June 1986

Report No. STAN-CS-86-1133
AlsonumberedCSL-TR-86-311

Distributed, Replicated Computer Bulletin Board Service

by

Judy Lynn Edighoffer

Department of Computer Science

Stanford University
Stanford, CA 94305

Distributed, Replicated Computer Bulletin Board Service

Judy Lynn Edighoffer

Abstract

Computer systems offer a variety of services to assist communication between people. This dissertation examines computer bulletin boards, one such facility that allows recipients to arrange for the delivery of messages on topics of personal interest. The thesis focuses on the problems of replication and cost scaling.

It is no longer necessarily true that users are closely tied to a single host, yet current methods for replicating bulletin boards do not provide a good way to represent what a person has seen that is independent of the copy read. Existing replication algorithms either don't support copy-independent read records or offer too little concurrency for this application. An original replication algorithm provides a copy-independent ordering for submissions using just a single copy of a bulletin board during the execution of the user operations. The algorithm works well even on a network frequently in a state of partition.

A more significant problem from the viewpoint of computer system administrators is the cost of a distributed bulletin board service. In existing mail systems and bulletin board systems, such as distribution lists on the Arpanet and USENET running under UNIX[1], the cost per participating computer tends to grow in proportion to the network size. The causes for this poor scaling will be examined, then it will be explained how a structured name space together with suitable operations on it leads to improved scaling by encouraging the creation of highly specialized bulletin boards.

This research was supported in part by the Defense Advanced Research Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431. It is a slightly modified version of the PhD thesis of the author. Special thanks go to Keith Lantz for his advice and support, to Joe Pallas for implementing the user-friendly user agent, to Chris Lauwers and Matt Zekauskas for implementing an SMTP gateway, and to the many people who quickly fixed bugs in the experimental version of the V-System, especially Rob Nagler and Lance Bcrc.

[1] trademark Bell Laboratories

Contents

CARNEGIE-MELLON UNIVERSITY
PITTSBURGH PENNSYLVANIA 15213

List of Figures

List of Tables

Chapter 1

Introduction

Computer-based message systems enable people to transmit messages to be read at the convenience of the recipients. Especially when extended over a network, these systems provide a very convenient method of communication, in some ways more convenient than other communications facilities. Nowadays, almost every computer system provides some sort of message facility. Networks such as the Arpanet, CSNet, and UUCP/USENET tie many computer science researchers together while commercial networks such as Tymnet and Telenet provide computer-based message services to business. A multitude of computer conferencing services exist, including Forum, NOTEPAD, and EIES.

Computer-based message systems pose a number of problems, ranging from political and legal aspects to purely technical issues. This thesis will examine the class of message systems henceforth referred to as *bulletin board systems.* Intuitively, a bulletin board system can be thought of as a mechanism for allowing people to speak about a topic rather than to a particular audience.

The focus will be on technical issues, particularly those related to the effects of increased network interconnection. To limit the difficulties of the problem, some matters will be ignored. In particular, this thesis will not address the problems of data representation on heterogeneous hardware, software protocols for implementing a store-and-forward network, and robustness in the face of corrupted data, forged messages, or incorrect or malicious agents.

While there are many interesting problems in the area of man/machine interaction, the subject of designing the ideal user interface is deemed to be beyond the scope of this work. The user interface will only be addressed in enough detail to determine what support must be provided to enable others to experiment with smart user interfaces. Legal and political issues, such as liability for libel suits and establishing standards, will not be addressed at all.

1.1 Demand for Bulletin Board Service

Computer-based message systems are useful[Hol80,HT78j. Of the many existing computer-based message systems that support some form of bulletin board, most are computer mail systems. Only a few are designed specifically to provide bulletin board services.

The usefulness of computer mail is reflected in the number of proposed standards, covering such diverse areas as the organization of the service, accessing messages from personal workstations, naming, and translation between different mail systems [Deu81,Hor83,Hor86,Nat83,Pal85,Pic79,Pos82,Ros83]. The interest is also reflected in the usage of computer mail and bulletin board systems. For example, Grapevine is a computer mail system that was offered to a large user community. By the summer of 1983, Grapevine served about 4,400 users[SBN84]. An average of almost two messages a day were sent by

each user. Each read an average of eight messages a day. Statistics gathered from several nearby computers participating in USENET indicate about 25% of all the users use that bulletin board system[Art>86]. This frequency of usage is certainly comparable to that of other major computer software packages.

The other side of the issue is the amount of time spent by individual users sorting through their messages. A moderate reading speed for a paperback novel is about 40 pages per hour. Sampling paperbacks indicates that a page contains roughly 2,250 characters. An moderately fast reader, then, could read roughly 90,000 characters per hour. An average of 108 messages a day were read or skipped over by readers of USENET according to a sample of the most recent reports on ba.news.ratings[Arb86]. Examining the size of the messages posted to the 'net.*' news groups at Navajo indicated the average size was 2025 characters, including all headers. Therefore, a user would spend about 2.43 hours a day reading the full text of messages posted to USENET, assuming that the overall average message size is the same as the average size of the messages typically read.

A similar examination of the messages posted to 'SU-BBoard[5] during the first two weeks of February 1986 yielded a mean of approximately 850 characters in the from line, the subject line, and the message body. Just the from and subject lines averaged about 80 characters. Posting statistics gathered by Lynch indicated an average of 28.3 messages a day were posted to SU-BBoard for the first half of the 1985-86 academic year[Lyn86]. A reader of SU-BBoard would spend 16 minutes a day if he read every message. Allowing for transmission of the messages at 1200 baud would, add roughly another 3 minutes. The disruption of scrolling would almost certainly keep the reading rate down below that used in these calculations. A person following several active bulletin boards could easily find himself spending an hour a day.

Usage is also reflected in the costs incurred by computer-based message systems. The proposal for CSNet [DHK83] estimated that each moderately active user would contribute about $250 per year in phone usage charges or about $75 per year in Telenetusage charges. Heavy users were estimated as producing more than double that traffic.

The costs of running USENET are of the same order of magnitude. Because its costs have been growing rapidly enough to worry many USENET administrators, R. Adams has been collecting statistics on the traffic on one machine[Ada85]. Figure 1.1 measures usage of USENET over the past year in terms of the number of bytes of all messages received by Seismo over two week intervals.

Many, if not most, sites receive a full copy of all these articles over ordinary modems. Estimating the average effective baud rate as 800, the transfer time amounts to about 2.5 hours. Multiple links of course result in multiple transmissions. Some small sites will choose to receive only a partial copy. Others will receive a full copy themselves and forward it to a number of others. Thus, servicing USENET can command a significant amount of CPU time and tie up a modem and phone line for a significant fraction of a day. When one considers that some of the transfers are long-distance calls, a computer facility can run up a noticeable phone bill, although it may still be a small fraction of the cost of the computer maintenance. Using an estimate of $.15 per minute, the cost would run about $22.5/day or $8200 per year. Notice, as well, that since most news groups are kept on line for a couple of weeks at the major sites, the storage requirement runs over 10 megabytes.

While these costs don't dominate the cost of running a large computer facility, neither are they negligible. Both demand and cost appear to be at least in the same order of magnitude as those of other computer facilities that have been the subject of research for many years. The demand for computer mail in general and bulletin boards in particular is great enough to justify concern about the efficiency of their implementations. Inefficiency can arise either because users are awkwardly trying to cope with services offering the wrong functionality or because the implementations are inefficient.

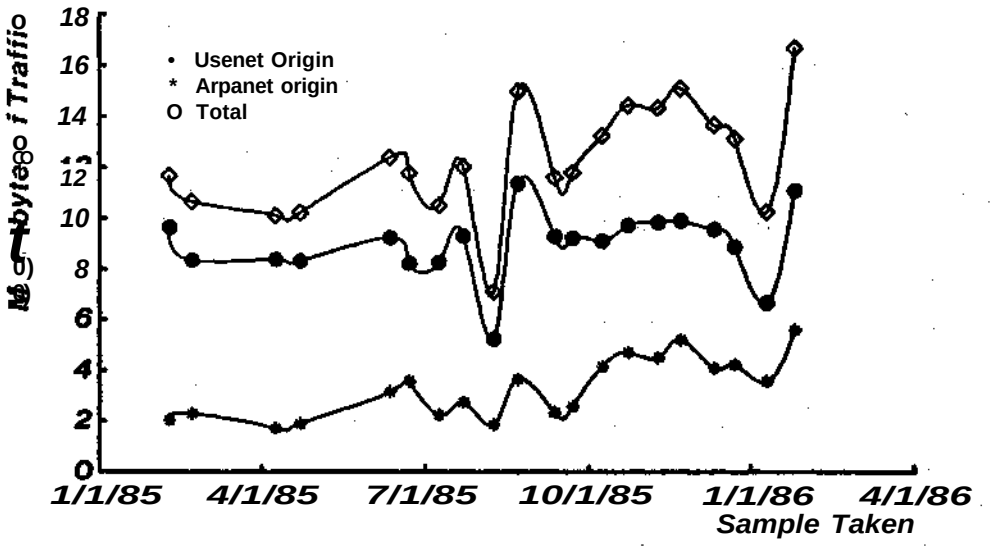

Figure 1.1: Volume of USENET Activity

1.2 Definition of a Bulletin Board

Before one can even think about how to implement a computer bulletin board facility, one must consider: what is a computer bulletin board? That question can be answered by considering what sort of communication activities a bulletin board is intended to support.

1.2.1 Patterns of Communication

The term *communication* covers a wide variety of activities. In everyday life, it is simple to identify basic patterns of communication. The most common one has two participants: a speaker and a listener. It occurs in letters sent via the post office, phone calk, and a host of other situations.

The second pattern of communication also has a distinct speaker to whom all are listening. In this case, however, the speaker delivers his message to every member of a group. Single speaker/multiple listener communication occurs in speeches, newspaper advertisements, posters stapled to telephone poles, and many other ways.

A third pattern of communication covers the remaining situations. It has no single speaker. Instead, the participants act roughly as peers, each addressing his contributions to the others in the group. Examples of such a peer interactions include business meetings and conference calls using the telephone system.

Each of these three patterns can be subdivided according to whether the speakers are able to precisely identify their peers/audience beforehand. Usually, letters are mailed to a known recipient, but they can be sent to a person fulfilling a particular function. For example, a letter complaining about a product might be sent to the customer relations department of the manufacturer rather than a particular employee working there. Phone calls can be made to a hot-line number or a horoscope service, as well as to acquaintances. Memos are typically circulated to an audience of known composition. Speeches may either be made at formal affairs to which the listeners are individually invited or they may be delivered to whoever happens to show up at an advertised time and place. In conference calls and formal group meetings, the membership of the group is usually well known in

advance. For a public town meeting, it may not be.

Communication can be further classified as to the duration of the interaction: one-shot versus an on-going conversation. For example, a letter or phone call might be part of a regular exchange or it might be a singular occurrence. A speech, memo, or advertisement may be a regularly scheduled event, perhaps intended to bring others up to date, or it may be a special one-shot event. A meeting can be a monthly progress review or it can be a reaction to a crisis with no expectation that there will ever be a follow up meeting.

There are other ways to classify forms of communication[KC85,Val84]. One approach is to contrast *communication* against *storage.* These terms distinguish what sort of rendezvous must be achieved by the speakers and the listeners. *Pure communication*requires all participants to get together at a particular point in time. The message may be transmitted to multiple locations, however, so that they need not be in the same place. The telephone, in the absence of a telephone answering machine, provides pure communication. At the other end of the spectrum, *pure storage* requires the speakers and listeners to get together at a common physical location, but not necessarily at a common time. The message is preserved in some manner and made available over a period of time. An everyday example of pure storage is the act of leaving a note on a desk when its occupant is absent. Some forms of communication require rendezvous in both time and place, such as a face-to-face meeting. Others, such as postal mail, require neither.

1.2.2 Types of Computer-Based Message Systems

The problem at hand is to deal with *computer-based message systems,* in which computers are used to assist in the transmission of messages. These services can be defined according to types of communication they support [Val84].

Computer mail handles a single speaker, provided the speaker knows the listener he wishes to address[BLNS82,DHM*81,DHK83,MM 81,NSH83,Str85,Vit81b]. It provides the benefits of both communication and storage, transmitting messages to other locations and storing them until the listener is ready to receive them. Most mail systems allow sustained interactions, but few of them provide good explicit support for conversations. Some researchers have been experimenting with ways to recognize when a set of users have begun a conversation (sustained interaction) or when one conversation has split into a collection of digressions, what data structures to use to represent conversations, and how to present them to users[Pet85].

Through the use of *distribution lists,* computer mail can also handle the single-speaker/multiple-listener case. Distribution lists, also known as *exploders,* provide a shorthand method for specifying a list of recipients. They simplify the task of regularly sending messages to the same audience.

Computer conferencing is a more specialized service focusing on handling structured peer group interactions[Dan81,HT78,KC85,Mee85,Val84]. In particular, they generally recognize special roles such as group moderators. The term /computer conferencing' has been applied to a range of services. While communication is provided by all, the amount of storage available varies from none to long-term storage of all messages.

Althbugh some existing systems labeled as computer mail or computer conferencing support access by an audience of unknown composition, for the purposes of this thesis, sxich services are defined as being those that handle patterns of communication in which the audience* is known. The identification of an individual can be in terms of his position or by membership in an organization. Note that while an individual might not personally know all those participating in a conference, it is assumed that there is a registration process or conference organizer to determine who participates. Users are not free to join and leave as they please.

If a computer-based message system is designed to support unknown audiences and

provides both communication and storage, it is a *bulletin board service*. A bulletin board service differs from other message systems because of the complexities in matching messages with listeners. Basically, the recipient is responsible for deciding whether to receive a particular message. A bulletin board service should be prepared for users who regularly inquire about new postings pertinent to their favorite topics. It must keep track of which messages each regular recipient has already seen.

The prototype bulletin computer board service described in this thesis deals only with text messages. However, bulletin board service is defined in terms of the type of speaker/listener interaction and rendezvous supported, not in terms of the format of messages. The use of text versus multi-media messages is a matter orthogonal to the definition of a bulletin board service.

1.3 Functional Requirements

Bulletin board systems are much like mail systems and so have many of the same requirements[HM79,Hui85,MD76]. By definition, a bulletin board system must provide both the communication and storage of messages. Some applications of bulletin boards suggest that messages should be kept around for as long as storage constraints permit. For example, consider a bulletin board holding for-sale messages. A person entering the market for a car would like be told about cars currently for sale, even if the for-sale messages were originally posted before he started reading the bulletin board.

The recipients of a message are not an enumerated list of specific individuals, but include anyone who expresses interest in the message, particularly in its topic. A bulletin board service must be able to inspect messages to determine their topics. Readers must also be able to describe their interests in a manner comprehensible to the bulletin board service. Essentially, these two requirements mean that the service must provide a method of naming messages according to their topics.

Of course, a bulletin board system must be palatable to those who use it. Proper choice of a naming mechanism is critical to acceptance. In the absence of an artificial intelligence capable of understanding natural language, the burden of identifying the content of a message must fall upon humans. A speaker must identify the subject of his message to the bulletin board system. This implies that users posting messages must be aware of what names are legal in the name space. This knowledge is also required of users specifying what they wish to receive. Ideally, names should reflect users' intuition, but because people differ in what they find intuitive, there should be a method for inquiring about existing names.

Another aspect of palatability is to minimize the amount of effort people must exert to use or maintain the system. For example, readers must be granted access to new bulletin boards. Insofar as protection concerns permit, the interested user should be able to arrange access directly with the bulletin board system. If this function is suitably automated, there will be no need to bother a local administrator.

Any distributed bulletin board service must be prepared to give good service despite having its clients spread out over a network. Since a large network can be in a state of network partition frequently, handling the common operations should not require action by a central authority. In fact, operations should be handled locally when possible. Fast response further requires that servicing a request must either involve few cooperating locations or be finished off-line so that no user must wait.

Another, related aspect is the the subject of transparency. Bulletin board systems are much easier to implement if it can be assumed that users will always connect from a particular site. However, the validity of such an assumption is becoming increasingly dubious. It has grown more common to dial in to a network from a mobile personal computer. Furthermore, there is a trend toward the use of workstations that depend on larger computers to offer services such as mail. For reliability, many services are replicated

on the larger machines. A bulletin board system should be prepared to let users access any copy of a bulletin board. It should be transparent to the user as to which copy is actually used.

In summary, a bulletin board system should:

1. Provide message storage and retrieval. Notices should normally be stored as long as storage constraints permit.
2. Name messages by encoding their subject matter.
3. Present a friendly user environment. In particular, it should use intuitive names, allow users to inquire as to what names are recognized, and provide access to resources anywhere on the network.
4. Minimize the work done by users.
5. Service requests in a distributed manner, not through a central authority.
6. Handle requests for which a user must wait using a minimal number of cooperating locations.
7. Not assume that users always will use the same copy of a replicated object.

There may be other desirable functions. Possibly, some representation of a conversation would be appropriate[Pet85]. In special cases, more sophisticated protection or classification schemes would be appropriate [JN81,LHM84]. Having messages take some responsibility for their own routing or presentation format is another possibility[Tsi84,Vit81a]. The bulletin board system might also provide some sort of confirmation as to delivery or reception[Sch81]. This thesis, however, focuses on those aspects of bulletin board systems that directly influence its costs. Thus, the matter of what added functionality to build on top of its basic facilities will be left as an open question.

1.4 Performance Requirements

The performance of a bulletin board system can be measured in several ways. Three important costs are those of storage space, volume of network traffic, and CPU usage. Not only the total and average values of the requirements should be examined, but the distribution of the costs for different operations and over individual machines as well. It does little good to have a constant CPU cost per user if the entire cost must be borne by a single location. That single computer would almost certainly become a major bottleneck as the network grows.

A critical performance requirement for a bulletin board system is the need to scale almost ideally as new users start participating: that is, the cost per user must remain nearly constant. The importance of good scaling can be seen when one considers how fast a network supporting a bulletin board system may grow. USENET connects a large number of computers running the Unix operating system via an informal network achieved largely through periodic exchanges over phone lines[NL79]. It started in the spring of 1980 with half a dozen or so hosts in North Carolina. By the summer of 1981, some 20 to 30 machines participated. The growth rate in the number of machines can be seen more dramatically in month-by-month data for the year 1984 collected by Mark Horton[Hor85]. He and others have worked to provide connectivity data so that sites can compute cost efficient paths to other sites. Additional figures were obtained by counting the number of sites listed in a local copy of computed paths, the file yusr/lib/news/maps/palias.glacier' on SU-Navajo.

As the plot in Figure 1.2 shows, the number of machines participating has been growing exponentially. If this growth rate continues and the bulletin board system either scales

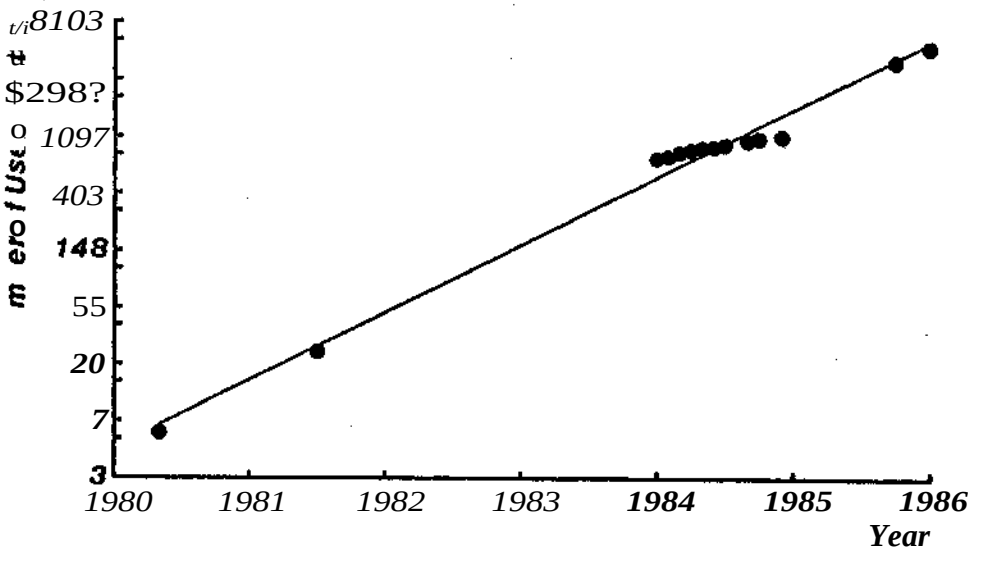

Figure 1.2: Number of Nodes Participating in USENET

poorly or does not divide the load evenly, the costs will soon become prohibitive.

Because current systems tend to scale poorly, they tend to inundate readers with more messages than they are actually interested in. This reduces the palatability of the system. One reflexive response to handling this problem is to designate moderators for the larger distribution lists. A moderator receives all the messages addressed to the distribution Sst audience. He or she is then responsible for censoring and editing the submissions and for periodically sending out digests to the audience. Moderators are often hard to come by since it is generally a job requiring a significant amount of time for no pay. The real solution here is to improve scaling.

Poor scaling leads to unpalatable system behavior in a related situation. When the readership is large, the number of responses is often large. The burden on the ordinary reader can be reduced by assigning someone the task of filtering out or summarizing replies. That moderator, however, will still have a lot of work to do.

1.5 Review of Existing Bulletin Board Systems

A few bulletin boards are implemented as files available only on a particular computer system. Most are produced by distribution lists using computer mail facilities. The latter method is used in the Grapevine[BLNS82] and Arpanet[Cro82,Pos82] environments. An entry of a distribution list may be an individual user, a shared bulletin board file for a site, or another, nested, distribution list. The functionality of a bulletin board is approximated through the use of a distribution list. Users contact the maintainer of the distribution list to either start or stop following a particular bulletin board.

USENET is a more sophisticated bulletin board service[Hor83,NL79]. Its participants are loosely organized. To join, a site must only get permission from some other participating site to exchange USENET messages. There is no global administration. Bulletin boards — referred to as *news groups —axe* named hierarchically. To encourage locality of reference, USENET includes some regional news groups and allows messages to be tagged as being for distribution only within a particular region. USENET software will present

the contents of news groups corresponding to all the subtopics of a given topic. In fact, a user can ask to see all subtopics except for an enumerated Hst of those he is not interested in. A user profile is kept for each reader listing the news groups followed together with what has been seen in each.

1.5.1 Scaling Problems

Existing bulletin board systems scale poorly. Distribution lists do not scale well in terms of the processing time needed to expand a distribution list. For example, in the Grapevine environment, each expansion of one list took more than 10 minutes [SBN84]. Performance depends heavily on how far the system has to go to get a copy of the distribution list. Trying to fully expand a distribution list by gathering together the contents of nested list is most expensive, time-wise. Since a distribution list must be expanded for every posting, expansion time becomes a significant cost.

