechanisms summarize the set of messages that have been received, and are used to detect messages that a principal or process has not yet received.

The Amoeba RPC implementation achieves serializability using a primary-copy update protocol. One process is designated to maintain the primary copy of an object, and all write operations are forwarded to that process. Different objects can use different processes to maintain their primary copy. Other processes can maintain read-only secondary copies, which are updated by the primary copy using a two-phase locking protocol. The run time system dynamically allocates secondary copies at those processes that perform frequent read-only operations.

4.4 Isis

The Isis distributed programming toolkit [Birman87, Birman91] is without doubt one of the most successful group communication systems yet developed. It has been used to develop many applications, ranging from replicated file systems to financial applications.

Isis provides atomic, interactive delivery with total or causal message orderings. Processes use a group membership service to join and leave process groups, and a process can belong to more than one group. Processes join groups either as a *member* or as a *client*. Group multicast is provided using either the **ABCAST** totally-ordered multicast protocol or the **CBCAST** causally-ordered protocol.

The newest Isis implementation is expected to be composed of a number of different components. Application processes include a toolkit library that implements some of the higher-level group membership services, and provides access to lower-level components. The basic group communication and membership protocols are implemented in another component, which in turn uses a raw communication component. Other services, including naming and failure detection, are implemented as user-level processes with which an application can communicate.

Isis addresses different problems than weak-consistency mechanisms do. Its intended audience is different: it is aimed at smaller systems that must often provide consistent, interactive service. It coordinates groups of transient processes, unlike the fault-tolerant "processes" assumed for wide area services. Isis is also intended as a toolkit that even unsophisticated programmers can use, and thus presents a safer, more comprehensive application interface than the framework I have developed.

4.5 Epsilon serializability

Epsilon serializability (ESR) is a correctness criterion for controlling transaction concurrency, and is not specific to distributed systems. Unlike most of the other mechanisms in this chapter, it is concerned with transactions where several operations must be executed as a group. It is an extension of strict serializability that allows transactions to observe controlled inconsistency [Pu91b, Pu91a]. Rather than enforcing a strict ordering on all transactions, perhaps using a two-phase locking protocol, orderings are applied only to update operations.

Pu and Leff [Pu91a] discuss four methods for implementing a replication control protocol that works with an ESR concurrency control protocol. These replica control protocols use asynchronous message propagation. Each update message contains the effects of an entire transaction. The

Ordered Update method executes update transactions in the same order at every process, which is trivial to implement with totally-ordered, reliable, eventual message delivery. The *Commutative Operations* method is limited to commutative update operations, while the *Read-independent Timestamped Updates* method is limited to operations that either append information or only overwrite older versions. These methods can use unordered, reliable, eventual message delivery. Some of the methods used in Refdbms for avoiding update conflicts take advantage of commutative and append-only operations. Finally, an optimistic method applies operations right away, then uses compensations to undo the effects of transactions that have caused or observed too much inconsistency or that have violated serializability.

An ESR concurrency control protocol allows applications to limit the amount of inconsistency a transaction can observe. Transactions can acquire a certain number of read-write or write-write locks on objects – locks that are disallowed under strict two-phase locking. If a transaction attempts to acquire more conflicting locks than the limit, the transaction is blocked. Commutative and timestamped operations introduce additional kinds of locks.

Because ESR requires locking, it is not a good choice for services that must scale to large numbers of replicas. However, it scales better than strictly serialized systems, and the definitions of operation compatibility can be used to build conflict-reduction mechanisms in weak-consistency systems that do not provide concurrency control (Section 3.2.3).

4.6 Psync

The Psync system [Mishra89] is a complete group communication system that provides causally consistent, atomic, interactive message delivery and group membership operations.

The system is based around the **Psync** communication protocol. Processes start communication by joining a group, which is called a *conversation* in Psync. Each message has an identifier. The protocol appends causal information to each message, and group members use this information to construct a graph of the causal relations between messages. The causal information consists of the identifiers of every message on which this message depends, which requires $O(n)$ space for each message.

Messages are sent to group members using a best-effort multicast. A recipient can detect when it has missed a message, because some other message will name it as a predecessor. The recipient can request a copy of the message it missed from the process that sent the other message.

Psync allows fine-grained control over delivery order. As messages are received and added to the dependence graph, some messages may become deliverable. They are delivered to the application, which performs the corresponding operations. The set of operations can be partitioned into sets of related commutative operations, and each partition assigned a priority. Within a partition, operations that are not causally related are executed in any order, while non-commutative operations are executed in order by priority.

Psync allows processes to join multiple groups, though it only ensures causal consistency within a group. Every message must be qualified by the identifier of the group to which it is sent. Processes join a group by executing a join protocol, after which they will begin receiving group messages. The protocol includes a special operation to remove a temporarily failed process from the group, and another operation to allow that process to recover.

The weak-consistency protocols used in Refdbms avoid the overhead of attaching $O(n)$ state to each message by attaching causal information to *batches* of messages. Each message only carries a single timestamp, which is used to identify the message, while communication sessions include $O(n)$ timestamps.

The Consul system [Mishra92] provides a more complete group communication system, adding failure detection, group membership, and stable storage modules to the basic **Psync** protocol. This modularization of a group communication system includes many more parts than my framework because it includes support for failure detection and recovery. This support is not as important in a weak-consistency system like Refdbms that uses long-lived principals and a message delivery protocol that hides transient host failure.

4.7 A reliable multicast protocol

Garcia-Molina and Kogan [Garcia-Molina88] have developed an internetwork multicast protocol that provides reliable interactive delivery. It uses an unreliable interactive (best-effort) multicast protocol to try to disseminate a message. Messages include sequencing information that allows receivers to detect when they have missed one or more earlier messages. When a receiver detects one or more missing messages, it contacts another principal to obtain a copy of them. As in the Orca unreliable multicast protocol, senders periodically send null messages if they have not recently sent a useful message, allowing receivers to detect missing messages quickly.

One unique feature of this protocol is that it takes advantage of network topology. Each principal has a prioritized list of other principals, and it selects in order from that list when it needs to contact another principal to retrieve a missing message. The protocol includes an algorithm for building priority lists that form a spanning tree over the principals. It uses *communication distance* between hosts, or the number of network links that messages between the two must traverse, to order principals.

This protocol is similar to the Orca protocol based on unreliable multicast. Both protocols attach sequencing information to messages, and use this information to detect messages that have been missed. Both send periodic null messages to ensure progress.

The way this protocol recovers from missed messages and partitions is similar to the timestamped anti-entropy protocol, except that it tries to deliver messages individually and interactively rather than queuing messages for delivery in batches. The protocol builds a spanning tree over the principals in the group, and messages are propagated along edges in that tree rather than between arbitrary pairs of principals. If the spanning tree is built carefully, this approach can minimize network traffic, though it can increase the time required to propagate a message. It is not clear how to build a group membership system that properly recomputes the spanning tree as principals join and leave a group without either centralizing the computation or involving the entire group.

4.8 OSCAR

OSCAR (Open System for Consistency and Replication) [Downing90a, Downing90b] implements weak-consistency replication using a mixture of distributed and centralized elements. It provides reliable eventual message delivery, with a variety of message orderings. The system is notable because it is implemented as a set of cooperating components, though the architecture is significantly different than that in Chapter 3.

Every replica is paired with a *replicator* and a *mediator*. During normal operation, whenever an update is initiated at a replica the corresponding replicator sends an unreliable multicast message to other replicators. The message includes a version number and timestamp. The replicators use this information to present update messages to their local replicas in a correct order. From time to time a master mediator polls every replicator, obtaining a version vector for every database item. The version vectors from different replicas are combined, and the result is multicast to every replicator. Replicators use the vector to detect messages they have missed, and to determine when messages in their local logs can be purged.

When the network partitions, one mediator becomes master in each partition. Mediators are prioritized, and a low-priority mediator becomes active when it has received no messages from higher-priority mediators for a certain period. When the network partition is repaired, the lower-priority mediator will become dormant, while the higher-priority mediator takes over for the entire partition. Updates missed by replicas in one partition or the other will be propagated the next time the mediator broadcasts a version vector.

This approach is not as efficient as my weak-consistency protocols over wide-area networks, and cannot scale as well. Replicators perform an interactive multicast every time they send a message. This causes much more network traffic than pairwise message exchange, which can approximate an optimal spanning tree. Likewise, mediators must communicate interactively with replicators, which causes a similar amount of network traffic and can overload the mediator if the group grows too large. Further, this approach cannot function if the logical network topology is not completely connected, while the timestamped anti-entropy protocol can.

4.9 Lazy Replication

The Lazy Replication system [Ladin90, Ladin91, Liskov87] provides reliable eventual message delivery with a mix of causally and totally consistent orderings. Applications can specify exactly what causal relations should be enforced between messages, so weaker orderings can be specified by omitting some specification. Applications can also specify that causal relationships caused by events outside the group should be considered when ordering messages.

The system uses message count vectors much like those in the Orca unreliable multicast RTS (Section 4.3). The message count vector summarizes a set of messages. Two message count vectors are attached to each message: one that specifies what messages must be delivered before this one, the other serving as a unique identifier. Each principal maintains a vector summarizing the messages that have been applied to the database, along with copies of the vectors from other principals.

The Lazy Replication protocol resembles a version of the timestamped anti-entropy protocol that allows unsynchronized clocks (Unsync TSAE). This protocol is detailed in Section 5.4.4. Both protocols store incoming messages in a log, and use message count or timestamp vectors to summarize sets of messages. Both lazily update principals, and use the vectors to guide message exchange and delivery.

The timestamped anti-entropy protocol is more efficient than Lazy Replication, because Lazy Replication does not take full advantage of the information available in its timestamps. It causes principals to transmit a message from one to the other many times, even when the message has already been observed. It also requires both $\Theta(n^2)$ state for acknowledgments and an extra message log to ensure that duplicate messages are not processed twice, even though it requires loosely-synchronized clocks for good performance. The TSAE protocols can use $\Theta(n)$ state when loosely-synchronized clocks are available. Lazy replication performs one-way updates in its gossip messages, instead of bidirectional updates.

4.10 Epidemic replication

The Xerox Clearinghouse service [Oppen81] is the name and location service for Xerox Network Systems. It maintains a distributed database that maps structured names into a set of properties, such as network addresses. The names are organized into a three-level hierarchical space, structured into organizations, domains within organizations, and local names within domains. The mappings for each domain are stored at one or more Clearinghouse servers. Clients can request any server to look up the binding for a name, and the server will forward the request to the appropriate server if necessary.

The Clearinghouse uses three different mechanisms to propagate updates between servers: *direct mail*, *anti-entropy*, and *rumor mongery* [Demers88, Demers89]. Direct mail is simply an unreliable best-effort multicast.

Rumor mongery provides unreliable eventual delivery, but it is more reliable than a one-time best-effort multicast. To be a rumor monger, a principal selects another principal and sends it one or more *hot rumors*. Hot rumors are recent update messages that the principal believes the other is not likely to have observed. Several different stopping rules will ensure that a message does not continue propagating forever, but none of the rules can ensure that a message has been propagated to every principal before stopping.

The Clearinghouse anti-entropy protocol guarantees reliable eventual delivery, as does the timestamped anti-entropy protocol. To execute the protocol, one principal selects another, and the two exchange update messages until they are mutually consistent. Unlike timestamped anti-entropy, messages are not timestamped, and the protocol does not maintain summaries of the messages that have been received. Instead, database contents are exchanged directly, using checksums to determine when enough entries have been exchange to make both principals mutually consistent. Demers et al. describe heuristics for reducing the expense of this exchange, including building an index on the message log so messages can be ordered from most recently received to least recent.

This anti-entropy protocol provided inspiration for the timestamped anti-entropy protocol. However, the Clearinghouse protocol can only provide unordered message delivery because it operates from the database rather than a message log. It provides no mechanism to detect when a message has been received everywhere, so database entries cannot be deleted reliably, and a principal can receive a message more than once. The Clearinghouse protocols therefore could not be used for applications like Refdbms that need stronger guarantees.

4.11 Summary

Many group communication systems have been proposed and implemented. They provide guarantees ranging from atomic, synchronous, totally-ordered message delivery to unreliable eventual delivery. Only a few provide reliable eventual delivery, the guarantee used in the Tattler and Refdbms systems. The Clearinghouse anti-entropy protocol and Lazy Replication are closest to the timestamped anti-entropy protocol presented in the next chapter.

The weak-consistency protocols I have developed improve on these systems by providing a combination of efficiency and well-defined guarantees. In particular, the timestamped anti-entropy protocol delivers a message at most once to any principal, allows correct detection of fully-acknowledged messages, and can be composed with a number of message ordering components. It improves efficiency by transmitting messages in batches rather than singly.

Chapter 5
Weak-consistency communication

The previous chapters introduced a framework for building a group communication system as the basis of a wide-area application. In this chapter I focus attention on the message delivery and ordering components of the framework. These two components deliver application messages to group members, and ensure that the messages are delivered in order. As discussed in Chapter 3, there are several guarantees that can be provided by an implementation of either component. For delivery component, the *timestamped anti-entropy protocol* provides reliable, eventual delivery. Various implementations of the corresponding message ordering component can provide any ordering guarantee.

5.1 Reliable, eventual message delivery

Systems like Refdbms and the Tattler use the *timestamped anti-entropy* protocol, which is used to build a message delivery component that provides reliable, eventual delivery. This means that every functioning group member will receive every message, but the message may require a finite unbounded time for delivery. These guarantees reflect a tension between an application's needs for timely information, accurate information, and reliability.

Reliable delivery was chosen because information services are generally expected to provide authoritative answers to queries. If one Refdbms replica were to miss an update, for example, the database copy at that replica could permanently diverge from others. Systems like location services that provide hints rather than authoritative answers are good candidates for unreliable mechanisms [Terry85, Oppen81, Jul88, Fowler85].

Eventual delivery trades latency for reliability. The message delivery component can mask out transient network and host failures by delaying messages and resending them after the fault is repaired. It also allows messages to be batched together form transmission, which is often more efficient than transmitting each message singly. Both of these features are especially important for mobile systems that can be disconnected from the Internet for extended periods.

The timestamped anti-entropy (TSAE) protocol is similar to the anti-entropy protocol used in the Xerox Clearinghouse [Demers88, Demers89]. Each principal periodically contacts another principal, and the two exchange messages from their logs until both logs contain the same set of messages. The TSAE protocol maintains extra data structures that summarize the messages each principal has received, and uses this information to guide the exchanges. There are two versions of the TSAE protocol: one that requires loosely-synchronized clocks, and one that does not. The loosely-synchronized version is presented in this section, while the general version is deferred until Section 5.4.4.

One important feature of TSAE is that it delivers messages in *batches*. Consider the stream of messages sent from a particular principal. Those messages will be received by other principals in batches, where each batch is a run of messages, with no messages missing in the middle of the run. When a principal receives a batch, the run of messages will immediately follow any messages already received from that sender. In this way principals receive streams of messages, without ever observing a "gap" in the sequence.

The TSAE protocol provides additional features that are necessary for information services. The protocol can be composed with a message ordering component to produce specific message ordering guarantees. The ordering component makes use of the batching property to reduce overhead. TSAE provides positive acknowledgment when all principals have received a message, so that the message can be removed from logs and so that death certificates can be removed from the database. It also provides a mechanism for two principals to measure how far out of date they are with respect to each other. Applications can use this feature to prioritize communication when resources are limited, and to prompt users of mobile systems to temporarily connect to a high-bandwidth link.

```
class timestamp {
      hostId host;
      clockSample clock;

      Boolean sameHost(timestamp t);
      Boolean lessThan(timestamp t);
      // (etc...)
}

timestamp CurrentTimestamp();
      // returns a unique timestamp for the local host
```

FIGURE 5.1: The timestamp data structure.

In this chapter, the group is assumed to have a fixed membership M of n principals. This restriction is removed in Chapter 6. Chapter 7 explores the performance of these protocols.

5.1.1 Data structures for timestamped anti-entropy

Timestamps are used in every component to represent temporal relations and to name events. As shown in Figure 5.1, a timestamp consists of a sample of the clock at a host, and is represented as the tuple $\langle$hostId, clock$\rangle$. The clock resolution must be fine enough that every important event in a principal, such as sending a message or performing anti-entropy, can be given a unique timestamp. Timestamps are compared based only on their clock samples, so that timestamps from different hosts can be compared. The more specialized case of only comparing samples from a single host is a trivial extension. Each host provides a function to retrieve a current timestamp; this function will be named CurrentTimestamp in this dissertation.

Timestamps can be organized into *timestamp vectors*. A timestamp vector is a set of timestamps, each from a different principal, indexed by their host identifiers (Figure 5.2). It represents a snapshot of the state of communication in a system. In particular, it represents a *cut* of the communication. Mattern [Mattern88] provides a well-written introduction to the use of time measures such as cuts in reasoning about the global states of distributed systems.

```
typedef set ⟨ principalId, timestamp ⟩ timestampSet;

class timestampVector {
      timestampSet timestamps;

      // update the entry for one principal
      update(principalId, timestamp);
      // merge in another vector, taking the elementwise maximum
      updateMax(timestampVector);

      // determine temporal relation of a timestamp
      Bool laterThan(timestamp);
      Bool earlierThan(timestamp);
      Bool concurrentWith(timestamp);

      // determine temporal relation of another vector
      Bool laterThan(timestampVector);
      Bool earlierThan(timestampVector);
      Bool concurrentWith(timestampVector);

      // return the smallest timestamp in this vector
      timestamp minElement();
}
```

FIGURE 5.2: The timestamp vector data structure.

Some of the operations on a timestamp vector deserve special mention. Two timestamp vectors can be combined by computing their elementwise maximum. A timestamp is considered *later* than a timestamp vector if the vector contains a lesser timestamp for the same host. A timestamp is considered *concurrent* with a vector if either the vector has exactly the same timestamp for the same host, or the vector does not include a timestamp for the host. Note that these comparisons are limited to one host, and do not consider possible temporal relations between different hosts. One timestamp vector is later than another if every timestamp in the first vector is greater than or equal to the corresponding timestamp in the other, and the two vectors are not equal.

Each principal executing the TSAE protocol must maintain three data structures: a *message log* and two timestamp vectors (Figure 5.3). These data structures must be maintained on stable storage, so they are not lost when the host or principal crashes.

```
timestampVector summary;
timestampVector ack;

typedef list ⟨ principalId, timestamp, message, delivered ⟩ msgList;

class msgLog {
    msgList messages;

    // manipulate messages in the log
    add(principalId, timestamp, message)
    deliver(principalId, timestamp)
    remove(principalId, timestamp, message)

    // query the log for all messages newer than some vector
    msgList listNewer(timestampVector)
    // query the log for all messages older than some timestamp
    msgList listOlder(timestamp)
    // query for all messages sent by a principal between two timestamps
    // can use special value 'ANY' for principalId
    msgList listMsgs(principalId, timestamp first, timestamp last)
}

msgLog log;
```

FIGURE 5.3: Data structures used by the TSAE communication protocol.

The message log contains messages that have been received by a principal. A timestamped message is entered into the log on receipt, and removed when all other principals have also received it Messages are eventually *delivered* from the log to the application. The log includes functions to retrieve messages that were sent later than the events recorded in a timestamp vector.

Not all applications will use a message log. Many applications, including the Tattler, can operate directly from the copy of group state maintained by the application application component, as discussed in Section 3.3.1. In that case the application must provide an interface to retrieve "messages" along with the associated principal identifier and timestamp from the group state.

Principals maintain a *summary timestamp vector* to record what updates they have observed. The vector contains one timestamp for every group member, and each member has received every message with lesser timestamps. Figure 5.4 shows how the summary vector relates to the messages

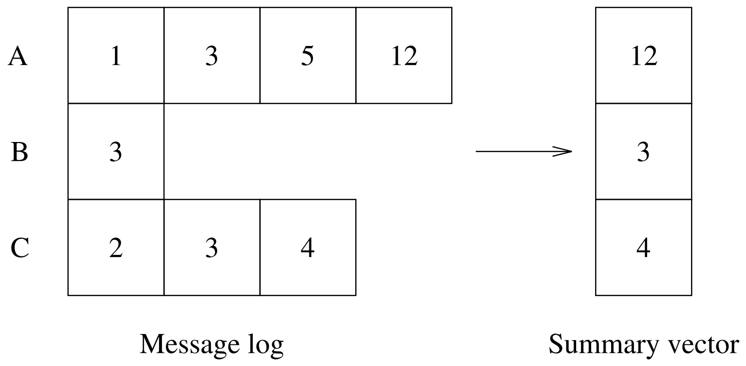


FIGURE 5.4: How the summary vector summarizes the messages in the log. Each message is marked with its timestamp. The timestamped anti-entropy protocol ensures that there are no "gaps" in the sequence of messages in the log, so that the last timestamp stands for every previous message.

---

in the log. Recall that messages are transmitted in batches, and that there are never gaps in the message sequence, so the timestamp of the latest message indicates that every message with an earlier timestamp has been received.

The summary vector provides a fast mechanism for transmitting information about the state of a principal. It is used during an anti-entropy exchange to determine which messages have not yet been received by a principal, and two principals can compare their summary vectors to measure how far out of date they are with respect to each other.

Formally, the summary vector maintained by principal A is written $\mathsf{summary}_A$; the subscript will be omitted when it is clear from context. Principal A records a timestamp t for principal B in $\mathsf{summary}_A(B)$ when A has received all messages generated at B with timestamps less than or equal to t. The timestamp t is measured from the clock at B. Each principal maintains one such timestamp for every principal in the group.

Each principal also maintains information about message acknowledgments. Rather than explicitly send an acknowledgment for every message, the information in the summary vector is used to build a summary acknowledgment. As long as principals use loosely-synchronized clocks, the smallest timestamp in the summary vector can be used as a single acknowledgement timestamp for all messages received by the principal (Figure 5.5). Clearly every message with a timestamp less than or equal to the minimum has been received, though there are later messages that are not yet

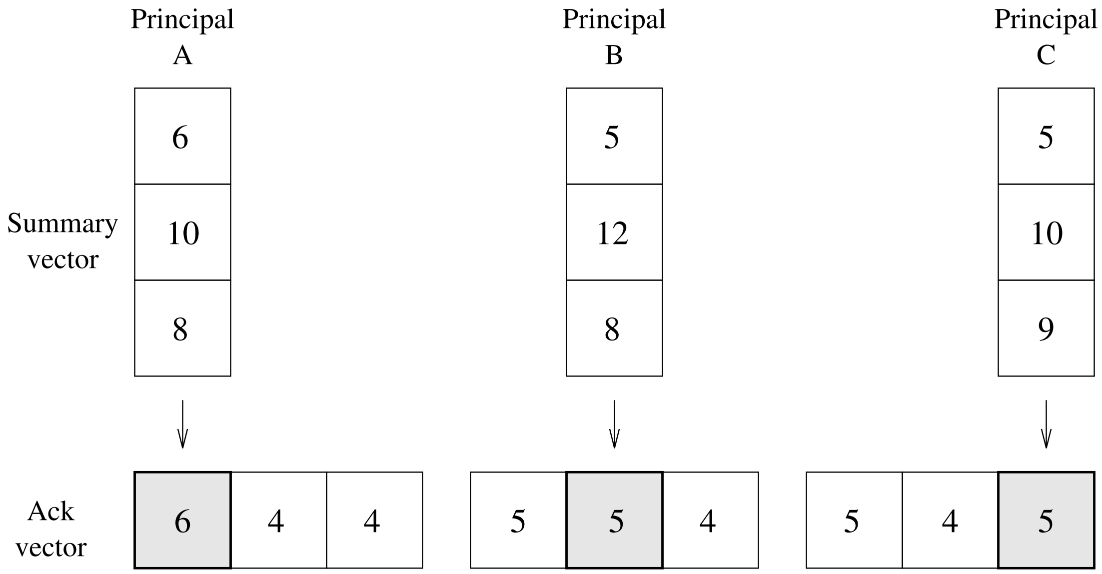


FIGURE 5.5: Summary and acknowledgment vectors for principals with loosely-synchronized clocks. The dark cell in the acknowledgment vector contains the minimum timestamp from the summary vector, while the other cells contain copies, usually slightly out of date, of the minima from other principals.

acknowledged. The TSAE protocol ensures that the acknowledgment timestamp makes forward progress, so every message will eventually be acknowledged.

Each principal stores a copy of the acknowledgment timestamp of every other group member. The TSAE protocol propagates acknowledgment timestamps just as it propagates application messages.

The acknowledgment timestamp vector at principal A is written $\mathtt{ack}_A$. Principal A holds a timestamp t for principal B as $\mathtt{ack}_A(B)$ if B has received every message from any sender with timestamps less than or equal to t. Principal B periodically sets its entry in its acknowledgment vector – that is, $\mathtt{ack}_B(B)$ – to the minimum timestamp recorded in its summary vector (Figure 5.5).

A principal can determine that every other group member has observed a particular message when the message timestamp is earlier than all entries in the local ack vector. This feature is used to purge messages from the log safely, and in handling dynamic group membership (Chapter 6).

5.1.2 The timestamped anti-entropy protocol

The anti-entropy protocol maintains the timestamp vectors and message log at each principal. It does so by periodically exchanging messages between pairs of principals.

From time to time, a principal A will select a partner principal B and start an *anti-entropy session.* A session begins with the two principals allocating a session timestamp, then exchanging their summary and acknowledgment vectors. Each principal determines if it has messages the other has not yet received, by seeing if some of its summary timestamps are greater than the corresponding ones of its partner. These messages are retrieved from the log and sent to the other principal using a reliable stream protocol. If any step of the exchange fails, either principal can abort the session, and any changes made to the state of either principal are discarded. The session ends with an exchange of acknowledgment messages.

At the end of a successful session, both principals have received the same set of messages. Principals A and B set their summary and acknowledgement vectors to the elementwise maximum of their current vector and the one received from the other principal.

Figure 5.6 shows what might happen to two principals during an anti-entropy session. The two principals start with the logs shown at the top of the figure, where A has messages from itself and from C that B has not yet received, while B has sent messages that A has not received. They determine which messages must be sent by comparing their summary vectors (middle row), discovering that the lightly shaded messages must be sent from A and the darker shaded messages must be sent from B. At the end of the session, both principals have received the same set of messages and update their summary vector to the value shown in the bottom row.

Figure 5.7 details the protocol executed by a principal originating an anti-entropy session, while Figure 5.8 shows the corresponding protocol the partner principal must follow.

After anti-entropy sessions have completed, the message ordering component can deliver messages from the log to the database and purge unneeded log entries. It uses the summary and acknowledgment vectors to guide this principal, as discussed in Sections 5.3 and 5.5.

By the end of an anti-entropy session, the originator and partner principals have both received any messages sent by either one of them up to the time the exchange started. In addition, one or

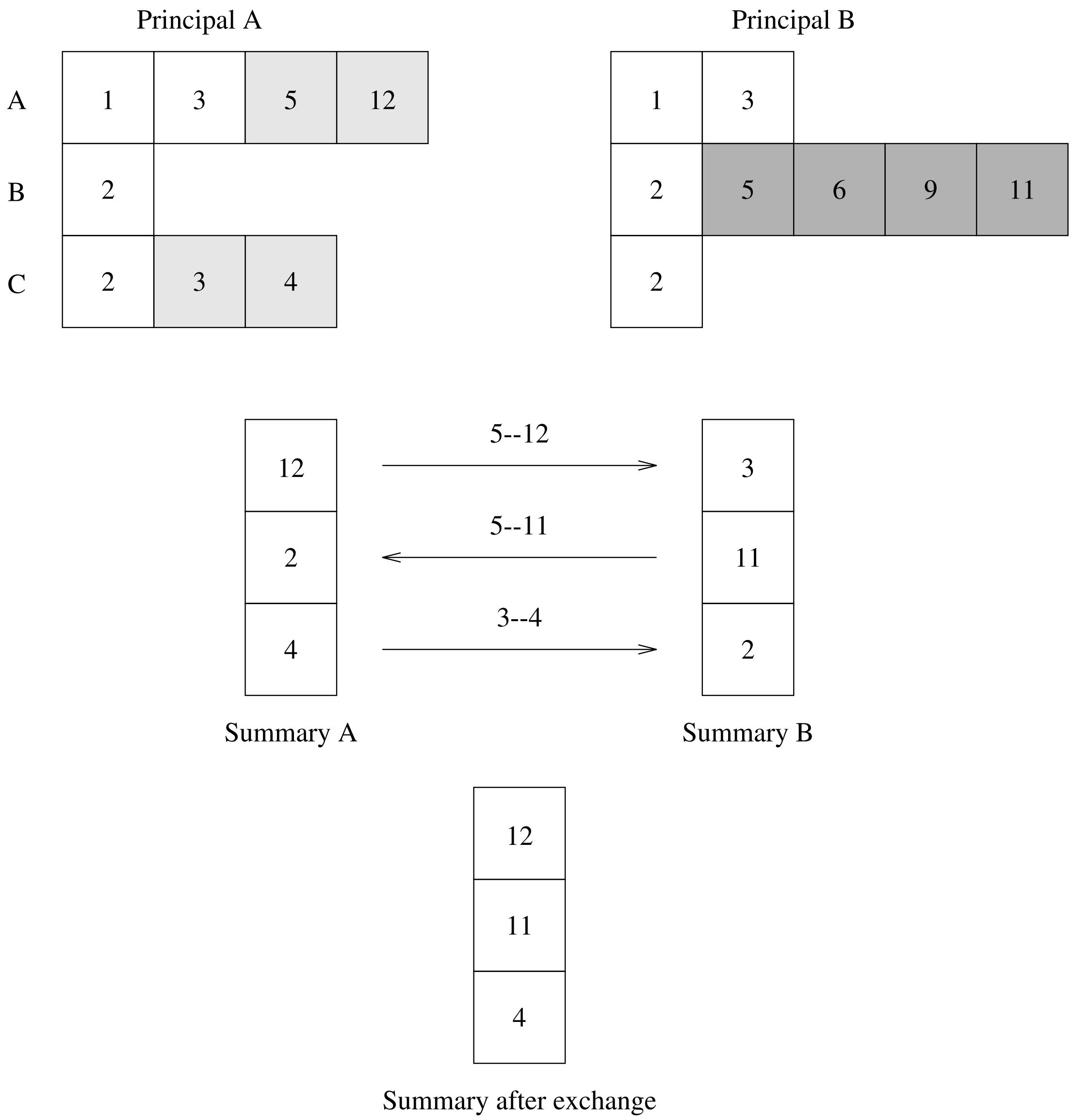


FIGURE 5.6: An example anti-entropy session. Principals A and B begin with the logs in the top of the figure. They exchange summary vectors, discovering that the shaded messages must be exchanged. After the exchange, they update their summary vectors to the bottom vector.

both will probably have received messages from principals other than its partner. In the example in Figure 5.6, principal A forwarded messages originally sent by C. This means that one principal need not directly contact another to receive its messages. Instead, some sequence of principals exchanging messages can eventually propagate the message. The correctness of TSAE is based on the reliability of this kind of diffusion, as discussed in the next section.

```
// Information about the partner
principalId partner;
timestampVector partnerSummary, partnerAck;
// A temporary copy of the local summary and ack vectors, to avoid
// timing problems with concurrent sessions
timestampVector localSummary, localAck;
msgList messages;

partner = selectPartner();

// update local vectors and exchange them with partner
summary[thisPrincipal] = CurrentTimestamp();
localSummary = summary;
ack[thisPrincipal] = minElement(localSummary);
localAck = ack;
send(partner, "AE request", localSummary, localAck);
receive(partner, partnerSummary, partnerAck);

// exchange messages
messages = log.listNewer(partnerSummary);
for ⟨ pid, timestamp, message ⟩ in messages:
      send(partner, pid, timestamp, message);
while (receive(partner, pid, timestamp, message)):
      log.add(pid,timestamp,message);

// finish communication
send(partner, "Acknowledged");
receive(partner, "Acknowledged");

// update summaries and trigger the message ordering component
summary.updateMax(partnerSummary);
ack.updateMax(partnerAck);
DeliverMessages();
```

FIGURE 5.7: Originator's protocol for TSAE with loosely-synchronized clocks. Note that error handling is not included to make the presentation readable. An anti-entropy session is aborted if either principal detects an error in communication, in which case any updates to the message log or timestamp vectors are discarded. The acknowledgment vector is updated by this protocol but used by the message ordering and log purging functions.

```
// Information about the other principal
principalId originator;
timestampVector originatorSummary, originatorAck;
// A temporary copy of the local summary and ack vectors, to avoid
// timing problems with concurrent sessions
timestampVector localSummary, localAck;
msgList messages;

// receive request from originator and update local state
receive(originator, "AE request", originatorSummary, originatorAck);
summary[thisPrincipal] = CurrentTimestamp();
localSummary = summary;
ack[thisPrincipal] = minElement(summary);
localAck = ack;
send(originator, localSummary, localAck)

// exchange messages
while (receive(originator, pid, timestamp, message)):
      log.add(pid,timestamp,message);
messages = log.listNewer(originatorSummary);
for ⟨ pid, timestamp, message ⟩ in messages:
      send(originator, pid, timestamp, message);

// finish communication
receive(originator, "Acknowledged");
send(originator, "Acknowledged");

// update summaries and trigger the message ordering component
summary.updateMax(originatorSummary);
ack.updateMax(originatorAck);
DeliverMessages();
```

FIGURE 5.8: Partner's protocol for TSAE with loosely-synchronized clocks.

5.2 Correctness

In this section I define correctness for reliable eventual delivery protocols, and establish it for TSAE. Discussion is limited to the version of TSAE that uses loosely-synchronized clocks.

Recall that the term *eventually* was defined in Chapter 2 to mean that an event occurs in a finite but unbounded time after some time t.

The reliability condition can then be stated formally:

Condition 5.1 *If a message is sent by principal p at real time t, the message will eventually be received at any group member principal q.*

Note that correctness is defined in terms of time as might be measured by an external observer, and not in terms of *virtual* time or clocks [Mattern88]. There are two reasons for this choice. First, I found it easier to reason about behavior of the protocol using time rather than clocks, especially when replica and message failure is involved. In addition, I believe that this formulation is more useful, since the intended application of these techniques is for applications interacting with people and physical devices. This implies that the applications will have channels of communication outside of the group communication system, and outside the domain covered by any virtual time measure. This is the same problem that motivated Lamport's *Strong Clock Condition* [Lamport78].

Recall also that the group membership is assumed static, that the network need not be fully connected, and that principals do not fail permanently. The static membership limitation is removed in Chapter 6.

Using the TSAE protocol, every principal periodically attempts to perform an anti-entropy session with each of its neighbor principals to deliver a message. Eventually, sessions will succeed with each of them, propagating the message. Each of those principals in turn will eventually contact all of their neighbors, and so on until all principals have received the message. In the next several sections I detail a proof of this property.

5.2.1 Logical communication topology

Consider the communications between principals. Recall that the network is connected but not necessarily complete (Section 2.4). The relation T defines the logical topology of the network by specifying which hosts are neighbors. P is the set of all principals.

Definition 5.2 *The* logical principal topology graph $G_T = (P, T)$ *is an undirected graph. The relation $T : P \to P$ is a symmetric relation that defines what pairs of principals can exchange messages, that is,*

$$(A, B) \in T \leftrightarrow A, B \in P \wedge A \text{ can communicate with } B.$$

The logical topology graph G_T must be connected. In an environment such as an Ethernet, G_T is the complete graph $C^{|P|}$, that is, every principal can communicate with every other principal. In systems such as Usenet, $G_T \subset C^{|P|}$ because each node only communicates with a few other nodes. While the physical links in most internetworks do not form a complete graph, the logical communication topology provided by the IP protocol is a complete graph. In practice the Internet is composed of a few communication cliques.

Definition 5.3 Communication cliques *are completely connected subgraphs of the topology graph G_T. If a principal is a member of a communication clique, it can communicate with every other principal in that clique.*

An application can elect to restrict the logical topology graph to a subset of the topology provided by the network protocols on which it is built. For example, it may be advantageous to structure the principals into a tree or a ring.

Recall that there are no permanent failures in the communication network or principals. That is, any pair of principals connected in the logical topology graph can eventually successfully send and receive a message, or more formally, the probability that on principal cannot successfully send a message to another principal during the time interval $(t, t + \delta)$ goes to zero as $\delta \rightarrow \infty$.

5.2.2 Eventual communication

The first step of the proof is to show that the TSAE protocol eventually performs anti-entropy sessions between every pair of principals connected in the logical topology graph.

Definition 5.4 (Attempted communication relation) *The relation $A(t, t + \delta) : P \rightarrow P$ is the set of ordered pairs of principals where the first principal has attempted to send one or more messages to the second principal during the period $(t, t + \delta)$.*

I assume that $A \subseteq T$, so that no attempt is ever made to communicate between two principals that the logical topology prevents from communicating directly. The graph G_A is defined in the obvious way.

Definition 5.5 (Successful communication relation) *The relation $S(t, t+\delta) : P \rightarrow P$ contains a pair for every principals that successfully sent a message to another principal during the time period $(t, t+\delta)$. The graph G_S is defined in the obvious way.*

$S(t, t+\delta)$ is clearly a subset of $A(t, t+\delta)$.

Lemma 5.6 (Eventual communication) *As $\delta \rightarrow \infty$, $S(t, t+\delta)$ converges to $A(t, t+\delta)$ as long as principals periodically retry messages that failed to be delivered.*

> *Proof:* If $S \not\rightarrow A$, then there is a pair $(a, b) \in A$ such that $(a, b) \notin S$ for all times $t+\delta$. However, if a message failed during a period $(t, t+\delta_1)$, by assumption the message will be retried during some period $(t+\delta_1, t+\delta_1+\delta_2)$. As $\delta_2 \rightarrow \infty$ the probability of the message not getting through goes to zero. Thus the probability of there being a pair (a, b) that have not been able to communicate goes to zero, and S converges to A. $\square$

Principals periodically perform anti-entropy sessions with neighbor principals. I assume there is an upper bound k on the time between attempts to communicate, and that a principal selects its partner so that every neighbor will eventually be selected after any time t. The probability that a principal a has not performed an anti-entropy session with a neighbor b during a period δ is thus bounded by the probability s_b of selecting neighbor b on each attempt, raised to the power of a lower bound on the number of times a has performed the anti-entropy algorithm:

$$\text{Pr}(a \text{ has not performed anti-entropy with } b \text{ during } \delta) \leq (1 - s_b)^{-1})^{\lfloor \frac{\delta}{k} \rfloor}. \tag{5.1}$$

As $\delta \rightarrow \infty$, the probability of not attempting anti-entropy goes to zero. This formulation accounts for both host failure, which must not be permanent to satisfy this constraint, and for the distribution times between anti-entropy sessions.

The attempted anti-entropy relation $A_{ae}(t, \delta)$ is defined in the same way as the attempted communication relation. Since the probability of not attempting an anti-entropy session with a neighbor goes to zero as δ goes to infinity, $A_{ae}(t, \delta)$ converges to the topology relation T as $\delta \to \infty$.

The relation $S_{ae}(t, \delta)$ is in turn defined as the set of anti-entropy sessions that have completed successfully during the period $(t, t + \delta)$. The protocol presented in Section 5.1.2 either completes successfully or aborts; an abort occurs only when one or both principals detect probable failure in either principal.

The mechanism for detecting failure must be accurate. That is, if both principals are functioning correctly, and they are able to communicate, the failure detection must not systematically report failure. However, if either principal or the network have failed, it must report failure. Formally, Pr(reporting failure| no actual failure) < 1. In practice, timeouts can be used to implement this kind of failure detector on the Internet.

Lemma 5.7 (Eventual anti-entropy) *If a principal repeatedly attempts to perform an anti-entropy session with another, starting at any time t, eventually one of these sessions must execute to completion.*

> *Proof:* Assume that some principal A repeatedly attempts to perform anti-entropy with principal B. By Lemma 5.6, A and B will eventually be able to exchange messages. The mean time-to-failure for principals was assumed to be much larger than the time required to execute the anti-entropy protocol, so the probability of one or both principals failing during the session is less than one. When both principals and the communication medium are functioning, probability that the failure detector will falsely report failure is also less than one. If the failure detector does not report failure, then the session can run to completion. Therefore the probability of any particular session failing is less than one, and the probability of every sessions failing goes to zero as the number of attempts goes to infinity. □

Since every principal repeatedly attempts to perform anti-entropy with its neighbors, and repeated attempts will eventually succeed, it follows that $S_{ae}(t, \delta)$ converges to $A_{ae}(t, \delta)$ as δ goes to infinity. Since $A_{ae}(t, \delta) \to T$, this implies $S_{ae}(t, \delta) \to T$.

5.2.3 Summary vector progress

Now that it has been shown that all principals will eventually perform anti-entropy with all their neighbors, the proof turns to the rôle of the summary timestamp vector in the TSAE protocol. The summary vector is only modified during successful anti-entropy sessions, and one value of the vector (stored in localSummary in Figure 5.7) is associated with the session. These vectors satisfy the requirements for a *vector time* measure as defined by Mattern [Mattern88]. The *happens-before* relation (or the equivalent *happens-after* relation) is defined between the timestamp vectors associated with two sessions to determine whether the sessions could be causally related:

Definition 5.8 (Happened before) *A timestamp vector v is said to have* happened before *another vector w, written $v \leq w$, iff $\forall i : v[i] \leq w[i]$.*

The final step in showing that the anti-entropy algorithm makes progress is to show that any given anti-entropy session will eventually happen before sessions involving every principal. The set of principals that have participated in sessions causally later than the session in question is defined as:

Definition 5.9 *If a successful anti-entropy session begins at time t_0 at principal p_0, the set $V(p_0, t_0, \delta)$ is the set of principals at time $t_0 + \delta$ that have performed a successful anti-entropy session that causally follows the anti-entropy session performed by p_0 at t_0.*

In each of these anti-entropy sessions, $\mathsf{summary}_p(p_0) \geq t_0$ since the session *happened-after* the session initiated at time t_0.

Lemma 5.10 (Diffusion) *As $\delta \to \infty$, $V(p_0, t_0, \delta)$ converges to the set of members M.*[1]

Proof: Obviously, $p_0 \in V(p_0, t_0, \delta)$ for any $\delta \geq 0$. Another principal $p_n \in V(p_0, t_0, \delta_n)$ if and only if it has performed anti-entropy with some predecessor principal that was in V at an earlier time. Formally, there must be another principal $p_{n-1} \in V(p_0, t_0, \delta_{n-1})$ at

[1] It is instructive to compare this definition and proof of diffusion to that used by Cristian [Cristian86, Cristian90] in his work on atomic broadcast.

$\delta_{n-1} < \delta_n$, such that $(p_{n-1}, p_n) \in S_{ae}(t + \delta_{n-1}, \delta_n - \delta_{n-1})$. The predecessor principal p_{n-1} can be equal to p_0. Since the communication topology graph G_T is connected, it is possible to construct at least one sequence of principals $(p_0, p_1, \ldots, p_{n-1}, p_n)$ to every principal p_n. Another way of stating this property is that a spanning tree rooted at p_0 can be constructed on the logical topology graph. Each pair (p_{i-1}, p_i) will eventually appear (in order) in $S_{ae}(t_0, t_0 + \delta)$ as δ increases, making each $p_i \in V(p_0, t_0, \delta)$ and eventually $p_n \in V(p_0, t_0, \delta)$. Thus $V(p_0, t_0, \delta) \to M$ as $\delta \to \infty$. $\square$

Note that anti-entropy sessions between principals p_{n-1} and p_n update $\mathsf{summary}_{n-1}(p_0)$ and $\mathsf{summary}_n(p_0)$ to be greater than or equal to t_0.

Since the TSAE protocol is eventually performed at every principal, it is easy to show that messages are delivered to every principal. Consider a message sent by principal p at time t. This event either occurs after some preceding anti-entropy session, or after the principal joins the group if it has not yet performed anti-entropy, and it occurs before the next anti-entropy session.

Definition 5.11 *The timestamp vector* $\mathrm{prev}_{p,t}$*, the* previous summary vector *at principal p at time t, is the summary timestamp vector* $\mathsf{summary}_p$ *produced by the anti-entropy algorithm with the latest value* $\mathsf{summary}_p(p) < t$*. This vector is associated with the* previous session. *Similarly the* next summary vector, $\mathrm{next}_{p,t}$*, is the copy of* $\mathsf{summary}_p$ *with the least* $\mathsf{summary}_p(p) > t$*. This vector is associated with the* next session.

Recall that there is an upper bound on the period between anti-entropy sessions at principals, so every event, such as sending a message, will have well-defined prev_p and next_p vectors. Recall also that all events at a principal are associated with unique timestamps, so there can never be an anti-entropy event with the same timestamp as a message.

The principal is certain to have a copy of the message when the next anti-entropy session occurs, even if there have been failures since it was sent. The state of the principal at the end of the preceding session, including the message, will have been preserved on stable storage.

Theorem 5.12 *If a message is sent at principal* p_0 *at time* t*, every principal eventually receives a copy of the message.*

Proof: Initially, the message has only been received by the principal p_0 that sent it. The message has timestamp $\tau = \text{clock}(p_0, t)$. At the beginning of the next anti-entropy session between p_0 and some other principal p_1, $\mathsf{summary}_1(p_0) \le \text{prev}_{0,t}(p_0) < \tau$, since the latest clock value that any other principal could have for principal p_0 is that in the previous summary vector $\text{prev}_{0,t}$. This relation implies that the message will be sent from p_0 to p_1 during the anti-entropy session. At the end of the session, $\mathsf{summary}_0(p_0) = \mathsf{summary}_1(p_0) = \text{next}_{0,t}(p_0) > \tau$.

Consider an anti-entropy session at time t_n, between principal p_{n-1}, which has already performed an anti-entropy session that causally follows the message and has therefore received a copy, and principal p_n, which has not yet done so. Since p_{n-1} has performed a causally-related anti-entropy session, $\mathsf{summary}_{n-1}(p_0) \ge \text{next}_{0,t}(p_0) > \tau$. Since p_n has not, $\mathsf{summary}_n(p_0) \le \text{prev}_{0,t}(p_0) < \tau$. Since $\mathsf{summary}_{n-1}(p_0) > \tau > \mathsf{summary}_n(p_0)$, the anti-entropy protocol will transmit a copy of the message to p_n during the anti-entropy session.

By Lemma 5.10, every principal will eventually perform an anti-entropy session that causally follows $\text{next}_{0,t}$. In each of these sessions, the message will be transmitted to the principal. Thus the message will eventually spread to every principal. □

One side-effect of this protocol is that every principal will receive each message exactly once. The summary vector entry for p_0 at every principal will transition from a value less than τ to one greater than τ at most once.

Note that the use of an unreliable multicast protocol in combination with TSAE does not invalidate this proof, since the proof relies solely on timestamps recorded during anti-entropy sessions and the *existence* of messages in the log. However, this can cause a principal to receive a message more than once. Section 5.4.3 discusses optimizations for this case.

```
PurgeLog()
{
    msgList messages;
    timestamp minAck;

    minAck = ack.minElement();
    msgList = log.listOlder(minAck);
    for ⟨ principalId, timestamp, message, delivered ⟩ in messages:
        if (delivered)
            log.remove(pid, timestamp, message);
}
```

FIGURE 5.9: A function to purge messages from the message log.

5.3 Purging the message log

The message log must be periodically purged so that it does not grow without bound. Even if there is no log, as in applications that work directly from an application database (Section 3.3.1), unneeded death certificates must be purged. Message purging is correct if it does not interfere with message propagation, and if every message is eventually purged.

Whether a message log is used or not, a message or death certificate can safely be removed when every member principal has received it. This condition can be detected when the message is earlier than all events in the acknowledgment timestamp vector. Figure 5.9 shows show this could be implemented.

The acknowledgment vector ack_p maintained at principal p is updated during every successful anti-entropy session. In each session, p acknowledges that it has received a set of messages by setting the acknowledgment timestamp for itself, $\mathsf{ack}_p(p)$. The minimum timestamp in $\mathsf{summary}_p$ acknowledges every message with a lesser timestamp. As a consequence of Lemma 5.10, as long as every principal regularly performs anti-entropy sessions, every timestamp in every summary vector will eventually pass the timestamp for a given message m. Every principal will eventually acknowledge that it received m when it sets its acknowledgment timestamp to a value greater than the timestamp on m. Just as with messages, this acknowledgment (or possibly a later one) will eventually propagate to every other principal, and every principal will learn that m has been received and acknowledged everywhere.

After a message m has been acknowledged everywhere, no principal can have yet to receive m, and so its deletion will not affect message receipt. Since every message is eventually acknowledged, every message will eventually be purged.

When the logical communication topology of the network is not complete, that is, if $G_T \subset C^{|P|}$, then there may be the possibility of purging log entries early at some principals. A message sent at time t can safely be purged at principal p if all the neighbor principals of p have acknowledged messages up to and including t. This is especially advantageous in systems where many principals have few neighbors, such as rings, trees, meshes, and lattices. In the Internet, it means that principals on closed subnets need only wait for each other and gateway principals, and not for principals on other parts of the Internet.

5.4 Extensions

The basic timestamped anti-entropy protocol can be extended in several ways. There are many ways that a principal can select a partner for anti-entropy sessions; TSAE can be combined with unreliable multicast; there are techniques to improve performance after transient failures; and the protocol can be modified to tolerate unsynchronized clocks and to reduce space requirements.

5.4.1 Selecting a session partner

There are several possible policies for selecting a partner for an anti-entropy session. Table 5.1 lists eight of them. The proof in Section 5.2 only requires that every neighbor eventually be contacted to ensure that messages are delivered reliably, and weaker constraints can work for some network topologies.

The policies can be divided into three classes: random, deterministic, and topological. Random policies assign a probability to each neighbor, then randomly select a partner for each session. The deterministic policies use a fixed rule to determine the neighbor to select as partner, possibly using some extra state such as a sequence counter. Topological policies organize the principals into some

TABLE 5.1: Partner selection policies.

Random policies:	
Uniform	Every neighbor principal has an equal probability of being randomly selected.
Distance-biased	Nearby neighbors have a greater probability than more distant neighbors of being randomly selected.
Oldest-biased	The probability of selecting a neighbor is proportional to the age of its entry in the summary vector.
Deterministic policies:	
Oldest-first	Always selects the neighbor n with the oldest value $\mathsf{summary}(n)$.
Latin squares	Builds a deterministic schedule guaranteed to propagate messages in $\Theta(\log n)$ rounds.
Topological policies:	
Ring	Organizes the principals into a ring.
Binary tree	Principals are organized into a binary tree, and propagate messages randomly along the arcs in the tree.
Mesh	Organizes the principals into a two-dimensional rectangular mesh.

fixed graph structure such as a ring or a mesh, and propagate messages along edges in the graph. Chapter 7 examines the performance implications of different policies.

The **uniform** policy assigns every neighbor an equal probability of selection, and selects randomly from them. This is a simple policy that meets the correctness requirement of contacting every neighbor.

Uniform selection can lead to overloaded network links in an internetwork where the physical topology is less connected than the logical. Demers et al. compared uniform to **distance-biased** selection for the Clearinghouse [Demers88]. Their study found that biasing partner selection by distance could reduce traffic on critical intercontinental links in the Xerox Corporate Internetwork by more than an order of magnitude. Selection can also be biased by the cost of communication, perhaps measured in terms of latency, or monetary cost of using a communication link.

Alon et al. [Alon87] proposed the **latin square** policy, which guarantees that a message is received by all principals in $O(\log n)$ time (assuming no principal failure). A latin square is an $N \times N$ matrix of N entries, where every row and column includes every entry once. The policy builds a communication schedule by constructing a random latin square, where the columns in the matrix are the schedules for each principal. A principal cycles through its schedule, contacting partners in the order given, and skipping over itself. It is not evident how to take advantage of topological information in this approach. It is also not clear how to extend it for dynamically changing principal groups without perform a consistent computation to build new schedules (Chapter 6), since each principal must build and follow the same schedule for selecting partners.

The **oldest-biased** and **oldest-first** policies attempt to produce the same effect as **latin squares** without computing a global schedule. **Oldest-biased** randomly selects a partner with probability proportional to the age of its entry in the summary vector. **Oldest-first** always selects the oldest entry, breaking ties by selecting the "closer" entry if it can be determined.

The topological policies, including **ring, binary tree,** and **mesh,** organize principals into a regular graph. Messages are propagated along edges in the graph. A topological policy can work well when its structure can be mapped onto the structure of the network.