Similar problems have arisen on the Arpanet. An extended discussion on the Arpanet MsgGroup distribution list brought up many of them[Msg80]. A number of individuals responsible for maintaining the larger lists complained of the pressure they were under to reduce costs. In reaction, distribution lists were often re-structured to contain references to other distribution lists. Expansion for users in a particular community such as at Xerox might be deferred until the posting is forwarded to the Xerox redistribution point. The size of the main distribution list is reduced as the expansion costs are more fairly distributed. This solution, however, requires continued watchfulness for new bottlenecks and manual restructuring of distributions lists to compensate.

In USENET, the regional distribution facilities are not utilized enough to prevent growth that threatens to swamp the system. Many network-wide bulletin boards are replicated at most sites. Thus, the costs expressed in Section 1.1 are incurred by all sites. As the network grows, the costs at each site grow as well. Unfortunately, it may well be difficult to do much more in the USENET environment. While the high degree of autonomy makes for an informal atmosphere that can be pleasant, it also makes it difficult even to get a full accounting of what sites are participating. There is no provision for figuring out where news groups are read and ensuring that copies reach only those destinations with a minimal burden on other sites.

1.5.2 Consequences of Imperfect Replication

The bulletin boards for the Stanford computer science community demonstrate many of the problems that arise from the use of ad hoc replication algorithms. The Stanford computer science community consists of a number of sub-communities. These have many overlapping interests, but also some distinct ones. There is an attempt to satisfy the common interest by creating the effect of a general bulletin board, 'SU-Bboard', by forwarding postings from one site to another. However, replication via distribution lists or mail forwarding does not always provide desirable results. It is very easy to define the forwarding incorrectly. Several times mistakes have introduced circular references that resulted in an infinite loop endlessly multiplying messages, much like that shown in Figure 1.3.

Another problem that manifests itself in both the Arpanet environment and in 'SU-Bboard' is the existence of non-identical bulletin boards with a high degree of overlap. Such bulletin boards are sometimes produced by a pair of distribution lists having a high degree of overlap in membership because they cover related topics. As Figure 1.4 illustrates, they can also result from an attempt to provide a network bulletin board by constructing a master distribution list covering several local bulletin boards without disallowing independent postings to the local bulletin boards.

Readers of such bulletin boards face an unpleasant decision: how many of the copies to

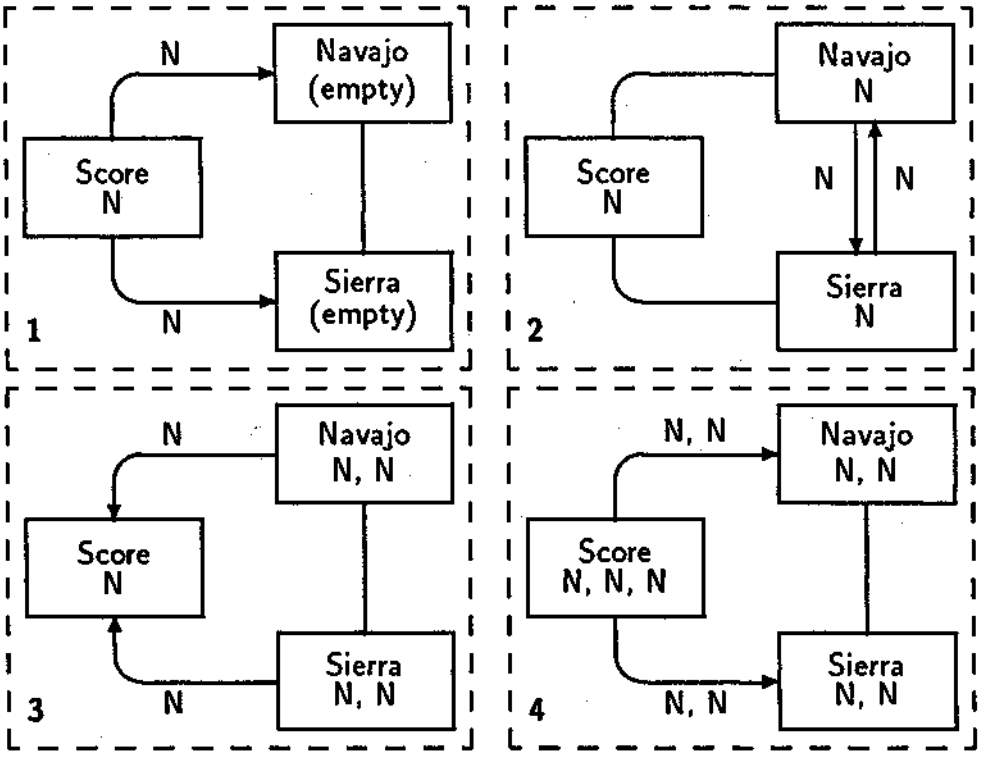

Figure 1.3: Danger of Circular References

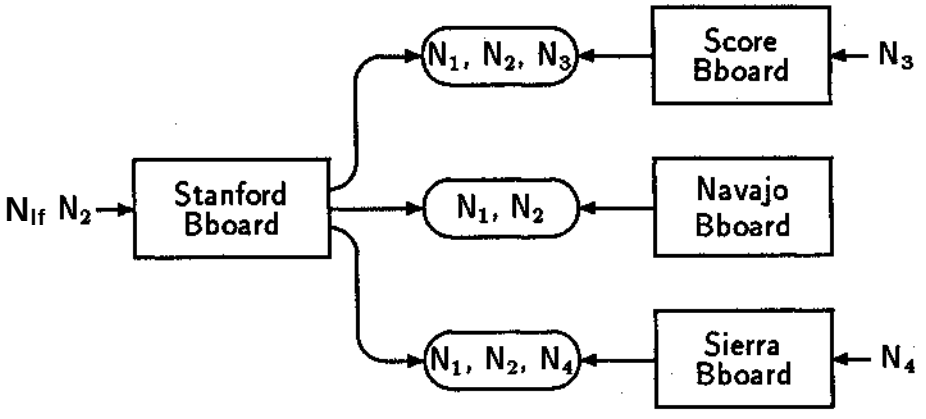

Figure 1.4: Nearly Identical Bulletin Boards

follow. If a reader follows all of them, he must expect to see some messages many times. In fact, the degree of overlap is increased by 'helpful' recipients who forward messages to other bulletin boards where they are relevant. Unfortunately, if users choose not to Follow all the partially overlapping bulletin boards, they will miss interesting messages.

1.5.3 Missing Functionality

Current bulletin board systems often are missing functionality. In the Arpanet environment, there is little support for locating bulletin boards or their maintainers. The file [SRI-NIC]<NETINFO>INTEREST-GROUPS.TXT contains a partial registry. Copies of this file can be obtained through the file transfer service. However, while many major distribution lists appear in this registry, the only entries are for lists whose maintainers have gone to the effort of having them entered.

Many existing bulletin board systems are also flawed because they either make it difficult to access bulletin boards from arbitrary points of the network or require an administrator to handle routine functions arranging access. In USENET, this is not the case. One simply need issue the command to read the news group. Most of the time, a copy will already be available on the local machine. In the Grapevine environment, this could be done by simple adding oneself to the appropriate list if the list has been set up to give casual users 'friends' privileges. Otherwise, for distribution lists, one must send a message to the list administrator asking to be added. If the bulletin board is kept as a file on a particular site, the problem is much worse. The user must either get an account on the machine or must find some way to call for a periodic transfer of the file to his or her own site.

Another instance of improper functionality is a general reliance on the assumption that a uper will always access the same copy of a bulletin board. In the Arpanet environment, mail is gathered only at a particular location for each user. The User must connect to that host to read his mail. In USENET, last-read-times are represented as a list of identifiers for messages that have been read. Each identifier is a local sequence number that has no meaning on any other USENET site. Grapevine, alone of the systems described earlier, recognizes that while users may have a primary location, they may login elsewhere. It provides access from many points and keeps a backup copy of each user's mailbox.

1.6 A Preview

This chapter defined what services a bulletin board system must provide and listed some of the flaws of existing implementations. Some of those problems arise because of the lack of integration of service or because the design is in some other way not a comprehensive response to the needs of the community. For example, lack of integration is manifested in ad hoc replication schemes that produce inconsistent copies and in prior read records that are not applicable at every site. This thesis will explore the difficulties and present a design for a bulletin board service that provides the desired functional and performance properties. This bulletin board system is named Taliesin, after a famous Celtic bard of King Arthur's court[1].

To establish the context in which the interesting problems can be discussed, the basic architecture of a bulletin board system will be presented first. Next the problem of achieving good scaling of costs will be addressed. Flat name spaces and failure to create

[1]Taliesin, Chief of the Bards of the West, the son of Saint Henwg, of Caerlleon upon Usk, ... the son of Bran, the son of Llyr Llediaith, King Paramount of all the Kings of Britain ... Taliesin became Chief Bard of the West, from having been appointed to preside over the chair of the Round Table, at Caerlleon upon Usk[Gue77, p. 496].

numerous, specialized bulletin boards will be identified as the prime causes of bad scaling in current systems. A keyword oriented name space will be presented as a potential solution to scaling problems.

The other hard problem is to replicate bulletin boards and other data structures in a manner that provides a consistent view to users and has enough concurrency to obtain good performance. A novel replication algorithm takes advantage of the semantics of bulletin board operations to achievemore concurrency than traditional database replication algorithms. To achieve good performance while retaining consistency, Taliesin couples a loosely synchronized algorithm with an algorithm enforcing more synchronization. A high degree of synchronization is only used when strict consistency is needed.

The latter half of the thesis will discuss implementation issues and experiences with a prototype. One of the strong points of the chosen replication algorithms is that they give administrators and implementors the freedom to pick from a wide variety of policies. Proper policy choices can produce good performance on a wide range of hardware and network configurations.

Most of the prototype implementation proved to be straight forward. Perhaps its most interesting aspect is that it provides a true keyword oriented name space. Most name spaces actually implemented are either flat or hierarchical in structure. Those working on IFIP naming have been concerned with the potential difficulties of implementing a keyword oriented name space[CCM85,Kil85]. The prototype successfully built its keyword name space on top of an underlying hierarchical one. However, some special support was needed in the queries on the underlying name space.

The thesis will conclude with an evaluation of the strengths and weaknesses of the design and prototype implementation of Taliesin. It will point out some areas that are particularly worthy of additional research.

Chapter 2

Architectural Overview

This chapter builds a framework for subsequent, more detailed descriptions of the interesting aspects of the design. The operations described are not novel. In fact, they closely resemble the facilities of computer mail systems. The terminology defined in the initial sections is adapted from that used for computer mail. The rest of the chapter is devoted to describing the architecture of the Taliesin bulletin board system at a high level of abstraction. Only the main points of the design are presented in this thesis. A complete description of the architecture is available in a related technical report [Edi86].

2.1 Underlying Network Architecture

Taliesin is a bulletin board system distributed over a network or an internet. An *internet* consists of a collection of interconnected networks. Each *network* is assumed to consist of a collection of *nodes* connected by *communication links.* The speed and resources of nodes may vary widely. In particular, some will provide long-term storage, but others may not.

At any instant, a state of *network partition* may exist in which some nodes are unable to contact others. It is assumed, however, that the nodes cooperate in providing a store-and-forwaxd service to transmit information between arbitrary pairs of nodes. A *message* between any pair of nodes will eventually be delivered. This topology covers local networks like ethernets, larger internets such as the Arpanet plus connected networks, and informal networks like USENET and dial-in bulletin boards/databases used by many home computer user groups. Figure 2.1 depicts a small network in a state of partition.

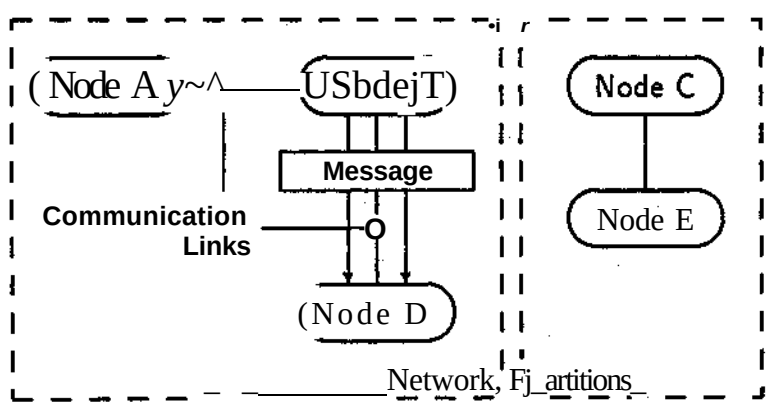

Figure 2.1: Network Model

Note that the term *message* refers to the logical unit of data transfer. A logical message may actually be fragmented into multiple packets or it may comprise a portion of the data exchanged using a virtual circuit. Such low level details will not be of concern in this document.

2.2 Logical Bulletin Board System Structure

From the perspective of a user, Taliesin closely resembles standard computer mail systems[Pic79,Sch81]. The basic functions of posting and reading are simply augmented to permit greater sharing. Its organization from a *logical* point of view will be described using terminology suggested by Deutsch[Deu81], with some modification to handle overloaded words. Note that an implementation may further divide or combine the logical services mentioned hereafter.

The basic function of a bulletin board system is to transmit a *notice* from its *originator* to a group of *recipients.* This is accomplished through the combined efforts of several (logical) *agents.* An agent is an active entity, such as a human *user,* a program, or a process. It may act in the role of a *server,* in which it responds to the requests of others, or it may act in the role of *client,* in which it makes requests of its own.

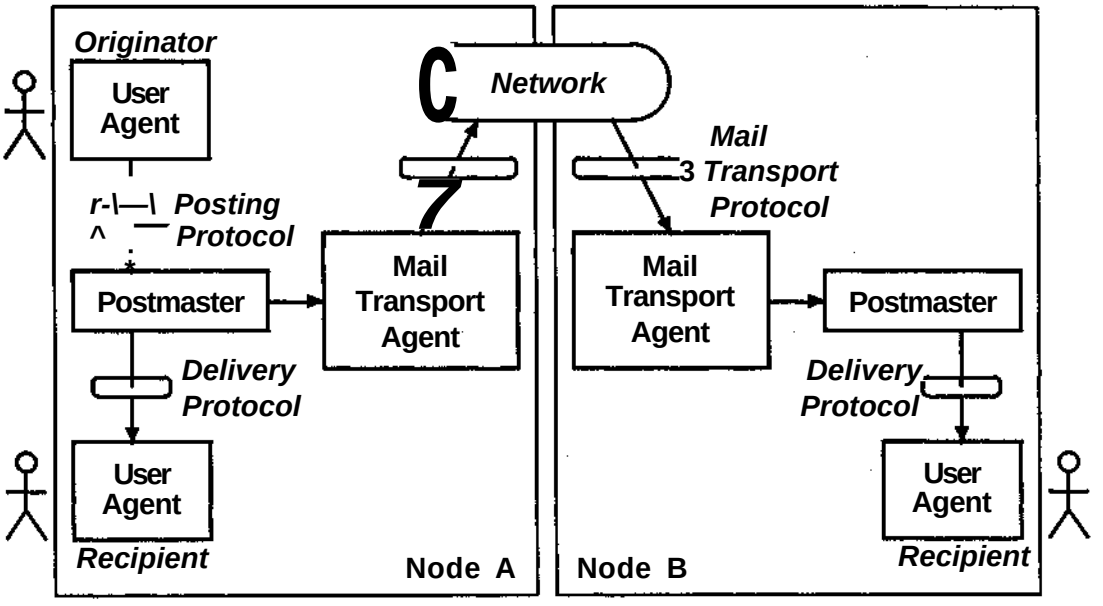

Figure 2.2: Notice Delivery Procedure

Logically, all direct interaction between the user and the bulletin board system takes place through a *user agent.* The user agent interprets input and displays responses from the rest of the bulletin board system. It provides text editing and may employ icons or menus in prompting for input. Although the logical user agent need not be implemented as a distinct entity, it is in many modern mail systems[BLNS82,Cro82,Uhl81]. User agents are beginning to be moved to nodes other than the node supplying mail service[BLNS82,Rey84]. Use of a distinct user agent allows users to pick an interface with which they feel comfortable and simplifies the implementation by cleanly subdividing the problem.

After parsing a command, the user agent transmits it to a *postmaster.* Postmasters cooperate to collectively implement bulletin boards and related objects. The interface between user agents and postmasters is defined strictly in terms of protocols. In particular,

after a notice is composed, it is sent to a postmaster using the *posting protocol.* Similarly, the user agent will use the *delivery protocol* to request the delivery of notices the user has asked to view. This logical arrangement of services in shown in Figure 2.2.

A variety of other agents are also involved in providing bulletin board service. Each postmaster implements some, but not necessarily all, bulletin boards. Some notices and requests must be forwarded to other postmasters. If an operation cannot be done locally, a postmaster hands off the request to a *mail transport agent.* Mail transport agents are responsible for forwarding requests to the proper nodes. Once a request arrives, it will be passed back up to a postmaster for servicing. Communication between mail transport agents and other agents is also specified in terms of a protocol, the *mail transport protocol.* For example, the Arpanet uses SMTP[Pos82] as its mail transport protocol.

Postmasters are responsible for protecting bulletin boards against unauthorized access. To do so, they must establish the identity of each client. This is done through an *authentication service.* In computer mail systems, this service is often provided by host operating systems.

A variety of other name-binding services are used. For example, a bulletin board name is bound to the locations of the postmasters implementing copies of it. These bindings are provided by a *name service.* It is increasingly common for name service to be implemented in a distinct agent[BLNS82,Moc83,Uhl81,LLNS83], although this is not required.

Figure 2.3 depicts a sample collection of the logical agents that cooperate to provide bulletin board service. How closely this structure corresponds to any particular implementation can vary widely. This basic architecture is followed by many regular mail systems, yet they vary from nearly monolithic implementations to separation into real agents corresponding to these logical agents.

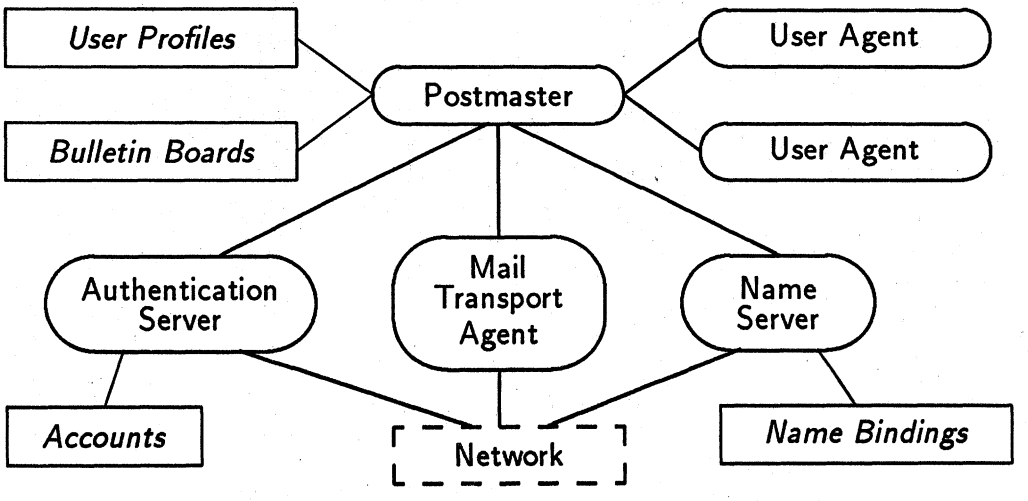

Figure 2.3: Logical Agent Structuring

2.3 Fundamental Object Types

As shown in Figure 2.4, the two primary data structures implemented by Taliesin are bulletin boards and user profiles. A *bulletin board* is a named storage location for notices. A bulletin board is like an ordinary mailbox, except that it is usually treated as a public rather than a private resource. From a user's point of view, a notice is just a communication from another user. It is manipulated, however, by user agents, postmasters, and mail

transport agents. Because these agents need to pass handling instructions to one another, notices consist of two portions: an *envelope* and a *body*.

The body is regarded as private information. It is composed and displayed by user agents, but no other agent may attempt to interpret or modify it. On the other hand, the envelope is understood and modified by all the agents. The user agent fills in the destinations and specifies any special handling characteristics. En route, the mail transport agents add routing and status information. Postmasters record the actual time of arrival.

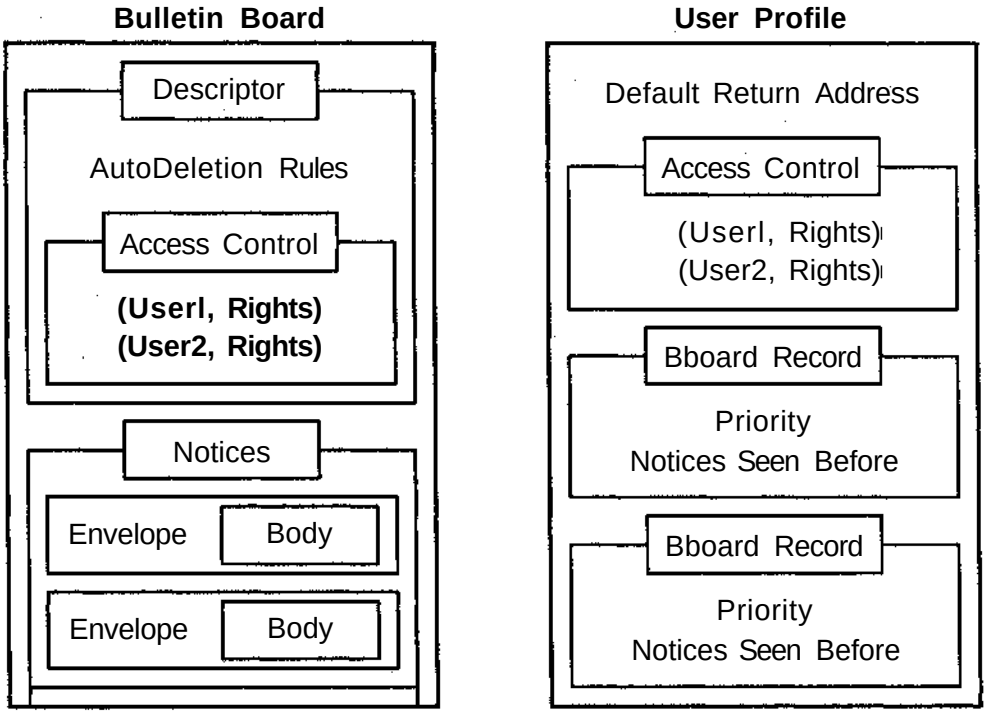

Figure 2.4: The Basic Objects of Taliesin

Taliesin supports *user profiles* as well as bulletin boards. User profiles provide important features that improve the palatability of the system. Users typically have interests in a variety of areas and so will follow multiple bulletin boards. Since users also tend to be lazy and will not want to type in the names of all interesting bulletin boards each time they read, Taliesin remembers the names in their user profiles.

Bulletin boards are typically monitored for new submissions. For example, a club may use one bulletin board to announce its meetings. Members must read it regularly to keep track of new announcements. The practice of sharing bulletin boards means that users do not have a personal copy of new postings which can be marked or deleted when read. This state information is stored in user profiles. A profile holds a list of the bulletin boards customarily read by a user together with an indication of what the user has seen in each.

Since one goal is to support the design of moderately intelligent user agents, additional state is recorded. A user profile also holds a reply-to address to be filled in automatically whenever the user sends a notice. Furthermore, it records a priority for each bulletin board that can be used by a smart user agent to organize its presentation.

2A Getting Started

When a user wants to use Taliesin, he starts up a user agent. The first action of the user agent is to establish a connection with some postmaster. Since Taliesin provides protection against unauthorized use, it establishes the user's identity as part of the process of setting up the connection.

StartSession(userCredentials) —> connectionId

The logical division of services calls for an authentication service to verify a user's identity. The authentication service can return either simple confirmation or an internal identifier representing the client. Because the client's identity is compared with stored privileges for bulletin boards and user profiles, Taliesin must use a form of user identification that is valid across invocations. If the authentication server does not return such an identifier, the bulletin board service must construct its own and be prepared to map all the user's external identities into it. Upon subsequent operations, the client need only present his connection-id to re-establish his identity.

The next step in getting started is to initialize the state of the user agent, both in the sense of tailoring its behavior to user specifications and in the sense of recalling what notices the user has already seen. Because this state information is saved in each user's user profile, the user agent calls the operation **ReadProfile.**

ReadProfile(connectionId, profileName) -» savedUserAgentState

Typically, each user will have exactly one user profile. If this were always the case, **ReadProflle** could be re-defined to automatically read the proper one. It is possible, however, that a user might act in different roles at different times and so would want to tailor what he reads to his current role. For example, a user might act as the maintainer of a set of software. When acting officially, he would read news related to the software package. At other times, the same user will read personal news. **ReadProflle** accepts the name of the desired user profile to handle such situations.

The operations **StartSession** and **ReadProflle** can be combined into a single operation since one user profile is always read at the beginning of a session. The reason for the distinction in Taliesin is purely historical.

2.5 Notice Operations

Once a user has established a connection, he may read and post notices. To post a notice, the user must interact with his user agent to compose the body of the notice. The user must also supply information used in filling out the envelope for the notice. The important fields are the reply-to address and the list of destination addresses. The destinations may be either bulletin boards or distribution lists. Optional special handling instructions can be given. For example, a notice may be assigned an expiration date. When all this information is ready, the user agent invokes the operation **PostNotices.** Postmasters take care of filling in the other fields of the envelope and storing a copy in each of the specified destinations.

PostNotices(connectionId, destinationList, noticesToSend)

To read a bulletin board, the user agent will usually invoke the operation **ReadNewNotices.** A list of all notices not previously read is returned. Alternatively, the operations **ReadNotices** or **ReadRecentNotices** may be invoked if the user wants to read selected notices whether seen before or not. The verbosity can be controlled so that notice identifiers, envelopes, or entire notices are returned.

ReadNewNotices(connectionId, bboardName, verbosity, priorReadInfo)
-> timeOfRead, TistOfNotices
ReadNotices(connectionId, bboardName, listOfNoticeIds)—> listOfNotices
ReadRecentNotices(connectionId, bboardName, timeInterval) —» listOfNotices

If a user wants to reply to a notice, the user agent picks out information from it to fill in the envelope of the reply. For example, the destination of the reply is normally the sender of the original notice. After offering the user an opportunity to override these defaults, the user agent posts the reply using the operation **PostNotices.**

Once the user is done reading, the user agent updates the record of what he has seen. **UpdateInterest** updates the user profile to reflect the reading of a particular bulletin board. It will compute the proper read record by combining what was stored in the profile with what the user saw during the latest read.

UpdateInterest(connectionId, profileName, bboardName, readInfo)

Once a user has read a notice, he may wish to delete it from the bulletin board. The operation **DeleteNotices** can be called for this purpose. If the user wants to reclaim storage, he can invoke the operation **DeleteOldNotices,** giving the minimum time a notice must have been stored before it is a candidate for deletion.

DeleteNotices(connectionId, bboardName, listOfNoticeIds)
DeleteOldNotices(connectionId, bboardName, minimumAgeToDelete)

2.6 Finishing Up

When the user has finished a session with the bulletin board system, his user agent performs some final tasks to clean up. It updates the user's profile to indicate what bulletin boards he has read, if it has not already done so. It invokes the operation **EndSession** to inform the postmaster that the client will not be engaged in any further requests for some time. The postmaster releases any resources it has allocated to handle the session.

EndSession(connectionId)

2.7 Customizing Bulletin Board Behaviors

One of the functional requirements for a bulletin board system is that it should minimize the workload of human moderators and maintainers. Indefinite storage of all notices is just not practical. Because manual deletion can be time consuming, Taliesin provides mechanisms to partially automate the process.

One mechanism was already mentioned. Taliesin allows users to specify an expiration date for notices and will automatically delete them as they expire. This feature is particularly useful for announcements of events such as seminars and concerts. They are known in advance to expire after the scheduled event has occurred.

However, most notices do not have a natural expiration date. Taliesin allows bulletin boards to be assigned handling characteristics that specify other conditions for automatically deleting notices. The operation **SetAutoDeletion** enables users to change those deletion conditions. The user must identify the bulletin board and the automatic deletion conditions.

SetAutoDeletion(connectionId, bboardName, deletionRules)

A variety of deletion rules are recognized. A bulletin board's rules can override the automatic flushing of expired notices. Independently, they can force the deletion of notices

that have been sitting in a copy for more than a specified interval of (real) time. Since notices may be delayed in arriving, this interval is measured from the time a notice arrives at the (copy of) the bulletin board. The length of the interval is under user control. Finally, a bulletin board may be set so that notices are to be deleted as soon as they are read. If no special deletion conditions are specified, notices will be deleted only in response to explicit requests.

2.8 Customizing the User Profile

A user may want to alter the record of his interests. For example, he might want to read a newly created bulletin board or to follow a bulletin board for only a limited time. Consider a bulletin board listing cars for sale. Few people will read this bulletin board with any regularity, but those individuals who are actively looking for a car will read it until their needs are satisfied.

Taliesin allows users to modify their user profiles at any time. They may change their minds as to which bulletin boards are interesting. To express interest, a user agent must invoke the operation **AddBulletinBoard.** The name of the bulletin board, its priority, and optionally, an initial prior read record must be supplied. The bulletin board system records the information in the specified user profile.

AddBulletinBoard(connectionId, profileName, bboardName, priority, readInfo)

If a user decides that a bulletin board is no longer worth following, he may call the operation **RemoveBulletinBoard** to delete its record from his user profile.

RemoveBulletinBoard(connectionId, profileName, bboardName)

Sometimes, a user may want to change the priority associated with a bulletin board or re-set the record of what he has seen before in some arbitrary fashion. The operation **EditBulletinBoard** enables users to do either.

EditBulletinBoard(connectionId, profileName, bboardName, priority, readInfo)

Finally, the user may change the defaults used by his user agent by calling for the operation **EditProfileDefaults.** In particular, he may change the default address filled into the reply field of the notices he sends.

EditProfileDefaults(connectionId, profileName, replyTo)

2.9 Protection

The bulletin board service is a facility shared among a human community^ This imposes a need for access control. To to avoid malicious deletion, protection is provided to restrict who may delete notices. Protection is useful for other reasons as well. Bulletin boards can be intended for a limited audience. A company might use a bulletin board to keep track of product development. Naturally, it would want to prevent individuals outside of the development group from learning the details of the product until it is formally announced.

Other groups may want to limit outside contributions to their bulletin board. They might be concerned that a few individuals will monopolize the bulletin board or they may intend to present only the opinions of experts. As an example of the latter, consider a bulletin board used to announce company policy. The company will want to make sure that what appears there is truly company policy.

Taliesin uses access lists, a standard protection mechanism[Jon79]. Four access rights are defined for bulletin boards: **read, post, delete,** and **other.** These correspond to the

operations of reading notices, posting notices, deleting notices, and other manipulations. User profile operations are normally granted only to their owners, so just one rights class really needs to be defined. However, rather than implement separate schemes for bulletin boards and user profiles, Taliesin's architecture uses the four bulletin board rights for both.

2.9.1 Access Control Lists

Protection is based upon access control lists, one list per bulletin board or user profile. To save storage and processing time, it is convenient to introduce user groups to compress the list. Each list entry, then, contains a user or user group identifier and the set of rights granted. A user is given the union of his rights as an individual and those granted to groups of which he is a member. Note that for this scheme to work, the authentication service must return not only the user's personal identifier, but a list of the user-groups he belongs to.

To change the protection on a bulletin board or a user profile, a family of three operations are offered. **GrantRights** can be called to give to a user new privileges in addition to those he already has. **RevokeRights** revokes the specified privileges while retaining any others. To completely redefine a user's rights, the operation **SetRights** may be invoked.

GrantRights(connectionId, bboardOrProfileName, userName, rights)
RevokeRights(connectionId, bboardOrProfileName, userName, rights)
SetRights(connectionId, bboardOrProfileName, userName, rights)

The pre-defined user group, **world,** is used to grant rights to all users. It need not be explicitly returned by the authentication service in the list of a user's user groups. Taliesin assumes that every user is a member of the group **world.** The protection scheme also allows specially privileged people to perform supervisory and maintenance tasks that ordinary users are not allowed to do. Taliesin identifies these users as being those who belong to the other pre-defined user group, **bboard-superuser.**

2.9.2 Authentication Service Operations

The authentication service maps from some sort of user credentials to a user identification that is meaningful at all nodes even across invocations of a user's agent. The credentials will be treated as consisting of a user name and a password. Although these operations are really provided in the authentication service rather than the bulletin board service proper, at least some such operations are visible to even the casual user. So, they are described here as if they were invoked via a postmaster.

Typically, a system administrator will create accounts for new users by calling the operation **DefineUser.** This requires him to choose a name for the user and assign an initial password. The user may change his password from time to time by calling upon the **ChangePassword** command. Since the postmaster already knows the user's old credentials, it need only be told the new password.

DefineUser(connectionId, userName, initial Password)
ChangePassword(connectionId, newPassword)

Protection is also defined in terms of user groups. A user group is treated exactly as if it were a user. It can be created by **DefineUser** as well. A user group identifier has the same format as a user identifier and is stored in the same manner in access control lists. An administrator may change the set of user groups to which a user belongs. **AddToUserGroup** will make a user a member of a user group while **RemoveFromUserGroup** will invalidate his membership.

AddToUserGroup(connectionId, userName, groupName)

RemoveFromUserGroup(connectionId, userName, groupName)

The authentication service is used by postmasters to verify the identities of users and to transform string names for users to the more compact user identifiers used in remembering what rights are given to various user. In support of the first need, the authentication service provides the operation Verify**User** to look up a user given a name and password. If the password is correct, it returns the user's identifier and a list of the user groups he belongs to. Otherwise, it returns only an error code.

VerifyUser(userName, password) —» validityStatus, userId, userGroupList
LookUpUserName(userName) —» userId, userGroupList
LookUpUserId(userId) —> userName

Postmasters use user identifiers rather than user names to record who may access Taliesin objects — bulletin boards and user profiles. However, to be friendly to users, user names are accepted as parameters to requests to modify access rights and are given in replies about access rights. The operation LookUpUserName is offered for mapping a user name to a user identifier. The operation **LookUpUserId** provides the inverse mapping.

2.10 Maintenance Operations

Not all operations are necessarily of interest to or within the power of the casual user. These remaining operations are of primary interest to the administrators of the bulletin board system. They enable the system to be fine tuned by monitoring its functions and by controlling the amount of replication.

2.10.1 Creation and Destruction

CreateBBoard may be invoked to create a bulletin board. The user agent must supply the name of the bulletin board, its automatic deletion conditions, initial access control list, and the location of the first copy. Of course, the user agent may supply default values for some of the parameters rather than prompting the user for them all. For example, the default might be to have the postmaster to which it is talking create the bulletin board locally.

CreateBBoard(connectionId, bboardName, deletionRules, accessControl, firstCopy)

Similarly, the operation **CreateUserProfile** can be called to create a new user profile. The user agent must supply a name for the profile, the user's default return address, the initial access control list, and the first copy's placement.

CreateUserProfile(connectionId, profileName, replyTo, accessControl, firstCopy)

At some point, a user or system maintainer may want to get rid of a bulletin board or user profile. The operations **DestroyBBoard** and **DestroyUserProfile** are defined to carry out those functions. They can only be called if a single copy of the object exists.

DestroyBBoard(connectionId, bboardName)
DestroyUserProfile(connectionId, profileName)

2.10.2 Replication

Taliesin provides bulletin board service distributed over a network. Many of its data structures are replicated. Users or administrators must be able to create and destroy copies. The operation **CreateCopy** may be called to create another copy of a bulletin

board, user profile, or name-binding. The user must indicate where he would like to place the new copy. In addition, each copy may have behavioral characteristics describing how it participates in the replication process. These are passed in as the final parameters. Taliesin is responsible for initializing the new copy and thereafter for keeping consistency between all copies.

CreateCopy(connectionid, objectType, objectName, copyNode, behaviors)
DestroyCopy(connectionId, objectType, objectName, copyNode)
ModifyCopy(connectionId, objectType, objectName, copyNode, behaviors)

The operation **Destroy Copy** may be called to get rid of a copy. This operation is subject to the restriction that the last copy may only be destroyed when the bulletin board, user profile, or name-binding is destroyed.

The administrators of the system or the owners of objects may want to tune the performance of the system by altering the behaviors of some of the copies. The operation **ModifyCopy** can be called to do so.

2.11 Query Operations

Taliesin supports a number of query operations. The most important one, **EnumerateBboardNames,** was called for as part of the functional requirements for any bulletin board system. It enables users to ask find out what bulletin boards exist pertaining to a particular topic. In particular, if a null search specification is given, **EnumerateBboardNames** will list every bulletin board.

EnumerateBboardNames(connectionId, searchSpecification) —» listOfBboardNames

Other queries allow users to discover the various attributes of bulletin boards and user profiles. This is an important feature if users are considering whether they wish to modify them. Users may ask about the current set of copies associated with a bulletin board, user profile, or name-binding by calling **QueryCopySet.** Similarly, they may ask about the access control list by invoking **QueryAccessList.** Inquiries as to the automatic deletion conditions associated with a bulletin board can be made using the operation **QueryAutoDeletion.**

QueryCopySet(connectionId, objectType, objectName) —> listOfCopies
QueryAccessList(connectionId, objectType, objectName) —* accessControlList
QueryAutoDeletion(connectionId, bboardName) —> deletionRules

Other queries are designed primarily for the user of system administrators. To support tuning of the system by changing copy placement and routing decisions, to watch for suspicious behavior that may indicate a bug, and for accounting reasons, the system keeps statistics on its usage. The system maintainers may get a report on the usage statistics by calling for the operation **ReportStatistics.**

ReportStatistics(connectionId, bboardName, outputFile)
DumpDataFile(connectionId, dataFileType, outputFile)

Each implementation also accept commands to print out its main data structures. The exact commands available will necessarily depend at least in part upon the implementation. The **DumpDataFile** command prints out the data structures whose existence and logical format is common across all implementations.

2.12 Interesting Architectural Issues

As was promised, the gross structure of Taliesin is patterned after that of traditional computer mail systems. The operations on notices are directly analogous to those of a mail system. Taliesin in many ways, however, is a more sophisticated service. The added sophistication can be seen best in its name space and the cooperation between agents to provide access to resources across a network.

Most computer mail systems name mailboxes after pre-existing names of users' accounts. Distribution lists are usually a concatenation of a node name and a string that hopefully describes the purpose of the list. Taliesin uses a separate name space for bulletin boards. The name-space has greater structure that reflects the topics of notices to be stored in each bulletin board. Chapter 3 describes Taliesin's name-space in detail and explains the motivation behind the design.

The other interesting aspect of Taliesin's architecture is its support for distributed service. Taliesin replicates bulletin boards and forwards requests across the network, if possible. Chapter 4 explains the need for replication and summarizes the major features of the chosen algorithms. Appendix A presents the replication algorithms in detail.

Chapter 3

A Name Space for Bulletin Boards

A distributed bulletin board system should scale well as new participants join. Ideally, the design should require a constant or near constant cost per user. The scaling properties depend primarily on two factors: message routing and user behavior. This chapter analyzes how the name space for bulletin boards affects costs. The basic idea for improving the scaling of costs is to select a name space that produces greater locality of reference by encouraging desirable user behaviors.

3*1 Query Language and Naming

Every user must translate his notion of an interesting notice into a *query*, a predicate that can be used by the bulletin board system to determine which notices to present to the user. Query processing is split between user agents and postmasters. The user gives his user agent a query. This version of the query may use nicknames or other shorthand forms of expression[Sol85]. Typically, it will not specify what was seen before. The user agent reads that information from the user's profile and transforms the original query into a series of queries that can be handled by postmasters. The user agent asks a postmaster to locate all the notices satisfying these queries, then uses the replies to finish processing the original query.

The unavailability of a single node cannot be allowed to suspend the processing of reads. Similarly, response time must not be degraded by requiring every postmaster to participate in the handling of each query. So, the postmaster initially receiving a query must decompose it into sub-queries to be executed at a subset of all postmasters. In this design, name-bindings record the locations of bulletin boards but do not otherwise characterize what notices might be found at a particular node. This implies that all bulletin boards must be searched unless the query is phrased in terms of bulletin board names. Hence, the name space should be designed so that names correspond to the most frequently submitted queries.

Two primary characteristics determine whether a notice is interesting: its topic and whether the user has seen it before. A user may also want to formulate a query based on other attributes and more complex relationships between notices. For example, a user might want to read all new replies to some notice except those posted by one author. Expressive power is split between postmasters and user agents. Adding expressiveness to postmasters may impose overhead on the execution of all queries, yet postmasters should provide enough expressiveness for efficient query processing. As might be expected, database research on optimizing query processing in a distributed system indicates that dividing the query according to location of the data is the key to efficiency[CH79,SN79,Won79]. Efficiency is highest when a query can be divided into subqueries that filter data at its storage point so that only the filtered results need be communicated for final processing.

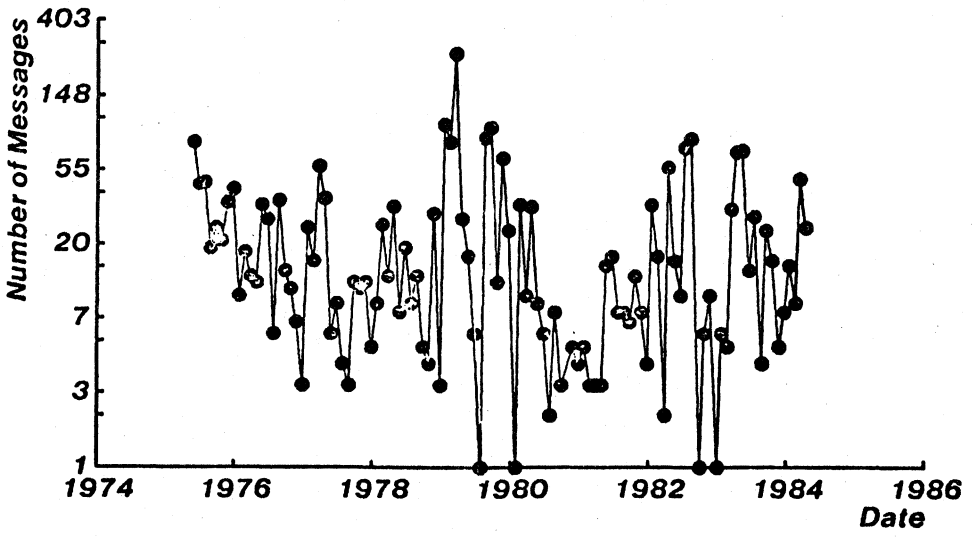

Figure 3.1: Submissions for MsgGroup (Unmoderated Arpanet List)

Since postmasters supervise the storage of bulletin boards, they are the logical candidates for doing the preliminary selection of notices of potential interest. Filtered data must be sent to a common location for further processing. As will be explained in Section 3.4.4, little or no additional processing of the intermediate results needs to be done. The initial data selected might as well be sent to the requesting user agent. In this design, postmaster queries identify a collection of bulletin boards and place selection constraints on the notices in each bulletin board in the collection. User agents may be written to perform whatever additional processing is desired.

3.2 Importance of Creating Bulletin Boards

There are a variety of reasons for creating many specialized bulletin boards. User friendliness is the most commonly recognized one. In a distributed bulletin board system, there are two other important reasons. The better known one is the need to make replication work quickly. The other reason, not heretofore recognized, is the fact that good scaling behavior can only be achieved if the number of bulletin boards grows with the number of users.

Current bulletin boards are characterized by bursts of activity, reflecting intense discussions about issues of temporary interest. Figures 3.1 and 3.2 show the notice posting patterns for a couple of Arpanet distribution lists[1][Msg84,Lis84]. Most users would like to avoid being presented with some of these debates. So, the bulletin board system should support the splintering off of subtopics.

Another reason to create many small bulletin boards is to reduce the cost of providing consistent, replicated copies. Replication requires communication between postmasters. The amount of communication and the speed of convergence depend on the number of copies. The fewer the number of copies, the better. Yet if a bulletin board is read at many nodes, fewer copies means worse response time and a greater chance that no copy

[1]The marked drop in traffic in early 1983 was caused by the disruption in Arpanet mail facilities when new mail format and transmission protocols were introduced.

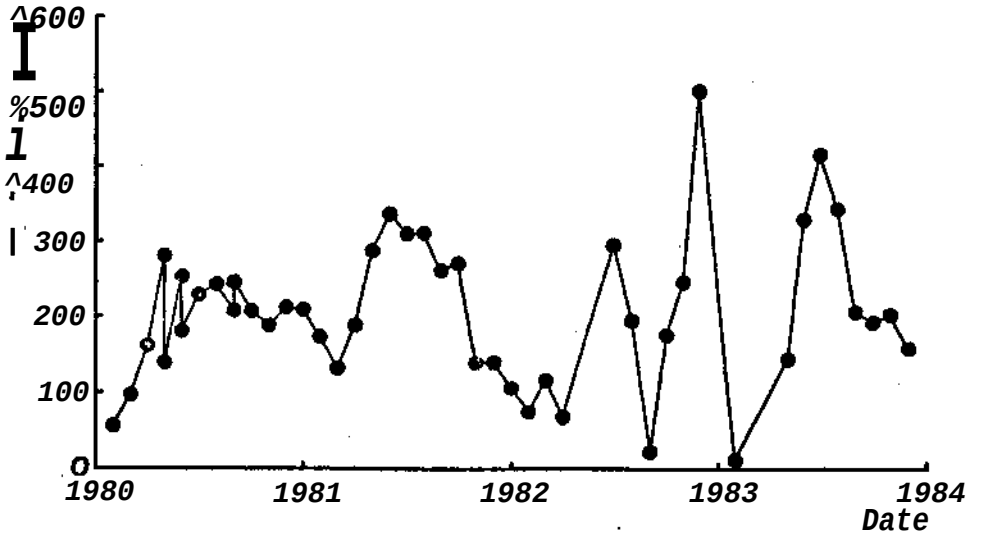

Figure 3.2: Submissions for SF-Lovers (Moderated Arpanet List)

will be available. Both of these outcomes are avoided if a bulletin board is used from a small number of nodes. If there are many small bulletin boards, it is more likely that each is referenced at only a few nodes.