5.4.2 Principal failure and volatile storage

Principals fail temporarily. When they recover they must recover their state from stable storage. If a principal takes any significant amount of time to repair and recover, it will likely be out of date by the time recovery is complete. It would be appropriate for the principal to immediately perform an anti-entropy session with another principal, to bring itself up to date. The process of purging message logs will have been delayed at other principals while the principal was unavailable, and so an immediate anti-entropy will update the recovering copy and allow other sites to begin purging their logs.

The proofs in Section 5.2 rely on messages and summaries being maintained on stable storage. If not implemented properly, using disk to approximate stable storage can be slow and can interfere with other operations on the host. The usual Unix approach is to use *delayed writeback* to avoid synchronous disk activity. Messages can be lost if a host fails before these data are written to disk. The analysis in Chapter 7 shows that the probability of this happening is negligible for well-written systems. However, in practice the update rates of many wide-area systems will be small enough that this problem is unimportant. Further, careful implementation can avoid most of the expense anyway [Birrell87].

5.4.3 Combining anti-entropy with unreliable multicast

Timestamped anti-entropy can be combined with an unreliable multicast to spread information rapidly. When a principal first sends a message, it can multicast it to other principals. Some of them will not receive the multicast, either because the network did not deliver the message or because the principal was temporarily unavailable. These principals will receive the message later when they conduct an anti-entropy session with another site that has received the message. This can speed dissemination when message order is not important.

The combination of unreliable multicast and TSAE is somewhat like the selective retransmission technique used in network protocols that implement reliable streams [Tanenbaum81, Section 4.2]: the TSAE protocol delivers messages that the receiver has missed. However, as the TSAE protocol was presented in Figure 5.7, a principal only considers summary vectors when deciding whether to transmit a message to a partner, regardless of whether the partner has already received the message by multicast. These duplicate transmissions are wasteful, and can be reduced by performing more accurate checks before sending a message.

Recall that a principal has received every message with timestamps earlier than the corresponding entry its summary vector. If TSAE is being used in isolation, it will have received no messages timestamped later than the summary vector entry. However, a multicast can deliver a message with a later timestamp. Messages timestamped later that the summary vector have been delivered to a principal *early*.

The problem of eliminating unnecessary retransmissions is to detect early messages. The obvious way to detect these messages would be for each principal to transmit a list of the identifiers of the early messages it has received along with its summary vector at the beginning of an anti-entropy session. However, in some applications the size of a message identifier may be a large fraction of the average message size, so transmitting the list would produce nearly as much network traffic as simply transmitting the early messages. One technique that is both simple and more efficient would be to exchange a checksum of all the early messages a principal has received. Each principal would first compute a checksum of its early messages, and send the checksum to its partner along with a copy of its summary summary vector. After receiving the partner's summary

```
class checksumVector {
      set⟨ partition, count, checksum ⟩ checksums;

      // add a message to the checksum for one partition
      addMessage(partition, message);
      // determine whether the checksum for a partition is different from
      // the checksum for that partition in another vector
      Boolean different(partition, checksumVector);
}
```

FIGURE 5.10: The checksum vector data type.

vector and checksum, the principal would compute a checksum for the messages in its log that are timestamped later than the partner's summary vector. If the received and computed checksums are the same, no messages need to be send to the partner.

This approach works well only when the probability of missing a multicast message is low. Even if the partner has missed only a single message, the principal will have to send every message that is timestamped later than the partner's summary vector. The simple approach can be improved by partitioning the messages, and computing checksums for each partition.

There are several ways to partition messages. One way is to apply a known hash function to the messages to divide them into a fixed number of partitions. A simpler approach is to group messages by their sender.

Once the messages have been partitioned, the participants in the anti-entropy session compute and exchange a checksum vector (Figure 5.10) to summarize the early messages they have received. The checksum vector is similar in structure to the summary vector, containing one checksum and a count of messages for each partition.

The modified TSAE protocol for the originator is shown in Figure 5.11. The partner's protocol is similar. The protocol has been augmented by adding steps to compute the checksum vectors and exchange them with the partner principal. Before each message is sent, the protocol checks whether both principals have received the same set of messages. If so, there is no need to send the message.

```
principalId partner;
timestampVector partnerSummary, partnerAck;
timestampVector localSummary, localAck;
checksumVector localCksum, partnerCksum, localPartnerCksum;
msgList messages;

partner = selectPartner();

// update local vectors and exchange them with partner
summary[thisPrincipal] = CurrentTimestamp();
localSummary = summary;
ack[thisPrincipal] = minElement(localSummary);
localAck = ack;
send(partner, "AE request", localSummary, localAck);
receive(partner, partnerSummary, partnerAck);

// compute and exchange checksum vectors
messages = log.listNewer(localSummary);
for ⟨ pid, timestamp, message ⟩ in messages:
      localCksum.addMessage(partition(message), message);
send(partner, localCksum);
receive(partner, partnerCksum);

// compute checksums on messages newer than partner's
// summary vector.
messages = log.listNewer(partnerSummary);
for ⟨ pid, timestamp, message ⟩ in messages:
      localPartnerCksum.addMessage(partition(message), message);

// exchange messages
for ⟨ pid, timestamp, message ⟩ in messages:
      if (localPartnerCksum.different(partition(message),partnerCksum)) then
        send(partner, pid, timestamp, message);
while (receive(partner, pid, timestamp, message)):
      log.add(pid,timestamp,message);

// finish communication
send(partner, "Acknowledged");
receive(partner, "Acknowledged");

// update summaries and trigger the message ordering component
summary.updateMax(partnerSummary);
ack.updateMax(partnerAck);
DeliverMessages();
```

FIGURE 5.11: Originator's protocol for TSAE combined with unreliable multicast. Partitions early messages using a partition function, and computes a checksum vector over the partitions to avoid retransmitting messages.

This mechanism does not completely eliminate duplicate transmissions, but it can significantly reduce them. Consider a principal that sends and multicasts a number of new messages, then is disconnected from the network before it can perform anti-entropy. No principal can advance its summary timestamp vector to include the multicast messages because it cannot perform anti-entropy with the sender, so they will always be eligible for exchange in the basic TSAE protocol. The checksum method allows principals to detect that these messages need not be exchanged at the cost of computing and exchanging the checksum vectors.

5.4.4 Anti-entropy with unsynchronized clocks

The TSAE protocol as presented requires loosely-synchronized clocks so that each principal can acknowledge messages using a single timestamp (Figure 5.5). If clocks are not synchronized, the clock at one principal may be much greater than the clock at another. If the minimum timestamp were selected to summarize the messages a principal has received, messages from the principal with the fast clock might never be acknowledged.

A principal's summary vector is a more general and exact measure of the messages that have been received. If the entire summary vector is used as an acknowledgment, then clock values from different hosts need never be compared.

To use summary vectors for acknowledgment, each principal must maintain a two-dimensional acknowledgment matrix of timestamps, as shown in Figure 5.12. The summary vector is part of the acknowledgment matrix: the ith column in the matrix is the summary vector for the local principal p_i. Other columns are old copies of the summary vectors from other principals.

The unsynchronized-clock version of the TSAE protocol is little different from the synchronized-clock version. During anti-entropy sessions, principals exchange the entire matrix and update the entire matrix using an elementwise maximum at the end of a session.

The only other difference arises when the message ordering component is called upon to determine whether a message has been acknowledged by every principal. Consider a message sent from principal p at time t. A principal q knows that every other principal has observed the message when every timestamp in the p row of the message vector at q is greater than t.

Principal A

A	B	C
12	3	3
2	2	2
4	1	2

Principal B

A	B	C
3	3	3
2	11	6
1	2	2

Principal C

A	B	C
3	3	3
2	6	6
2	2	8

FIGURE 5.12: Summary and acknowledgment data structures for TSAE for unsynchronized clocks. The dark column is the summary timestamp vector, while the other columns are snapshots of the summary vectors from other principals.

The acknowledgment matrix requires $\Theta(n^2)$ space, which is not useful if the principal group is to include thousands of members. The techniques in the next section can be used to reduce the space requirement. In addition, the space can sometimes be dramatically reduced by noting that there is no need to store a *row* of the acknowledgment matrix unless the principal associated with the row has messages in the message log. If s principals are sending messages, the storage space is then $O(sn)$.

The unsynchronized-clocks TSAE protocol was developed independently by Agrawal and Malpani [Agrawal91]. However, their work did not consider the effects of dynamic group membership.

5.5 Message ordering

The message ordering component ensures that messages are delivered from the message log to the application in order (Section 3.4). It also purges log entries, as discussed in Section 5.3. Both these operations use the timestamps in the summary vector and on messages to compute the order.

Other distributed protocols that support strong message orderings, such as Psync (Section 4.6), Lazy Replication (Section 4.9), or the Orca unreliable multicast RTS (Section 4.3), append a number of timestamps to each message. The TSAE protocol reduces this overhead by transmitting messages in batches, and appending some of the necessary temporal information (in the form of the summary vector) to the entire batch, rather than to individual messages.

Section 3.4 listed five possible message orderings. A *total, causal* ordering ensures that every principal receives every message in the same order, and that the order respects causality. A *total, noncausal* ordering only ensures that every principal receives messages in the same order. A *causal* ordering ensures that a message that is causally dependent upon another message is not delivered first, but allows messages that are unrelated to be delivered in any order. A *FIFO* ordering delivers messages sent by each principal in the order they were sent, but makes no guarantees about the interleaving of streams of messages from different principals. Finally, it is possible to guarantee no particular order.

It is trivial to construct a message ordering component that provides no ordering guarantees: messages can simply be delivered when they are received.

Unordered delivery works well when TSAE is combined with an unreliable multicast. However, the implementations for all the other orderings in this section rely on the reliability and batching properties of TSAE, and therefore cannot make use of unreliable multicast.

A component that delivers messages in a per-principal FIFO order is only slightly more difficult than unordered delivery. At the end of every anti-entropy session, a principal has received a batch of zero or more messages sent by another principal. Further, all the messages received in the batch follow the messages in previous batches, precede all messages in later batches, without gaps anywhere in the sequence. To deliver messages in FIFO order, therefore, the ordering component only needs to sort messages in a batch by timestamp and deliver batches in the order they occur. Figure 5.13 shows a way to implement this.

Total, noncausal orders are only slightly more complex, as long as clocks are loosely synchronized. As with the FIFO ordering, messages are sorted and delivered in timestamp order. However, it is necessary to delay delivery until the ordering component can be sure that no messages with lesser timestamps will be received. No messages will be received with timestamps greater than the minimum timestamp in the summary vector. The principalId is used to break ordering ties. The function in Figure 5.14 presents this approach.

```
deliveredMessages: timestampVector;

DeliverMessages()
{
      timestampVector localSummary;
      principalId pid;
      msgList messages;

      localSummary = summary;
      for each pid in group:
            messages = log.listMsgs(pid,deliveredMessages(pid),localSummary(pid));
            sort messages by timestamp;
            for ⟨ pid, timestamp, message ⟩ in messages:
                  log.deliver(pid, timestamp);

      deliveredMessages.updateMax(localSummary);
      PurgeLog();
}
```

FIGURE 5.13: Function to deliver messages in per-principal FIFO order. Each anti-entropy session produces a batch of messages from each principal. They are arranged into a FIFO order by sorting by timestamp.

This ordering method is slightly biased, in that messages from principals with slow-running clocks are delivered before messages from principals with faster-running clocks. Since clocks are assumed to be loosely synchronized to within some ϵ, this bias is limited. Further, if the time between updates is greater than ϵ the bias has no effect.

The simple total order will not respect potential causal relations unless the timestamps appended to each message already reflect causality. Since system clocks are unlikely to include causal information, they can be augmented by maintaining a *logical clock* [Lamport78] at each principal. A logical clock is a counter that is incremented every time a principal sends a message or performs an anti-entropy session. The counter is appended to every message or anti-entropy session. Every time a counter value is received from another principal in a message or anti-entropy session, the local counter is set to a value larger than the counter in the message. In this way, if there is a potential causal relation between two events, then the timestamp for one event will be greater than the other. (Note that the converse is not true: a relation between two timestamps does not imply potential

```
lastTimestamp: timestamp;

DeliverMessages()
{
    timestamp localTimestamp;
    msgList messages;

    localTimestamp = summary.minElement();
    messages = log.listMsgs(ANY,lastTimestamp,localTimestamp);
    sort messages by timestamp, and by pid within timestamp;
    for ⟨ pid, timestamp, message ⟩ in messages:
        log.deliver(pid, timestamp);

    lastTimestamp = localTimestamp;
    PurgeLog();
}
```

FIGURE 5.14: Function to deliver messages in a total order. No messages can be received with timestamps less than the minimum entry in the summary vector, so any undelivered messages in the log timestamped earlier than the minimum are delivered in timestamp order.

---

causality.) The existing timestamp class is easily extended to include a logical clock counter as well as a system clock sample. A total, causal ordering is obtained by sorting and delivering messages in logical clock order.

Causal, but not total, orderings are used when causal relations are important, and messages should be delivered as early as possible. There is a simple implementation of this ordering that requires slight modifications to the interface to the DeliverMessages function and to the message log. The implementation is only correct when TSAE is used in isolation, and not with an early-delivery mechanism like an unreliable multicast.

Consider a message being transmitted during an anti-entropy session. Assume that the originator of the anti-entropy session is also the principal that sent the message to the group. During the anti-entropy session, the originator will send to the partner every other message it has received; in particular, it will send every message on which the message in question could be causally dependent that the partner has not yet received. Furthermore, the partner could not yet have received any

```
DeliverMessages(msgList messages)
{
      sort messages by delivery timestamp;
      for ⟨ pid, timestamp, delivery timestamp, message ⟩ in messages:
            log.deliver(pid, timestamp);

      PurgeLog();
}
```

FIGURE 5.15: Function to deliver messages in a causal order. This implementation relies on the way TSAE delivers batches of messages. It requires that the message log be modified to maintain two timestamps for each message: one set by the message sender, and another set whenever the message is delivered. The delivery timestamp is used to reconstruct causal relationships within a batch of messages.

---

messages that are causally dependent upon the message, because it would already have received the message when it received the dependent message.

This property can be exploited to ensure that messages are delivered in a causal order. Batches can be delivered upon receipt, because there is no need to wait for messages on which the messages in the batch could depend. However, the partner must deliver messages in a way that respects causality *within* the batch.

Figure 5.15 shows how this can be done. It requires two modifications to the system as it has been defined: messages must be timestamped twice, once by the sender and once when they are delivered, and the anti-entropy session must pass a list of the messages in each batch to the DeliverMessages function. When a message is delivered from the message log to the application, the ordering component must timestamp it. When the message is sent to the partner in an anti-entropy session, the delivery timestamp is transmitted as well. The partner then uses the delivery timestamp to order the messages in the batch so that messages are delivered after the ones on which they are dependent. The partner will overwrite the local delivery timestamp on the message when it does so.

The Lazy Replication system, which uses a mechanism similar to TSAE, allows applications to specify *external* causal consistency constraints. That is, applications can specify causal constraints created using some system other than the group. Applications can present a timestamp vector along with messages, where the vector summarizes the messages that must be delivered first. This vector could potentially include timestamps from principals outside the group.

The Psync protocol can take advantage of commutativity between different kinds of messages. The message ordering component must be provided with commutative information to be able to make this possible.

5.6 Summary

This chapter has introduced complementary implementations of the message delivery and message ordering components. The message delivery component uses the timestamped anti-entropy protocol, which provides reliable, eventual delivery. Several message ordering implementations were listed, which provide a range of ordering possibilities. Together they provide weak-consistency group communication.

The timestamped anti-entropy (TSAE) protocol performs periodic exchanges of messages between pairs of principals, called anti-entropy sessions. The sessions deliver batches of messages so that each batch immediately follows any earlier batches the principal has received, and so that there are never "gaps" in the message sequence. This property can be exploited by build simple mechanisms to summarize the messages that a principal has received and to propagate message acknowledgments throughout the group.

A reliable, eventual message delivery protocol is correct if every principal eventually receives a copy of every message sent to the group. The TSAE protocol has been proven to meet this criterion, even when the underlying network is not a completely-connected graph. Furthermore, there is a simple algorithm for purging messages from the message log based on acknowledgments that does not interfere with message propagation.

The basic TSAE protocol can be customized in several ways. There are several policies that can be used to select a partner for anti-entropy sessions. Anti-entropy can be combined with an unreliable multicast that will delivery messages more rapidly than TSAE will propagate them. Finally, there is a generalized version of the TSAE protocol that work when clocks are not loosely synchronized, though at the expense of extra state at each principal.

The message ordering component can be implemented using one of several algorithms, providing ordering guarantees ranging from total, causal orders to no ordering. Most of the implementations are very simple, using timestamp information in messages and the batching behavior of TSAE to advantage. A causal non-total ordering is slightly more complex, but much simpler and more efficient than other systems that must attach complex causal information to each message.

Chapter 6
Group membership

A *group membership* mechanism is the final component in a group communication system. This mechanism allows principals to join and leave a group dynamically. For example, consider a replication service in which each data item is to be stored at some number of replicas. If that number increases or some of the replicas are removed from service, other replicas will need to join the group dynamically in order to maintain the required resilience.

Every principal in the group maintains a *view* of the membership. This view is the set of principals it believes are in the group. A member will always have itself in its view. The view may also contain status information and timestamps to help coordinate updates to the set. The message delivery component uses the view to identify what principals should receive messages, and some implementations of the message ordering component use group information when determining whether a message is deliverable or not.

I have developed a new weakly consistent group membership mechanism. The mechanism allows temporary inconsistencies in the view each principal maintains of the group, in exchange for low communication overhead and fault tolerance. The mechanism only allows a principal to be a member of one group at a time.

A group comes into existence when the first member initializes it, and ceases to exist when the last member leaves. Each group has a unique identity, so every newly-initialized group is distinct from all other groups, past or present. A principal becomes a member by executing a **join** protocol with one or more group members. Those members, which *sponsor* the new principal, provide it with a copy of the application-maintained group state as well as a copy of the data structures used by the group communication system. Principals stop being members by executing a **leave** protocol.

For completeness, I also consider how the group membership system can **eject** a member that has failed, even though principals have been assumed to be free of failure.

In contrast to the weak consistency mechanism presented in this chapter, other group membership mechanisms ensure greater consistency of group views at the expense of latency and communication overhead. Both the ISIS system [Birman87, Birman91] and a group membership mechanism by Cristian [Cristian89] are built on top of atomic broadcast protocols, and hence provide each principal with the same sequence of group views. Ricciardi [Ricciardi91] is investigating an alternative group membership mechanism for Isis that does not use the underlying atomic broadcast. However, it uses two- and three-phase commit protocols to maintain consistent group views. The Arjuna system [Little90] maintains a logically centralized group view via atomic transactions.

This chapter begins by considering how dynamic principal group membership interacts with message reliability. The reliability guarantee developed in Chapter 5 assumed static group membership. This is followed by definitions of correctness and fault tolerance for membership views. Finally, the protocols for initializing, joining, and leaving a group are presented, and they are shown to maintain correct and fault-tolerant views.

6.1 Message delivery and dynamic membership

The timestamped anti-entropy (TSAE) protocol presented in Chapter 5 provides reliable, eventual message delivery for a static member principal population. A reliable delivery guarantee implies that every principal eventually receives every message. Likewise, the TSAE protocol provides mechanisms to detect when every member principal has acknowledged a message. These merit special definition if the group membership is changing.

Consider a message sent at time t. As before, *time* means time as measured by an external observer. The simplest definition of reliable delivery is that every principal that is a member at t must receive a copy of the message. But what about principals that join after it is sent? And what of principals that leave before they can receive the message?

One answer to these questions is to order group operations along with application messages. This was the approach taken in Isis [Birman87, Birman91]. If messages are delivered in a total order, then the set of member principals is well-defined and consistent at every principal whenever a message is delivered. Principal join and leave operations will be delayed while other messages are being propagated. A principal that joins after a message is sent will be given a copy of the group state that includes the effects of the message. A principal that tries to leave the group after a message is sent will receive the message before it can finish leaving.

There are problems with this approach. The membership is only well-defined if a total message ordering is applied. Isis, for example, enforces extra message ordering constraints when the group membership changes. Applications that could otherwise use a weaker ordering will have to wait unnecessarily for message delivery. In addition, it will cause group operations to be delayed so they can be ordered with respect to messages from other principals. New principals will have to wait for their request to be delivered before they can begin participating in the group.

The Refdbms and Tattler systems use a different approach. A message sent at time t will eventually be received by every principal that is a member at t, even if the principal that sends the message does not yet know the other principal has joined. Any principal that joins after t will either receive the message, or will receive a copy of the group state that reflects the contents of the message. This definition allows a principal to join immediately, though it requires somewhat longer delay on leaving than the total-ordering approach.

The message delivery component records acknowledgments so that other components can detect when a message has been delivered everywhere. This is used to determine when messages can be purged from the log. If some principal purges a message too early, it will not be able to propagate that message to another principal that has not yet received it. The method in Section 5.3 purges a message when the acknowledgment timestamp for every principal in the view is later than that of the message. The membership protocols in Section 6.4 do not compromise this method.

6.2 Correctness

As noted earlier, other group membership protocols provide *consistent* views of group membership; that is, every group member observes changes to the membership in the same order. The weak consistency protocols guarantee that all principals will eventually converge to a single consistent view if all membership changes cease. More specifically, given the last such change in membership at time t, the probability that two principals, p and q, disagree on group membership at time $t + \delta$ goes to zero as δ increases. The same condition holds with respect to each principal's view of the membership status of individual principals. To avoid pathological situations, the set of principals in the universe is assumed to be finite.

The view information for one group can be represented as a *view relation* $V = P \times P$. For principals p and $q \in P$, there will be a pair (p, q) in the relation if q is in p's view, and a pair (q, p) if p is in q's view. There will be an pair (p, p) for every principal that believes it is part of the group. The set of such principals is the *member* set M.

The structure of the view relation is *correct* at any time iff all principals can eventually converge to the same view. One principal can only propagate information to another when it has the other principal in its view. Therefore, every member principal should be in the transitive closure of every other member's view.

Definition 6.1 (View correctness) *Let M be the set of principals $p \in P$ for which there is a pair (p, p) in the view relation V; that is, the set of group members. Let V^* be the transitive closure of V. V is* correct *iff* $(\forall p \in M)(\forall q \in M)((p, q) \in V^*)$.

Every group operation must preserve the correctness of the view relation. As part of this, whenever a principal joins or leaves the group, it must find other principals to act as its *sponsor*. The sponsors are the source of the new member's state, and membership information propagates from the sponsors to other existing members.

6.3 Fault tolerance

The membership system should be able to withstand some number k of simultaneous permanent principal failures without compromising the correctness of members' views. If principals never fail permanently, then the TSAE delivery protocol coupled with the membership protocols in Section 6.4 will work correctly if every principal obtains one sponsor when they join. However, if principals can permanently fail then it is possible to obtain an incorrect view relation. A view relation that can withstand up to k simultaneous principal failures and still be correct is called *k-resilient.*

Note that this definition of resilience is only concerned with the correctness of group membership information. Principal failures can cause a message to be lost if the message has not propagated to other (non-faulty) principals. However, k-resilience implies that messages *will* propagate from a non-faulty sender to other non-faulty principals in the face of k failures.

The fault tolerance arguments in this chapter apply only to systems implemented on networks with a completely connected logical topology. While the extension to non-completely connected networks is important, it complicates the exposition of the protocols and does not contribute to the basic analysis.

In the analyses that follow, it is sometimes useful to treat the view relation as a directed graph $K = (P, V)$ where the vertices are principals and the edges are pairs in the view relation.

Lemma 6.2 *For any two members $m, m' \in M$, a correct view digraph K can be viewed as a flow graph with source m and sink m'. The view relation V is k-resilient if, for all members $m, m' \in M$, the minimum vertex cut of the associated flow graph with source m and sink m' in K is at least* $k + 1$.

> *Proof:* If the minimum vertex cut of the flow graph from m to m' is at least $k + 1$, then up to k vertices can be removed from K while maintaining a path from m to m'. Call the set of failed principals F; $|F| \leq k$. Since there is a path from m to m' in $K - F$, there is an edge (m, m') in the transitive closure of $K - F$. Since this condition holds for all pairs of members, the view relation is still correct after removing the failed principals F. $\square$

The membership protocols defined in the next section ensure that this condition holds by constructing a $(k + 2)$-clique in the view graph around every member as it joins, and ensures that every principal remains part of a $(k + 2)$-clique as principals leave or fail.

6.4 Protocols

The group membership protocols allow for creating a new group, joining and leaving a group, and handling member failure. Each protocol will be presented in this section, along with a discussion of how these protocols provide k-resilience, and why they do not interfere with the reliable message delivery.

There are four group protocols. **Initialization** creates a new group. **Join** adds a new principal to a group, and transfers a copy of the group state to the new principal. A member can voluntarily **leave.** The group can **eject** a failed member, then recover from the failure.

The membership protocols *preserve correctness* if they transform one correct view relation into another. They are k-resilient if they transform one correct, k-resilient relation into another.

6.4.1 Data structures

Each member principal maintains a view of the group membership, defining the set of principals it believes are a part of the group. The view includes a set containing the identity of each member, its status, and a timestamp on that status. Figure 6.1 shows the details. In a practical implementation such as Refdbms, the membership view, summary vector, and acknowledgment vector can be stored together.

The view data structure includes a groupId, which uniquely identifies the group. The details of how the group identifier is constructed have been omitted to avoid cluttering the presentation. The type need only support initialization and checking for equality with another group identifier. A timestamp would be sufficient.

Updates to the view are propagated using information in the view set, rather than by keeping an explicit log of group membership changes. This implies that *death certificates* must be maintained after a principal has left the group (Section 3.3.1) until every principal has observed the change. Death certificates are purged in just the same way as the message log (Section 5.3).

Every principal in the view has a status. A principal that is part of the group will have status member, while one that has voluntarily left the group will have status left. A failed principal is marked as failed. The left and failed records are death certificates.

```
class groupId {
      // an identification key
}

typedef enum { member, leaving, failed, pendingMember } memberStatus;

class memberEntry {
      principalId pid;
      memberStatus status;
      timestamp t;
}

typedef set⟨memberEntry⟩ memberSet;

class memberView {
      groupId group;
      memberSet members;

      // maintain the view
      add(principalId, memberStatus, timestamp);
      update(principalId, memberStatus, timestamp);
      delete(principalId);
      merge(memberView);
      // list entries
      memberSet listLater(timestampVector);
      memberSet listEarlier(timestamp);
}

memberView view;
```

FIGURE 6.1: The group membership view data structure.