The costs of a computer bulletin board system can be evaluated in terms of the amount of storage required to save notices, the network traffic generated by transmitting notices, and the amount of time users spend reading notices. The amount of network traffic is directly related to the average number of notices presented to each user agent, D_u. How much filtering is provided by user agents influences how many notices users see. D_u is useful as an approximation of the user burden.

Before estimating the costs, some notation will be defined. A bulletin board is the basic unit that may be read under this model of query processing. The number of bulletin boards in existence, $J5_n$, is assumed to be finite but may be a function of the number of users. Using any enumeration of the bulletin boards, let $S_u(i)$ denote the size of the i-th one (new notices per time period). U will denote the number of users who read and/or post notices. User behavior will be parameterized by two quantities, both of which may be functions of the number of users. P_u denotes the average number of notices posted per user in one time period while $R_u(i)$ is the number of readers of the i-th bulletin board. *1RZ* and 3£ denote the mean readership and bulletin board size. Both bulletin board sizes and network traffic will be computed in terms of notices appended or transmitted per time period. The notation $E(X)$ denotes the mean of the random variable X.

3.2.1 Network Traffic

The amount of network traffic depends on the distribution of the readership of each bulletin board. If an arbitrary number of users can be clustered on a single node, there may be no need to transmit notices over the network. However, technology imposes an upper bound on the number of users per node in the near-term.

Assumption 1 There are at most v users per node.

Given this assumption, a lower bound can be derived on the number of remote notice-fetches and hence on network traffic.

Lemma 1 Network traffic is at least $U(D_u/v - P_u)$.

Proof The i-th bulletin board is read by $R_u(i)$ users. At most v of these can be on any one node so it must be read on at least $R_u(i)/v$ nodes. One node per notice may receive the notice as a local posting. Each additional node must get a copy over the network. So, the minimum number of notice fetches for the i-th bulletin board is $S_u(i)(R_u(i)/u - 1)$. The total network traffic is at least:

$$\sum_{i=1}^{B_u} S_u(i)(R_u(i)/u - 1) = u^{-1} \sum_{i=1}^{B_u} S_u(i)(R_u(i) - \sum_{i=1}^{B_u} S_u(i)$$

The sum of $S_u(i)$ is the total size of all bulletin boards. It is equal to the total number of postings by all users, UP_U. The value of the other sum can be expressed in terms of D_u. The product UD_U is the total number of notice-fetches over all bulletin boards. Since postmasters return all new notices in a bulletin board each time it is read, the number of notice-fetches for a particular bulletin board is the product of its size and its readership.

$$D_u = U^{-1} \sum_{i=1}^{B_u} S_u(i)R_u(i)$$

$$\text{Network traffic} = UD_u/v - UP_uD$$

So, network traffic, scaled down to a per user basis, grows as D_u grows. D_u will be estimated in the next two subsections based on average and worst case behaviors.

3.2.2 Average Case

Ideal scaling requires not only that the total storage cost remain proportional to the number of users, but that the burden be fairly distributed. If the average size of bulletin boards grows, those nodes storing average or above average sized bulletin boards will see their storage costs grow as new users are added to the system. The average size will be computed to estimate the potential for fairness and as the first step in determining overall storage costs.

Lemma 2 $\overline{Su} = UP_u/B_u$.

Proof In each time period, the U users submit an average of P_u notices each, for a total of UP_U. These notices are distributed among the B_u bulletin boards. The average traffic per bulletin board is UP_U/B_U. •

As Lemma 2 indicates, there are only two ways to keep the storage costs in line. The first is to reduce P_u. The more users there are, the less each may say. At first, the reduction may be painless. For example, a user may choose to reply to queries in areas where he feels he has minimal competence if the user community is small. In a larger community (more apt to include an expert), the user may remain silent.

In the long run, however, this approach is fundamentally hostile to users. The utility of the bulletin board service is deliberately decreased as network connectivity improves. The only other way to control storage costs is to increase the number of bulletin boards. In fact, if ideal scaling is to be achieved without decreasing P_u, the number of bulletin boards must grow in proportion to the number of users.

To estimate £)$_u$, the behavior of users must be predicted. Specifically, it is necessary to know how many readers each bulletin board has. Two user models lead to similar results. The first directly computes average costs as a function of the actual readership distribution.

Lemma 3 $D_u = P_U\bar{R}\hat{} + \text{Cov}(j?_w, S_U)B_U/U$

Proof The proof of Lemma 1 gave a general formula for D_u. Putting that together with the definition of the covariance of two random variables and the value for $''S\hat{}$ given in Lemma 2 gives:

$$D_u = U^{-1}\sum_{i=1}^{B_u} S_u(i)R_u(i)$$

$$Cov(S_u, IQ = BZ^{1}\sum_{}^{B_u} £\ S_u(i)Ru(t) - \hat{}\overline{R_u}$$

$$D_u = U\sim\backslash UP_U/B_U)(\bar{R_U}'B_U) + U\sim^{1}B_uCov(R_u,S_u)$$

$$D_u = \bar{P}uR\hat{}+CoviRu.S\hat{}BjU\ \text{D}$$

The sum of the readerships of all bulletin boards is almost certainly greater than U. Only if there are a large number of users who post notices but never read any bulletin board would this be false. This implies that almost certainly, $\overline{R_u} \geq U/B_u$. The relationship between the size of a bulletin board and the number of its readers is a function partly of human nature and partly of the changing interest in the available bulletin boards. Intuitively, the size of a bulletin board is positively correlated with the number of its readers. This is confirmed in Table 3.3 [2] for a small number of samples of covariances for the readership of USENET news groups[Arb86]. Hence, if U/B_u -» oo as U —» oo and Pu ~h 0> then D_u —» oo as well. There may, however, be a local minimum should the covariance decrease.

Another approach to estimating D_u uses the fact that humans have finite abilities and a finite amount of time they can devote to composing notices. The number of notices a user can compose per time period is bounded in practice by $\hat{P}$.

Assumption 2 No user posts more than P notices per time period.

This model of user behavior needs to predict the readership of a bulletin board given the number of writers. The chief use of bulletin boards is as forums for discussing a particular topic or the affairs of some community. In this context, it is reasonable to assume that the people posting to a bulletin board also read it.

Assumption 3 Users only post notices to bulletin boards they read.

Current bulletin boards have far more readers than writers, as suggested by Figure 3.5. The analysis leading to Lemma 4 goes through without change as long as it is not the case that the more users post to a bulletin board, the less likely people are to read the bulletin board.

Lemma 4 $D_u \geq \{UPl)/\{\hat{P}B_u)$

Proof Since no user can contribute more than $\hat{P}$ notices, at least $S_u(i)/\hat{P}$ users post to a bulletin board of size $S_u(i)$. Assumption 3 requires each writer to read the bulletin board. D_u has a lower bound, then, of:

[2] Data from ba.news.ratings news group. Only news groups read by at least one user on the machine were used to compute the statistics. Data collection times may differ.

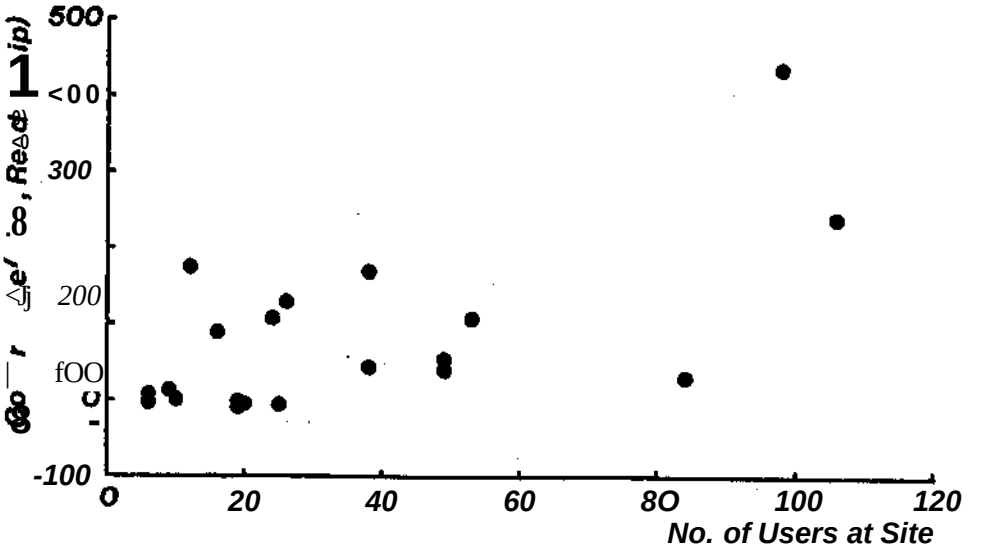

Figure 3.3: Correlation of Readership and Traffic

$$D_u = U^{-1} \sum_{i=1}^{B_u} R_u(i) S_u(i) \geq U^{-1} \sum_{i=1}^{B_u} S_u^2(i)/\hat{P}$$

A elementary result from statistics is that $E(X^2) \geq (E(X))^2$.

$$D_u \geq (U\hat{P})^{-1} B_u \overline{S_u^2} \geq (U\hat{P})^{-1} B_u (\overline{S_u})^2$$

Plugging in the mean size of bulletin boards from Lemma 2 yields:

$$D_U > (B_U/UP)(U\hat{P}_U/B_U)^2 = \frac{UP^2_U}{\hat{P}B_U} \square$$

Again, the number of notices presented to each user agent grows unless either the number of notices posted per user drops or the number of bulletin boards grows at legist as fast as the number of users. The link between D_u and network traffic stated in Lemma 1 implies that the communication costs also scale ideally only if B_u ex U. Note that controlling D_u by decreasing P_u may not keep the volume of network traffic under control.

Existing bulletin board systems do not meet the goal of creating bulletin boards. In particular, USENET administrators try to control costs by reducing P_u. One tact taken is to limit the number of topics that can be discussed on USENET by holding down the number of news groups. Figure 3.4 shows that B_u is not growing anywhere near as fast as the number of nodes. The data for 1984 was supplied by Horton[Hor85]. The other data was taken from USENET on SU-Navajo.

3.2.3 Worst Case

Even if U/B_u is constant, costs may not scale well. Bad scaling for the worst cases, *universal bulletin boards,* can prevent good scaling overall. Universal bulletin boards are

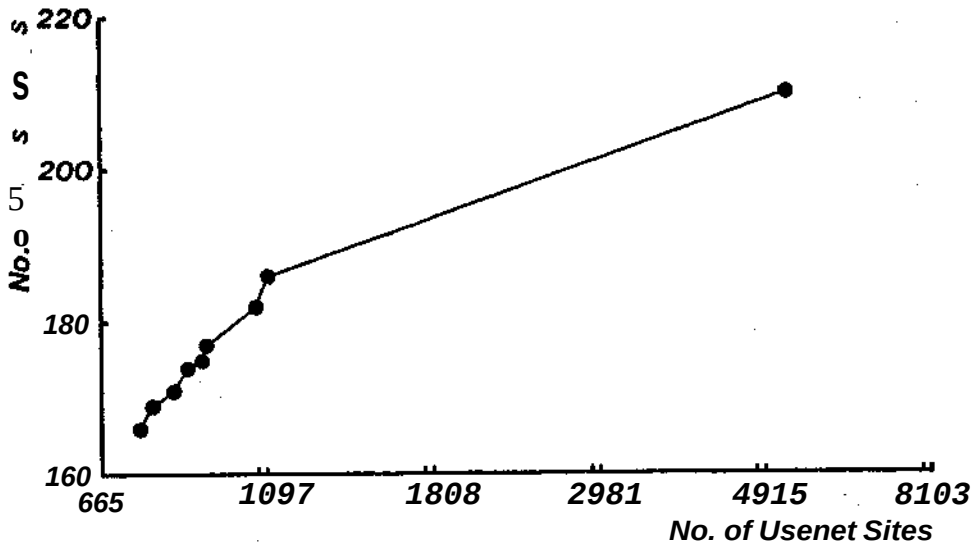

Figure 3.4: Creation of USENET News Groups

those followed by a constant fraction of the user community. It is certainly reasonable to believe that some subjects like politics are of interest to a fraction of the general population that is relatively stable over time. Experience with the Grapevine computer mail system indicates that popular bulletin boards do exist[SBN84]. For example, Tax^.pa was read by about 1/6 the user community.

Assumption 4 At least one bulletin board, POPULARJ3B, is read by a constant fraction^ /it, of the total number of users.

The bad behavior of universal bulletin boards also depends on the willingness of users to post to popular bulletin boards. Let N_u be the average number of notices posted to POPULARJBB by a reader. First of all, the traffic to POPULARJ3B will be calculated.

Lemma 5 At least fiN_uU notices are posted to POPULARJBB.

Proof By Assumption 4, $)J,U$ users follow P0PULARJ3B. Their average posting rate is N_u, so the total number of notices posted is $jiUN_u$ per time period. □

Knowing the size of POPULARJBB, a lower bound can be calculated on the value of D_u over all bulletin boards.

Lemma 6 The average over all users of D_u is at least fjL^2N_uU.

Proof Each of the $JJLU$ readers of POPULARJBB is presented with the fiN_uU new notices that are posted to it according to Lemma 5. So, POPULARJBB itself contributes at least $fi^2N_uU^2$ notice deliveries. This forces the average notice deliveries per user to be at least $/JL^2N_UU$. D

Lemmas 5 and 6 show that storage costs and traffic costs are determined by the interaction of N_u and U. If N_u oc $C7^{""1}$, the size of even a universal bulletin board remains constant and network traffic may scale ideally. However, if N_u remains constant, both costs would scale poorly. It is worth examining existing data to attempt to determine how N_u is influenced by the size of the user community and the number of readers.

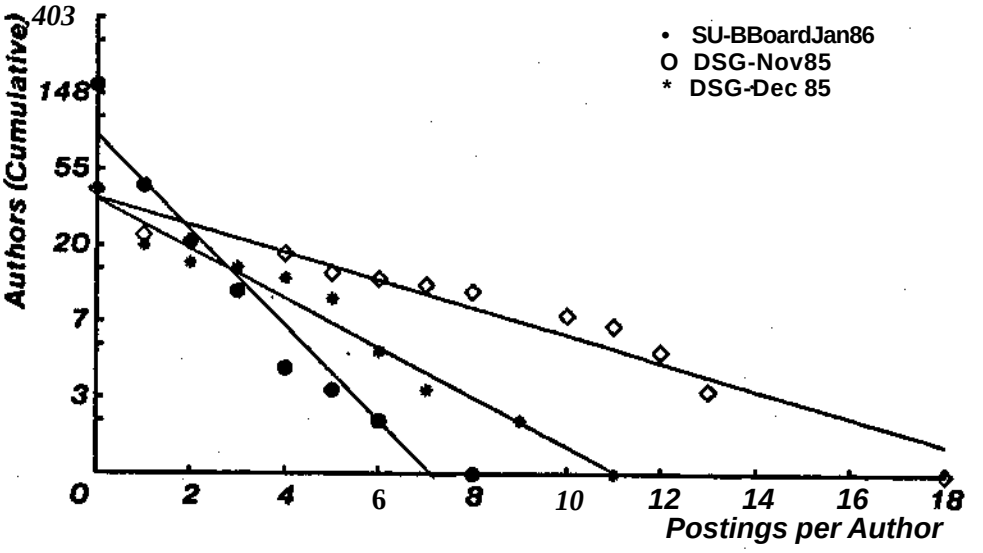

Figure 3.5: Posting Behavior

Historical data on N_u for a common environment, changing only the number of users, was not readily available. To estimate the impact of readership on posting behavior, the current behavior of USENET users was examined. The data provides some information on the effect of the size of the community on human posting behavior, although the effect may be confused with differences based on varying interests in topics.

USENET is an informal, cooperative effort with minimal overall administration and no official figures on readership. To estimate readership, the distribution of the number of postings per user was examined for a collection of bulletin boards and distribution lists. Let *Pi* be the random variable measuring how many notices were posted by the i-th reader. An empirical cumulative distribution function can be calculated as:

$$F(n) = U^{-1} \sum_{i=0}^{i=U} \mathcal{I}(P_i \leq n)$$

$$\mathcal{I}(P_i \leq n) = \begin{cases} 1 & \text{if } P_i \leq n \\ 0 & \text{otherwise} \end{cases}$$

Examining the empirical density and cumulative distribution functions revealed that the log of their tails appear to drop off smoothly. Figure 3.5 shows plots of the tail of the cumulative distribution with linear regression curves fitted on the log of the number of users posting at least a particular number of notices. The samples were the DSG distribution list for November, DSG for December, posting to SU-BBoard by everyone with accounts on Pescadero, Gregorio, or Navajo, and the first 100 notices sent to the MsgGroup distribution list[Msg75]. The smooth nature of the curve allows the number of users who don't post to be estimated from the posting behavior of those who do.

This process was then used to estimate the readership of a random sampling of USENET news groups to see how the size of the news group readership varied with the average number of notices posted per user. This readership estimate is low because it only estimates the numbers of readers whose contributions reached the computer Navajo. Brian Reid attempted to get a more direct estimate by sampling the readerships at a number of USENET

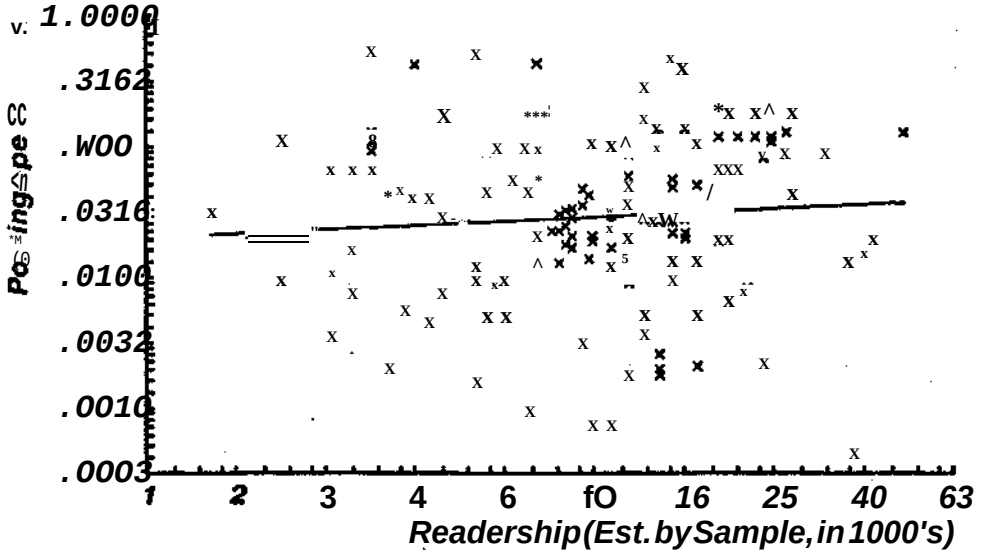

Figure 3^6: Impact of Readership on Postings (direct estimate)

sites[Arb86,Rei86]. The data from the most recent reports posted on ba.news.ratings (18 sites) were used to compute a more direct estimate of the readership. The maximum reported traffic to each iaews group was reported. The relationships between the number of postings per user and the readerships of news groups is plotted in Figures 3.6 and 3.7.

These graphs show the log of the readership against the log of the average number of postings. Figure 3.6, in particular, suggests that there is an inverse relationship. Linear regression on the two data sets yielded the following estimates for the relationship between N_u and U:

$$N_u = 0.113C7^{161}$$
$$N_u = 7.761T'^{500}$$

The first equation is derived from the direct estimates. The second is based on analysis of the empirical cumulative distributions. The slope of line is shallow enough to suggest that popular bulletin boards will grow in size. In fact, the data sample has some inherent biases that tend to exaggerate how quickly N_u falls off. News groups with excessive traffic are removed from USENET. This means that if there are topics that would be read by many readers who on the average post a moderate to high number of notices each, the topics will not be covered by a USENET news group. Local USENET administrators also try to cooperate to avoid creating new groups for which there is little interest. Thus, topics with small readerships and low posting rates are also not likely to be represented by news groups. This evidence strongly suggests, then, that N_u will not fall off rapidly enough to compensate for the effect of expanding the readership of a popular bulletin board.

3.3 Encouraging Bulletin Board Creation

The primary problem with increasing the number of bulletin boards is in making the appearance of new bulletin boards palatable to users. Current bulletin board systems

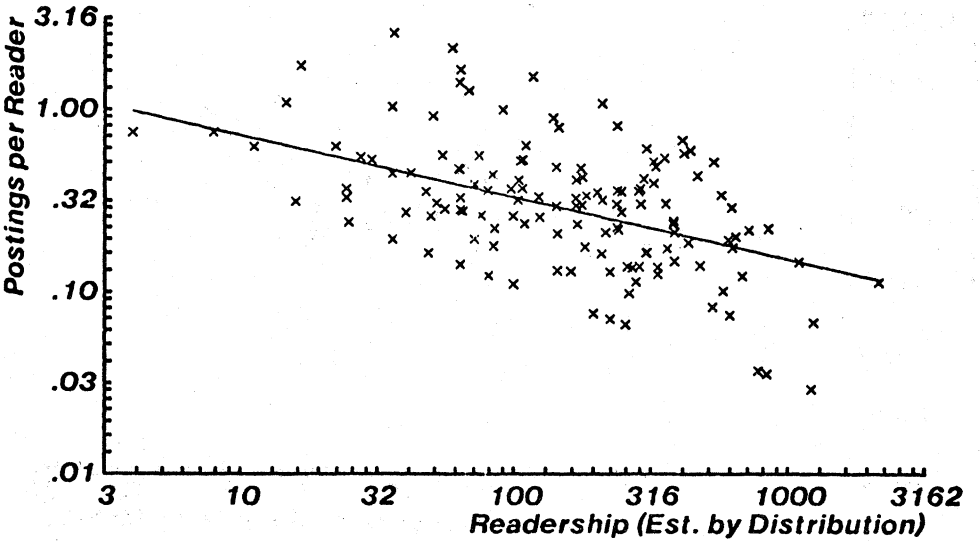

Figure 3.7: Impact of Readership on Postings (indirect estimate)

often lack a means of informing all potentially interested people of the creation of a new bulletin board. Even if everyone is told, most people are lazy enough that they don't want to memorize the names of all of the interesting sub-topic bulletin boards or to take the necessary steps to read them. Under these conditions, people tend to know about and interact with a few widely known bulletin boards. These potentially universal bulletin boards are apt to prevent good cost scaling.

It is particularly important for a bulletin board system to ease the transition when new, sub-topic bulletin boards are introduced. The name space affects what facilities can be provided. Current bulletin board systems usually have flat name spaces. Because users can only define their interests by enumerating bulletin boards and because there is no indication of the relationships between the topics of bulletin boards, it is difficult to locate the users who might be interested in the newly created subtopics. What needs to be done is to provide a name space and operations on that name space such that:

- Names are intuitive.
- Query operators exist for enumerating bulletin board names.
- Collections of bulletin boards can be specified in terms of the name space structure.
- The determination of which bulletin boards are part of a collection is made during query processing.
- Default reply addresses are automatically generated.