The timestamp for principal p records the clock at p, clock(p, t) when the principal entered its current state. For failed status the timestamp can either be infinity, or an approximation obtained from the clock at another principal. The approximation must be distinct from and later than any clock value from the failed principal. This will be taken up in more detail in Section 6.4.5.

```
InitializeView(groupId gid)
{
    view.group = gid;
    view.add(thisPrincipal,member,0);
    summary.update(thisPrincipal,0);
    ack.update(thisPrincipal,0);
    inform location service;
}
```

FIGURE 6.2: Initializing a new group. Code for creating the group identifier has been omitted for clarity.

6.4.2 Initializing a new group

A principal can create a new group by performing an **initialization** operation. This has two effects. First, it creates a new group identity. Second, it sets up a membership view at the principal. The view contains only the initializing principal, with status member at time zero (Figure 6.2).

Since there is only one principal in the group, there are no concerns about consistency or failure resilience. Until k additional members have joined the group, there is no possibility of k-resilience. The **join** protocol presented in the next section will form a complete view graph when the membership is small (i.e. $n \leq k + 2$).

A newly-created group will need to be registered with a location service so that other principals can locate it. The **join** protocol expects to obtain a list of possible group members from the location service. The location service is discussed further in Chapter 8.

6.4.3 Group join

A principal joins a group by finding one or more group members, then contacting them until it has obtained enough *sponsors* among the current membership to ensure k-resilience. Anti-entropy sessions will eventually propagate information about the new member throughout the group. As noted in Chapter 2, this protocol assumes the existence of a fault-tolerant location service. The location service must always provide at least one location that is a member. Principals leaving the group will return forwarding addresses to other members.

To keep the view relation k-resilient, a principal joining the group must obtain $k + 1$ sponsors before becoming a full-fledged group member. If this is impossible because the group is too small, the graph is kept as resilient as possible by using all members as sponsors. These sponsors put the principal in their view, meeting the fault-tolerance criterion. The protocol followed by a new principal is shown in Figure 6.3. The joining principal is considered to be a group member after changing its status to member. The sponsors and new member propagate the updated membership view to the rest of the group in anti-entropy sessions.

The getFirstSponsor and getAdditionalSponsor routines are the heart of the protocol (Figure 6.4). In them, the principal contacts another principal that is possibly a member, and requests that principal to sponsor it. If the principal is not a member or is leaving the group, it declines sponsorship. Each sponsor must add the new principal to its view, then send a copy of the updated view to the new member. The first sponsor must also send a copy of the application database, and the message log, summary vector, and acknowledgment vector used by other group communication components.

If a sponsor fails while the join protocol is being executed, one of two events can occur. The new member may detect the failure while trying to interact with the failed member, in which case the new member can simply try another sponsor. If the sponsor fails after the new member has finished interacting with it, the failure can be ignored because at most k of the $k + 1$ sponsors can fail.

The implementation in Figure 6.4 can be extended to allow a possible sponsor to return a forwarding address for one or more members if it is not itself a member [Fowler85, Jul88]. This allows the location service to maintain information that is less up-to-date, as long as the principals that have left the group provide forwarding addresses for some period. This topic is taken up in more detail in Chapter 8.

Theorem 6.3 *The* **join** *protocol preserves k-resilience in the view relation.*

> *Proof:* The principal j is joining a group with members M. Assume that the set $S \subseteq M$ is the set of sponsors it obtains; $|S| = k + 1$. Denote the view graph before this protocol is executed as K_M, which is assumed to be k-resilient.

```
JoinGroup(groupId gid, int nsponsors)
{
      set⟨principalId⟩ possibleMembers;
      principalId possible;
      int nfound = 0;

      // Obtain a list of possible members from the location service.
      // This list must contain at least one group member.
      possibleMembers = LocationService.find(gid);

      // Initialize the local view
      view.group = gid;
      view.add(thisPrincipal,pendingMember,0);

      // Initialize message delivery data structures
      summary.update(thisPrincipal,0);
      ack.update(thisPrincipal,0);

      // Get the first sponsor
      do {
            possible = possibleMembers.pickAndDeleteRandom();
            if (getFirstSponsor(possible,gid))
                  nfound = 1;
      } while ((nfound == 0) && !possibleMembers.empty());

      // get additional sponsors
      while ((nfound < nsponsors) && !possibleMembers.empty()) {
            possible = possibleMembers.pickAndDeleteRandom();
            if (getAdditionalSponsor(possible,gid)) {
                  nfound = nfound+1;
            }
      }

      view.update(thisPrincipal, member, CurrentTimestamp());
      // optional: initiate anti-entropy session with some other member
}
```

FIGURE 6.3: The **join** protocol followed by a new member. Error handling has been elided for clarity.

After executing this protocol, up to k failures occur in the set $F = \{f_1, \ldots, f_k\}$. Assume that $j \notin F$. The failed principals can be divided into two sets: the members of $F_S = F \cap S$ are sponsors for j, and those of $F_M = F \cap (M - S)$ are not.

For every member m that has not failed ($m \in M - F$), there is a path $m, \ldots, m_s$ in the view graph, where $m_s \in S$, since $|S| > |F|$ and the original view relation is k-resilient.

```
Bool
getFirstSponsor(principalId possible, groupId gid)
{
      memberView view;

      send(possible, "Request First Sponsorship", gid);
      receive(possible, status, otherView);
      if (status == ACCEPTED) {
            view.merge(otherView);
            transfer group state;
            transfer message log and summary vectors;
            return True;
      } else {
            // request failed or was declined
            return False;
      }
}
```

FIGURE 6.4: Obtaining the first sponsor. Obtaining additional sponsors is similar, except that the transfers of application and group communication state do not occur.

Since there is an edge (m_s, j) in the new view graph, for every functioning member m there is a path $m, \ldots, m_s, j$.

The same argument holds in reverse for paths from j to all other members in $M - F$, so the new view graph is correct after up to k failures. Therefore it is k-resilient. □

In addition, the protocol forms an n-clique in the view graph when the number of members n is not more than $k + 2$. Once a $(k + 2)$-clique is reached, k-resilience is established.

Members joining a group do not interfere with message delivery. A message sent at time t by principal A is to be delivered to every principal that is a member at t. The proof of diffusion (Lemma 5.10) can be extended by noting that the group membership at t is a fixed finite set, and by substituting the view relation for the logical communication topology in the proofs. A correct view relation is a connected graph, so every principal that is a member at t will eventually perform an anti-entropy session that is causally dependent upon the message-sending event at t. If a B joins after t, it will either receive the message or be given group state from a sponsor that already received the message.

6.4.4 Group leave

When a member leaves the group, it must not cause the view relation to become less than k-resilient or incorrect. This can happen when removing the member from the group causes the minimum vertex-cut to drop below $k + 1$ vertices. The protocol must also ensure that all messages have been propagated to other principals.

To alleviate this problem, a principal that wants to leave the group must do so in two steps.

1. The principal declares its intent to leave by changing its status to left and performing anti-entropy with one or more other members.

2. The principal then waits until all principals that were group members at the time it declared its intent have observed the status change. Then the member can destroy its state.

During this delay, the principal cannot send new messages, and so should not accept operations from clients or sponsor other principals. However, it must maintain all its state and actively participate in anti-entropy sessions with any other group member.

The first step of the protocol must complete eventually, since a principal can change its own state in minimal time and any principal can eventually complete an anti-entropy session with another.

For the second step to complete, every principal that is a member at the time of declaration t must perform an anti-entropy session that causally follows the declaration event. Since the group membership at t is fixed and finite, and the view relation is correct, it follows from Lemma 5.10 on diffusion that every member will eventually perform such an anti-entropy session.

Further, if the group is k-resilient at t and no more than k principals fail, the member that is leaving can still communicate with the remaining group members to complete the protocol.

When a member A declares its intent to leave, another other member B may already have declared its intent to leave. Principal B may collect its acknowledgments and leave the group before explicitly acknowledging the declaration from A. However, A will also detect that B collected its acknowledgments and will delete B from its view. A need then only wait for acknowledgments from the other group members.

The delay in leaving ensures that the view relation cannot become vulnerable to k or fewer failures, and that no messages sent from the principal will be lost. This also ensures that the leaving principal does not compromise the correctness condition for reliable message delivery. The delay completes when all members have acknowledged that the principal has expressed its intent to leave. This requires a chain of anti-entropy events starting after time t between it and all other members, then a chain of anti-entropy sessions from every other member back to the leaving principal. The return sessions ensure leaving principal will receive every message sent up to time t along with acknowledgement of its state change.

When the last principal leaves a group, the group ceases to exist. The location services should be so informed.

Theorem 6.4 *The* **leave** *protocol preserves k-resilience in the view relation as long as at least $k+1$ principals remain in the group.*

Proof: Consider any path $m_1, \ldots, m_n$ in the view graph. Assume some subset $\{l_1, \ldots, l_m\} \subseteq \{m_2, \ldots, m_{n-1}\}$ of these have declared their intent to leave the group. Without loss of generality, assume that l_1 is the first to find that its change has been observed by all group members, and that this occurs at time t_1. At time $t_2 \geq t_1$, the second principal l_2 will find that it has met its condition and will cease to exist. Assume that the view graph is k-resilient for $t \leq t_1$. Then for the period $t_1 < t \leq t_2$ the part of the path m_{k-1}, l_1, m_{k+1} can be replaced by the sequence m_{k-1}, m_{k+1} since m_{k-1} must have every member in its view that l_1 had. These steps can be repeated for each member as it leaves the group.

Since the graph was originally k-resilient, after some k failures there must still be some path $m_1, \ldots, m_n$ between any two principals. This path will still exist after $l_1 \ldots l_m$ leave the group. $\Box$

As with the **join** protocol, this protocol forms a n-clique when there are fewer than $k + 1$ group members.

6.4.5 Failure recovery

Principals can exhibit either temporary performance failures or permanent failures. Host rebooting, transient load, and network router failure are typical temporary failures. A principal that has permanently failed will never recover, or recovery will take so long that it might as well be forever. Extended repair, disaster, or unexpected removal from service are permanent failures.

The mechanism for detecting that a principal has failed is beyond the scope of this dissertation. Standard probabilistic failure detection mechanisms, such as timeouts, do not apply to the long-lived, crash-resilient principals used to build information services. For many applications, failure will be sufficiently rare that human detection is feasible.

The TSAE protocol already handles temporary failures. Soon after the principal recovers, it will start to perform anti-entropy sessions. Between state that was saved on stable storage and the information preserved at other principals, these operations will catch it up to the current state of the system.

Permanent failures are a different problem. Most seriously, they compromise reliable delivery guarantees. Information that has not propagated out of the failed principal may be trapped there and lost. Without remedy, this would be especially serious for group information, as it could create an incorrect view relation. For example, a single principal that has just joined the group with one sponsor can become isolated from the rest of the group if its sponsor fails.

The only way to solve this problem is to ensure that information is recorded at several principals before a group membership operation is complete. The group membership protocols in this chapter maintain a k-resilient view relation, where information is always recorded by at least $k+1$ principals.

A principal that has failed must be *ejected* from the group. Ejection proceeds as follows: when principal p finds that principal f has failed, it marks f as status failed in its view, and sets the timestamp of the status change to $\mathsf{infinity}$. This will prevent a principal from recovering and claiming to be a group member, then propagating messages that have been delivered and purged by other members. Anti-entropy sessions then propagate the failure information to all other group members. Principal f is ejected, and the failure recovered from, when all group members have observed and acknowledged its failure. As with group leaves, k-resilience is then restored.

When a principal must be ejected from the group, the view relation can become less than k-resilient, just as with group leaves. If the system were only to experience k failures over all time, the loss of one failed principal would not cause a loss of resilience; only $k - 1$ failures would be possible and $k - 1$ resilience still holds.

However, it is not possible to bound the number of failures a long-lived system will incur. The membership mechanisms in this chapter re-obtain k-resilience even though as many as k principals have already failed. As long as the number of members has not dropped below $k + 1$, anti-entropy among the members will eventually restore k-resilience. In fact, anti-entropy will eventually generate a complete graph among the members if the membership is stable long enough.

This approach leaves the group vulnerable for the time it takes to restore k-resilience. The length of time the group is vulnerable can be decreased by increasing the rate at which members perform anti-entropy. The duration, and degree, of vulnerability is the subject of Section 7.3.

6.5 Summary

The group membership component can provide weak consistency semantics: all principals will eventually see membership changes, but only one or a few see the change initially. The basic timestamped anti-entropy message delivery protocol is correct when used with this group implementation, even when the group membership changes dynamically.

The group membership protocols provide for four operations: initialization, join, leave, and failure recovery. Each of these protocols proceeds immediately at only a few principals, providing better scalability than existing consistent group membership protocols. The operations maintain a *view* of group membership at each principal that records the members that principal knows about. The views define a *view relation,* which is correct if its transitive closure is equal to the group membership.

The protocols take care to maintain a correct view relation even in the presence of a few failures. In particular, the graph is said to be k-resilient if up to k members can fail and not compromise the graph. The join and leave protocols preserve k-resilience. These algorithms handle permanent failure – as distinguished from transient failure – though it reduces the resilience of the view relation until it is restored by normal message propagation.

Refdbms uses these protocols in its group membership mechanism.

Chapter 7
Performance of weak-consistency protocols

The last several chapters have presented an architecture for a weak consistency group communication system and protocols for constructing it. How well can these protocols be expected to work? How do they compare to other approaches? In this chapter I investigate a number of performance measures.

Several of the measures are related to the guarantees provided by the components. *Message reliability* measures how often the message delivery component will fail to deliver a message because principals are removed from service without notice, while *message latency* measures how long the component takes to deliver a message. *Membership resilience* measures how well the group join and leave protocols preserve correctness and resilience.

Two additional measures are not related to any specific component. *Message traffic* indicates how many messages applications built using the TSAE protocol will generate, and indirectly how they interfere with other network activity, while *consistency* measures how up-to-date each principal can be expected to be.

I conclude the chapter with a brief comparison of the performance of the TSAE protocol with the protocols surveyed in Chapter 4.

7.1 Message reliability

The timestamped anti-entropy message delivery protocol provides reliable eventual delivery. However, reliable delivery does not guarantee that a message is delivered when its sender fails. For the TSAE protocol to fail to deliver a message, every principal that has received a copy – which includes the sender – must fail. This section examines how often this happens.

Delivery reliability can be measured by the probability that a message will be delivered to every principal before all recipients can fail. The probability is affected by the rate at which anti-entropy sessions propagate messages, and the rate at which principals fail.

In earlier chapters, principals were assumed not to fail. In practice, the only way a principal can fail is in a sudden, catastrophic removal from service – a fire or hardware failure, for example. This sort of failure is extremely rare for systems on the Internet. The analysis in this section, however, explores a wide range of failure rates.

7.1.1 Analytical modeling

Message loss can be modeled using a state transition system like that shown in Figure 7.1. Each state is labeled with a pair $\langle m, f\rangle$, where m is the number of functioning principals that have observed a message, and f is the total number of functioning principals. The system starts in state $\langle 1, n\rangle$, with one principal having observed a message out of n possible (5 in the example). The system can then either propagate the information using anti-entropy, in which case the system moves to state $\langle 2, n\rangle$, or a principal can be removed from service and the system moves into state $\langle 0, n-1\rangle$. The message has been lost when the system reaches a state $\langle 0, x\rangle$, and it is delivered when it reaches $\langle x, x\rangle$.

Anti-entropy and principal failure are treated as Poisson processes with rate λ_a and λ_f, because Poisson processes are easy to model and analyze. Real systems often follow more complex distributions, but this simplifying assumption is common practice.

The rate of *useful* anti-entropy sessions, where a principal that has received the message contacts one that has not, is a function of m, f, and the partner selection policy. In particular, f principals will be initiating anti-entropy sessions. If principals choose their partners randomly, each principal that has observed the update has a chance $(f-m)/(f-1)$ of contacting a principal that has not yet observed the update. Since anti-entropy is a Poisson process, the rate of useful anti-entropy sessions is

$$m\frac{f-m}{f-1}\lambda_a.$$

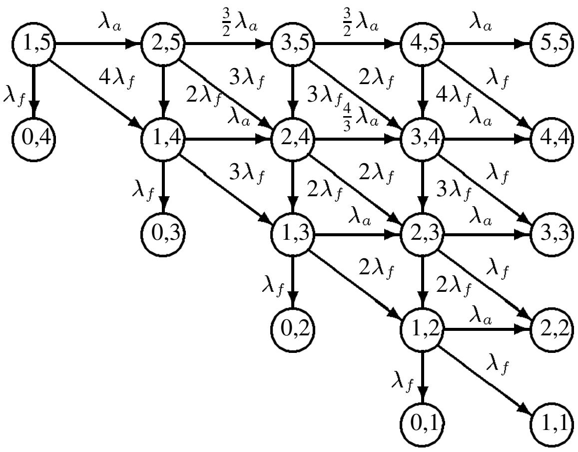


FIGURE 7.1: Model of message receipt and failure for five principals. Each state is labeled $\langle m, f \rangle$, where m is the number of functioning principals that have received the message, and f is the total number of functioning principals. This model only includes permanent failure; transient failure and recovery would add an additional dimension of states.

Since removal from service is a permanent event, the state transition graph is acyclic, with $\Theta(n^2)$ states in the number of principals. The probability p_i of reaching each state i can be computed using a sequential walk of the states. The probability density functions $p_i(t)$ of the time at which the system enters each state can be derived analytically or numerically. The analytic solution for $p_i(t)$ can be found by convolving the entry-time distribution $p_j(t)$ for each predecessor state j with the probability density of the time required for the transition from j to i. Alternately, the system can be solved numerically using a simple Monte Carlo evaluation.

7.1.2 Results

Figures 7.2 and 7.3 show the probability of removal from service interfering with message delivery for different numbers of principals. The probability is a function of $\rho = \lambda_a/\lambda_f$, the ratio of the anti-entropy rate to the permanent site failure rate. The two graphs are identical in content, but are plotted using different vertical scales to properly show the behavior for both small and large values of ρ.

Internet sites generally are removed from service after several years of service, and then usually with enough notice to run a leave protocol. The anti-entropy rate is therefore likely to be many

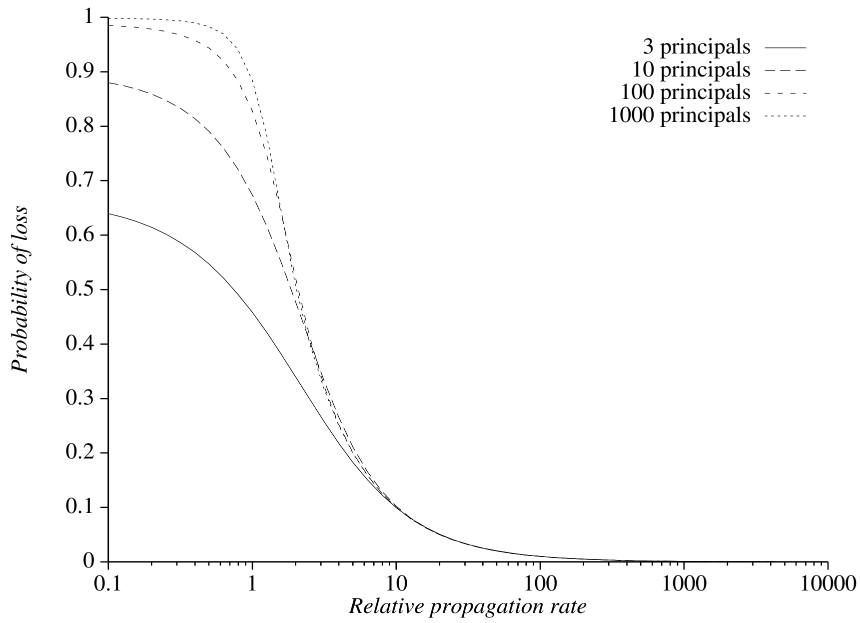


FIGURE 7.2: Probability of failing to deliver a message to all sites (linear vertical scale). The relative propagation rate ρ is the ratio of the anti-entropy rate λ_a to the permanent site failure rate λ_f. The linear scale emphasizes the effect of the number of principals for small values of ρ.

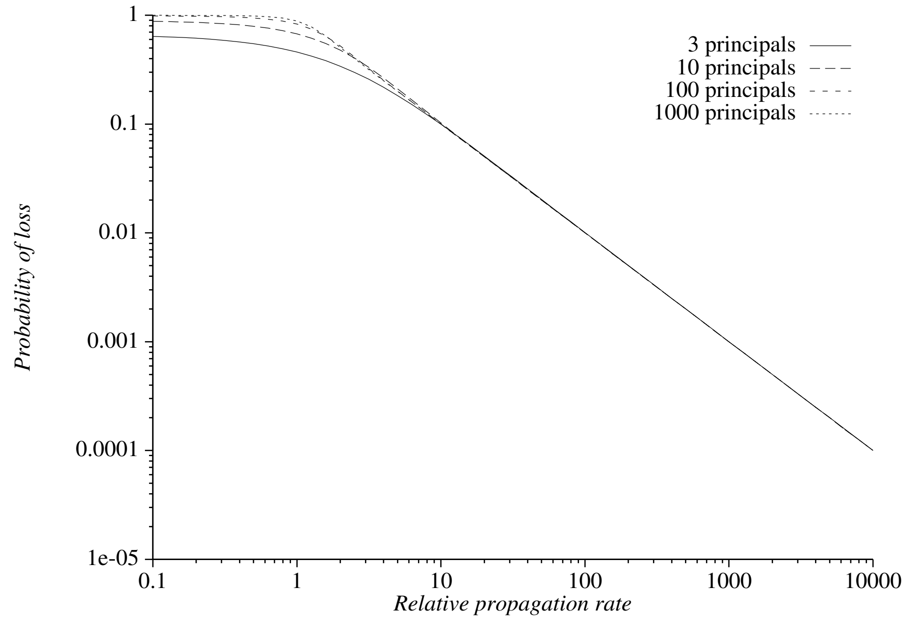


FIGURE 7.3: Probability of failing to deliver a message to all sites (logarithmic vertical scale). The logarithmic scale emphasizes the effect of large values of ρ: the probability decreases as ρ increases.

thousands of times higher than the permanent failure rate. As a result, there will be almost no messages lost because of removal from service.

7.1.3 Volatile storage

Some implementations may buffer messages in volatile storage before copying them to the stable message log. This is the default behavior of several operating systems, including the Unix file system. These implementations will lose the information in volatile storage when a principal temporarily fails and recovers.

Volatile storage complicates the state transition model. States must be labeled with four values: the number of functioning principals that have not observed a message, the number that have written it to volatile store, the number that have written it to disk, and the number that have temporarily failed. The state transitions are complex and the solution is impractical for realistic numbers of principals.

However, the effect of volatile storage can be bounded by considering the probability that a failure will occur while there are messages that have not been made stable. Assume that temporary failure is a Poisson process with rate λ_t and that volatile data is flushed to stable storage every s time units. The probability that a failure occurs before writeback is

$$p = \frac{-2e^{-s\lambda_t} + s^2\lambda_t^2 - 2s\lambda_t + 2}{2s\lambda_t^2}.$$

For a typical value of $s = 30$ seconds and $1/\lambda_t = 15$ days, p is so close to zero as to be negligible.

7.2 Message latency

The message component provides latency guarantees as well as reliability. The TSAE protocol only guarantees eventual delivery, but in practice messages propagate to every principal rapidly.

If information is propagated quickly, clients using different principals will not often observe different information, and loss of an update from site failure will be unlikely. The size of the message log is related to this measure, since messages are removed from the log when acknowledgments have been received from every principal.

7.2.1 Simulation modeling

I constructed a discrete event simulation model of the timestamped anti-entropy protocol to measure propagation latency. The latency simulator measured the time required for an update message, entered at time zero, and its acknowledgments to propagate to all available principals. The time required to send a message from one principal to another was assumed to be negligible compared to the time between anti-entropy sessions. The simulator could be parameterized to use different partner selection policies and numbers of sites. The simulator was run until either the 95% confidence intervals were less than 5%, or 10 000 updates had been processed. In practice 95% confidence intervals were generally between 1 and 2%.

The simulation modeled only the TSAE protocol, and did not consider the effect of combining TSAE with a best-effort multicast. Therefore the results in this section represent worst-cast behavior that would be improved if a multicast were added.

7.2.2 Results

Figure 7.4 shows the cumulative probability over time that a message has been received by all principals for varying numbers of principals. Time is measured as multiples of the mean interval at which principals initiate anti-entropy events. The simulations in this graph use **uniform** partner selection. The time required to propagate a message appears to scale well with the number of sites.

The time required to propagate message acknowledgments everywhere is also an important measure, because it determines how quickly messages can be purged from the message log. Figure 7.5 shows the latency required from the time a message is sent to the time acknowledgments are received by every principal from every principal. Once again the time required appears to scale well.

The partner selection policy also affects the speed of message propagation. Figure 7.6 shows the mean time required to propagate a message to every principal for several policies as the number of

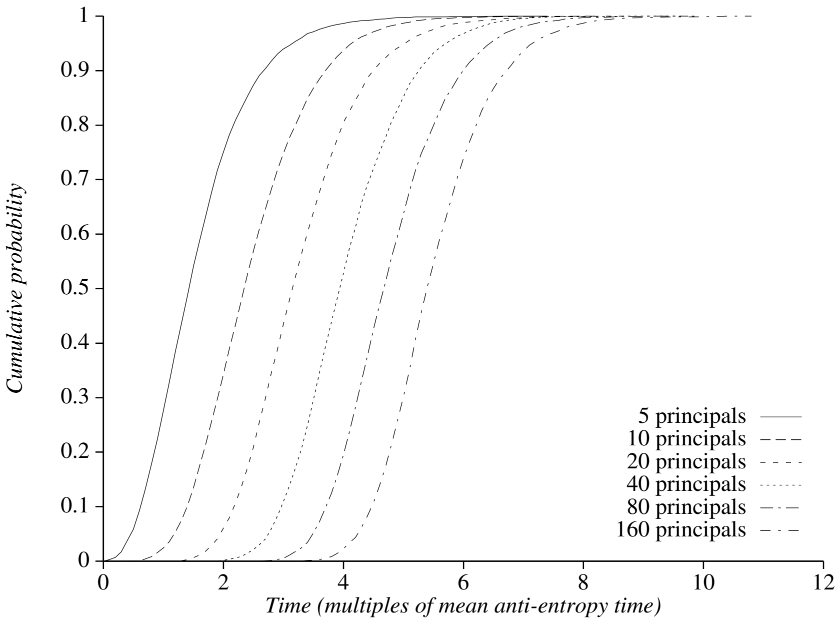


FIGURE 7.4: Cumulative probability distribution for propagating a message to all principals. Measured for **uniform** partner selection.

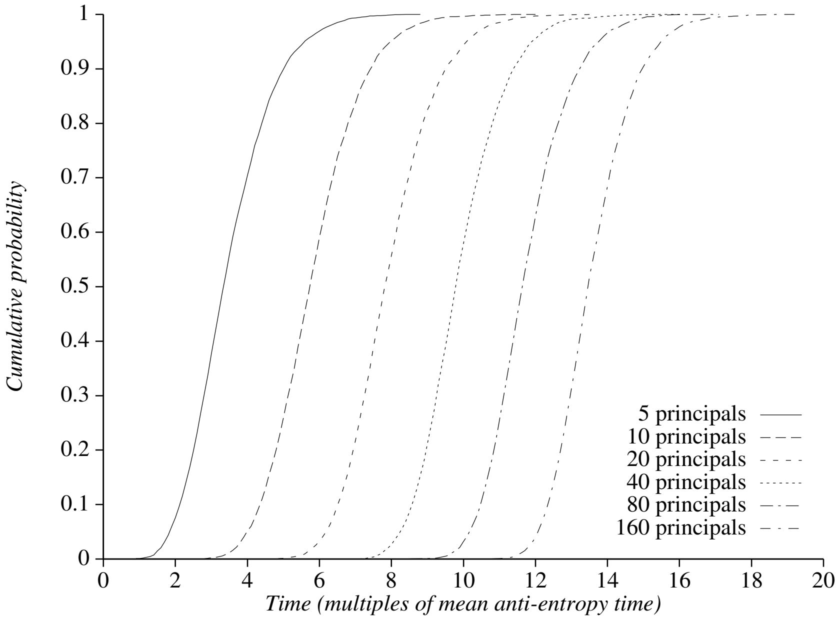


FIGURE 7.5: Cumulative probability distribution for receiving an acknowledgment from all principals. Measured for **uniform** partner selection.

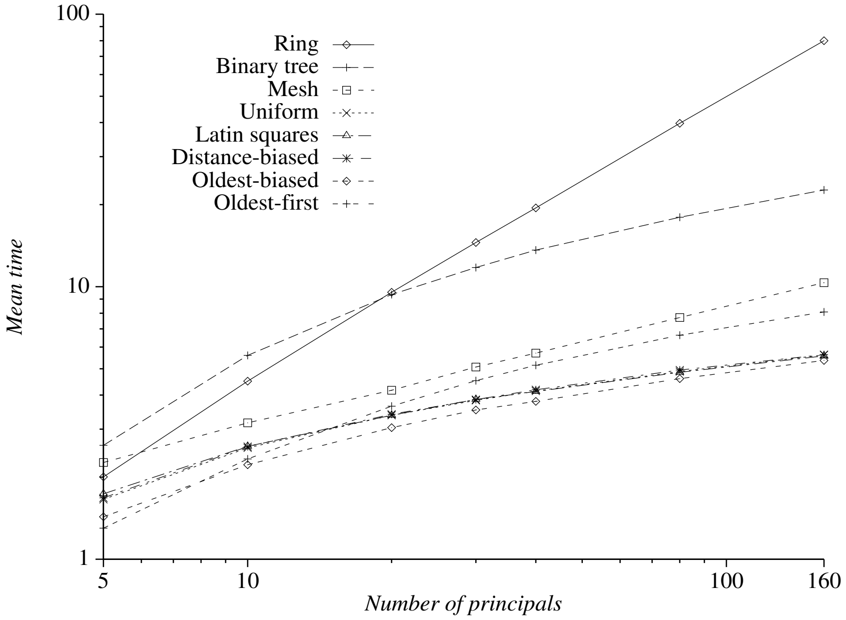


FIGURE 7.6: Effect of partner selection policy on scaling of propagation time.

---

sites increases. The **uniform, latin squares,** and **distance-biased** policies give essentially identical performance. **Age-biased** appears to provide slightly better performance, which would appear to contradict the claim by Alon that the **latin squares** policy is fastest [Alon87]. I believe the difference arises from a slight difference in implementation: Alon's implementation requires that every principal propagate messages in well-defined rounds, while this simulation allows propagation to occur at random intervals. This may mitigate some of the benefit derived from Alon's **latin squares** policy. The policies that simulate a fixed topology – **ring, mesh,** or **binary tree** – have the worst performance and scaling.

The results for acknowledgment time (Figure 7.7) are similar. Again **uniform, distance-biased,** and **latin squares** all have essentially the same latency, while **age-biased** performs slightly better. The **oldest-first** policy performs best of all for small groups, and is about the same as the random policies for large groups. The **oldest-first** policy was notably worse than these protocols for message propagation latency. Again the fixed-topology policies perform worse than all the others.

These results indicate that simple random policies, such as uniform selection or age biasing, perform quite well. **Uniform** partner selection was therefore selected for the Refdbms system.

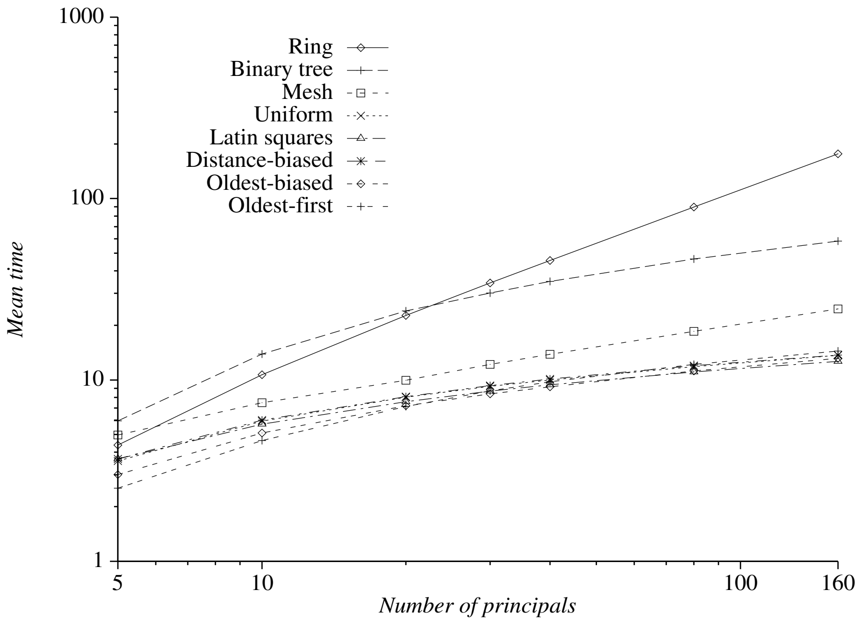


FIGURE 7.7: Effect of partner selection policy on scaling of mean time to acknowledgment.

7.3 Group membership resilience

The group membership component (Chapter 6) maintains weakly consistent *views* of the membership at every principal. It provides protocols for principals to join and leave the group, and to eject a principal that has failed. The views define a *view relation* between principals. The view relation is correct if its transitive closure is equal to the group membership. The view relation can be made resilient to some number of failing principals by adjusting the number of sponsors a principal must obtain upon joining the group.

The latency of the membership protocols depends on the message and acknowledgment latencies investigated in the previous section. The join protocol executes immediately, and notice of a new member is then propagated throughout the group. The leave and eject protocols require a message to be disseminated to and acknowledged by the group.

There are two related measures that can be taken of the view relation. The *in-degree* of a principal measures how many other principals have it in their view. The in-degree measures how far knowledge of a principal's membership has spread. The *minimum vertex cut* between two principals determines how many principals can fail before disconnecting them. This measures the failure resilience of the view relation.

Both steady state and transient behavior can be determined for each measure. In practice the steady-state behavior is not interesting because information spreads rapidly through a group. Transient behavior is more interesting, showing how quickly members' views converge.

7.3.1 Simulation modeling

A discrete-event simulation was used to model a system of n principals. For each run of the simulator, each of the n principals joined the group, acquiring k randomly-selected sponsors during the join protocol. Some number f of the principals, selected at random, then failed. The principals then began conducting anti-entropy sessions, with sessions occurring as a Poisson process. The sessions continued until all member views converged, or until the view relation became incorrect.

Two measures were collected at the end of each run: the mean time required for views to converge, and the probability that they converged. Six additional measures were sampled during the run: the average, minimum, and maximum in-degree of principals, and the average, minimum, and maximum minimum vertex cut of the relation. The mean time between anti-entropy sessions was used as the time base for all measurements.

Computing the minimum vertex cut proved to be an expensive operation. Finding the minimum vertex cut between any two principals requires $\Theta(n^2)$ applications of a maximum flow algorithm, which requires $O(n^2)$ time. The simulator required several hours on a SparcStation 2 to complete 3000 runs of a 25-principal system.

The simulator used the batch-means method [Jain91] to determine when the confidence interval of the sampled mean time-to-convergence was less than 2%. Each batch consisted of 500 runs, and at most 3000 batches were collected.

7.3.2 Results

Membership views converge rapidly, reaching nearly complete consistency after only a few rounds of anti-entropy. Figure 7.8 shows the time-varying measures collected for a system of 25 principals,

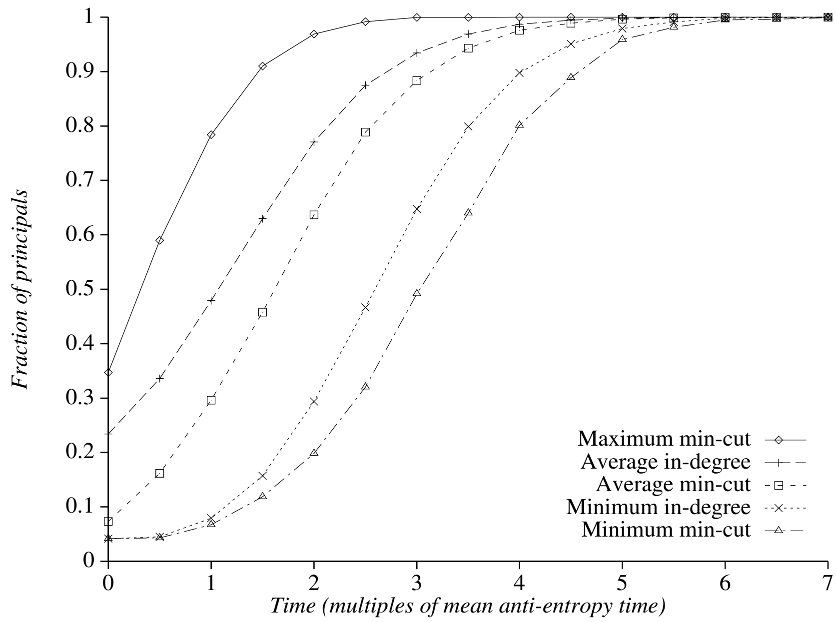


FIGURE 7.8: Progress of the minimum cut and in-degree measures in a group of 25 principals, using one sponsor, with no failures.

---

none of which failed, with principals obtaining one sponsor when joining. The minimum min-cut (the lowest curve on the graph) reports the smallest fraction of the group that can fail to render the view relation incorrect. This begins at $1/24$th, since initially some principal is known by only one of the other 24 principals. The resilience increases rapidly, and within six time units the views have almost always converged.

The average and maximum min-cut curves in Figure 7.8 show the range of principals that can fail before rendering the view relation incorrect. The expected resilience is always better than the minimum. The graph also shows that the average and minimum in-degree increase quickly as the group views converge.

Figure 7.9 shows how using two sponsors and one failed principal affect the measures. The views converge slightly faster than in the previous figure, partly because there are only 24 principals left in the group, but more because each new member was known by more members at the start.

The number of sponsors has a significant effect on the resilience and speed of convergence. Figure 7.10 shows the minimum resilience (min-cut) for a group of 25 principals, varying the

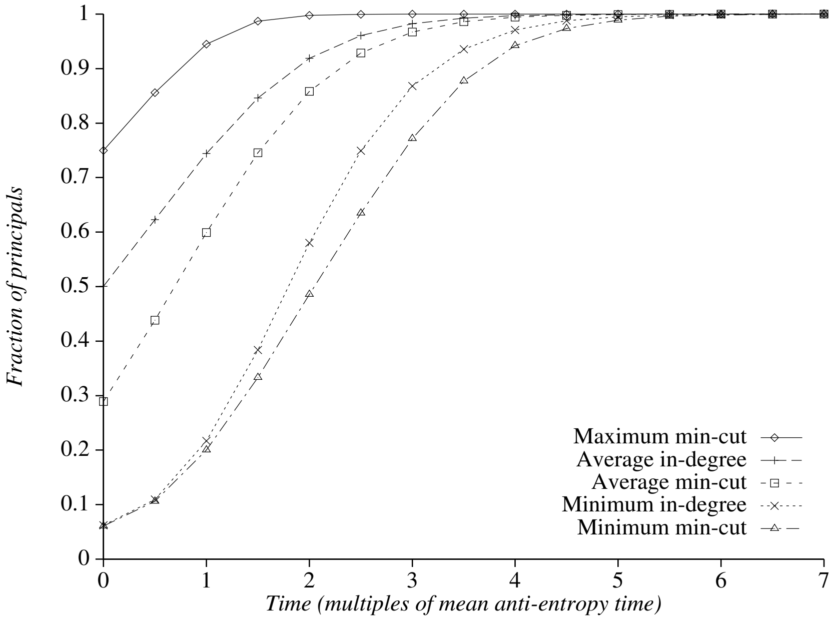


FIGURE 7.9: Progress of the minimum cut and in-degree measures in a group of 25 principals, using two sponsors, with one initial failure.

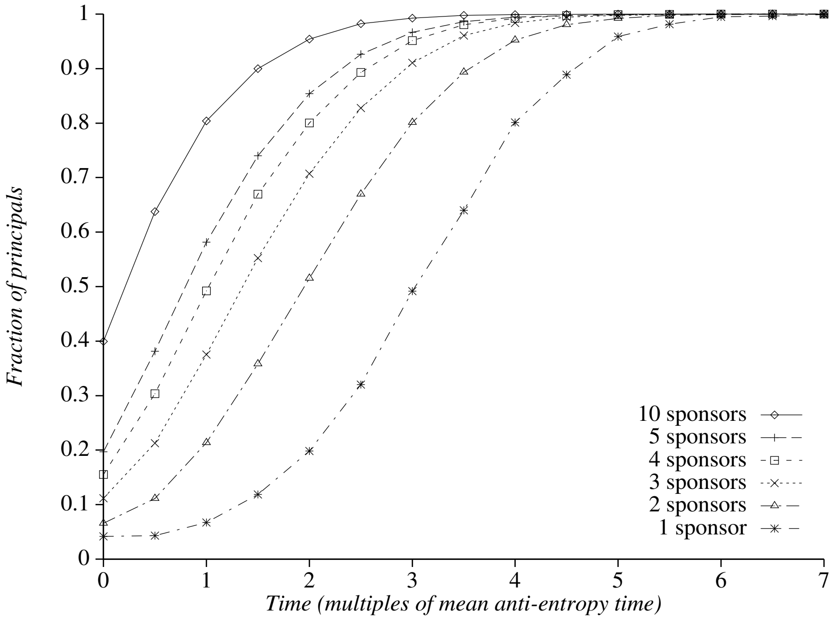


FIGURE 7.10: Progress of the group membership resilience, with varying numbers of sponsors. Measured for a group of 25 principals. The minimum cut is reported as the fraction of the principals that must fail to render the view relation incorrect.

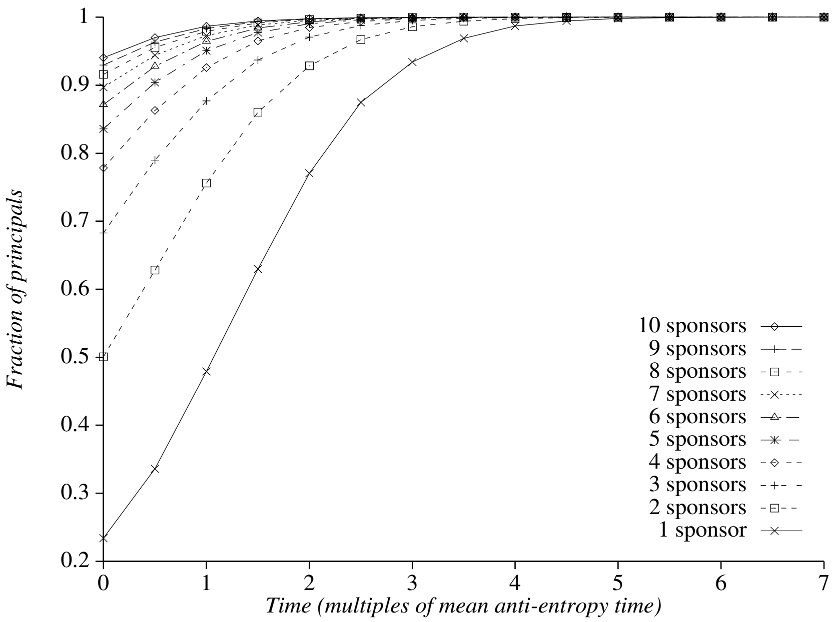


FIGURE 7.11: Progress of the average in-degree as anti-entropy propagates membership information. Measured for a group of 25 principals. The in-degree is measured as a fraction of the 24 principals that could know about each principal.

---

number of sponsors from one to ten. The views converge much more rapidly as sponsors are added. However, the benefit of adding sponsors decreases as they are added: using two sponsors instead of one produces a much greater improvement in resilience than adding a third sponsor.

Figure 7.11 shows how the number of sponsors affects the in-degree. Again, adding sponsors can significantly increase the fraction of the group that knows about each principal, and the greatest increase is obtained between one and two sponsors. This measure shows that even though the minimum resilience of the view relation may start low, the actual connectivity (as shown by the average min-cut and in-degree) is dense from the start. The low resilience comes from a small number of principals that are known only to a few others, while most of the rest of the group are known by several other members.

Figures 7.12 and 7.13 summarize the effect of the number of sponsors and number of failures on the time required for views to converge. Increasing the number of sponsors speeds convergence, and views converge faster with fewer failures. Increasing the number of failures causes only a slight increase in the mean time to convergence because the loss of information is offset by a decrease in

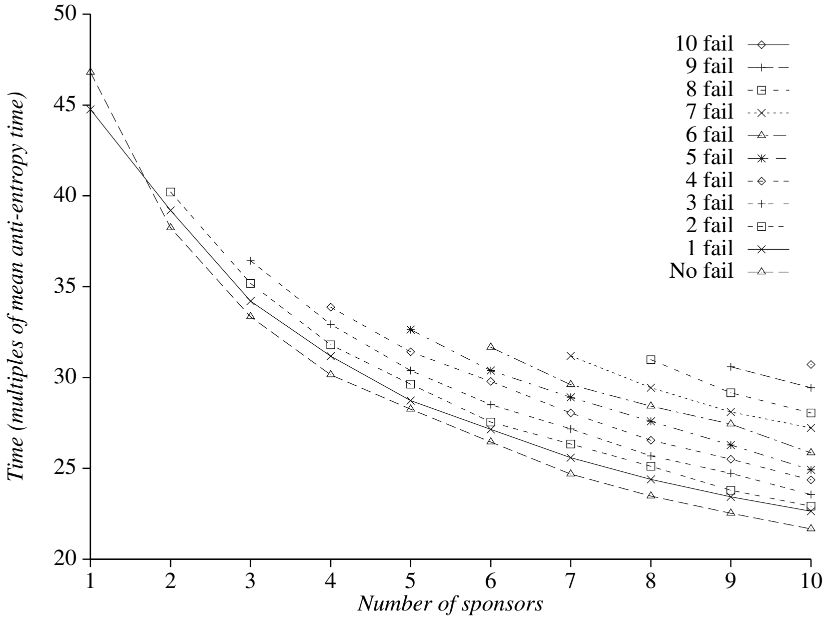


FIGURE 7.12: Mean time for views to converge, varying number of sponsors. Measured for 25 principals, of which between zero and ten fail. Note that the number of failures was not allowed to exceed the number of sponsors, to avoid an incorrect view relation. The discrepancy between zero and one failures with one sponsor arises because the single failure does not significantly affect the connectivity of the relation, but decreases the number of principals that must reach consensus.

---

the number of principals. There appears to be a marked decrease when the number of failures is only one or two less than the number of sponsors.

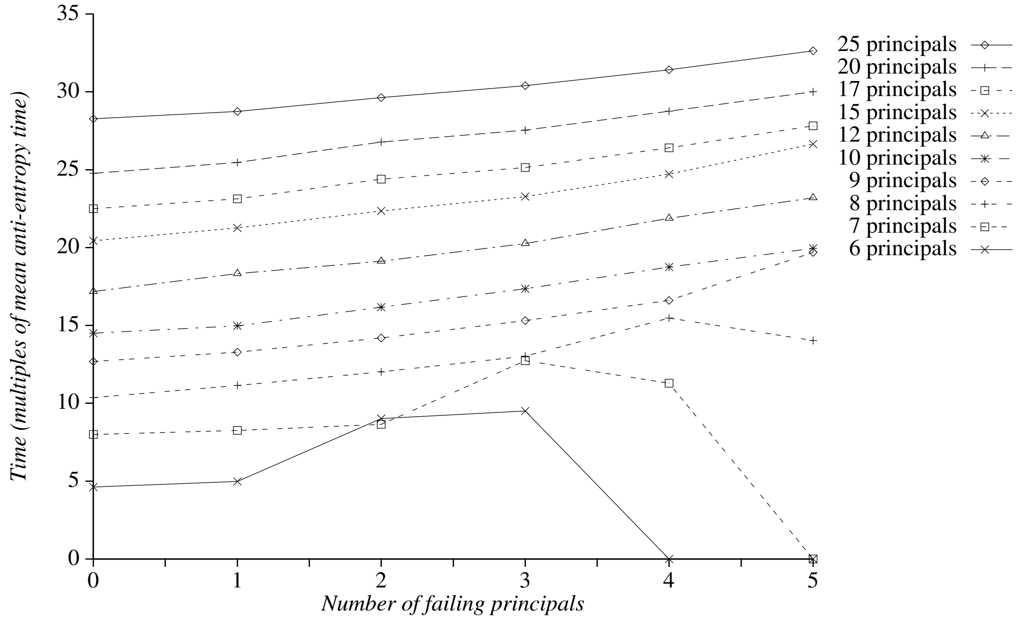


FIGURE 7.13: Mean time for views to converge, varying number of failing principals. Joining principals obtained five sponsors. As the number of failures approaches the number of principals, the time to convergence decreases because the number of principals that much reach consensus becomes small.

7.4 Traffic

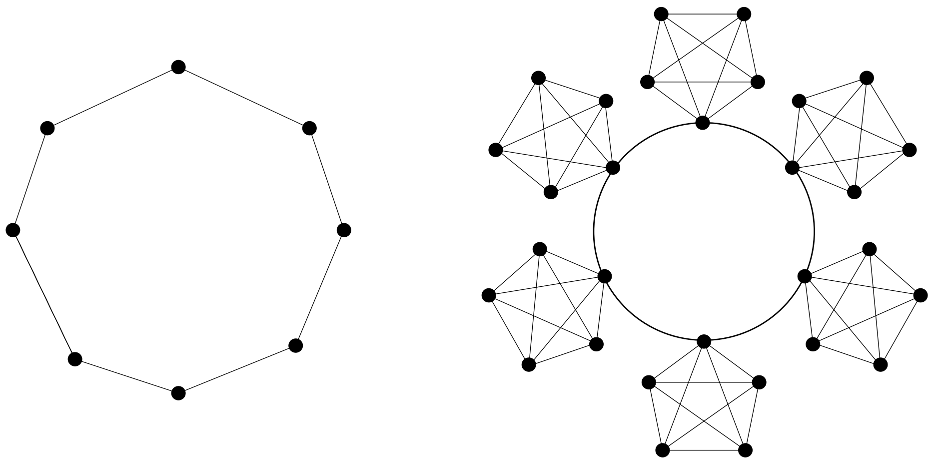
A group communication system must not overload the network on which it operates. This is particularly important if the group is to scale to a large size. The *traffic* induced by a system, as measured by the number of network packets that are sent and by the distance the packets traverse, is the primary measure of how the system will affect the network.

Researchers at Xerox PARC found that the original version of the Clearinghouse system overloaded their internetwork [Demers88, Demers89]. The original implementation used anti-entropy sessions with a best-effort multicast, and used a uniform partner selection policy during anti-entropy. A revised implementation reduced the network load by a combination of fixing implementation bugs, investigating a distance-biased partner selection policy, and adding a rumor mongery propagation method. The Clearinghouse protocols were discussed in Section 4.10.

Network traffic is measured by the number of packets sent, and by how many network links they must traverse. The number of packets is determined by the number of messages each principal sends, their size, and how often principals perform anti-entropy. In the absence of principal or network failures, the TSAE protocol ensures that every message is sent exactly once to each principal. The number of links each packet must cross is determined by the topology of the network and by the partner selection policy that principals use when performing anti-entropy.

Different network links can have different costs. For example, intercontinental links have long latencies, while IP running over a local Ethernet usually delivers a packet in a few milliseconds. Traffic measures and partner selection policies should account for this cost.

For this performance evaluation, I introduced two partner selection policies in addition to those discussed in Section 5.4.1. The **cost-biased** policies preferentially select low cost partners. This is different from **distance-biased** selection when network links have different costs. The **cost-biased** policy selects a principal with probability proportional to the inverse of the difference between its cost and the highest cost in the group. The **cost-squared-biased** selects with probability proportional to the difference squared, increasing the probability that low-cost partners will be selected.



(a) (b)

FIGURE 7.14: The ring and backbone physical topologies simulated for traffic analysis. (a) Principals are connected by a ring. (b) Principals are organized into 5-cliques; one principal acts as a gateway onto a backbone ring.

7.4.1 Simulation modeling

The system was modeled in a discrete-event simulation. Principals initiated anti-entropy sessions according to a Poisson process. Each run of the simulation performed 1000 anti-entropy sessions, collecting link traffic and cost statistics. At least 100 runs were performed, and they were repeated either until 1000 runs completed or until a batch-means analysis indicated that the 95% confidence interval width for each measure was less than 1% of the measured value. A complete set of runs required between half an hour and six hours on a DECstation 5000/200, depending on the number of principals and the partner selection policy.