The first two properties enable users to find out what bulletin boards exist, including newly created bulletin boards. The remaining properties help automate the transition when sub-topic bulletin boards are introduced. Queries are phrased in terms of collections, rather than particular bulletin boards. For example, a user might read a collection consisting of the bulletin boards covering Pascal and all its sub-topics.

The bulletin board system determines which bulletin boards are members of a collection as part of query processing. Suppose a new bulletin board is created to hold notices discussing Pascal compilers. The user reading about Pascal and all its sub-topics will automatically see the notices sent to the new Pascal compilers bulletin board. Should

the traffic to a topic plus all its sub-topics grow too large, a user can change his query to specify a new, more restrictive collection. The hard part of this process is to choose subdivisions that do a good job of matching differences in user interests.

The other way in which users must react to the subdivision of a bulletin board is by changing where they post notices. The re-direction of notices to the new bulletin board is eased by two user agent facilities. These use the fact that a notice's envelope identifies which bulletin board the notice was stored in. The user agent should automatically fill in the destination address for replies based on the origin of the notice being replied to. For example, notices sent to the new bulletin board will be stamped as coming from the Pascal compilers bulletin boa;rd. Replies to them should be addressed to Pascal compilers.

Without understanding the content of messages, a bulletin board system can't guarantee that notices will be directed to a newly created bulletin board rather than an older one covering its super-topic. However, a user agent can encourage the use of the new bulletin board by informing the user of what bulletin boards are part of a collection or by displaying the names of notice origins. Consider the problem of getting users to post notices to the bulletin board covering Pascal compilers instead of that for Pascal. A user reading all the sub-topics of Pascal could either be told that the system read the bulletin boards Pascal and Pascal compilers or he could just be told that a particular message he read came from Pascal compilers. In either case, the user is told that there is now a bulletin board covering Pascal compilers.

Another useful, but less essential, feature of a bulletin board system is a facility for automatically subdividing bulletin boards. Creation could occur in response to excessive traffic or the start of a significant discussion. Sorting notices by conversation has been done by at least one computer mail system[CP85]. The possibility of applying similar techniques to bulletin boards is not explored in this thesis.

What would happen if a structured name space were adopted? The hope is that with such a name space, users would be willing to subdivide large bulletin boards. Costs will be estimated for a graph structured name space under several different models of user behavior. With good user behavior, costs will scale ideally. More interestingly, logarithmic cost growth is possible even when universal bulletin boards exist. Because logarithmic growth can be tolerated over a wide range of network sizes, it is an acceptable scaling rate. Logarithmic growth is a vast improvement over linear growth.

3.3.1 Potential for Ideal Scaling

Suppose the goal were met in the sense that bulletin boards are split or merged so that the traffic to every bulletin board is between $\hat{s}$ and $\hat{S}$. Then the ratio of users to bulletin boards remains bounded if posting behavior doesn't change.

Assumption 5 For any bulletin board i, $\hat{s} \leq S_u(i) \leq \hat{S}$.

Lemma 7 If Assumption 5 holds, then $UP_U/\hat{S} < B_U \leq UP_u/\hat{s}$.

Proof The total number of notices posted per time period is UP_U. Each bulletin board receives at least $\hat{s}$ and at most $\hat{S}$ of these according to Assumption 5. So, $UP_U/\hat{S} \leq B_U \leq UP_u/\hat{s}$. D

Lemma 7 ensures that the ratio of U/B_u does not grow as U grows, other than through changes to P_u. Individual bulletin boards will have ideal storage cost scaling without pressuring users to post less according to Lemma 2. The storage cost per user depends on the number of bulletin boards read. In the short term, this number can grow. However, there is a long term bound determined by human abilities.

Human capability limits the total length of notices read by a single person in a single

time period. Furthermore, notice size cannot be reduced indefinitely without losing meaning. For example, not every notice can be composed of a single word and still contain information worth reading. Effectively, there is an upper bound on the number of notices any single person can read.

Assumption 6 The reading speed of human beings is bounded.

Assumption 7 Notice sizes have a lower bound.

Observation 8 Each user reads at most V notices.

This upper bound translates into an upper bound on the number of bulletin boards read, assuming that users don't read bulletin boards that never hold interesting notices. Senseless reads irritate users by increasing the wait for query processing.

Assumption 8 No user asks his user agent to read a bulletin board unless the user is willing to read one notice from it (per time period).

Observation 9 Each user agent reads at most V bulletin boards.

Observation 9 can be applied to show that the communication and storage cost per user is bounded, despite the existence of multiple copies of bulletin boards.

Lemma 10 $D_u \leq V\hat{S}$ notices per time period.

Proof Each user reads at most V bulletin boards according to Observation 9. Each of these can have no more than $\hat{S}$ notices according to Assumption 5. So, the total number of notices in all these bulletin boards is at most $V\hat{S}$. *O*

The notice storage requirement is clearly limited to saving a copy of each notice read. Likewise, network traffic is limited to fetching D_u notices per user. So, Lemma 10 shows that all the costs scale ideally.

3.3.2 Improved, Non-ideal Scaling

The assumption that all bulletin boards can be subdivided to reduce their posting volume below some fixed bound may not hold true. For example, consider a name space organized around topics. If a notice doesn't belong to one of the defined sub-topic bulletin boards, it would naturally be posted in a more general one. This means the number of postings to the general bulletin board may not fall off after the major sub-topic bulletin boards have been created. There is still hope, however, of getting logarithmic cost growth rather than the linear growth suggested in Section **3.2.3.**

Simplifying Assumptions

Any analysis is difficult when the name space is a DAG. It is much easier to investigate trees, especially complete balanced trees. Assume that the users and/or administrators have organized the name space as a complete, balanced tree with branching /3.

Assumption 9 The name space for bulletin boards is a complete, balanced tree with branching */3* for some $fi > 1$.

Hierarchical name spaces arise naturally in several contexts. Consider the situation in which each node supports a single bulletin board intended to contain news of strictly local interest. These are leaves of the tree. Some news will be of interest to larger user communities. In fact, the communities can be organized into a hierarchy, with communities composed of sub-communities. Each community in the hierarchy has its own bulletin

board. Figure 3.8 shows a name space constructed in this manner.

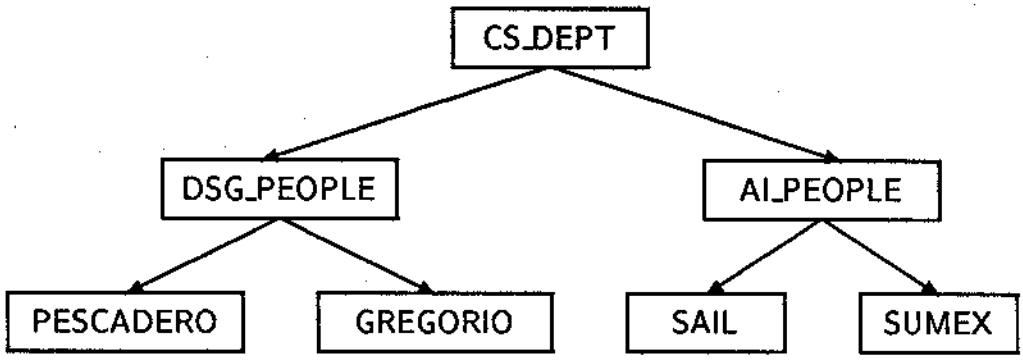

Figure 3.8: Sample Tree Name Space

Consider a different situation. Let bulletin boards reflect topics. Again, there is a natural ranking based on the notion of topic versus sub-topic. This hierarchy can be approximated by a tr^e. In general, a name space tree will not be balanced nor will every interior node have the same number of children. How deviations from this ideal affect the situation will be explained in Section 3.3.2.

General Analysis

In this scenario, it will be assumed that creation takes the form of sub-dividing leaf bulletin boards. Such sub-division will occur when necessary to ensure that the readership of leaf bulletin boards does not grow beyond an upper bound. It will also be assumed that the naming tree is re-organized as needed so that each leaf bulletin board has at least a minimal number of readers. Since a larger readership is correlated with a larger bulletin board, Assumption 10| is a rough re-stating of Assumption 5 except that it applies only to leaves, not all nodes.

Assumption 10 The number of readers of each leaf bulletin board is at least Ai and is at most A2.

The size of the tree depends on user behavior. Each user follows specialized (leaf) bulletin boards corresponding to his interests plus possibly their more general ancestors. If a user wants to read a general bulletin board but not its sub-topics, D_u will only be overestimated if the user is modeled as reading some leaf under that bulletin board plus its ancestors. The analysis to come will assume that users read some leaf bulletin boards plus their ancestors. Posting behavior will be parameterized to account for the fact that users may not actually post to all these bulletin boards.

Assumption 11 A user reads no bulletin board unless it is either a leaf bulletin board he follows or an ancestor of a leaf bulletin board he follows.

Let F_u denote the average number of leaf bulletin boards followed by a user. Assumption 3 is assumed to still hold: interest in a bulletin board implies the desire to read and post notices. Let H_u denote the depth of the tree. A tree consisting of just a root node is defined to have depth 1. A node is defined to be at level k if it is k edges removed from the root. The root of the tree is at level 0. The assumptions made so far allow the computation of the depth of the tree and the number of bulletin boards at each level in the tree.

Lemma 11 There are P^k bulletin boards at level k.

Proof There is 1 = $f3°$ root node. If there are 0^k bulletin boards at level fc, then since each of these has /? children, there must be $/?^{*+1}$ at level k + **1.** D

Lemma 12 $1 + log_p(UF_u/X_2) \_ < H_U \leq 1 + \mathbf{log\wedge l/lfy\ A_x)}$

Proof Leaf nodes are by definition at level H_u — 1. Lemma 11 says then that there must be $f3^{Hu\sim x}$ of them. The total number of user-interests in leaf bulletin boards is F_UU. No leaf bulletin board is read by more than A_2 users or less than Ai. So, there must be between $F_uU/\2$ and $F_UUjX\$ leaf bulletin boards:

$$F_UU/X_2 \_ </3^H \wedge <\_ F_UU/X_1$$
$$l+\backslash o_{g(3}(UF_u l\backslash_2) \quad \leq H_U < \_ \quad l + \backslash og_p(UF_u/X_1)n$$

Costs may scale ideally, as the number of bulletin boards is growing at least in proportion to the number of users.

Corollary 13 $B_u > (/3- l)^{-1}(f3UF_u/X_2 - 1)$

Proof The total number of bulletin boards is the sum over all levels of the number of bulletin boards at each level. Lemmas 11 and 12 yield a total number of bulletin boards of:

$$\mathbf{p} \quad \mathbf{v}\wedge \ nk \quad \frac{P \quad ^l}{P-!}$$
$$k=0$$
$$B_u \_> \ (P\text{-}l)^{-1}(\beta^{1+\log_\beta(UF_u/\lambda_2)} - 1)$$
$$B_U \_> \ ip\text{-}iy^1(\beta UF_u/\lambda_2 - 1) \ \square$$

The next step in estimating the costs is to determine the sizes of the bulletin boards. Parameterizing the relationship between posting behavior and level in the tree yields some general formulas. These will be applied to special cases, reflecting different predictions of user behavior.

Assumption 12 The fraction of notices posted by a user to bulletin boards at level k is $p_uK\{k)j3^k$. This fraction is further subdivided among however many level k bulletin boards the user follows.

Knowing where users post their notices makes it possible to determine the sizes of bulletin boards at each level in the naming tree. Let $S_u(k)$ denote the size of a level k bulletin board.

Lemma 14 Suppose all users post P_u notices and read exactly F_u leaf bulletin boards plus all their ancestors. Then the size of a bulletin board (notices per time period) is:

$$\frac{\lambda_1}{F_u^2 \sum_{n=0}^{H_u-1} K(n} \cdot \cdot \tilde{} \cdots < \frac{\lambda_2 P_u K(k) \beta^{H_u-1}}{\sum_{n=0}^{H_u-1} K(n) \beta^n}$$

Proof A bulletin board at level k is itself at the root of a subtree of height H_u — k. Lemma 11 implies that there are $/?^{Hu''''1\sim/:}$ leaf bulletin boards under each level k bulletin board. By hypothesis, no more than A_2 users follow each of these

leaf bulletin boards. So, at most $\backslash 20^{H_u \sim k \sim 1}$ users follow the level k bulletin board. On the average, users post P_u notices per time period. The more bulletin boards a user follows, the more thinly he must spread his postings. The greatest number of postings per user to a single level k bulletin board is obtained when the user is only dividing them among a single leaf bulletin board plus its ancestors. Putting these facts together yields:

$$\begin{aligned} S_u(k) &\leq \beta^{H_u-k-1}\lambda_2\rho_u K(k)\beta^k P_u \\ S_u(k) &\leq \beta^{H_u-1}\lambda_2\rho_u K(k)P_u \end{aligned}$$

On the other hand, at least Ai users follow each of the leaf bulletin boards. The sum of the readerships of the leaves under a level k bulletin board is at least $\wedge pHu\text{-}k\text{-}i$.$\mathrm{H_{OW}e_{ver9}}$ this may count the same user several times. Since all users follow F_u leaf bulletin boards, there must be at least $\backslash ifl^{Hu \sim k \sim \sim 1}/F_u$ distinct users reading each level k bulletin board. Each user can spread his postings most thinly by posting equally to F_u distinct ancestral bulletin boards. The least number of postings possible is $p_u K\{k)f5^k P_u jF_u$

$$\begin{aligned} S_u(k) &\geq fr^{H_u-k-1}\lambda_1 F_u^{-1}\rho_u K(k)\beta^k P_u F_u^{-1} \\ S_u(k) &\geq \$<^{H_u-1}\lambda_1\rho_u K(k)P_u F_u^{-2} \end{aligned}$$

The factor p_M provides a scaling factor to account for the fact the fraction sent to each levej will fall off as the number of levels increases. This adjustment can be computed by noting that its sum over all bulletin boards followed is 1.

$$\begin{aligned} \mathbf{1} &= \sum_{n=0} \rho_u K(n)\beta^n \\ pu &= \sum_{n=0} \end{aligned}$$

Plugging these facts together yields the desired result. D

Bounds on the number of notices presented to user agents are obtained next.

Lemma 15 Suppose all users post P_u notices and read exactly F_u leaf bulletin boards plus all their ancestors. Then:

$$\begin{aligned} Du \wedge XiPuF\text{-}_u^2 \wedge &\frac{\beta^{H_u-1}\sum_{k=0}^{H_u-1} K(k)}{\sum_{k=0}^{H_u-1} K(k)\beta^k} \\ U_u 2^* F_u P_u *2 — &\frac{\beta^{H_u-1}\sum_{k=0}^{H_u-1} K(k)}{\sum_{k=0}^{H_u-1} K(k)\beta^k} \end{aligned}$$

Proof Each user reads F_u leaf bulletin boards. In addition, the ancestors of these bulletin boards may be read. The number of ancestors followed at each level is at most the number of leaf bulletin boards followed. Lemma 14 provides bounds on the sizes of each bulletin board.

$$D_u \leq \sum_{k=0}^{H_u-1} F_u S_u(k)$$

$$D_u \leq \sum_{k=0}^{H_u-1} F_u p^{H} \text{«-}^{x}\backslash_2 K\{k) P_u \quad (^{H}\wedge 2K(n) fi^{n}) \quad)^{-1}$$

$$D_u \leq F_u P_u \lambda_2 \beta^{H_u-1} \frac{\sum_{k=0}^{H_u-1} K(k)}{\sum_{k=0}^{H_u-1} K(k)\beta^k}$$

Each user is assumed to read at least one leaf bulletin board (JF$_W$ ≥ 1) plus all its ancestors. So, the number of ancestors is at least 1 per user at each level of the tree. The total number of notices read is at least the sum:

$$D_u \geq \sum_{k=0}^{H_u-1} S_u(k)$$

$$D_u \geq \sum_{k=0}^{H_u-1} \beta^{H_u-1} \lambda_1 K(k) P_u F_u^{-2} \left(\sum_{n=0}^{H_u-1} K(n)\beta^n \right)^{-1}$$

$$D_u \geq \beta^{H_u-1} \lambda_1 P_u F_u^{-2} \frac{\sum_{k=0}^{H_u-1} K(k)}{\text{Efe}^{1} K(kW^{k}} \square$$

Good Posting Behavior

Users must post fewer notices to bulletin boards near the root to achieve good scaling. Users might behave in this way naturally. If the name space reflects user communities, the decrease might occur because of a tendency to interact most often with those in the same local community. In a name space reflecting topics, users may talk most often about their specialties. Consider the special case in which the reduction in posting to bulletin boards near the root matches the branching factor of the tree. This produces a tolerable, logarithmic growth in costs.

Assumption 13 In Assumption 12, Vfe $K(k) = c$.

Application of this assumption to the general result of Lemma 14 shows an upper bound on the size of an individual bulletin board that is proportional only to the average number of notices posted per user.

Corollary 16 Under Assumption 13, $S_u(k) \leq 2A_2 \#''\sim^{1}(£ - \mathbf{l})\mathbf{P_{tt}}$.

Proof Plugging in the value of $K\{k)$ given in Assumption 13 into the relation given in Lemma 14 yields:

$$S_u(k) \leq \beta^{H_u-1} \lambda_2 c P_u \left(\sum_{k=0}^{H_u-1} c\beta^k \right)^{-1} = \lambda_2(\beta - 1) P_u \frac{\beta^{H_u-1}}{\beta^{H_u} - 1}$$

By definition, the depth of any tree is at least 1. Furthermore, /?> 1.

$$\frac{\beta^{H_u}}{\beta^{H_u}-1} = 1 + (P^{H''} - \mathbf{I})^{''1} \leq 1 + (0 - \mathbf{I})^{''1} \leq 2$$

$$S_u(k) \leq \cdot 2X_2 p^{-1}(p-1)P_u \square$$

An upper bound on the number of notices presented to each user agent can likewise be determined that is a function of P_w, F_w, and the logarithm of U.

Corollary 17 Under Assumption 13, there are constants a, 7, and *6* such that $D_u \leq aFuPulogpifUFu) + SF_UP_U$

Proof Lemma 15 in this special case takes the form:

$$D_u \_< \sum_{k=0}^{-1} \left(\frac{\beta^{H_u}}{\beta^{H_u}-1} \right) P^{H} \text{»} — I$$

$$D_u \_< 2\backslash_2 /3'^{-1}(P\text{-}l)F_u P_u H_u$$

A upper bound on the value of H_u is given in Lemma 12 .

$$D_u \prec 2\backslash_2 p^{-1}(\beta - 1)F_u P_u(1 + \log_\beta(UF_u/\lambda_1))$$

The desired result holds if $a = 2A_2 £\sim^1(/3 - 1)$, $7 = Aj''\backslash$ and $6 = 2A_2 \wedge -$. D

When usage of a btdletin board system is high, $F_u \ll V$. Similarly, the theoretical bound on P_u of Assumption 2 produces the effect of $P_u \ll P$. In this domain of high usage, the size of individual bulletin boards remains fixed. The number of notices presented to users is only growing logarithmically in U. This leads to logarithmic growth of network traffic costs. Likewise, the total storage per user grows logarithmically.

Explosive Potential

Users may not post fewer notices to bulletin boards near the root of the tree. Some users like the publicity of speaking to the large audiences reachable through the bulletin boards near the root. In addition, divisions according to topic may do a poor job of classifying notices. General bulletin boards could receive many unclassifiable notices/The special case in which notice postings do not fall off near the root will be examined here. As is hardly surprising, the root bulletin board acts much like a universal bulletin board so the costs scale poorly.

Assumption 14 In Assumption 12, $K(k)\bullet == fi\sim^k$.

This choice for *K(jc)* causes the root bulletin board to grows roughly in proportion to $U/\log 17$, assuming P_u and F_u are constant.

Lemma 18 Under Assumption 14:

$$S_u(k) \geq \frac{\lambda_1 \beta^{-k} P_u U}{X_2 F_u (l + \log_{l3}(F_u U / X_1))}$$

Proof Plugging Assumption 14 into Lemma 14 yields:

$$S_u(k) \geq \beta^{H_u-1}\lambda_1\beta^{-k}P_uF_u^{-2}\left(\sum_{n=0}^{H_u-1}\beta^{-n}\beta^n\right)^{-1}$$
$$S_u(k) \geq \beta^{H_u-1}\lambda_1\beta^{-k}P_uF_u^{-2}H_u^{-1}$$

Bounds for the value of H_u were given in Lemma 12.

$$S_u(k) \geq \frac{\lambda_1\lambda_2^{-1}\beta^{-k}UP_u}{F_u(1+\log_\beta(UF_u/\lambda_1))} \square$$

The communications costs as estimated by D_u behave similarly.

Lemma 19 Under Assumption 14:

$$D_u \geq \frac{\lambda_1 P_u(\beta UFJX, \quad -1)}{\beta(1-\beta^{-1})F_u^2(1+\log_\beta(UF_u/\lambda_1))}$$

Proof Plugging Assumption 14 into Lemma 15 yields the following bound on D_u:

$$P_uF_u^{"2}\ \frac{\sum_{k=0}^{H-1}\beta^{-k}}{\overline{0}^{-1}\beta^{-k}\beta^k}$$
$$F_u^{-2}\frac{\beta^{-H_u}-1}{\beta^{-1}-1}H_u^{-1}$$
$$D_U \geq x_x p_u F_{\bar{u}}^{-2}(\beta^{H_u}-1)(1-\beta^{-1})^{-1}H_u^{-1}$$

Substituting in bounds on H_u yields the desired result:

$$D_u \geq \frac{\lambda_1 P_u(\beta UF_u/\lambda_2-1)}{\beta(1-\beta^{-1})F_u^2(1+\log_\beta(UF_u/\lambda_1))} \square$$

Having costs grow roughly as $U/\log U$ isn't quite as bad as the cost of having a universal bulletin board, but is not tolerable over a very wide range of U. Figure 3.9 shows the differences in the estimates of D_u from each of the scenarios. V was estimated from the number of notices that could be read in two hours using the reading rate estimate of Section 1.1. Bulletin board sizes are assumed to range from 1 to 20 notices per day. A popular bulletin board is considered to interest 10% of the user community. The mean posting rate to it of 0.04 notices per day is taken from the plots of typical posting rates in Figure 3.6.

For the tree structured bulletin boards, the readership of leaf bulletin boards is assumed to vary from 10 to 500. The branching factor of the tree is taken to be 10. The daily postings per user is taken to be 0.35 because that is roughly what a typical follower of SU-BBoard and the DSG mailing list would post. USENET readership statistics appearing in ba.news.ratings were used to estimate the number of bulletin boards each user would follow: F_u « 15.0[Arb86]. To show the relative growth rates more clearly, the calculated estimates of D_u were re-scaled so that JDioo = 10-

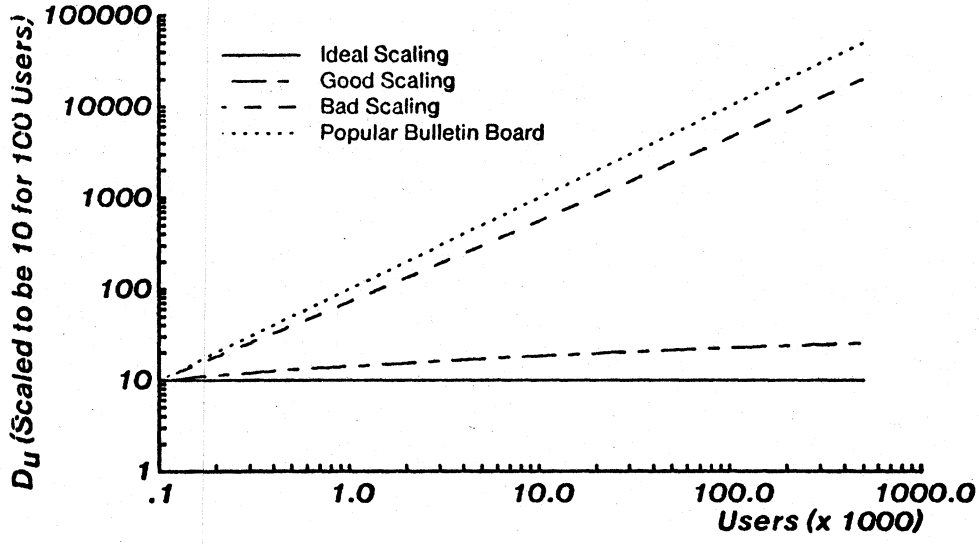

Figure 3.9: Estimated Network Cost Scaling

Other Naming Graphs

The preceding analysis assumed that the name space is a complete, balanced β tree. Unbalanced trees are generally more favorable. A balanced tree places every leaf as close as possible to the root. Unbalancing the tree pushes more users farther away. Thus, the average user will post fewer notices to bulletin boards near the root unless users preferentially post to near the root. Incomplete trees are favorable for much the same reason. The missing descendants reduce traffic to the parent node because there are fewer readers of leaves and increase the average leaf/root distance by unbalancing the tree.

On the other hand, the costs can be far worse than estimated for a general DAG. The number of bulletin boards read may be a power of U, far higher than estimated, because the number of ancestors of a leaf can grow geometrically until it includes all the remote ancestors. The only hope for good behavior with a general DAG is that users are compelled by time constraints to limit the number of ancestral bulletin boards they read.

3.4 The Taliesin Name Space

The need to encourage bulletin board creation led to the recognition that a bulletin board system requires a structured name space. The design for Taliesin uses an attribute oriented name space structured so that sub-topic and super-topic relationships can be identified.

3.4.1 Name Space Syntax

Names should correspond to the most commonly used notice selection criteria. A selection can be phrased as *attribute relation value*: e.g., author = John Doe. Taliesin structures a bulletin board name as a set of *(attribute, value)* pairs. At any one time, only a finite number of bulletin boards exist, but new ones may be created. While runtime creation of new attributes allows greater flexibility, this design opts for the simplicity of using a predetermined set.

Attribute Choice

The most important attribute is *topic.* It may have as its value a list of keywords specifying topic categories. The intent is that the topics discussed within a bulletin board are in the intersection of the categories named by the keywords. For example, if the topic attribute is **{graphics, terminals},** the bulletin board should hold notices pertaining to both graphics and terminals. It could contain discussions about terminal support in various graphics software packages, about the merits of terminals capable of graphical display, or a variety of similar topics.

This design uses two other attributes, *site* and *organization.* The site attribute encourages creation of bulletin boards that are relevant only to particular locations. Hence, it promotes locality of use. The site attribute may also be used to guide the decision as to where to place the copies of a bulletin board. The organization attribute reflects another type of locality, based on community or organizational ties. It, too, may be useful for guiding copy placement or assigning initial access rights. If a bulletin board is associated with a particular person or role, the organization attribute will be used to identify the person or role. Other attributes such as *author* are not included because it was felt that they do little to encourage locality of reference or to guide copy placement or other policies.

Attribute Values

An attribute value is a *set* of keywords. Each of the keywords can be either a simple name or a qualified name.

SITE:	**(none given)**
ORGANIZATION:	**stanford/vlsi-project**
TOPIC:	**terminals, graphics/3-D**

The order of appearance of the keywords is not used to distinguish between different names. This feature has user friendly aspects. People need to remember only the keywords, not their order. It is easier to keep in mind that a bulletin board pertains to graphics and terminals than to remember whether the ordering is **graphics/terminals** or **terminals/graphics.**

Sometimes, however, it makes sense to have hierarchy within a keyword. For example, a value for the organization attribute might reflect the organizational hierarchy of a company. It can also be useful to distinguish between the names of the communities, when a bulletin board is of interest to several communities. For example, an organization attribute value of {CMU, **Stanford, EEJDept, CSJDept},** is confusing while the name **{CMU/EEJDept, Stanford/CSJDept}** clearly indicates that the bulletin board is for the department of electrical engineering at Carnegie Mellon and for the computer science department at Stanford. Similarly, hierarchy is very useful for clarity in the site attribute. Without hierarchy, the site attribute **{Pennsylvania, Peru}** doesn't distinguish between a bulletin board devoted to the town Peru, Pennsylvania and one devoted to interactions between the state of Pennsylvania and the country of Peru.

Values of the topic attribute should ideally use single nouns as keywords. Separating the nouns as distinct keywords avoids imposing an arbitrary hierarchical ordering. Sometimes it makes sense to qualify a category, much as nouns are sensibly qualified by adjectives. Thus, topic keywords such as **graphics/3-D** are acceptable. A good test for whether a word should appear as a qualifier is to ask if qualification in the reverse order would be nonsensical. For example, consider the problem of choosing between **{graphics, 3-D}** and **{graphics/3-D}.** The test is to ask 'What kind of 3-D (thing) is discussed?'.

Because the answer [6]A graphical 3-D' does not make sense, it is clear that the term 3-D modifies **graphics** and is not a separate category.

3.4.2 Name Space Structure

Collections of bulletin boards are specified using the structure of the name space. The structure is derived from sub-topic/super-topic relationships defined between pairs of bulletin boards. This produces a name space in the form of a DAG.

Implicit vs. Explicit Relationships

The syntax of bulletin board names implicitly represents some relationships. These could be used as the sole b$,sis for the name space structure. Explicitly defined relationships would provide more flexibility, but suffer several major disadvantages, as discussed below.

Explicit declarations must be stored. Moreover, they must be kept consistent. Inconsistency can arise as disagreement between the name space and the explicitly declared relationships. Figure 3.10 shows a situation in which the natural link indicating that IBM microcomputers is a subtopic of microcomputers is missing. A user asking about microcomputers won't receive notices about IBM microcomputers. Inconsistency with the name space cannot arise when relationships are computed based upon the syntax of names.

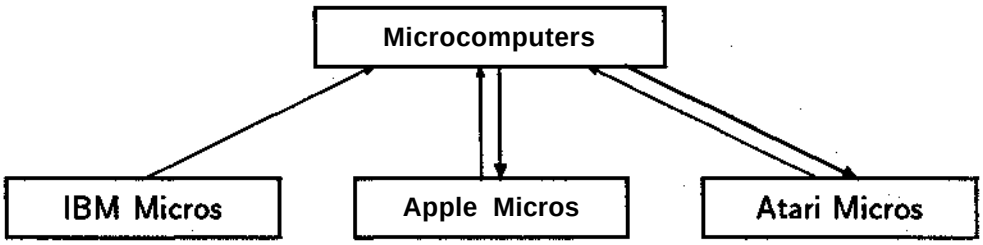

Figure 3.10: Potential for Inconsistency

Using explicitly defined relationships also entails checking to ensure that the name space graph remains a DAG, For example, consider the name space in Figure 3.11. Because this name space graph contains a cycle, a user asking to to read about astrophysics might put a postmaster into an infinite loop trying to locate all its subtopics. Checks must be made to disallow the creation of cycles. There is no such risk with implicitly defined relationships.

Explicitly defined relationships have other weaknesses. Because they are explicitly defined, users must be vigilant enough to create all the proper links for new bulletin boards. Using explicit relationships also makes it more difficult to avoid generating duplicate notices when reading. Notices are duplicated if a query is sent along multiple paths, as shown in Figure 3.12. The list of bulletin boards in a collection must be checked for duplicates. With implicitly defined names, name look-up need only locate all names matching a syntactic pattern. No duplicates are generated as part of the scan. The look-up using explicitly defined relationships cannot be defined as a simple pattern matching operation based on name syntax. Hence, it is more time consuming and more apt to be mistakenly implemented in a way that produces duplicates.

3.4.3 Chosen Relationships

Taliesin implicitly represents relationships between bulletin boards. A bulletin board covers a subtopic of another if it has the same name, but with the addition of keywords to further

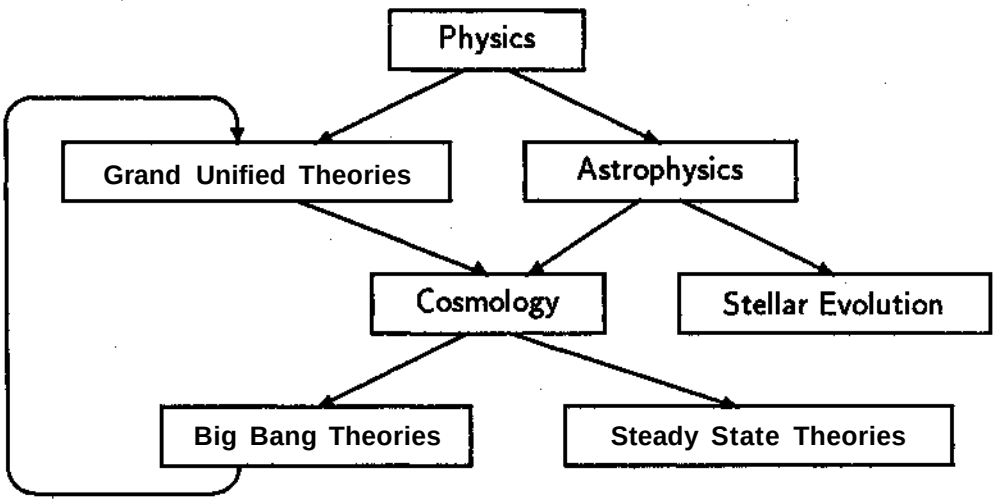

Figure 3.11: Danger Due to Cyclic Relationships (Non-DAG)

restrict the topic. Six relationships are recognized.

- super-regions
- super-organizations
- super-topics
- sub-regions
- sub-organizations
- sub-topics

Collections of bulletin boards are defined in terms of a root bulletin board plus all other bulletin boards related in the specified ways. For example, a user might express interest in the topic *graphics* and all its sub-topics.

Consider the name of Bulletin Board 1 in Figure 3.13. Its descendants — sub-regions, sub-organizations, or sub-topics — will contain the same keywords plus one or more new keywords of their own. Thus, Bulletin Board 2 is a sub-organization and Bulletin Board 3 is a sub-region. Bulletin Board 4, however, is not related. In practice, this simple definition of descendant is extended by allowing not only the addition of keywords, but the appending of qualifiers onto old keywords. Thus, Bulletin Board 5 is also a subtopic of Bulletin Board 1.

Note that this structuring of the name space allows it to be viewed as a graph. Each bulletin board corresponds to a vertex in the graph. If a bulletin board *A* is a descendant of a second bulletin board J5, then a link can be drawn from *B* to *A.*

If links were defined explicitly, the structure of the graph need not resemble in any way the structure of the name space. For example, the graph in Figure^.!4 would be a perfectly legitimate way of organizing bulletin boards. Different implicit relationships are possible, too. For example, no distinction might be made for subtopic/supertopic relationships based on attribute type. Taliesin keeps the distinction because it gives users the opportunity to avoid queries searching remote sites and locations. Remote queries may have significantly slower response times.

Implications of **Structure**

The chosen method for structuring the name space and the adoption of queries based upon that structure affects the design and efficiency of the system. For instance, the operations **AddBulletinBoard, EditBulletinBoard,** and **UpdateInterest** defined in Chapter 2 actually

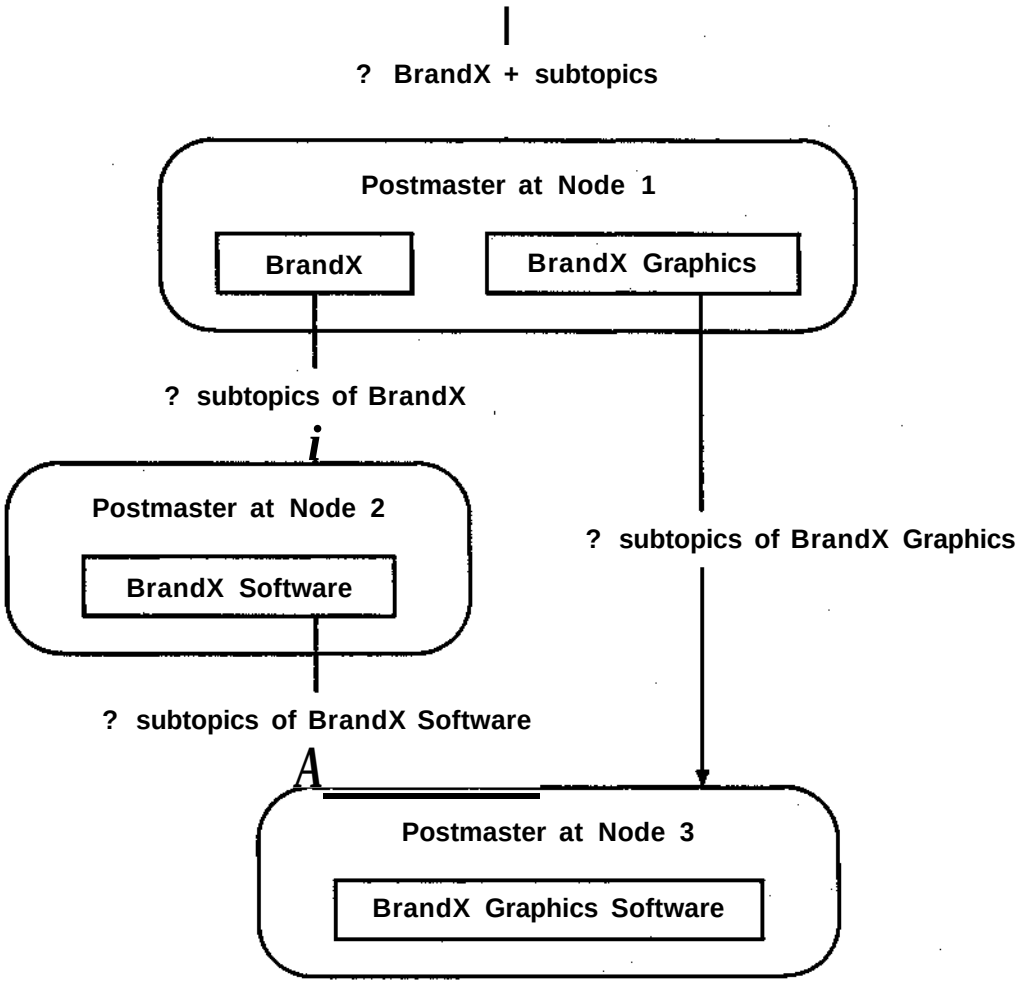

Figure 3.12: Generation of Duplicate Notices

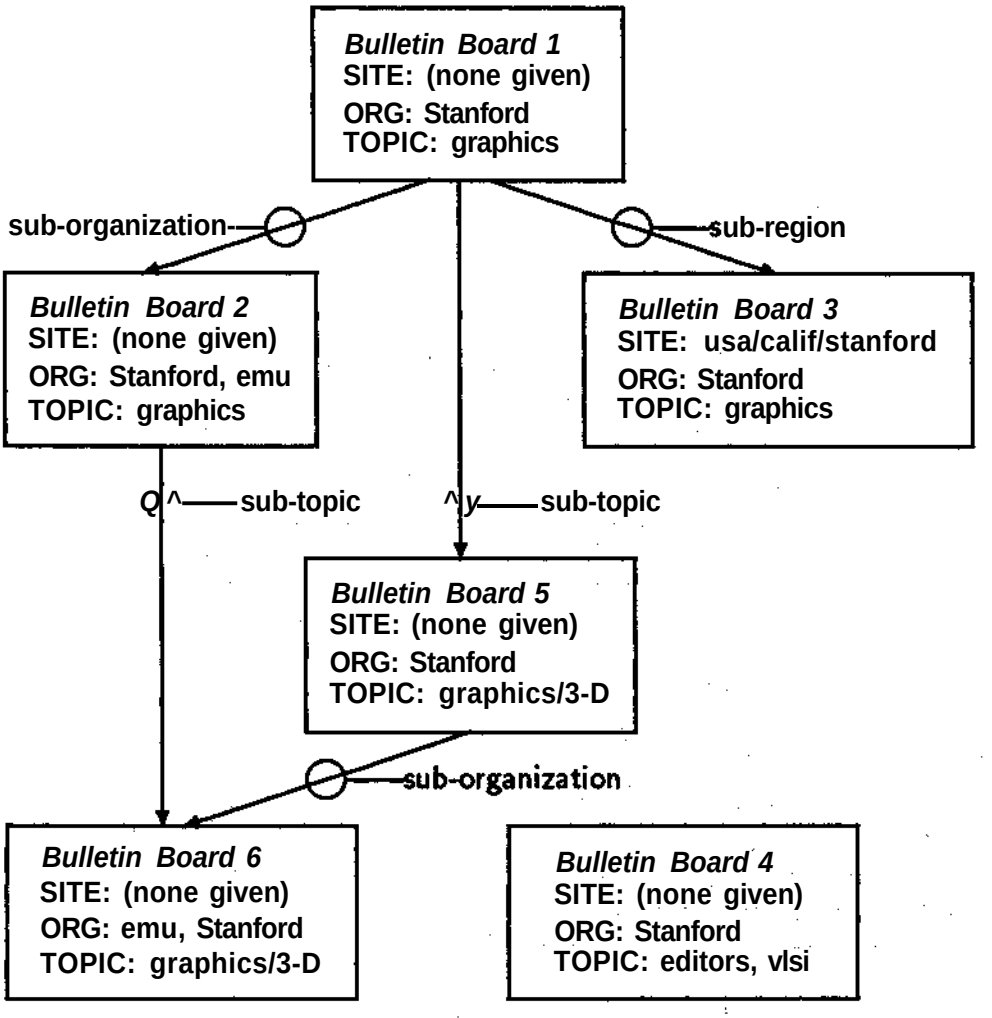

Figure 3.13: Sample Name Space Graph

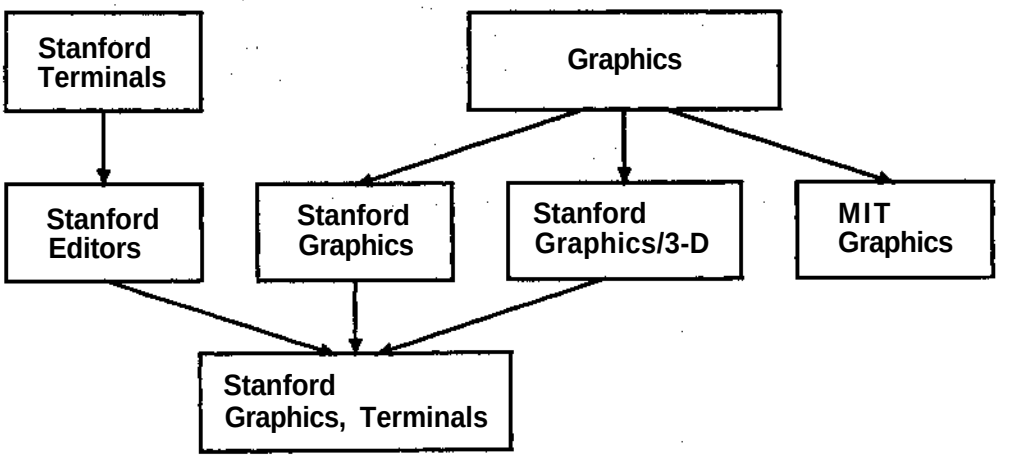

Figure 3.14: An Alternative Name Space Graph

do not take the name of a bulletin board as a parameter. They take a specification of a collection. Furthermore, while the profile of a user reading the collection of bulletin boards relating to graphics in Figure 3.13 will contain a single record stating interest in that collection, the u^er is actually reading five bulletin boards. His user profile must record what notices have been viewed in the entire collection, not just in one of the five bulletin boards.

Because collections can contain multiple bulletin boards, **ReadNewNotices** returns an array of records, one per bulletin board read. It would have been nice to return a single record, but that does not appear to be feasible. The replication algorithm cannot give good performance if constrained to use a common time scale across all bulletin boards. If there is no common time scale, the bulletin board service must provide some method of translating between the times at all the copies of the many different bulletin boards. Such translation requites additional storage, just as an array of records does, but doing the translation would almost certainly be more complex than looking for the proper read-time record in the array.

It is possible that a user won't see every bulletin board in a collection each time he attempts to read the collection. A transient condition such as network partition may temporarily hide a bulletin board. Unavailability once does not mean that a bulletin board has been destroyed. However, long term unavailability is highly suggestive of destruction. When a bulletin board is destroyed, the record of when it was last read should be expunged, but not before then. So, a second field is added to the collection of defaults in user profiles. It states how many successive failures to find a bulletin board must occur before it should be assumed that it has been destroyed.

Another effect of reading all the bulletin boards in a collection, particularly since collections are organized around a common topic, is that users are apt to encounter notices that have been posted to more than one of the bulletin boards. Taliesin postmasters attempt to avoid presenting users with the same notice twice. To this end, Taliesin returns only oiae copy of any notice even if it is present in more than one bulletin board in the collection.

3.4.4 Expressive Power

The other aspect of the chosen name space that has important implications is that it limits the queries that user agents can issue to postmasters. To explain how much network traffic may be increased because unwanted notices are being filtered out at user agents rather than postmasters, the chosen query language will be compared with relational algebra, a general database query language[U1182]. The operators used in relational algebra are union, difference, intersection, selection, join, and projection.

In a bulletin board system, the only object of interest to users is the notice. Join is needed only to reconstruct notices from the underlying relations or to pick out notices indirectly through their relationships to other notices: for example, to fetch all notices written by the same author as a notice sent at 10:42 AM. If the author were known in this case, no join would be needed. Such complex queries seem likely to be infrequently used in a bulletin board system.

Projection is used to adjust the arity of tuples in the intermediate stages and to get rid of unwanted fields in the final answer. Intermediate results, in this case, are always notices so the arity need never be changed. Taliesin postmasters support some pre-defined projections: for example, just the envelope of a notice can be returned. Efficiency should not be greatly reduced by the fact that only a small number of projections are provided.

Queries using union, difference, intersection, and selection map into logical formulas, which can be put into disjunctive normal form. Union corresponds to OR, intersection to AND, and difference to AND NOT. Selection corresponds to unary predicates such as AUTHOR(tuple) = EDIGHOFFER.