The simulator programs modeled two different physical network topologies (Figure 7.14). Neither topology corresponds to the real structure of the Internet. Instead, they were selected as representative of topologies that can concentrate traffic onto a few links. The first topology connects principals in a ring. Rings are simple to model, and are the simplest structure that shows how distance can affect traffic without creating the edge effects produced by linear networks. The second topology groups principals into 5-cliques, and connects the cliques through a backbone ring. This is similar to the structure of the Internet today: regions of high connectivity, with regions weakly connected through a backbone network. Communication within a region is fast, while

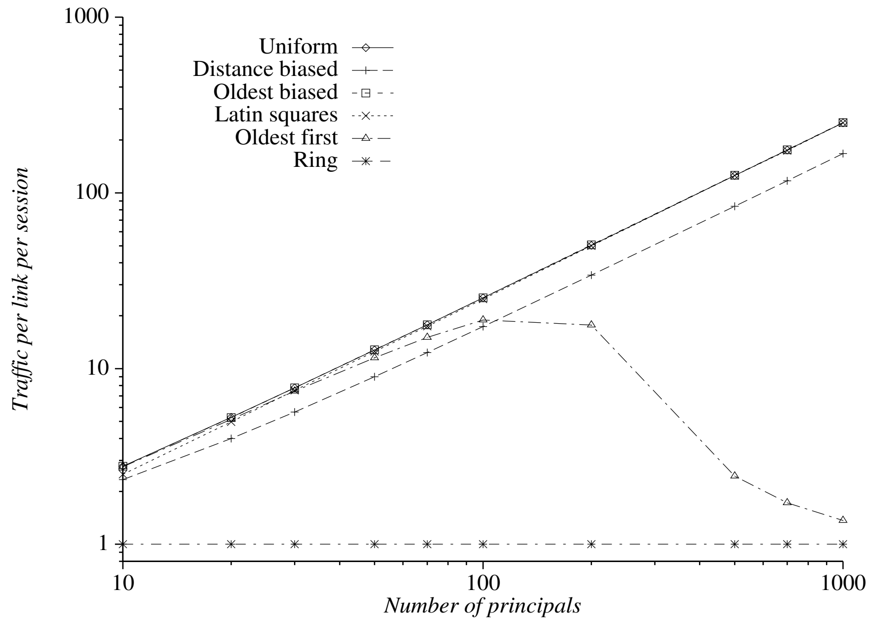


FIGURE 7.15: Traffic per network link on a ring network, varying the number of principals.

communication between regions can be expensive. Demers et al. noted that this kind of topology was a problem for the original Clearinghouse protocols [Demers88].

Each simulator run was parameterized by the partner selection policy and the topology. The binary tree and mesh partner selection policies were not tested because they were patently inappropriate on ring-like topologies. For the ring topology, the number of principals could be specified; this ranged from ten to one thousand. For the backbone topology, the number of principals was fixed at 30 (six 5-cliques) but the cost of the backbone links relative to the intra-clique cost could be specified. This cost ranged from 1 through 160.

7.4.2 Results using ring topology

As expected, the amount of traffic scales with the number of principals (Figure 7.15). The **uniform, oldest-biased,** and **latin squares** policies all produced the same amount of traffic, which is not surprising since they also exhibited similar propagation delays (Figure 7.6). Distance biasing produced some improvement, while the **ring** policy produced a constant traffic of one link per anti-entropy session. I believe that the anomalous result for the **oldest-first** policy is due to a minor implementation detail: when several principals are equally old, the nearest one will be selected. I suspect that when the group becomes large the policy often propagates a message from one principal to its neighbor in a “wave” of propagation, which would behave much like the **ring** policy.

7.4.3 Results using backbone topology

The clique-and-backbone topology is more interesting than the simple ring because it provides a more realistic basis for comparing policies. The backbone links can have a different cost than the intra-clique links, and the connectivity between principals is more varied.

The **cost-biased** partner selection policies are appropriate for this topology. They are similar to distance-biasing, except that the likelihood of selecting a principal as partner is proportional to the difference between the minimal communication cost and the cost of communicating with that partner. The **cost-squared-biased** policy sets the likelihood to be proportional to the difference squared, favoring nearby sites. The advantage of these policies is that they are not based on an arbitrary topology, but rather upon observable performance measures. This makes them appropriate for use in a wide-area internetwork where topological information is likely to be unavailable.

The **ring** and **distance-biased** polices had to be mapped onto the backbone topology. Principals were assigned an index, starting with zero at the gateway of one clique. The other four principals in the clique where numbered one through four, the gateway of the next clique clockwise around the ring was numbered five, and so on. The **ring** policy used this ordering for its logical ring, while the **distance-biased** policy used the difference between indexes as its “distance”.

Figure 7.16 shows the average cost of the links traversed by an anti-entropy session. As on a simple ring, the **uniform, oldest-biased, oldest-first,** and **latin squares** policies all performed about the same, while distance-biasing improves performance somewhat. The **ring** policy is somewhat better than all of these, though it scales in about the same way. The cost-biasing protocols produce somewhat more traffic when the backbone cost is low, but scale better as the backbone cost increases.

The final two figures in this section, Figures 7.17 and 7.18, show how the anti-entropy traffic is divided between the cliques and the backbone. In Figure 7.17 the cost-biasing policies decrease the traffic per link, approaching one as the cost of the backbone increases. The other policies do

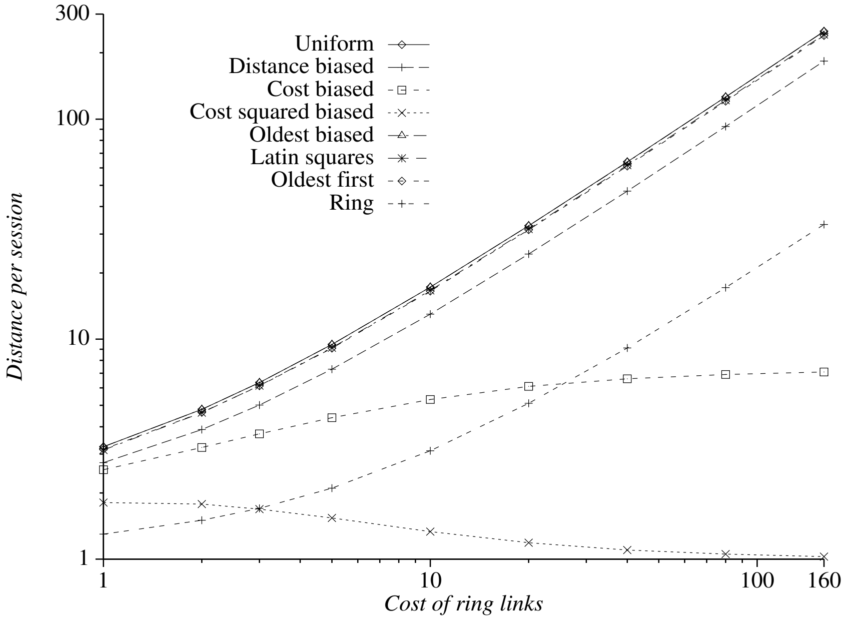


FIGURE 7.16: Effect of partner selection policy on the average number of network links used in an anti-entropy session. Measured on a ring of six 5-cliques. The cost of the backbone ring links varied from 1 to 160.

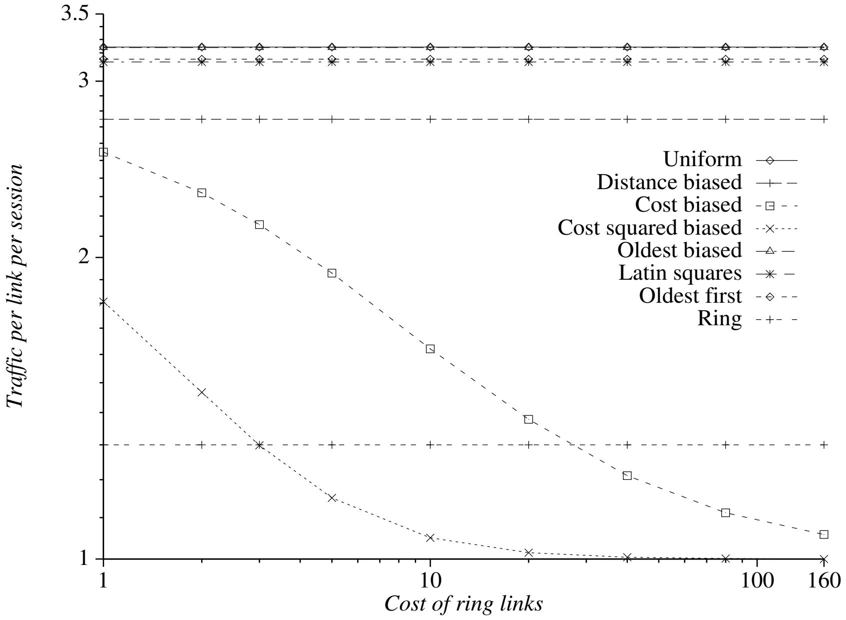


FIGURE 7.17: Effect of partner selection policy on the mean traffic per link, for all links. The cost of the backbone ring links varied from 1 to 160. Only the cost-biased policies adapt to the cost of backbone links.

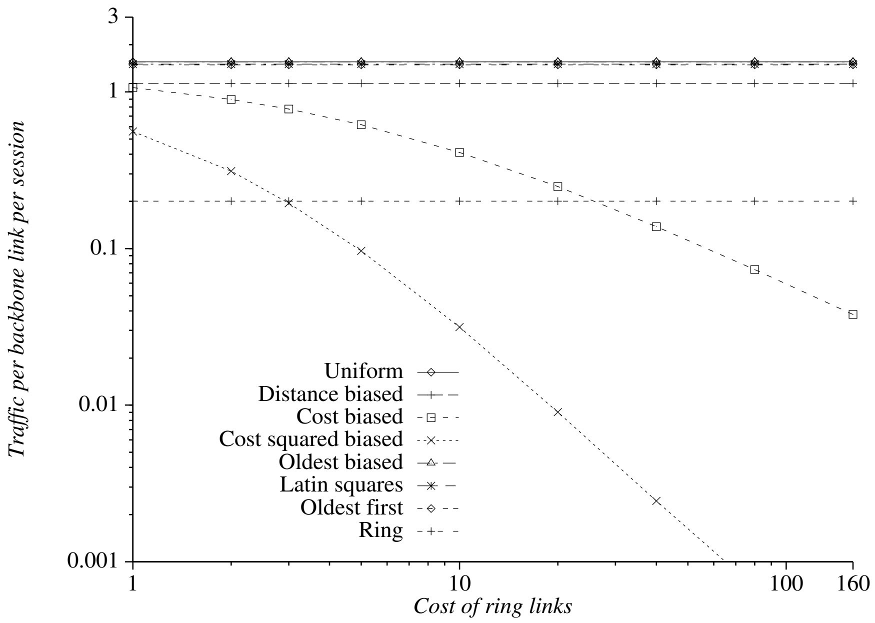


FIGURE 7.18: Effect of partner selection policy on the mean traffic per backbone ring link.

---

not account for link cost, and the link traffic remains constant as link cost changes. Figure 7.18 shows that the cost decrease is achieved by concentrating communication within a clique. When backbone links cost 80 times as much as a clique link, the **cost-biased** policy only allows between 1 and 2% of all anti-entropy sessions to cross between cliques. Fewer than 0.3% of all sessions cross backbone links when cost-squared-biasing is used. Clearly cost-biasing ensures that communication is predominantly local.

7.4.4 Traffic and propagation time

While one might want to reduce network traffic as much as possible, a reduction in traffic generally requires an increase in the time required to propagate a message. Figure 7.19 shows the relationship between the two. For a membership of 30 principals, the time required to propagate a message is plotted as a function of the link traffic for each of several partner selection protocols. All of the policies excepting the **ring** policy appear to fit on a smooth curve. Figure 7.20 shows that a similar result holds for the time required to propagate acknowledgments.

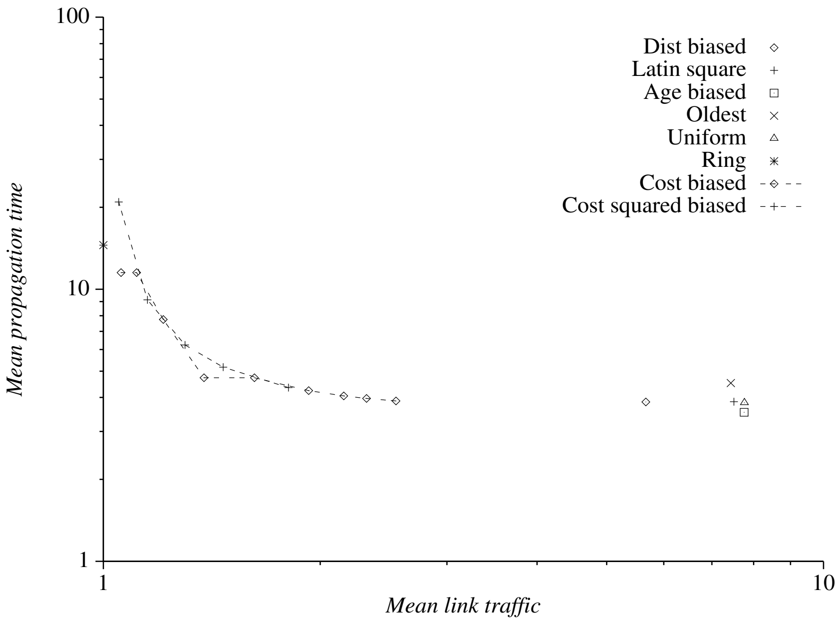


FIGURE 7.19: Scatterplot of the relationship between link traffic and propagation delay. Measured for 30 principals. Time is measured in multiples of mean time between anti-entropy sessions. Every protocol except **cost-biased** and **cost-squared-biased** is represented by a single point. Multiple points are reported for the cost-biased protocols, showing how they respond to different backbone link costs.

7.5 Consistency

Weak consistency protocols allow principals to contain out-of-date information. There are two related measures of this effect – one concerning the propagation of a single message, the other concerning the consistency of group state. The time required to propagate an message from one principal to others shows how quickly information will be made available to clients; this was investigated in Section 7.2. The likelihood that a principal is out-of-date with respect to other principals, and the difference between them, aggregates the effects of several messages.

7.5.1 Simulation modeling

A discrete event simulation modeled the TSAE protocol to measure information age. The simulator used five events: one each to start and stop the simulation, one to send a message, one to perform anti-entropy, and one to sample the state of a principal. The simulation was first allowed to run for 1 000 time units so it would reach steady state, then measurements began. The simulation ended

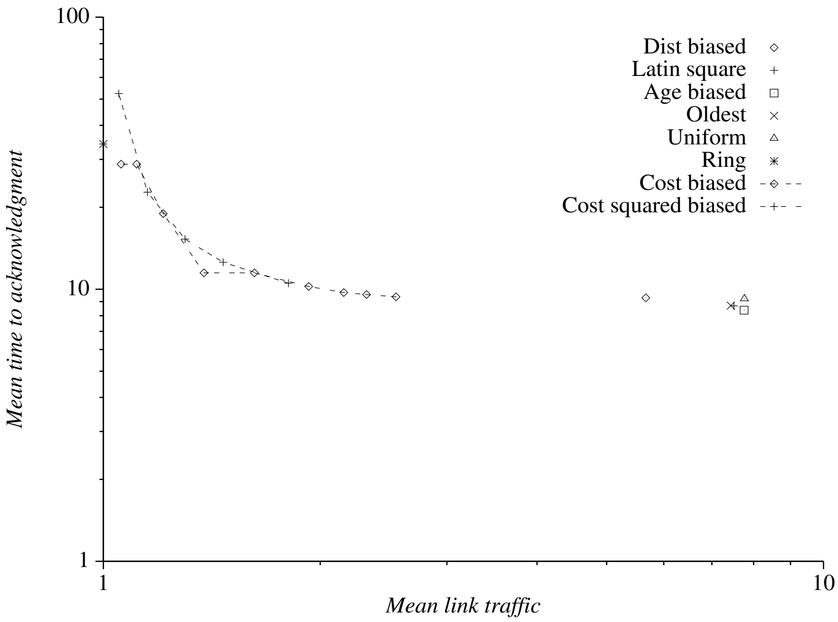


FIGURE 7.20: Relationship between link traffic and time to acknowledgment. Measured for 30 principals. Time is measured in multiples of mean time between anti-entropy session. See Figure 7.19 for more details.

---

at 50 000 time units. Read, write, and anti-entropy events were modeled as Poisson processes with parameterizable rates. These rates were measured per principal. The simulator included different partner selection protocols and an optional unreliable multicast on writes.

The state of a principal was modeled as a *single* data value, and messages were treated as updates to that value. In this way the simulation results show the currency of each data value in a principal, given the rate at which the value is updated.

The simulator maintained two data structures for each principal: the anti-entropy summary vector and a message number. It also maintained a global message counter. When a message was sent, the global counter was incremented and the sender's message number was assigned that value. If an unreliable multicast was being used, the message number would be copied to other principals if a the datagram was received. Anti-entropy events propagated message numbers between principals, as well as updating the principals' summary vectors.

Sampling events were used to collect measures of the expected age of data and the probability of finding old data. A principal was selected at random, and the message number for that principal was compared to the global counter. The difference showed how many messages the principal had yet to receive.

7.5.2 Results

A system like Refdbms stores thousands of values in its group state, while the simulation results concern a single data value. The results can be viewed in two ways. Each data value can be considered separately, and the performance results considered in terms of the update rate *per value.* Alternately, the the values can be considered collectively, and the results interpreted in terms of the *overall* update rate. In Refdbms the overall update rate will be thousands of times greater than the rate per item.

The age of a principal's state depends on the ratio of the anti-entropy rate to the update rate for the state. Many wide-area services have extremely low update rates; some services write new entries and never change them. A low update rate means that anti-entropy has a better chance of propagating an update before another update enters the system. In the Domain Name Service [Mockapetris87], a host name or address rarely changes more than once every few months. In systems like Refdbms, new entries are added, corrected quickly, then remain stable. I expect the update rate for most wide-area services to be about a thousand times lower than the anti-entropy rate. Most of the graphs in this section were generated using a mean time-to-update of 1 000 time units; the maximum anti-entropy rate investigated was only 200 times greater, giving a mean time-to-anti-entropy of five. This implies that all the results presented here are more pessimistic than would actually be observed.

Figure 7.21 shows the likelihood of a principal holding an out-of-date value, while Figure 7.22 shows the expected age of that value. Clearly, adding an unreliable multicast on write significantly improves both measures. The message success probability is the most important influence on information age in large groups of principals. For small numbers of principals, increasing the anti-entropy rate dramatically improves both the probability of getting up-to-date information and the expected age.

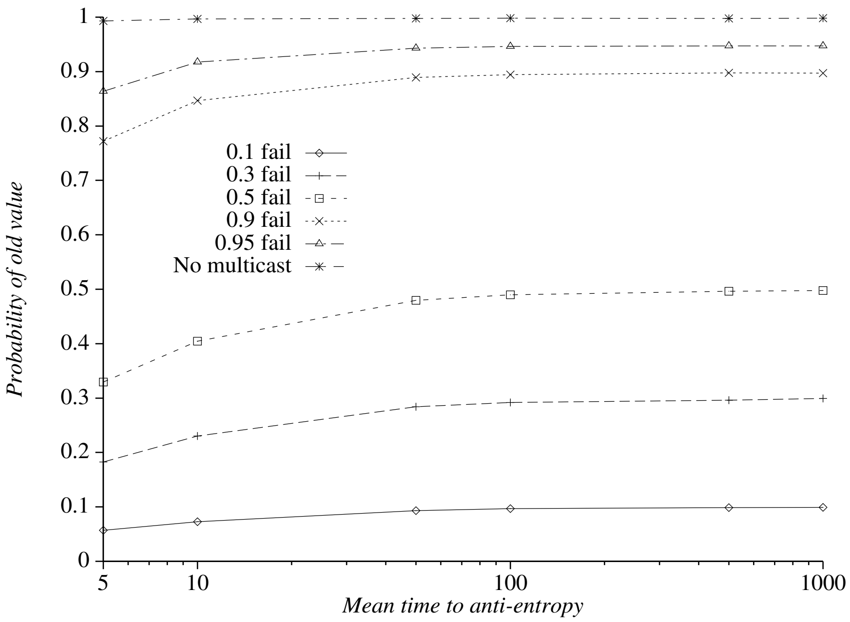


FIGURE 7.21: Probability of getting old value as the per-principal anti-entropy rate varies, for 500 principals. Mean time-to-write 1 000; **uniform** partner selection. Anti-entropy was combined with a best-effort multicast. The different curves show the effect of the probability of multicast message failure.

Figures 7.23 and 7.24 show how consistency depends on the number of principals. For these simulations the anti-entropy rate was fixed at 100 times that of writes. This value might be typical for a Refdbms entry soon after it is entered, when updates are most likely. Later updates will be less frequent and the ratio will increase, improving the consistency. Once again an unreliable multicast provides considerable improvement.

I also investigated the effect of partner selection policy on information age, as shown in Figure 7.25. The results show that expected age is related to propagation time, since the policies are ranked in exactly the same order as in Figure 7.6, which shows the mean propagation time for the different policies. The topological policies (**Ring, binary tree, and mesh**) propagate more slowly, and give a greater expected age, than other policies. The other policies are nearly equal, though **oldest-first** has a slight advantage.

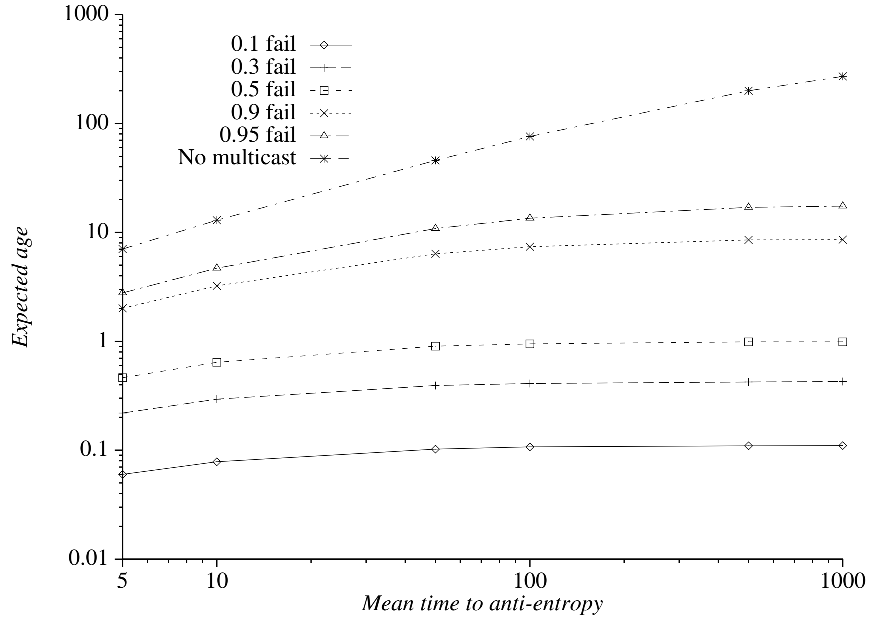


FIGURE 7.22: Expected data age as anti-entropy rate varies, for 500 principals. Mean time-to-write 1 000; **uniform** partner selection. Again, anti-entropy was combined with a best-effort multicast, for which the message failure rate varied.

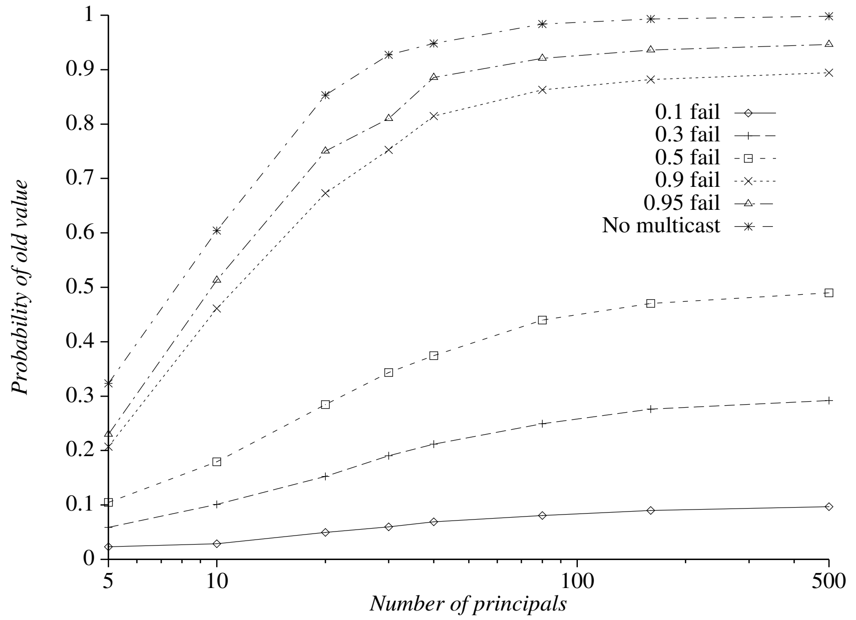


FIGURE 7.23: Probability of getting old value as the number of principals varies, with anti-entropy occurring 100 times as often as writes. **Uniform** partner selection, combined with best-effort multicast.