Such formulas can be processed as a series of queries, one per term in the disjunction. No extra notices are transmitted as a result. Conjunctions may map onto sub-graphs of the name space. For example, in Figure 3.13 Bulletin Boards 1, 2, 3, 5, and 6 correspond to the conjunction (TOPIC=Graphics) AND (ORGANIZATION=Stanford). Because only a finite set of bulletin board names exist, only a finite number of conjunctions are representable. Because not all conjunctions are representable, user agents must sometimes ask for more than is really wanted, a potentially serious cause of unnecessary traffic.

Another important reason why unwanted notices may be transmitted is the restricted availability of NOT. This effect of NOT is only produced with the chosen query language when a user asks to see a bulletin board but not all of its sub-topics. For example, (ORG=Stanford) AND (TOPIC=Graphics) AND NOT TOPIC=Graphics/3-D identifies just Bulletin Boards 1, 2 and 3. Most of the time, a user agent cannot ask to avoid pieces of the name space tree. It can, however, be more precise about what parts the user does want to see: for example, by specifically asking for Bulletin Boards 1, 2, and 3.

The final major reason for unnecessary notice traffic is the limited choice of predicates for selection. Postmasters select only on the basis of bulletin board names, notice identifiers, and posting times. A significant amount of efficiency is lost if users often decide what to view based on other attributes of notices.

Taliesin's name space has reasonable expressiveness. It is anticipated that the queries it supports will be the ones users most often want to ask so that excessive unnecessary network traffic will be avoided. However, there are still reasons for building more sophisticated selection functions into the user agent. In particular, users are apt to want to be able to select on other properties of notices, such as author, and they may want to exclude more keywords.

3*5 Lessons Learned about Naming

An important aspect of providing a distributed bulletin board service is to find a way to make it pleasant for people to use while encouraging them to avoid straining the system by flooding it with too much traffic. The choice of a name space influences user behavior by changing the relative difficulty of the various operations. Adopting a keyword-oriented name space solves two problems. A set of unordered keywords is easier to remember than either an arbitrary, long name from a flat name space or a particular ordering from a hierarchical name space.

A keyword-based name space also provides structure that can be used by a bulletin board system to recognize topic/subtopic relationships between bulletin boards. That recognition enables users to state what topics are interesting, independent of what bulletin boards exist at a particular time, and eases the introduction of new bulletin boards. Easing the introduction of bulletin boards makes it at least plausible to hope that users can be persuaded to follow a small number of bulletin boards covering topics of particular interest. If this happens, users will not be inundated with large numbers of notices they don't want to see and bulletin boards will neither grow in size nor be replicated at an ever growing number of locations. The scaling of all costs will be improved.

A structured name-space is also useful for a number of reasons totally unrelated to the need to encourage favorable user behavior. Structure can be used to partition the name space into pieces, avoiding the need for complete replication at all nodes and giving greater administrative control over local names [LEH85,Moc83,MP85,Ter83].

The analysis in this chapter confirms the growing recognition that structured name-spaces should be used for large computer message systems. As others have noted, such name-spaces can be more user friendly than flat name spaces and they are easier to administer on a large network. These reasons were recognized by IFIP WG 6.5 on directory management and led to exploration of ways to implement a structured name-space

standard[CCM85,Kil85]. In addition, the work in this chapter indicates that structure can be used to reduce the costs of a bulletin board system.

Chapter 4

Replication

To be a useful tool, a bulletin board system must offer reliable, fast access to the notices it stores. Bulletin boards are located by reading name-bindings. To determine which bulletin boards and notices they want to view, users must first read their user profiles. This means fast, reliable access is required for user profiles and name-bindings as well. Replication of agents and the objects they manage is the key to achieving these performance goals. It improves response time by placing copies near the point of use and enhances availability by allowing one copy to be used if another is unavailable.

4.1 Concurrency Considerations

Replication does not necessarily bring improved performance. For example, if operations reference a large number of copies, response will be slowed by the more distant or isolated copies. Availability will be higher when using a single copy on a reliable node unless operations can be performed concurrently on only a few copies. By taking advantage of application-specific semantics, it is possible to design a replication algorithm with more concurrency [BG84,Gar83,SJRN83,SS83,SS84]. The chosen algorithm must have enough concurrency so that its performance will meet the needs of the application.

4.1.1 Desired Performance

Performance in a bulletin board system is judged primarily in terms of whether response is fast enough to keep users from becoming impatient. Response time is most crucial for reading a bulletin board. Since this operation is normally preceded by reading a user profile and requires parsing names, all reads must be fast.

In addition, changes must become visible soon enough that users do not become annoyed at the delay. However, the importance of rapid visibility of updates depends on what is being changed. User profiles have the most critical need. Users should reasonably demand that writes be completed by the next time they start a session with the bulletin board system. Fortunately, a user profile will typically be accessed by a single person and then only from a handful of nodes corresponding to the user's normal login nodes. The interval between sessions is apt to be on the order of hours. So, writing user profiles is not demanding in terms of how much concurrency is required.

Users will also want the notices they post to become visible quickly. This provides reassurance that the notices have really made it to their destinations and allows quick answers to questions. Notices should become visible quickly locally even if network speed or frequency of partition make it impossible to propagate them quickly[1]. Because notices

[1]Grapevine discourages ill-thought replies by enforcing next-day service rather than making postings

typically are posted simultaneously at many widely distributed nodes, the notice replication algorithm should offer a lot of concurrency.

The other types of updates are creation or destruction of objects and changes to access control lists, automatic deletion rules, and copy sets. These do not require particularly fast handling or a great dejal of concurrency. For one thing, such updates occur less frequently. For another, users are not apt to feel obligated to sit around and wait for their completion. They typically will be satisfied to accept an immediate indication that the bulletin board system will act in good faith to carry out the action. By use of the various queries, users can inquire to see if their changes have gone into effect. Delays on the order of a day should not be particularly distressing. Even delays of several days may be tolerable.

All these factors add up to widely varying concurrency needs within a bulletin board system. Read operations must provide very fast response and offer high availability. Notice updates require a high degree of concurrency and should complete quickly, as least as long as a local copy of the bulletin board is available. Fortunately, the other updates do not require as much concurrency.

4.1-2 Potential Conflicts

Concurrency allows the interleaving of sequences of operations. Correctness depends on identifying those operations whose activities might influence one another and providing synchronization [EGLT76,Pap79,SS83,TGGL82,Wei85]. Simple reads have minimal interaction with other operations. This type of read includes queries about the attributes of user profiles and bulletin boards and searches over the name space. Since they leave behind no trace in the bulletin board system, simple reads don't interfere with one another. The results returned do depend on what writes have been completed, however.

The dependence oil write order is usually of little import in a bulletin board system. Consider a simple query. It involves parsing one name and examining the named object. If the name has been defined but the object has not yet been created, a reader will be told that the operation failed. Similarly, if the object has been created but the name has not, a user will be told that the system was unable to find an object by that name. Both are transient failures not apt to confuse people so long as the anomaly is resolved fairly quickly Other dependencies on the order of writing are even less noticeable. For example, there might be a race between changing the automatic deletion rules to permit the deletion of expired notices and changing the access control list to allow a new user to read the bulletin board. At worst, the order of the writes determines whether the new reader sees expired notices. The atomicity of writes ensures that readers do not see a half-written fields. For example, users are not denied access to a bulletin board because their names happen to be on the unwritten part of a half-written access control list.

Other read operations are not so simple. In particular, reading a bulletin board is normally done by reading a user profile, reading the recommended bulletin boards, and then writing the user profile. So, the full operation of **ReadNewNotices** in a very real sense interacts with calls to **PostNotice** or **DeleteNotice.** It conflicts with modifications to the user profile as well.

The operations of posting and deleting notices seem to be independent as long as they are applied to different notices. Technically, however, these operations do involve reading a bulletin board and Writing a new version with the selected notices added or removed. If the ordering of postings is not needed by the system, users could cope with seeing different orders at different copies. However, the need for a compact, copy-independent measure of what has been seen before implies that the system needs to know the ordering.

As defined in Section 2.10.1, bulletin boards and user profiles are created at a single initial copy. The only possible conflict is in the allocation of names. Other modifications

visible as soon as possible.

cannot be invoked until creation is complete. The other updates are handled by postmasters manipulating a single field: reading the current value, computing anew value based on the command parameters, and writing out the result. Updates to most attributes interfere only with updates to the same attribute, with destruction of the object, and possibly with changes to the copy set. Some attributes have structure and so greater concurrency is theoretically possible. In particular, different entries in an access control list and records for different bulletin boards in a user profile can be modified simultaneously.

Taliesin creates, modifies, and destroys name-bindings. There is no conflict between alterations as long as they apply to different name-bindings. Concurrency does needs to be controlled to guarantee that two objects are not created with the same name. If all name service agents must agree upon whether a name is not already in use, creation would take a long time and produce a lot of communication over the network. Taliesin therefore partitions the name space up into *replication groups.* Each name belongs to a single replication group. Only those agents keeping a copy of that group need to come to agreement on the legality of allocating a name.

4*2 Algorithm Choice

The concurrency possibilities and performance demands differ radically for the operations to be provided by Taliesin. This raises the possibility that the needs might be best met by using multiple algorithms. There is precedent for such an approach: for example, the SDD-1 database system used four update protocols[BRGP79]. Its protocols provide varying degrees of synchronization, but the ones requiring less synchronization are applicable to fewer situations. In a similar fashion, Taliesin will use different protocols to take advantage of the semantics of the different operations.

4.2.1 An Algorithm for Most Updates

Knowledge of what interactions are cause for concern can be used to determine what interleavings of actions produce correct results. If operations that produce legal transformations are executed one at a time, a database certainly remains *consistent* Consistency has two aspects. Each copy must remain *internally consistent:* that is, the semantic relationships must be preserved. For example, a bank's database of accounts must not create or lose track of money.

In addition, the concurrency control algorithms must ensure *mutual consistency* between copies. Intuitively, mutual consistency can be thought of as meaning that the copies undergo the same sequence of operations. If a bank's database is not mutually consistent, a client might be overdrawn at one branch because a withdrawal was processed before a deposit. At different branch, the same client might not be, because the deposit was processed first. This would create disagreement over whether the client should be forced to pay some sort of overdraft penalty.

Most often, database researchers adopt the criterion of *serializability* as the test for correctness[Pap79,SJRN83,TGGL82,Wei85]. According to the definition of serializability, a concurrency control algorithm is correct only if every way it might possibly interleave the steps in a sequence of updates has the same effect as could be produced by some serial execution. Mutual consistency is preserved if the operations not only appear as if they were applied sequentially, but the apparent sequencing is the same at every copy.

Traditional concurrency control algorithms support the operations *read* and *write.* The database is partitioned into a collection of items that can be independently modified. Conflicts between operations are determined on the basis of whether the usage of any item conflicts. Figure 4.1 shows a typical compatibility matrix. Of course, there is no conflict if

operations apply to different objects. Several existing replicated file systems use read/write semantics[F081,Svo81,WPE*83].

Operation	Read	Write
Read	UK	Conflict
Write	Conflict	Conflict

Figure 4.1: Traditional Read/Write Semantics

Section 4.1.1 concluded that minimal concurrency is needed for updates except for notice operations. Read/write semantics are acceptable for everything except replicating notices and, in a sense^ for name-bindings. Essentially, name-bindings are subject to read/write semantics with each binding being considered a separate object. Replication of notices within bulletin boards will be done by a separate novel algorithm that provides agreement on posting times yet has a high degree of concurrency.

One requirement of the general update algorithm is that it must allow the copy set to be changed. Because changes to the copy set interact with the propagation of notices to all copies of a bulletin board, the general algorithm must have certain properties. A compact, copy-independent representation of what a user has read creates a need for global sequencing of notice postings. Exactly why this is so will be explained in more detail in Section 4.2.2. If such sequence numbers are generated, then disagreement about the order of changes to the copy set, or *schism,* produces incorrect behavior. Consider the following situation in which two copies, *A* and JB, exist.

Copy	Sequence Counter	Copy Set	Accepted Notices	Notices to be Posted
A	2	*A, B*	N_1@1	N_2
B	2	*A, B*	N_1@1	N_3

Suppose a loosely synchronized replication algorithm were used that allows a copy to propose a change and then to act on the change until such time as it hears of some other, superceding change. If a state of network partition separates nodes *A* and J5, it would be possible for *A* to decide to delete *B* and for *B* to decide to delete *A.* This would lead to the temporarily inconsistent state:

Copy	Sequence Counter	Copy Set	Accepted Notices
A	6^	*AT^*	N_1@1, N_2@5
B	7	*B*	JVi@l' N^{3}@4

This state is one in which a schism exists: actions are being taken on the basis of sequence of copy set changes that may not be legal. If user *U* reads copy J3, giving a prior read time of 2, he will be shown notice JV3 and told that his new read time is 7, Now, suppose the network remerges. The recovery algorithm must resolve the inconsistent beliefs as to the set of copies in one way or another. Suppose that the consistency restoration algorithm gets rid of copy *B.*

Copy	Sequence Counter	Copy Set	Accepted Notices
A	7	*A*	• iViPl, iV2v95, iV3O:

Suppose user *U* now reads copy *A* using his prior* read time of 7. He will be shown no notices and is told his new read time is also 7. This means he has forever missed notice

The potential for schism arises whenever it is possible to retract or otherwise undo the decision to destroy a copy of a bulletin board. In particular, none of the optimistic algo-

rithms that attempt to back out of erroneous updates will work. There are, however, many published algorithms that satisfy the need to avoid schism. The functional requirements discussed in Section 1.3 further guide the decision.

Taliesin uses a modified majority vote algorithm modeled after that described by Thomas[Tho79]. Gifford also developed a voting algorithm, but it handles just read-only and write-only updates[Gif81]. Thomas's algorithm produces agreement on the order of changes to the copy set so schism will not arise. All copies are peers, yet differences in their availability or response time can be compensated for by the vote assignment policy. Most importantly, majority vote requires availability of only a majority of the copies, not all of them. However, some adaptations are needed. These will be described in Section 4.5. An approach suggested by Greene has a number of similar merits, but it was felt that sites frequently cut off by network partition would tend to be slighted[Gre81].

4.2.2 An Algorithm for Notice Replication

Replication algorithms based on read/write semantics, including majority vote, do not provide enough concurrency for the notice operations. In a bulletin board system, it is typical for many notices to be posted to a bulletin board concurrently. Using voting, locking, or similar techniques to enforce a globally agreed upon ordering for notice postings takes too much time because no concurrency is permitted. If postings are not assigned an agreed upon order, the representation of what has been seen before is a problem. Local time values are not acceptable because the functional requirement for transparent access to different copies would not be met. There is no rule for translating the local time at one copy to the local time at another copy. In fact, even if the translation between the two times were known, there is no guarantee that the postings would appear in the same order.

Even getting global agreement as to which notices were posted prior to a read operation is too slow when copies are scattered across a large or partitioned network. If no global time is agreed upon as part of the read operation, the record of what a user has seen must include the identity of each notice seen. Such a list grows with time. If not pruned, it would consume an undesirable amount of storage.

Taliesin gets good performance by using an algorithm tailored to fit bulletin board semantics. To decide when concurrency results in correct behavior, it is necessary to identify which operations conflict. Conflicts over notice operations arise primarily as a result of the need to save a record of what has been seen before. Bulletin boards, in this sense, are a form of historical database[CW83]. However, bulletin boards are not true historical databases. A bulletin board is more naturally viewed as a collection of notices, being updated by adding or deletion notices. Thus, while the readers do want to phrase queries that have a time element, they are interested in the present contents of the bulletin board system. In a historical database, the queries would be geared toward asking what the bulletin board held at an arbitrary time.

One way to get a higher degree of concurrency is to use directory semantics [BG84,BLNS82,DS83,OD81,PWC*81]. Under directory semantics, posting and deleting notices are independent operations as long as different notices are involved. Such an approach is used successfully in the Clearinghouse directory service [0D81] and the Grapevine name service[BLNS82]. Normally, convergence between copies is ensured by periodically computing the union of the sets at different copies. If there are multiple updates affecting the same object, their ordering is resolved by locally generated update time-stamps. An alternative way of assigning an ordering to creation and deletion events was suggested by Daniels and Spector[DS83].

Unfortunately, bulletin boards do not quite have the same semantics as directories. Like read/write semantics, directory semantics treats read operations as conflicting with postings so that it can be definitively said that a notice was posted either before or after

a particular read. This treatment produces even less concurrency than techniques that uniquely order postings because reads are more common than postings.

Other approaches to getting more concurrency have been proposed. One tact is to act optimistically then undo any troublesome results[Dav84,MPM78]. If the semantics are known, it is possible to make use of an *undo* operator for each operation[SS84,SS83]. Many conflicts are to be expected with bulletin boards, however, so that most notice postings and/or updates to user profiles would be unwound. Thus, these approaches are unacceptable.

Another approach to gaining concurrency is to allow the simultaneous existence of more than one version of an object[BG83,Svo81]. Under this view, a transaction creates a version with the writing of a data item. Traditionally, transactions conflict when they try to concurrently manipulate the sole version of an object. In the case of read/write semantics, conflicts occur when one transaction tries to write the object when the other is trying to either read or Write it. When multiple versions are allowed, write/write conflicts are no longer a problem. Unfortunately, this doesn't help because posting a notice really consists of reading the bulletin board and writing a new version with the notice appended.

Because the replication and concurrency control methods developed by others are insufficient, a novel algorithm will be presented. It is based on a slightly different notion of correctness. Correctness in viewing notices is not measured by whether operations can be serialized in the same fashion at every node. It is defined in terms of presenting a serial view to each user. The ordering seen by different users need not be the same. Except in pathological cases of rapid changes to the copy set, users will see every notice exactly once. Rarely will replies be presented before the original notice. If the agents managing bulletin boards consist of multiples processes or interleave requests from multiple users, each agent must still implement local concurrency control to serialize access to its data files based on read/write conflicts.

4.2.3 Algorithms for Reading

Fast response requires that the number of copies accessed as part of any read operation should be kept to a minimum. Gifford's work on a similar voting scheme designed for file replication shows that reading a single copy can result in outdated or inconsistent views[Gif81]. A consistent, current view is obtained only if the votes needed to approve an update plus the number of votes of copies scanned in a read is guaranteed to exceed the total number of votes in the pool. However, if a query is purely informational, the asker will rarely be hurt if an older view is shown for a little while longer. Changes based on old information will be rejected by the system because the update algorithms enforce serializability. Other uses of old versions of user profiles, bulletin boards, and name-bindings likewise do little harm. A user may be given the wrong access rights for a while longer if his requests are directed to out-dated servers. Possibly, a copy of ^a bulletin board or user profile will be temporarily inaccessible. All these effects are a minor price to pay for fast response, especially since users will normally try to re-read bulletin boards and so will later see the changes.

Returning the wrong data when reading a user profile is the most obnoxious outcome since it can result in a user being presented with notices that he has seen before. Fortunately, reads of many bulletin boards are apt to be separated by a span of hours so that updates will have plenty of time to propagate to the handful of locations associated with a user. If a user wants to continuously monitor certain bulletin boards or mailboxes, he could do so by using a user agent that runs in background during the day; Only when the agent is started up in the morning would it need to read the user's profile. The bulletin board system would have all night to finish writing the^updates from the previous day. Reads of user profiles, in any case, are most likely be directed to a copy at least 'near'

the copy to which the prior update was directed. Even if there is a moderate delay in propagating updates to all copies, no user is likely to notice.

In no case, then, is there a need for returning the most recent, consistent data in response to a read operation. Therefore, Taliesin adopts a policy of reading exactly one copy to meet the demand for fast response and high availability. Local read/write synchronization on the data files is still used to avoid reading a partially updated value.

To find a copy of a user profile or bulletin board, a postmaster first must read its name-binding. The underlying name space is structured so parsing a name may involve reading a number of directories. All parsing starts with the root directory of the *root replication group.* Every name server supports a copy of the root replication group. Each also maintains a table, the *replication group table,* holding the current locations of all other replication groups referenced by any locally supported replication group. The parsing of a name starts at any name server. It parses the name as far as possible locally. If it cannot complete the parse locally, it vises the replication group table to forward the request to any other name server that can continue. At each step in the parse, only one copy of a name-binding is ever read.

Once a postmaster has found the name-binding, it has in hand a list of copy locations. It will try to access these in turn until it succeeds in finding one. Preferably, the copies will be tried in increasing order of response time, cost of access, or some similar measure of suitability. The contents of that one copy's version of the bulletin board or user profile is returned. Here, too, only a single copy of any replicated data structure is used.

4.3 Notice Replication

As was pointed out before, the semantics of the notice operations closely resemble those of directories. The notice replication algorithm uses techniques similar to those used by Grapevine and the Clearinghouse. Notices are initially posted or deleted at a single copy. A notice's posting time is assigned using a local logical clock. Because this clock only needs to distinguish whether a notice was locally accepted before or after a user read the bulletin board, its granularity can be quite coarse. Henceforth, the values of the clock will be referred to as *epochs.* The epoch at which a notice is stored by the first copy is called its *signing epoch.*

Local changes are propagated to other copies whenever convenient through periodic transmission of *reconciliation reports.* It is not necessary for every postmaster with a copy of a bulletin board to directly issue a reconciliation report to every other postmaster with a copy. It is only necessary that there be a chain of communication between every pair of copies. Reconciliation reports need to be sent directly only between pairs of nodes forming the links in the chain. The decisions of when and where to send reconciliation reports are left as implementation issues. The architecture provides flexibility for a variety of network configurations.

The most important part of a reconciliation report is a list of notice postings and deletions that the intended recipient might not have heard of before. To simplify the process of checking to see if a notice has already been seen before, each is assigned a unique identifier. This unique identifier plus the signing epoch comprise the *signature* of a notice.

The identifier is generated locally[Wat81]. It is assumed that each postmaster has a unique identifier assigned through human administrative channels. The identifier given to a notice, then, consists of a local sequence number concatenated with the postmaster's identifier. Assuming that local sequence numbers run over a large enough range that they won't wrap around, generation of globally unique notice identifiers is possible using information available at only a single site.

It is easy to see how a new posting can appear in a reconciliation report. A notice

deletion, on the other hand, is reflected only as an absence in the local copy of a bulletin board. In particular, a postmaster getting a report needs to be told of notices that were deleted by the report's composer. Therefore, each copy remembers that a deletion has occurred by storing a *deletion marker* in the bulletin board. A deletion marker is a just signature plus a tag indicating that the attached object is a deletion marker rather than an envelope. The notice identifier is that of the notice to be deleted, while the signing epoch reflects the time of the deletion. Like notices, deletion markers are passed in reconciliation reports. Any time a postmaster receives a deletion marker, it stores the marker and deletes the original notice.