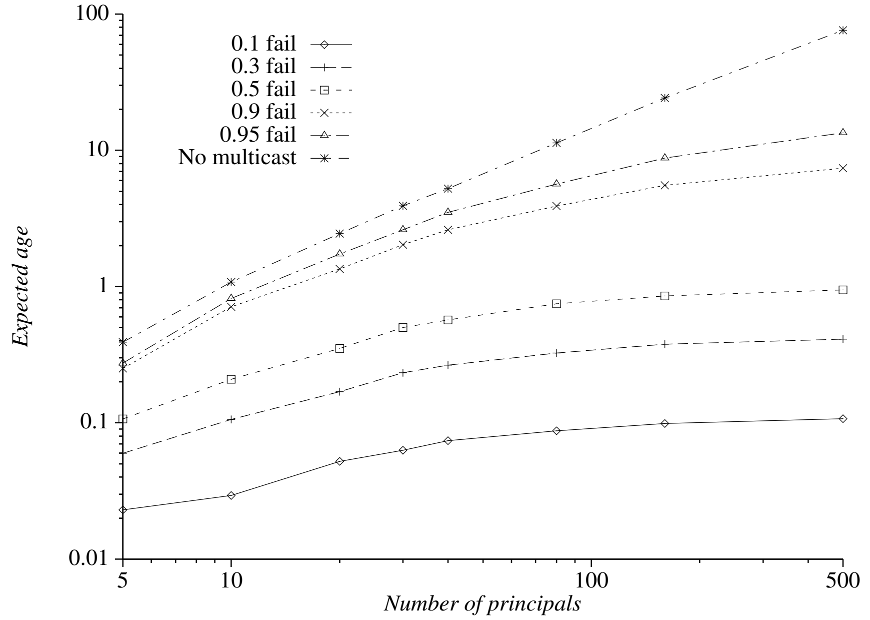


FIGURE 7.24: Expected data age as the number of principals varies, with anti-entropy occurring 100 times as often as writes. Uses **uniform** partner selection. Also shows the effect of varying message failure rates in a best-effort multicast.

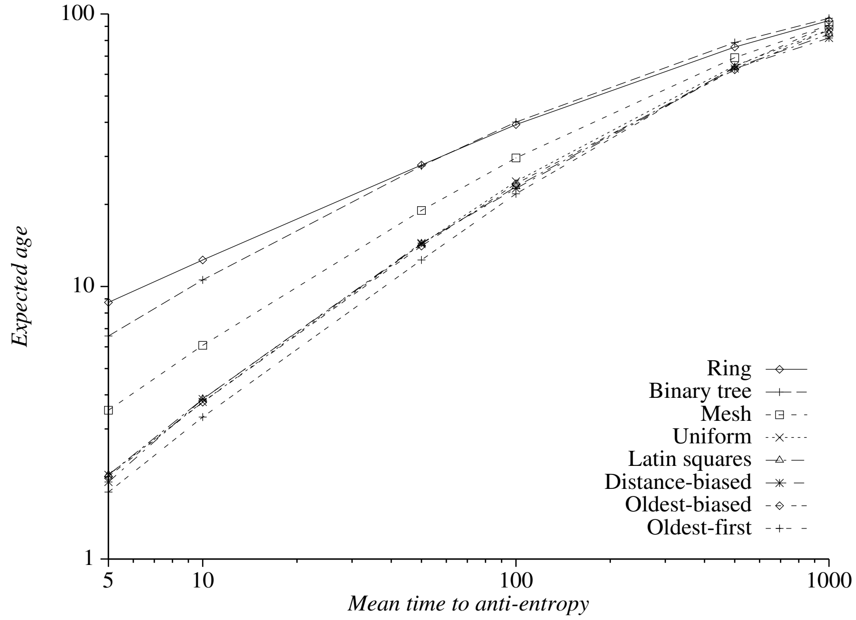


FIGURE 7.25: Effect of partner selection policy on expected data age. 160 principals; mean time-to-write 1 000. No best-effort multicast was used for this graph.

7.6 Comparison

The preceding sections have presented an analysis of the performance of the timestamped anti-entropy protocol. In this section I compare my framework, using timestamped anti-entropy, with other systems.

7.6.1 Efficiency

Table 7.1 compares the amount of state and network traffic that TSAE and the surveyed systems will produce, along with the guarantees each can provide. Any protocol will require at least $O(1)$ state per message in addition to the message contents, assuming the message must be marked with the sender's identity. Likewise, every principal must maintain a list of group members, so the minimum state per principal is $O(n)$. Each application-level message will induce some number of network packets. At best one packet would be sent to each recipient. Hardware multicast can reduce this traffic somewhat, but in a wide-area network it is unlikely that many of the principals would reside on common networks.

Timestamped anti-entropy compares favorably with the other systems. It can produce total or causal orders without either adding extra state per message or sending extra messages. The approach of attaching causal information to batches appears to be a good idea.

The centralized systems and consistent replication all use a minimal amount of state per message because messages are processed one-at-a-time. They also produce low network traffic. The drawback is that they must operate synchronously, which makes them infeasible for wide-area networks.

The systems that provide interactive delivery all begin by sending an unreliable multicast that is backed by a reliable protocol. Most of them transmit extra information with each message so that missing messages can be detected, and most require principals to periodically transmit null messages if they have not generated any real traffic.

Eventual delivery, however, appears to require much less state per message. Excepting Lazy Replication, the eventual delivery systems attach $O(1)$ state to each message. The anti-entropy protocols require only a small amount of extra network traffic per message. The Lazy Replication protocol can probably use some of the techniques in TSAE to reduce its per-principal state and per-message traffic, but it cannot reduce its per-message state because it guarantees causal consistency with events outside the group.

7.6.2 Implementation effort

The implementation of TSAE, group communication, and message ordering in Refdbms can be compared with the implementation of another system, such as Isis version 2.1. The differences between the two are striking. I used lines of code as a simple complexity metric.

These measures must be used with caution. The comparison is not entirely fair, because the Refdbms implementation is not a general-purpose toolkit. Isis is a more mature package than Refdbms, having been in development for several years, while Refdbms is only in early testing. The size of code and implementation effort should be amortized over the number of applications that use the code, in which case the complexity of *interfaces* is a more useful, though unquantifiable, measure.

The Isis V2.1 toolkit is structured into four major components: two libraries that are linked with user programs, and two programs that provide communication services between hosts and fault tolerance services. The toolkit implements the **ABCAST** and **CBCAST** protocols already discussed and several variations on them. It also provides a number of higher-level services, including failure detection and recovery, threads, persistent storage, a primitive name service, and a number of high-level distributed services.

Refdbms consists of several dozen programs, mostly concerned with user interfaces, the search engine, and output formatting. For this analysis I have only considered those programs that relate to distributed operation, and I have removed parts of those programs that implement specific Refdbms functions (such as constructing the indices).

Table 7.2 compares TSAE, as implemented in Refdbms, with Isis. TSAE appears to be of a similar complexity to an Isis protocol, although the amount of comment lines in Refdbms (1̃5%) is much higher than in the Isis sources, and the Refdbms numbers include some application code as well as “toolkit” code. However, TSAE uses only a total of 2 600 lines for communication, while

TABLE 7.1: Performance comparison of several group communication systems.

	Guarantees		State per		Traffic per
System	Delivery	Order	message	principal	message
Optimum	—	—	$O(1)$	$O(n)$	n
TSAE	R,E	any	$O(1)$	$O(n)$	$O(n) + n/k$ [5] $k > 0$
Centralized	A,S	total	$O(1)$	1	$2n$
Consistent replication	A,S	total	$O(1)$	$O(n)$	$O(n)$ [2]
Isis ABCAST	A,I	total	$O(n)$	$O(n)$	$(1 + 1/k)(2n + 2\phi n)$, $k \geq 1$ [6]
Isis CBCAST	A,I	causal	$O(n)$	$O(n)$	$2n + 2\phi n$
Psync	A,I	causal	$O(n)$	$O(n)$	$n + 2\phi n$
Orca RTS [1]	R,I	total	$O(n)$	$O(n)$	$n + 2\phi n$
Rel. multicast	R,I	?	?	$O(n)$	$2n + 2\phi n$ [5]
OSCAR	R,I	none, FIFO, or total	$O(1)$	$O(n)$	$n + 2\phi n + 2n^2/k$, $k \geq 1$ [6]
Lazy Replication	R,E	causal [3]	$O(n)$	$O(n^2)$	$O(n^2)$
Clearinghouse anti-entropy	R,E	none	$O(1)$	$O(n)$	$O(n) + 1/k$ [5]
Rumor mongery	U,E	none	$O(1)$	$O(n)$	large [4,5]

Abbreviations:

- n Number of principals.
- ϕ Probability of message failure for an unreliable multicast.
- k Number of ordinary messages per control message or batch.
- R Reliable delivery.
- U Unreliable delivery.
- S Synchronous delivery.
- I Interactive delivery.
- E Eventual delivery.

Notes:

[1] Orca is represented by the implementation based on unreliable multicast.

[2] Traffic for consistent replication depends on the particular protocol chosen, how that protocol is parameterized, and the ratio of read to write operations.

[3] Lazy replication can maintain causal relations generated outside the group.

[4] Depending on the stopping rules used, rumor mongery can send a message many times to each principal.

[5] These systems can take advantage of network topology to reduce communication distance.

[6] Isis and OSCAR both use a coordinator. The coordinator sends control messages every k normal messages on average.

TABLE 7.2: Implementation complexity of Isis compared with TSAE in Refdbms. Lines rounded to the nearest hundred.

	Isis		TSAE/Refdbms	
Component	Subcomponent	Lines	Subcomponent	Lines
Ordered messages	ABCAST	700	Ordering	100
	GBCAST	1 000	Anti-entropy	1 400
	CBCAST	1 100	Send message	100
		2 800		1 600
Low-level communication		8 500		1 000
Groups		3 400	General	500
			Join	1 400
			Leave	200
		3 400		2 100
Other	Failure	1 900		
	Threads	2 100		
Total size		52 000		9 200

Isis uses 11 300 because it implements three different protocols, and it uses a much more elaborate infrastructure to implement them. Isis requires half again as much code to implement its groups, though it provides functions, such as subsetting a group, that Refdbms does not.

The total size of the two packages is striking: Isis is more than five times larger than the distributed portions of Refdbms. This suggests that an implementation tailored to application requirements can be simpler and more efficient.

7.7 Summary

This chapter presented a number of performance evaluations of the components of the weak-consistency group communication system, including network traffic, message latency, fault tolerance, and consistency.

The TSAE message delivery protocol scales well to large groups of principals. The time required to propagate an update from one principal to all others increases as the log of the size of the group, and the partner selection policy can be chosen to control network traffic as the membership grows. Each principal need only store $O(n)$ state in the group size.

Fault tolerance is the ability of a system to provide correct service even when some parts fail. The TSAE protocol provides excellent fault tolerance by delaying communication until a principal is available. This implies that the message delivery component will not fail to deliver a message to the group unless the sender and several other principals fail, and this was found to be an unlikely event.

The negative aspect of weak consistency protocols is that principals will have out-of-date information. This investigation found that an unreliable multicast can mitigate most of this problem, and that at reasonable propagation rates principals are rarely more than a few updates behind. Many applications, including name services and bibliographic databases, work well with this level of inconsistency.

Chapter 8
Multiple membership roles

The group communication mechanisms presented in the last several chapters are correct and efficient, and have been used to build the Refdbms system. Other features are needed to build a complete wide-area information service. In this chapter I discuss how the additional pieces can be constructed from weak-consistency components.

Group members and their clients may vary in their roles within an application, and consequently vary in certain behaviors and privileges. For example, some principals may be restricted from initiating certain operations on the group state. Some principals will act as *clients* of the application. Other principals may store only a *subset* of the group state. A *location service* is needed to allow principals to locate the group.

Consider an information service client. The client must be able to find a group member in order to provide service to a user. This is the job of the location service. If the client is running on a disconnected mobile computer, it will not be able to contact the service unless a copy has already been placed locally. When a mobile computer has disk space, techniques to store a subset of the service database reduce storage requirements. Nearby principals that store a subset are useful even when a client is connected to the network, because communication with these nearby principals will be faster than with distant authoritative principals.

8.1 Limiting write access

In Refdbms, not every principal is allowed to submit changes to the database. For example, the UCSC Technical Reports database should only be updated by users at UCSC; other sites should only read the database. This is implemented by giving members a *privilege* level of either full,

TABLE 8.1: Refdbms privilege levels.

Level	Can sponsor	Privilege
Full	Full	Can perform any operation, and can sponsor another principal with any privilege level.
Read-write	Read-only	Can read and update the database, but can only sponsor a principal with read-only privilege.
Read-only	None	Can only read the database; cannot sponsor another principal.

read-write, or read-only. A member's privileges are set when it joins the group, and are derived from the privileges of its sponsors.

Table 8.1 shows the privilege levels used in Refdbms. Full and read-write members can perform any operation on the database, while read-only members cannot update it. A principal obtains its privilege level from its first sponsor. If that sponsor is a full member, the new member is also a full member; if it is a read-write member, the new member is read-only. Read-only principals cannot serve as primary sponsors. (This last design decision was arbitrary, and will be revisited in future versions of the system.)

A more general solution could use *capabilities* to define what privileges a principal has. For example, the Amoeba distributed operating system uses encrypted capabilities that can be managed in user code [Mullender86, Tanenbaum86]. These capabilities include one bit for each kind of operation that can be performed on the group state, allowing fine-grained privileges.

The Refdbms system has no safeguards to ensure that a principal does not incorrectly claim to have greater privilege than it should. Authentication and checking for correct behavior are important problems, but current approaches are generally centralized and will not work well for very large systems. Section 9.5 discusses some possible approaches.

Multiple privilege levels can be used to reduce the amount of space required by the summary timestamp vector. This vector only needs to include a timestamp for every principal that has modified the group state, and so read-only members can be excluded from the vector. However, read-only principals cannot be excluded from the acknowledgment vector, since they must receive each message.

```
Client(groupId gid, request)
{
    list⟨principal⟩ possibleMembers;

    // Obtain a list of possible members from the location service.
    // This list must contain at least one group member.
    possibleMembers = LocationService.find(gid);

    // Order the members by distance
    PerformanceService.order(possibleMembers, criterion);

    // Try each possible member, starting with those nearby
    for each ⟨pid⟩ in possibleMembers {
        send(pid,request);
        receive(pid, returnCode, results);
        if (returnCode == SUCCESS)
            return SUCCESS;
        }
    }
    return FAILURE;
}
```

FIGURE 8.1: A skeleton client. The location service provides one or more possible members, and the performance prediction service orders them from best to worst. The ordering can be based on several criteria, including message latency, available bandwidth, and reliability.

8.2 Clients

A client is transient, unlike principals. It requests that one operation be performed on the group state, then exits. It does not become a part of the group, but instead sends its requests to a member. This is unlike the Isis system (Section 4.4), where clients join a group in a special role. A client in that system multicasts its requests to the non-clients in the group, and receives one or more replies.

Figure 8.1 shows a skeleton of what a client must do. The location service provides the client with the addresses of group members. As noted earlier, it must provide at least one address as long as the group exists. The client orders the addresses using the performance prediction service (Section 2.5) to determine which principals are nearby. It contacts the best available member to perform the operation.

Refdbms includes clients to add, change, and delete references, to search the keyword index, and to retrieve reference entries. These clients communicate with a local group member through a database residing on a common file system, typically a local file system or one mounted from an NFS server. This approach distributes searching to local workstations and eliminates the need for a location service. However, clients cannot use database replicas at other sites if the local copy is unavailable. This is not a problem for Refdbms clients since it is unlikely that any workstations will function unless the shared file system is available.

8.3 Storing a subset of group state

A properly-implemented group containing several members will provide a highly reliable, scalable service. With enough members, the network distance between clients and members should be small on the average – perhaps on the same continent, or in the same region. However, mobile computers require that group state be stored *locally,* and copies on a nearby server can improve performance of workstation clients. Most of these systems, particularly mobile computers, will not have the disk space to store the group state in its entirety. Instead, they can store a subset of it.

There are two ways the subset can be defined: either as an arbitrary set of state items, or as all items that satisfy a query. The former are called *caches,* and the latter are called *slices.* Some principals may maintain a combination.

Cache principals maintain copies of recently-accessed state items. This can improve performance if a user or organization repeatedly accesses a small set of items. In Refdbms, this might happen while a user is writing a paper: the references cited in the text would be retrieved repeatedly as the paper was edited and reformated.

Slice principals prefetch items according to user-specified interests. Users specify a query, and the slice principal maintains a copy of every item that matches the query. Alonso, Barbará, and Garcia-Molina have researched similar issues for systems that maintain *quasi-copies* of a central database [Alonso90b]. They point out that slices are similar to materialized views in a relational database. As with database views, the items in a slice are determined by evaluating an expression that has the same form as a query on the group state. In the quasi-copy system, each slice also has a coherency condition that specifies how far out of date the items in a slice can be.

Caches and slices differ in their handling of new state items: slice principals will store a copy of a new item if it matches some predicate, while cache principals will not.

When a principal stores only part of the group state, it will not be able to answer every query. Instead, some queries must be forwarded to some other principal. As in the Domain Name Service, the forwarding can be *recursive,* where the principal forwards the query to another principal, or *iterative,* where it informs the client of other principals that might answer the query. Recursive forwarding makes for simple clients, but increases the dependence of clients on the correctness of the members.

Refdbms does not currently implement caches or slices.

8.3.1 Caches

A cache principal maintains a message log and summary and acknowledgment vectors, just as ordinary members do. It periodically performs anti-entropy sessions with principals maintaining a full copy of the group state, propagating any updates it originated to other principals and receiving updates to the items it has cached.

Update propagation between a cache and another principal is asymmetric, unlike a normal anti-entropy session. The cache may discard many of the messages it receives because it is not maintaining a copy of the items to which they apply. Another principal therefore cannot rely on receiving a full set of update messages from the cache principal, other than those messages originally sent by the cache principal. During an anti-entropy session, the cache principal sends its partner a complete copy of its summary and acknowledgment timestamp vectors, while the partner sends only the single timestamp it maintains for the cache along with a complete copy of its acknowledgment vector.

Every group member, including every cache and slice principal, must still maintain an acknowledgment timestamp (or timestamp vector, if clocks are not synchronized) for every other member. Cache principals must not purge messages from their log until every member has received the message, and the conditions for doing so are not affected by maintaining only a subset of the group state.

A cache principal cannot authoritatively answer all user queries. For example, if a user asks for all references related to marsupials, there is no guarantee that the cache will have those references. Instead, the cache principal must forward the query request to another member.

Many different policies can be applied to determine what items should be cached. The burden of identifying items that are candidates for caching can be placed on clients, which can explicitly request that the cache principal maintain certain items. Alternately, all retrieval requests can be directed to the cache principal, which can add items to the cache set and remove them when the disk space required to store the items has grown past some bound.

8.3.2 Slices

Slice principals act much like cache principals, except that they store a subset of the group state determined by a query, or *selection condition,* rather than an arbitrary set of items. The principal must store this query in addition to the group state and communication data structures. The selection condition is a predicate on items. For simplicity assume the predicate is in disjunctive normal form; that is, it has the form

$$(p_1 \wedge p_2) \vee (p_1 \wedge p_3) \vee (p_4).$$

When a client queries a slice principal, the slice can determine whether it can satisfy the query if the query is equivalent to a subset of the slice predicates. For example, a Refdbms slice principal could answer a query on **marsupials** if it stored references on **marsupials** $\vee$ **australian animals**.

The slice principal conducts anti-entropy sessions to maintain its information. As with caches, information flow must be asymmetric because the slice only contains a subset of the group state. The slice server must select as its partner a principal that maintains a superset of its state . Usually, this will be a principal maintaining a full copy. The slice principal and its partner exchange complete summary and acknowledgment vectors during an anti-entropy session. The partner will send updates from any other group member, while the slice principal only sends messages it has originated. If network bandwidth is limited, the slice can send its selection condition, and the partner can transmit updates for only those items that meet the condition.

Several users on a local network may share a common slice principal. The combined selection condition is the union of individual users' conditions, which can be computed in $O(n \log n)$ time in the size of the predicates if they are in disjunctive normal form.

The selection conditions will need to be changed from time to time to reflect changing user interest. When a slice principal changes its condition, it potentially increases or decreases the information it maintains. The principal computes the difference between the old and new conditions, then performs a special anti-entropy session to both become consistent with another member and retrieve items matching the difference condition. If the change narrows the scope of the slice, the principal can discard consistent items without communicating with other principals.

The slicing mechanism is particularly useful for portable computing systems. These systems may be disconnected from the network, or connected only by a low-bandwidth wireless link. A user can create a small slice principal on their system to keep important information local. The volume of updates to the slice may then be small enough to send over the wireless link. A slice principal can also obtain a summary timestamp vector from a full-copy principal to determine how many updates the slice lacks. When the difference exceeds some bound, the slice principal can prompt the user to connect their machine to a higher bandwidth network, perhaps an Ethernet or a telephone connection, to get the missing updates.

8.3.3 Using slices for resource discovery

Complex information systems should proactively find useful information for their users. Users can specify a query and expect to be notified whenever new information that matches the query becomes available. In Refdbms, the user should be notified when references matching one or more keywords is added to a database.

Many information services, including Refdbms, split their information into several separate databases, so users can have private copies and to reflect different administrative domains. When

```
class locationService {
    // retrieve the principals associated with a group
    memberSet find(groupId gid);
    // merge a sample of a member's view into the service copy
    update(groupId gid, memberSet view, timestamp acktime);
}

locationService LocationService;
```

FIGURE 8.2: The interface to the location service.

multiple databases exist, there is the separate problem of finding out about databases as they are added.

The usual solution is to have a meta-database that lists the databases that are available, and a description of their contents. The meta-database can be built using this architecture as well, perhaps as part of a descriptive naming service [Bowman90]. A user can specify what databases they want to use by specifying a selection condition on the meta-database entries. This condition can be used to build a slice of the meta-database, and user queries can be routed to those databases. As new databases become available, they will be added to the slice and thus become available to the user. The user can install an agent to automatically create a slice of each new database using the user's selection condition.

8.4 Location service

The location service is responsible for mapping a group identifier into a set of addresses of possible group members. Among those possibilities there must be at least one member, as long as the group still exists. This service is used by clients to locate members that can perform operations, and by new members as they join the group.

Weak consistency group communication should be used to build the location service. The combination of the group members and the location service should create less network traffic and require less state than simply placing a principal everywhere people needed to use the service.

Figure 8.2 shows the interface that the service provides. It maintains a database that maps a group identifier onto a timestamped membership view.

From time to time group members send the location service a copy of their membership view and their minimum acknowledgement timestamp. The view and timestamp are merged into the copy the service already has, and the merged values are then propagated as an update message throughout the location service group (Figure 8.3). Recall that the memberSet type that represents a membership view includes a principal identifier, status, and the time at which the principal entered that status. The minimum acknowledgment timestamp is used in the leave protocol to determine when every member has observed the declaration that a principal is leaving the group.

The rules for merging two membership views A and B in the location service are simple, but they are slightly different than those used in the group membership component (Chapter 6). As long as both A and B have an entry for principal p, the later entry is used. However, if A has an entry for p but B does not, the rules must determine whether B is lacking because p has just joined the group, or because p has left the group and A is out of date. The acknowledgment timestamp acts as an implicit death certificate for principals that have failed or left the group: if A's entry for p is less than B's acknowledgment timestamp, then B must have more up-to-date information and p has left the group. On the other hand, if the entry at A is later than B's acknowledgment timestamp, p must have joined the group. This approach allows the location service to receive group membership samples and eventually become consistent with the actual group membership.

The client specification in Section 8.2 and the group join protocol in Section 6.4.3 both assume that the location service will provide at least one group member. The approach presented so far cannot make this guarantee, because any sample of a membership view can become arbitrarily far out of date. It is possible that every principal in that view has failed or left the group since it was recorded. If those principals leave *forwarding addresses* [Fowler85, Jul88] for other group members, then there is always a way to find a member based on location service information.

Often, a change in group membership is most important to nearby clients. For example, when a new principal is added to the group, nearby clients may want to start using the new principal instead of another. The location service can take advantage of this property by using an anti-entropy partner

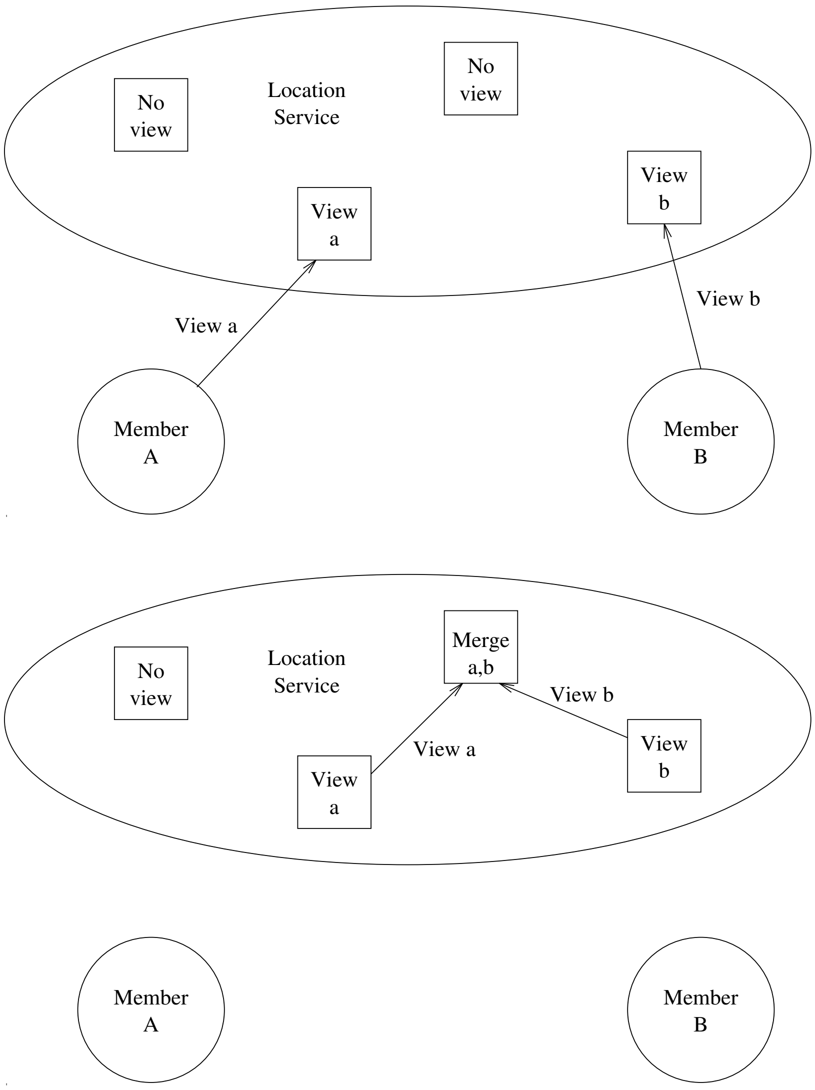


FIGURE 8.3: How the location service receives and propagates samples of membership views. Different group members can send samples to different location servers. These samples are exchanged using TSAE, and eventually merged.

---

selection policy that prefers nearby partners. The performance analysis in Chapter 7 shows that these policies substantially reduce network traffic without adversely affecting message delivery. In this way, a new server becomes visible quickly in its area of the net while eventually becoming visible to more distant regions.

8.4.1 Existing location services

A number of location or name services have been constructed. Most of them can be implemented with multiple replicas, but most do not provide adequate support for locating groups with dynamic membership.

The Internet Domain Name System (DNS) [Mockapetris87] is in use throughout the world, primarily for translating host names to IP addresses. The names are organized hierarchically, and portions of the name space are under separate administrative control. All the information in an administrative *zone* is maintained by an authoritative name server. Name servers can be replicated, and no particular replication technique is required. However, the DNS protocol provides a mechanism for primary-copy replication with periodic refreshes of secondary copies. Information can be cached by local name servers, and a timeout is provided to limit inconsistency.

The division of the name space into different zones is one of the most important features of the DNS. Division of authority keeps the load on any individual server low, allows for a lower degree of replication, and encourages independence between organizations. It also complicates query processing, since a single server can only answer queries related to its zone. DNS zones are linked together at their boundaries using special resource entries, and this information is used to direct a query to a more appropriate server.

The DNS can be extended into the location service described above. A new resource type can be added to the DNS that maintains group view information, and replicated name servers can exchange this information. The current name space is oriented toward host names, which usually possess a high degree of locality within a name zone, but this is not intrinsic to the design.

The Clearinghouse name service was very similar to the DNS, excepting that it could map a name into a list of names, as well as into an address. It used weak-consistency replication rather than primary-copy replication for replicated name servers.

The Cambridge Universal Name Service (UNS) [Ma92] combines consistent and inconsistent replication by introducing two classes of name server. The first class servers are called replicas, and use a consistent update protocol. The second class servers contain read-only copies, and are updated asynchronously from the first class servers. Caches act as an unofficial third class of service to which updates are not propagated.

The Advanced Networked Systems Architecture (ANSA) defines an architecture for object-oriented distributed computing [ANSA89, ANSA91]. It includes many components, including group communication. The *Trader* provides a descriptive name service, which maps content-oriented queries, which include typing information, into references to objects that implement the appropriate service. The references can be used to perform a location-transparent invocation of operations on the service. The underlying mechanism appears to use forwarding addresses to accommodate migration.

Chapter 9
Continuing work

As always in work of this scope, there are many subjects worthy of further investigation.

9.1 Performance

The performance evaluations reported in Chapter 7 are all based on simulation and analysis. As the Refdbms system comes into wider use, many of these performance measures can be re-evaluated based on its logs. The traffic analysis would also benefit from a re-evaluation using real network costs.

It appears that there is a relationship between network traffic and the time required to propagate a message (Section 7.4.4). This relationship is worthy of further investigation; I suspect that there is a simple relation between the two.

9.2 Fault tolerance

The analyses of fault tolerance in the weak-consistency group membership implementation in Chapter 6 assume a completely connected network topology. While it appears that the results in that chapter are easily extended to less-connected networks, I have not yet completed the proofs.

9.3 Reducing space requirements

The TSAE protocol requires maintenance of $\Theta(n)$ timestamps, in addition to the message log. Every principal must maintain information on every other group member, leading to a complete view relation between principals. For some systems, even $\Theta(n)$ state is unacceptably large. This space can be reduced by imposing structure on the view relation so that principals need only know about a few others.

One way to reduce the amount of state is to divide the group into subgroups, and impose a hierarchical structure the subgroups. Members would maintain information on the other members in their subgroup. Some of those members would act as *representatives* to a supergroup. Messages would be sent throughout a subgroup, then forwarded to other subgroups by exchanges between representatives in the supergroup. The hierarchy could be extended to additional layers if needed.

A similar possibility is to structure subgroups as a b-tree. Each subgroup would contain a number of members. When that number becomes greater than a limit, some members would either be added to sibling subgroups, or the group would split. Likewise, if the number becomes too small, subgroups could be coalesced. Messages would be propagated between subgroups using representatives.

A final possibility is to use a simpler, sparsely-connected logical communication topology. The Usenet messaging system [Quarterman86] uses the Unix-to-Unix copy program (UUCP) to propagate messages. The logical (and physical) topology of that network is sparsely connected. Few sites are connected to more than two or three other sites, each site only knows about its neighbors. The Usenet software implements a simple flooding protocol [Tanenbaum81] that forwards each message to neighbor sites. Each site must maintain a log of the message identifiers it has received so that messages do not propagate forever. The message identifiers must be stored long enough that there is no possibility of a message still being in transit somewhere, usually a few weeks to a month.

9.4 Hybrid consistency

There are several variations on the weak consistency model. In one, principals would be clustered into cliques, where all of the principals in a clique would use a consistent (reliable, interactive) message delivery protocol. Messages would be sent between cliques using eventual delivery protocols. It has been conjectured that this approach could reduce the amount of state principals would maintain, allow message logs to be purged more rapidly, and produce lower message traffic than the system presented here.

9.5 Authentication

Every authentication system of which I am aware either requires a trusted authentication service, or the *a priori* presence of shared information [Burrows89b, Burrows89a, Lampson91]. For example, the Kerberos authentication service [Steiner88] uses a trusted server. This violates the assumption that no part of the system is centralized. These systems also implicitly assume a completely connected logical network topology. If the network provides a different logical topology – as the Usenet does – the traditional authentication model no longer holds. It appears that a new model of authentication will be required for mobile, weakly-connected systems. I expect that a probabilistic, rather than absolute, model of truth and belief will be required. I am investigating this problem further.

9.6 Location services

The location service model presented in Section 8.4 uses stable forwarding addresses to ensure that a principal can always find a group member when acting as a client or joining the group. Stable forwarding addresses are somewhat unrealistic, because they require that a principal continue to exist for a long time. When a principal leaves the group, it is likely that the system on which it was operating is being removed from service or changed, and the burden of maintaining forwarding information is not likely to be welcome.

Another possibility is to delegate the maintenance of a forwarding address to another principal. In particular, all requests for a forwarding address could be forwarded to a nearby location server. The forwarding address can then take the form of a group membership view that is maintained just as all views in the location service are. How to locate the location server to which the problem has been delegated, and how to ensure that the location service can eventually purge the forwarding address, are matters for future consideration.

9.7 Refdbms

The Refdbms system is a prototype, and at the time of writing several parts of the distributed system architecture have not been implemented. It uses the TSAE message delivery and the group membership protocols, but without any frills. It does not now provide slices or caches, and clients communicate with a group member through a shared file system.

I plan to add a meta-database and location service to Refdbms. The metadatabase will allow users to find databases of interest by querying database descriptions. The location service will provide a simple database-to-membership mapping until a sufficiently functional standard Internet location service is available. This will also allow clients to contact any group member, rather than requiring each site that will use a database to maintain a copy. Slices, along with a mechanism for submitting persistent queries to notify a user when interesting new references are added to the database, are other priorities.

Chapter 10
Summary

Wide-area distributed systems can be built conveniently, efficiently, and correctly using the group communication framework presented in this dissertation. In the framework, a group communication system is constructed from four components, providing message delivery, message ordering, group membership, and application functions. The group communication system provides ways for one member to multicast a message to the other group members, and for members to join and leave the group.

The system as a whole provides *weak consistency.* A weakly consistent group allows the principals that make up the group to differ at any particular moment on their views of group state, as long as they will eventually reach consensus. The framework provides weak consistency using a reliable, eventual message delivery protocol. Reliable delivery assures that consensus will be reached when every principal receives group messages. Eventual delivery allows the system to delay messages in order to improve efficiency and to wait for transient system faults to be repaired.

This approach to constructing a wide-area system is different from the way most wide-area applications have been developed to date. Most of these applications have been developed in an ad hoc fashion, with no attempt to systematize the process of building wide-area distributed systems. The group communication architecture I have presented is a step toward developing a set of tools and a discipline to make system construction simpler. The architecture is useful as a way to reason about and classify distributed systems, and to define interfaces for reusable modules.

The group communication framework is different from other group communication systems because of its emphasis on weak consistency and on customization to meet application requirements. Systems such as Isis, Arjuna, Psync, and others provide tools for building distributed systems on a smaller scale. They can provide fast, efficient mechanisms with strong guarantees for local-area systems, but they do not provide convenient and efficient mechanisms for wide-area groups that might scale into the thousands of principals. The architecture presented here defines a small number of interfaces between components, so that different implementations of each component can be provided to meet application needs. This is unlike most other distributed programming systems, which provide a fixed, limited set of protocols.

I have developed the timestamped anti-entropy (TSAE) message delivery protocol, which implements reliable, eventual delivery. TSAE uses periodic exchange of messages between pairs of principals to propagate a message throughout the group. These exchanges are called *anti-entropy sessions*. Incoming messages are stored in a message log, and later delivered to the application in some well-defined order. The TSAE protocol maintains summary information on the messages it has received, which is used to improve the efficiency of the exchange, to purge messages from the log, and to order messages for delivery to the application.

The group membership protocol avoids interactive communication between members without compromising the message delivery guarantee. Each principal in the group maintains a *view* of the membership, and these views are exchanged during anti-entropy sessions. Eventually, every group member will receive every membership change and the group will reach consensus. A new member joins the group by obtaining one or more sponsors from among the existing members. These sponsors provide it with a copy of the state of the group. Members can leave the group by declaring their intent to do so, then waiting for the declaration to propagate throughout the membership.

The TSAE message delivery protocol and the related group membership protocol exhibit good performance. I have investigated the fault tolerance of message delivery and group membership, and found that the system responds quickly to failure. The latency in receiving a message is small, and scales well in the number of principals. Likewise, the message traffic imposed on the network scales well with the size of the group. The policy used to select partners influences both message latency and traffic, and policies can be selected to trade latency for traffic or vice versa. Finally, given reasonably frequent anti-entropy sessions, principals will maintain up-to-date views of group state.

Bibliography

[Agrawal91] D. Agrawal and A. Malpani. Efficient dissemination of information in computer networks. *Computer Journal*, **34**(6):534–41 (December 1991).

[Alon87] Noga Alon, Amnon Barak, and Udi Manber. On disseminating information reliably without broadcasting. *Proceedings of 7th International Conference on Distributed Computing Systems* (Berlin), pages 74–81 (September 1987).

[Alonso90a] Rafael Alonso, Daniel Barbará, and Luis L. Cova. Using stashing to increase node autonomy in distributed file systems. *Proceedings of the 9th IEEE Symposium on Reliable Distributed Systems* (October 1990).

[Alonso90b] Rafael Alonso, Daniel Barbará, and Hector Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, **15**(3) (September 1990).

[ANSA89] Architecture Projects Management Ltd. *ANSA: An Engineer's Introduction to the Architecture* (November 1989).

[ANSA91] Architecture Projects Management Ltd. *ANSA: A System Designer's Introduction to the Architecture* (April 1991).

[Bal89] Henri E. Bal. *The Shared Data-Object Model as a Paradigm for Programming Distributed Systems*. PhD thesis (1989). Vrije Universiteit Amsterdam.

[Bal90] Henri E. Bal, Andrew S. Tanenbaum, and M. Frans Kaashoek. Orca: a language for distributed programming. *SIGPLAN Notices*, **25**(5):17–24 (May 1990).

[Barbará86] Daniel Barbará, Hector Garcia-Molina, and Annemarie Spauster. Policies for dynamic vote reassignment. *Proceedings of the 6th International Conference on Distributed Computing Systems* (Cambridge, Massachusetts), pages 37–44 (May 1986).

[Barbará90] Daniel Barbará and Hector Garcia-Molina. The case for controlled inconsistency in replicated data (position paper). *Proceedings of the Workshop on the Management of Replicated Data* (Houston, Texas), pages 35–8 (November 1990).

[Berners-Lee92] Tim Berners-Lee, Robert Cailliau, Jean-François Groff, and Bernd Pollermann. World-Wide Web: the information universe. *Electronic Networking: Research, Applications, and Policy*, **1**(2) (Spring 1992).

[Bernstein84] Philip A. Bernstein and Nathan Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, **9**(4):596–615 (December 1984).

[Bernstein87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems* (1987). Addison-Wesley, Reading, Massachusetts.

[Birman87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, **5**(1):47–76 (February 1987).

[Birman90] Kenneth Birman, Andre Schiper, and Pat Stephenson. Fast causal multicast. Technical report TR–1105 (13 April 1990). Department of Computer Science, Cornell University.

[Birman91] Kenneth P. Birman, Robert Cooper, and Barry Gleeson. Programming with process groups: group and multicast semantics. Technical report TR–91–1185 (29 January 1991). Department of Computer Science, Cornell University.

[Birrell87] Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber. A simple and efficient implementation for small databases. *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, Texas). Published as *Operating Systems Review*, **21**(5):149–54 (November 1987).

[Bloch87] Joshua J. Bloch, Dean S. Daniels, and Alfred Z. Spector. A weighted voting algorithm for replicated directories. *Journal of the ACM*, **34**(4):859–909 (October 1987).

[Bowman90] C. Mic Bowman. Univers: the construction of an internet-wide descriptive naming system. Technical report TR 90–27 (10 August 1990). Department of Computer Science, University of Arizona.

[Burrows89a] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (Litchfield Park, Arizona). Published as *Operating Systems Review*, **23**(5):1–13 (December 1989).

[Burrows89b] Michael Burrows, Martín Abadi, and Roger Needham. A login of authentication. Technical report 39 (February 1989). Digital Equipment Corporation Systems Research Center, Palo Alto, CA.

[Campbell92] Roy H. Campbell, Nayeem Islam, and Peter Madany. Choices, frameworks, and refinement. *Computing Systems*, **5**(3):217–57 (Summer 1992). Usenix Association.

[Cheriton84] David R. Cheriton and Willy Zwaenepoel. One-to-many interprocess communication in the V-system. Technical report STAN–CS–84–1011 (August 1984). Computer Systems Laboratory, Department of Computer Science, Stanford University.

[Comer88] Douglas Comer. *Internetworking with TCP/IP: principles, protocols, and architecture* (1988). Prentice Hall, Englewood Cliffs, NJ.

[Cristian86] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: from simple message diffusion to Byzantine agreement. Technical report RJ 5244 (30 July 1986). IBM Almaden Research Center.

[Cristian89] Flaviu Cristian. A probabilistic approach to distributed clock synchronization. *Proceedings of the 9th International Conference on Distributed Computing Systems* (Newport Beach, CA), pages 288–96 (1989).

[Cristian90] Flaviu Cristian. Synchronous atomic broadcast for redundant broadcast channels. Technical report RJ 7203 (December 1989, revised April 1990). IBM Almaden Research Center.

[Cristian91] Flaviu Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, **4**(4):175–87 (1991).

[Davčev85] Daňco Davčev and Walter A. Burkhard. Consistency and recovery control for replicated files. *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, **19**(5):87–96 (December 1985).

[Demers88] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *Operating Systems Review*, **22**(1):8–32 (January 1988).

[Demers89] Alan Demers, Mark Gealy, Dan Greene, Carl Hauser, Wes Irish, John Larson, Sue Manning, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry, and Don Woods. Epidemic algorithms for replicated database maintenance. Technical report CSL–89–1 (January 1989). Xerox Palo Alto Research Center, CA.

[Deutsch92] Peter Deutsch. Resource discovery in an Internet environment. Master's thesis (June 1992). School of Computer Science, McGill University.

[Downing90a] Alan R. Downing, Ira B. Greenberg, and Jon M. Peha. OSCAR: an architecture for weak-consistency replication. *Proceedings of IEEE PARBASE-90* (March 1990).

[Downing90b] Alan R. Downing, Ira B. Greenberg, and Jon M. Peha. OSCAR: a system for weak-consistency replication. *Proceedings of the Workshop on Management of Replicated Data* (Houston, Texas), pages 26–30 (November 1990).

[El-Abbadi86] A. El-Abbadi and S. Toueg. Availability in partitioned replicated databases. *Proceedings of the 5th SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 240–51 (1986).

[Emtage92a] Alan Emtage. Personal communication (1992). Electronic mail message.

[Emtage92b] Alan Emtage and Peter Deutsch. Archie – an electronic directory service for the Internet. *Proceedings of the Winter 1992 Usenix Conference* (San Francisco), pages 93–110 (January 1992).

[Fischer85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, **32**(2):374–82 (April 1985).

[Fowler85] Robert Joseph Fowler. *Decentralized object finding using forwarding addresses*. PhD thesis, published as Technical report 85–12–1 (December 1985). University of Washington.

[Ganatra92] Nitin K. Ganatra. Census: collecting host information on a wide area network. Technical report UCSC–CRL–92–34 (June 1992). Computer and Information Sciences Board, University of California at Santa Cruz.

[Garcia-Molina88] Hector Garcia-Molina and Boris Kogan. An implementation of reliable broadcast using an unreliable multicast facility. *Proceedings of the 7th Symposium on Reliable Distributed Systems* (Ohio State University, Columbus, OH), pages 101–11 (10–12 October 1988).

[Gifford79] D. K. Gifford. Weighted voting for replicated data. *Proceedings of the 7th ACM Symposium on Operating Systems Principles* (Pacific Grove, California), pages 150–62 (December 1979).

[Golding91a] Richard Golding and Darrell D. E. Long. Accessing replicated data in a large-scale distributed system. *International Journal in Computer Simulation*, **1**(2) (1991, to appear).

[Golding91b] Richard A. Golding. Accessing replicated data in a large-scale distributed systems. Master's thesis; published as Technical report UCSC–CRL–91–18 (June 1991). Computer and Information Sciences Board, University of California at Santa Cruz.

[Golding92a] Richard Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, **5**(4) (Fall 1992). Usenix Association.

[Golding92b] Richard Golding. End-to-end performance prediction for the Internet – progress report. Technical report UCSC–CRL–92–26 (June 1992). Computer and Information Sciences Board, University of California at Santa Cruz.

[Golding92c] Richard A. Golding. A weak-consistency architecture for distributed information services. Technical report UCSC–CRL–92–30 (June 1992). Computer and Information Sciences Board, University of California at Santa Cruz.

[Golding92d] Richard A. Golding and Darrell D. E. Long. Quorum-oriented multicast protocols for data replication. *Proceedings of the 8th International Conference on Data Engineering* (Tempe, Arizona), pages 490–7 (February 1992).

[Gray86] Jim Gray. Why do computers stop and what can be done about it? *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 3–11 (1986).

[Heidemann92] John S. Heidemann, Thomas W. Page, Richard G. Guy, and Gerald J. Popek. Primarily disconnected operation: experiences with Ficus. *Proceedings of 2nd Workshop on the Management of Replicated Data* (Monterey, CA), pages 2–5 (November 1992).

[Hisgen90] Andy Hisgen, Andrew Birrell, Chuck Jerian, Timothy Mann, Michael Schroeder, and Garret Swart. Granularity and semantic level of replication in the Echo distributed file system. *Proceedings of the Workshop on the Management of Replicated Data* (Houston, Texas), pages 2–4 (November 1990).

[Islam92] Nayeem Islam and Roy H. Campbell. Object-oriented framework design and implementation. Technical report UIUCDCS–R–92–1737 (March 1992). Department of Computer Science, University of Illinois at Urbana-Champaign.

[Jain91] Raj Jain. *The Art of Computer Systems Performance Analysis* (1991). John Wiley, New York.

[Jajodia87] Sushil Jajodia and David Mutchler. Dynamic voting. *Proceedings of the ACM SIGMOD 1987 Annual Conference*, pages 227–38 (May 1987).

[Jul88] Eric Jul. *Object mobility in a distributed object-oriented system*. PhD thesis, published as Technical report 88–12–06 (December 1988). Computer Science Department, University of Washington.

[Kahle89] Brewster Kahle. Wide area information server concepts. Technical report TMC–202 (3 November 1989). Thinking Machines Corporation.

[Kahle91] Brewster Kahle and Art Medlar. An information system for corporate users: wide area information servers. Technical report TMC–199 (8 April 1991). Thinking Machines Corporation.

[Kistler91] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (Pacific Grove, CA), pages 213–25 (13 October 1991).

[Ladin90] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Lazy replication: exploiting the semantics of distributed services. Technical report MIT/LCS/TR–484 (July 1990). Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

[Ladin91] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: exploiting the semantics of distributed services. *Position paper for 4th ACM-SIGOPS European Workshop* (Bologna). Published as *Operating Systems Review*, **25**(1):49–55 (January 1991).

[Lamport78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**(7):558–65 (1978).

[Lampson91] Butler Lampson, Martĭn Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 165–82 (13 October 1991).

[Leffler89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *Design and implementation of the 4.3BSD UNIX operating system* (1989). Addison-Wesley.

[Lesk78] M. E. Lesk. Some applications of inverted indexes on the UNIX system. Computing Science technical report 69 (June 1978). Bell Laboratories.

[Liskov87] Barbara Liskov. Highly-available distributed services. Programming Methodology Group Memo 52 (February 1987). Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

[Little90] Mark C. Little and Santosh K. Shrivastava. Replicated k-resilient objects in Arjuna. *Proceedings of the Workshop on Management of Replicated Data* (Houston, Texas), pages 53–8 (November 1990).

[Long88] Darrell D. E. Long and Jehan-François Pâris. A realistic evaluation of optimistic dynamic voting. *Proceedings of the 7th IEEE Symposium on Reliable Distributed Systems* (Columbus, OH), pages 129–37 (October 1988).

[Long91] Darrell D. E. Long, John L. Carroll, and C. J. Park. A study of the reliability of Internet sites. *Proceedings of the 10th IEEE Symposium on Reliable in Distributed Systems* (Pisa, Italy), pages 177–86 (September 1991).

[Long92] Darrell D. E. Long. A replicated monitoring tool. *Proceedings of the 2nd Workshop on the Management of Replicated Data* (November 1992).

[Lottor92] Mark K. Lottor. Internet growth (1981–1991). RFC 1296 (January 1992). Network Information Systems Center, SRI International.

[Ma92] Chaoying Ma. *Designing a Universal Name Service*. PhD thesis (1992). University of Cambridge Computer Laboratory.

[Mann89] Timothy Mann, Andy Hisgen, and Garret Swart. An algorithm for data replication. Report #46 (June 1989). Digital Equipment Corporation Systems Research Center, Palo Alto, CA.

[Mattern88] Friedemann Mattern. Virtual time and global states of distributed systems. Technical report SFB124–38/88 (October 1988). Department of Computer Science, University of Kaiserslautern.

[Mills88] D. Mills. Network Time Protocol (version 1) specification and implementation. Network Working Group RFC 1059 (July 1988). Network Information Center.

[Mishra89] Shikavant Mishra, Larry L. Peterson, and Richard D. Schlichting. Imnplementing fault-tolerant replicated objects using Psync. *Proceedings of the 8th Symposium on Reliable Distributed Systems* (Seattle, WA), pages 42–52 (10–12 October 1989).

[Mishra92] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Protocol modularity in systems for managing replicated data. *Proceedings of the 2nd Workshop on the Management of Replicated Data* (Monterey, CA), pages 78–81 (November 1992).

[Mockapetris87] P. Mockapetris. Domain names – concepts and facilities. RFC 1034 (November 1987). ARPA Network Working Group.

[Mullender86] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based distributed operating system. *Computer Journal*, **29**(4):289–99 (1986).

[Mullender90] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: a distributed system for the 1990s. *IEEE Computer*, **23**(5):44–53 (May 1990).

[Oppen81] D. C. Oppen and Y. K. Dahl. The Clearinghouse: a decentralized agent for locating named objects in a distributed environment. Technical report OPD–T8103 (1981). Xerox Office Products Division, Palo Alto, Ca.

[PageJones88] Meilir Page-Jones. *The Practical Guide to Structured Systems Design*, second edition (1988). Yourdon Press.

[Postel80] J. Postel. *Transmission Control Protocol*, RFC 761 (January 1980). USC Information Sciences Institute.

[Pu91a] Calton Pu and Avraham Leff. Epsilon-serializability. Technical report CUCS–054–90 (15 January 1991). Department of Computer Science, Columbia University.

[Pu91b] Calton Pu and Avraham Leff. Replica control in distributed systems: an asynchronous approach. Technical report CUCS–053–090 (8 January 1991). Department of Computer Science, Columbia University.

[Quarterman86] John S. Quarterman and Josiah C. Hoskins. Notable computer networks. *Communications of the ACM*, **29**(10):932–71 (October 1986).

[Ricciardi91] Aleta M. Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environments. Technical report TR91–1188 (7 February 1991). Department of Computer Science, Cornell University.

[Rumbaugh91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design* (1991). Prentice-Hall, Englewood Cliffs, NJ.

[Schatz90] Bruce Raymond Schatz. Interactive retrieval in information spaces distributed across a wide-area network. Technical report TR 90–35 (December 1990). Department of Computer Science, University of Arizona.

[Seltzer90] Margo Seltzer and Michael Stonebraker. Transaction support in read optimized and write optimized file systems. Technical report UCB/ERL M90/37 (April 1990). Electronics Research Laboratory, College of Engineering, University of California at Berkeley.

[Steiner88] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: an authentication service for open network systems. *USENIX Winter Conference* (Dallas, TX), pages 191–202 (9–12 February 1988).

[Sullivan92] Mark Sullivan and Michael Olson. An index implementation supporting fast recovery for the POSTGRES storage system. *Proceedings of the 8th International Conference on Data Engineering*, pages 293–300 (February 1992).

[Sun88] Sun Microsystems, Incorporated. *Network Programming* (1988).

[Tanenbaum81] A. S. Tanenbaum. *Computer networks* (1981). Prentice-Hall.

[Tanenbaum86] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert Van Renesse. Using sparse capabilities in a distributed operating system. *Proceedings of the 6th International Conference on Distributed Computing Systems* (Cambridge, Mass), pages 558–63 (May 1986).

[Terry85] Douglas Brian Terry. *Distributed name servers: naming and caching in large distributed computing environments*. PhD thesis, published as Technical report CSL–85–1 (February 1985). Xerox Palo Alto Research Center, CA.

[Thomas79] R. H. Thomas. A majority consensus approach to concurrency control. *ACM Transactions on Database Systems*, **4**:180–209 (1979).

[Turek92] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, **25**(6):8–17 (June 1992).

[Tuthill83] Bill Tuthill. Development of refer: Bug Fixes and Enhancements (or (unofficially) “Refer Madness”). *Usenix Conference Proceedings* (San Diego, CA), pages 99–103 (Winter 1983). Usenix Association.

[Wilkes91] John Wilkes. The refdbms bibliography database user guide and reference manual. Technical report HPL–CSP–91–11 (20 May 1991). Hewlett-Packard Laboratories.