4.3.1 Simulation of a Global Clock

As described so far, epochs form a purely local time scale. However, a global clock is wanted to compactly record what a user has read. It is possible to compute an epoch that acts like a global clock time from the local epoch clock values. In fact, two global clock times are important. The first reflects how far a copy has progressed in gathering information about the actions taken by other copies. The maximal epoch for which a copy knows it has complete information will be called the *agreement epoch.* The other useful time on the global clock measures how far other copies have gotten in their efforts to gain complete information. The *expunging epoch* is defined to be the maximal epoch for which a copy knows that every action taken by any copy is known at every other copy. Henceforth, the local time on the epoch clock will be called the *posting epoch.* Each copy of a bulletin board is divided into three parts based on its posting, agreement, and expunging epochs as shown in Figure 4.2.

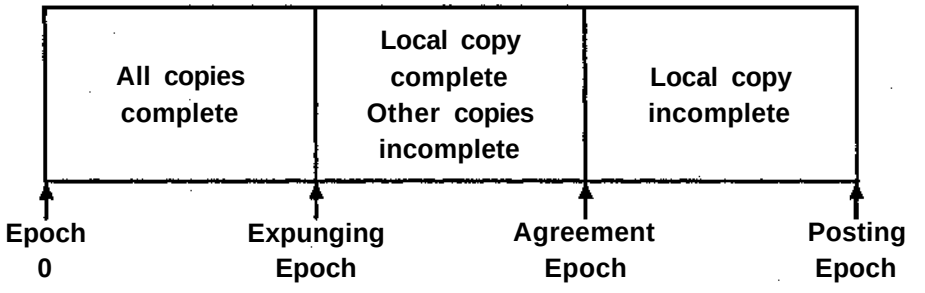

Figure 4.2: The Epoch Time Scale

A copy has a complete version of an epoch if it knows what notices were posted or deleted during that epoch by every copy. Suppose reconciliation reports contain the entire contents of one copy of a bulletin board. Receiving a report generated at epoch E by copy A implies that the receiver knows everything done by copy A at every epoch up through E. The local copy of a bulletin board is complete up through epoch 12, then, if reconciliation reports have been received from all other copies generated when the other copies were at epoch E or later. Accordingly, each copy stores an array of *posting epoch bounds.* The agreement epoch is the minimum value in this array. The epoch bounds are in many ways analogous to version vectors[PP83,WPE*83]. However, they are used to record the progress of other copies rather than to determine if an inconsistent state has arisen.

Reports do not have to be directly exchanged between every pair of nodes if each reconciliation report contains the full array of posting epoch bounds known to the composer. The reason for this is as follows. A posting epoch bound is advanced for only two reasons. A copy A can advance its own posting epoch at will, but it can advance that of another only if it has received a reconciliation report bearing the new, higher value. If a copy does advance the posting epoch bound of some other copy B to E, it does so ultimately because

B sent a reconciliation report composed at posting epoch *E*. The original report included all of J5's actions up through epoch *E*. Each link in the forwarding chain must also have included those actions in their reconciliation reports. So, *A* must have gotten word of JETs actions up through epoch *E*. Figure 4.3 shows how exchanging a reconciliation report affects posting epoch bounds.

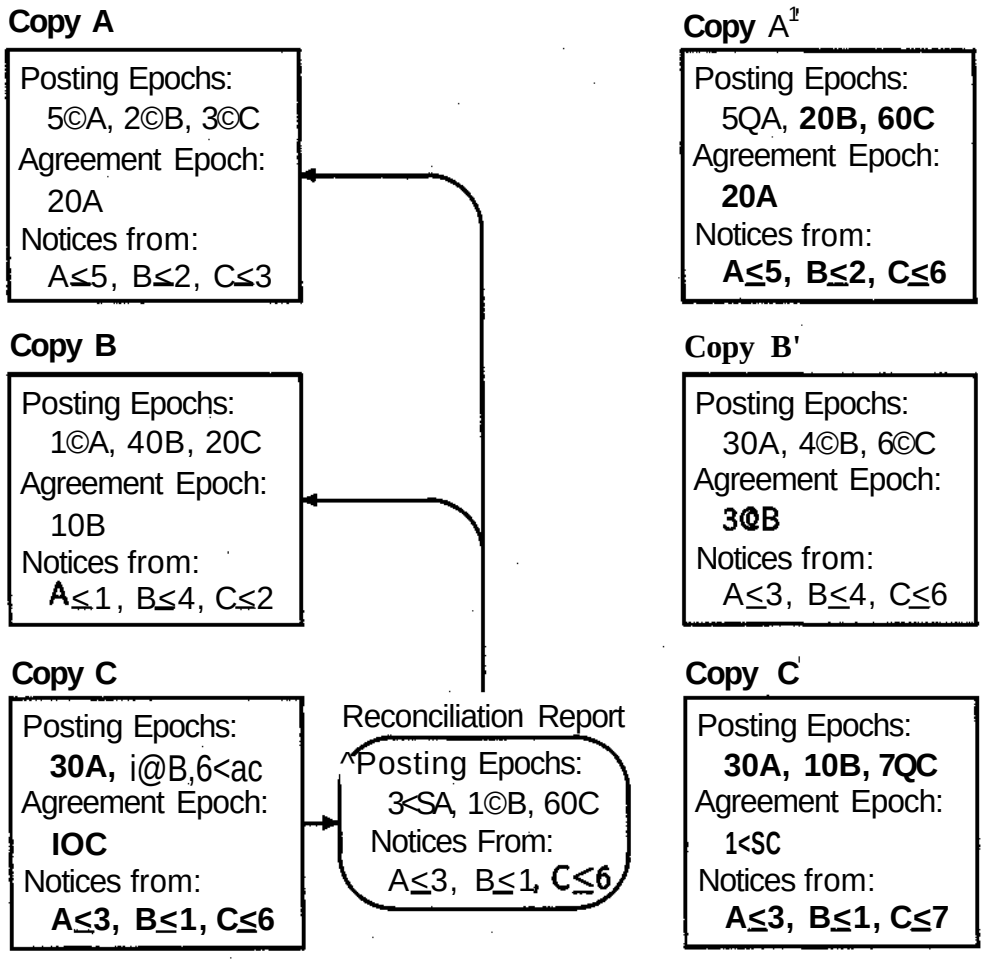

Figure 4.3: Computation of Epochs

Expunging epochs are derived in a fashion like that of agreement epochs. Reconciliation reports contain an array of *agreement epoch bounds*. By remembering how far other copies have progressed in accumulating complete knowledge, a copy can compute which copy is most ignorant. The minimum value of the agreement epoch bounds is the expunging epoch.

The expunging epoch is useful in determining what information can be safely discarded. Any deletion marker dated on or before a copy's expunging epoch can be discarded. Every other copy is known to have already been told about the deletion. Similarly, only notices dated after the expunging epoch need to be included in reconciliation reports.

4.3o2 Recording What Users Have Seen

Any copy knows of all notices posted anywhere dated with epochs tip to its agreement epoch. If a user has been told that he has seen all notices signed up to a copy's agreement epoch *E*, he would see only duplicates at any copy if he were to ask for notices dated *E* or earlier. The primary measure of what a user has seen, his *read epoch*, is the latest agreement epoch of any copy he has read. Note that because some copies may receive reconciliation reports sooner than others, their agreement epochs may differ. A user may read a copy with an earlier agreement epoch than that of a copy previously read, in which case the only notices reported will be those from the inconsistent portion (dated after the user's read epoch and, hence, the copy's agreement epoch).

Figure 4.4 depicts a bulletin board with two copies. Because all notices posted to either copy for epochs 1 and 2 have been forwarded to both copies, the agreement epoch is 2. If a user were to ask to read notices posted after epoch 1, he would be told of notices 2, 3, and 6 if he reads copy A or 2, 4 and 5 if he reads copy B. In either case, he would be told that the time of the read is 2. Note that if the reader then goes to the other copy for his next read, he will still see the notices that he missed. The missed ones are stamped as arriving after epoch 2, his new read epoch.

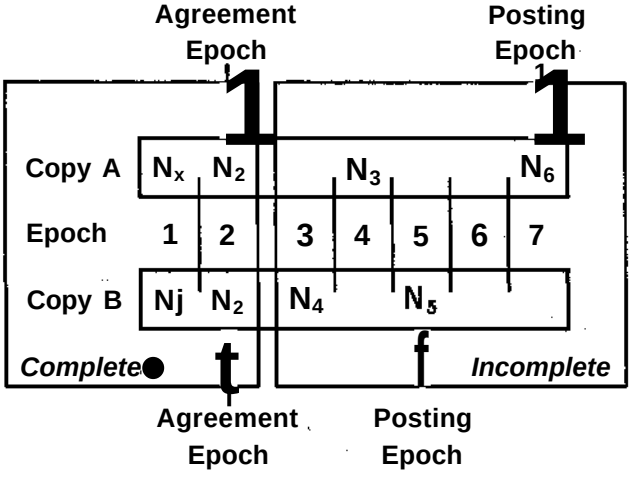

Figure 4.4: Epochs and **ReadNewNotices**

If only notices dated on or before a copy's agreement epoch were returned, new postings would not become visible immediately. Postmasters return all new notices, even those dated after a bulletin board's agreement epoch. A user's record of what he has seen includes a list of identifiers for notices dated after the reported read epoch for the bulletin board. Finally, Taliesin recognizes that users may stop a session before they have finished with the notices fetched by their user agent. A record of what was viewed includes a list of notices that the client wishes to view again.

One parameter to the operation **ReadNewNotices** is such a three part record of what was seen before. Postmasters pick out the notices to be viewed again plus any others that were not seen before. The operation returns the notices plus a read epoch equal to the agreement epoch of the copy read. User profiles store the same information. For convenience, when responding to **Updatelnterest,** postmasters sort out which notices are covered by the read epoch bound if presented with a complete list of those read.

4.3.3 Advancing the Epoch Clock

The reason an epoch clock is needed is to make sense of the relative ordering of notice operations. Because effects of calls to **PostNotice** and **DeleteNotice** are not externally visible until a user reads the bulletin board, there is no need to advance the current value of the clock, the posting epoch, as long as only they are invoked. However, the need for users to know if they read a copy before or after a notice was posted means that the posting epoch must be advanced after each read operation.

The derivation of the agreement epoch implicitly assumes that a copy never gives a new notice a signing epoch of *E* if it has issued a reconciliation report that purports to cover all its actions up through epoch *E*. So, the local posting epoch must be incremented whenever a reconciliation report is being composed. Of course, the posting epoch may also be incremented more frequently and can even be taken from a real time clock.

If posting, agreement, or expunging epochs diverge, the performance of the notice replication algorithm falls off. Reconciliation reports grow bigger, deletion markers consume more storage, and the agreement epoch becomes a poorer estimate of what users have seen. To ease the divergence problem, the posting epoch is transformed into a variation of a Lamport clock[Lam78]. Whenever a reconciliation report is received, the local posting epoch is advanced, if need be, so that it is at least as large as any reported epoch. This helps to keep the posting epoch from falling behind other copies' posting epochs. Strictly speaking, the posting epoch is not a true Lamport clock because no unique node identifier is concatenated to make epoch values unique and because it does not advance to become greater than the the last reported epoch. Uniqueness is not a necessary trait for epochs.

4*4 Modifications to Majority Vote

The majority vote algorithm proposed by Thomas uses time-stamps to detect conflicts and to determine which transactions to abort and possibly restart[Tho79]. One copy of everything needed to compute the new value is read. Each of these items has associated with it a time-stamp indicating when it was last modified.

Taliesin keeps only complete copies of bulletin board descriptors, user profiles, and name-binding replication groups. For example, every field of a bulletin board descriptor is stored at every copy, Ilowever, the granularity of locking is kept small to increase concurrency. Each name-binding has its own time-stamp. The copy set, access control list, and automatic deletion rules fields similarly have their own time-stamps. For user profiles, the collection of defaults has one time-stamp, the copy set has another, and the access control list a third. Similarly, each bulletin board's summary record holding its automatic deletion rules has its own time-stamp.

A proposed update requires majority approval. In Taliesin, operations are applied to only a single explicitly requested object: one bulletin board descriptor, one user profile, or one distribution list. Even implicitly, only one object is manipulated, although implicitly the name-binding for the explicitly requested object is always used. The only operations that change the name-binding are those creating or destroying copies. In fact, the only time when the update to the name-binding is a critical part of a Taliesin operation is when a user profile, bulletin board, or distribution list is created or destroyed. However, because creation and destruction boards only apply to unreplicated objects, the only replicated object involved is a name-binding. This means that approval is needed only from one set of agents: those implementing the name-binding or those implementing the bulletin board, user profile, or distribution list. There is no need for multiple votes for a single update.

A proposal describing the update must be circulated to the other agents with copies of the object. The proposal identifies which version of the field was used to compute the new value by identifying the field and giving its previous time-stamp. The voting process checks

to make sure that each update uses the most recent versions by checking time-stamps. 'No' votes indicate that an update is illegal because it was based on stale data. Simultaneous conflicting updates are handled by a race to get 'yes' votes. Those who have voted 'yes' for one update will vote 'pass' for subsequent lower priority concurrent updates and defer voting on higher priority ones. Taliesin uses the time-stamp assigned to the fields to be written as a priority.

The implementation of Taliesin uses a single coordinator. While vote proposals can still be daisy-chained from one voting node to the next, Taliesin requires the initiating site to retain responsibility for determining whether to commit or abort. Allowing any node to do so would greatly complicate the algorithm and make it nearly impossible to abort under additional conditions, such as taking excessive time to gather votes.

Taliesin makes a couple of important modifications to the original majority vote scheme by Thomas. The most significant change is that the majority vote algorithm is used to change the very copy sets that determine what constitutes a majority. The correctness of the majority vote algorithm derives from the fact that two conflicting updates cannot be passed because that would require approval in two majorities. Any two majorities in the same pool of voters has a common member. Under the voting rules, that common member is not permitted to vote 'yes' on two conflicting updates.

Changing the copy set changes the definition of a majority and so could produce incorrect behavior. However, if changes to the copy set are considered to be in conflict with changes to any other field of the same object, the problem goes away. Members of an old majority cannot vote for any change to any field occurring concurrently or after a change to the copy set. Actually all fields of the object need to Be written when the copy set is changed, even if only the old values are only re-written. This policy also provides a clear definition of the original state of newly created objects.

4.5 Accommodating Interactions

The majority vote scheme may be used to change the votes assigned to copies, but the notice replication algorithm requires accurate knowledge of what copies exist in order to know when to increment the agreement and expunging epochs. It is necessary to ensure that the two algorithms interact correctly. Consider the problem of creating a new copy, C, when two copies, *A* an4 B, already exist. The initial value of the new copy is computed as part of this operation. The table below shows the epochs for each of the copies, including those for *C* should the creation proposal be approved. Bounds on posting epochs are placed in parentheses. A question mark is placed after the plausible choices for the posting epoch, when the algorithm for selecting the proper one has not been defined.

	Posting Epochs			Agreement	
Copy	*A*	*B*	*C*	Epoch	Notices
A	4	(2)	—	2	N_x
B	(1)	5	—	1	N_t
C	(4)	2	4? 5?	2	JVi

If the two algorithms are not modified, there is no interaction between circulating vote proposals and reconciliation reports. Suppose *A* draws up the vote proposal to create *C*. Suppose the network greatly delays the voting messages, but a reconciliation report composed by *A* at epoch 7 and sent to *B* arrives quickly. Naturally, *B* updates its epochs in the normal fashion.

Copy	Posting Epochs A	B	C	Agreement Epoch	Notices
A	8	(2)	—	2	N_1
B	(7)	8		2	N_1
C	(4)	(2)	4? 5?	2	N_1

Now suppose that the. voting process approves the creation of *C*. *B* claims to have complete knowledge of all notices posted up through epoch 7 even though it has never even heard of copy *C* or its actions. Anyone who reads copy *B* is likely to miss out on the first few notices posted to copy *C*. This problem arose because lost or delayed messages can trick a copy into advancing its agreement epoch when it should be waiting to hear about the outcome of a copy creation proposal. The solution is to tag updates changing the copy set with a *voting epoch* and include them in reconciliation reports. The voting epoch should be the posting epoch of the coordinating copy for the same reason that notices are signed with the posting epoch. The defining property of the agreement epoch now ensures that copies hear about changes to the copy set in time to react on the epoch time scale.

Postmasters also must know when to react to copy deletion on the epoch time scale. Before it occurs, the remaining copies must wait for reconciliation reports from the destroyed copy. Afterwards, they may advance their agreement epochs without waiting for reports from the newly deleted copy. Many values will work for the effective deletion epoch as long as a single value is agreed upon. The value normally used by Taliesin postmasters is determined by a need to deal with copies isolated by prolonged network partition or by the failure of the node they are stored on. Such copies prevent the advance of the agreement and expunging epochs at all other copies. To ameliorate this problem, the normal choice for a deletion epoch is one plus the agreement epoch of the coordinating postmaster. Note that this *effective epoch* for a deletion cannot be predicted given only the voting epoch.

Creation of a new copy of a bulletin board must also be synchronized with the advancement of the agreement epochs at the other copies. The new copy can start off with any existing copy's notices and epoch bounds provided that the initial posting epoch is high enough that the new copy's reports will be expected by the other copies. The obvious choice for an effective epoch for copy creation is the voting epoch of the change. However, this doesn't work. Figure 4.5 shows a sequence of notice and epoch bounds when a second copy is created effective the voting epoch. In the final version, both copies are claiming to have an complete version of all notices posted up through epoch 2. Both copies are also still marking new notices as arriving in epoch 2. If a user reads copy A, he will be told his read epoch is 2. Subsequent reads will never include notice *N2* because its signing epoch of 2 will be interpreted as indicating it was read before.

The flawed outcome demonstrated in Figure 4.5 explains two rules postmasters must obey when creating a new copy. The effective epoch, which is also the newly created copy's initial posting epoch, must be at least one plus the voting epoch to ensure that the new copy will not assume that it can post notices until after other copies are ready to hear about them. Furthermore, the coordinating postmaster must increment its posting epoch after computing the initial epoch bounds for the new copy. That way the coordinating copy will not take any more actions during the epoch of the creation than it told the new copy about.

The critical timing information associated with a change in the copy set is saved in the form of a *history event* Each event identifies a copy, whether it was created or destroyed, and the voting and effective epochs for the change. To store history events so that they may be used in computing epochs, they are saved as a list in the *history field* of bulletin board descriptors. Postmasters are required to update the history field using the majority vote algorithm every time they create or destroy a copy of the bulletin board. However, storage for an individual event is reclaimed on a local basis without a vote when the expunging epoch is advanced past the voting epoch of the event.

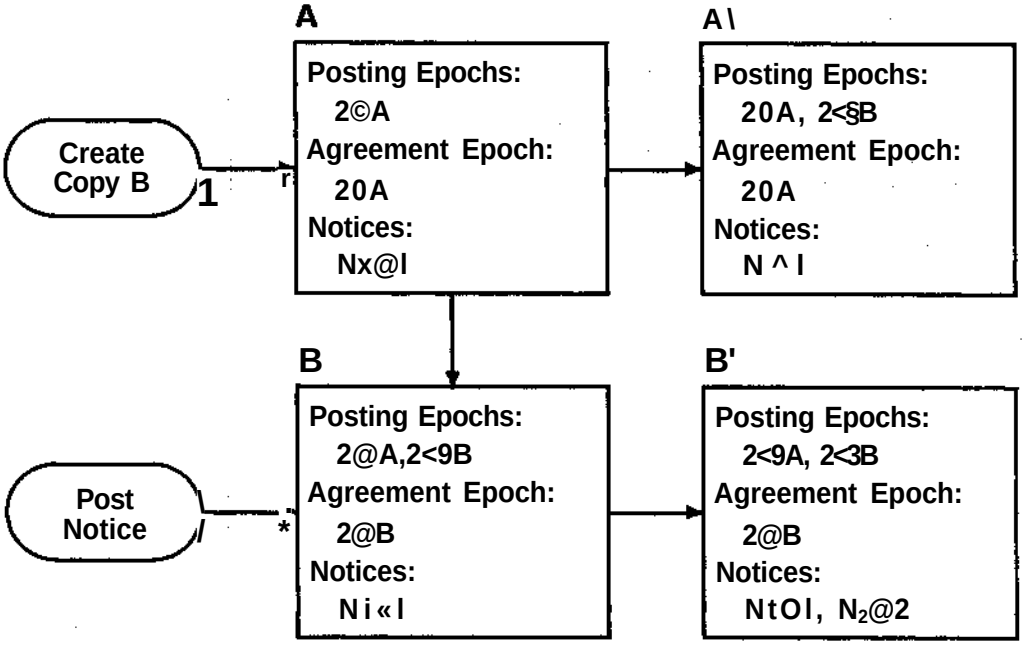

Figure 4.5: An Improper Effective Creation Epoch

Each reconciliation report includes information on changes in the copy set. Specifically, each report includes the current values of the history and copy set fields plus a list of all the proposals to update the history field that are still being voted upon. If a copy knows of a proposal to create a copy, it may not advance its agreement epoch beyond the voting epoch of the update.

4.5.1 Deletion Side-effects

In the sense that a copy is generally deleted with effective epoch less than the deleted copy's posting epoch, deletion occurs retroactively. This raises the problem of what to do with the notices accepted by the deleted copy. Consider a scenario in which originally there are three copies of a bulletin board, at nodes A, JB, and *C:*

Copy	Posting Epochs *A*	*B*	*C*	Agreement Epoch	Notices
A	6	(4)	(2)	2	iV_1@2
B	(4)	6	(5)	4	N_1 @2, N_2 @3
C	(2)	(5)	6	2	N_1 @2, iV_2 @3, N_3 @6

The system manager at A, observing the isolation of node C, might decide to delete Cs copy. So, a proposal to delete it effective epoch 3 is introduced at node *A.* Once the deletion has gone through, the situation is:

Copy	Posting Epochs *A*	*B*	*C*	Agreement Epoch	Notices
	j	(4)	—	T	N_1 @2
B	(4)	6	—	4	N_1 @2, N_2 @3

However, the isolation of node *C* was not as complete as the system manager thought. It accepted two notices after epoch 3 and even told *B* about one in a reconciliation report.

Consequently, copy *A* is claiming to have complete knowledge of all postings accepted through epoch 4, but copy *B* thinks it knows of one more notice signed epoch 3. Notice N_3 has been lost altogether.

To avoid confusion, notices that might be lost must be identified and forwarded to another copy. Similarly, to avoid partially propagating notices that were originally accepted by a deleted copy, there must be some way of detecting them. The signature of notices and deletion markers includes a signature number and the identity of the signing postmaster as well as a notice identifier and signing epoch. The identification of the postmaster makes it possible to check if the notice was accepted after a copy deletion on the epoch time scale. The signature number is to prevent re-submitted notices from being erroneously rejected as duplicates by copies whose versions were invalidated by a retroactive copy deletion. With the addition of signature numbers (shown in parentheses after the accepting copy and posting epoch for each notice), the original scenario looks like:

	Posting Epochs			Agreement	
Copy	*A*	*B*	*C*	Epoch	Notices
A	**6**	**(4)**	**(-)**	**2**	N_1 @A.2(1)
B		**6**	**(▌)**	**4**	N_1 @A.2(1), N_2 @C3.3m
C		(5)	**6**	**2**	N_1 @A.2(1), N_2 @C3.3(1), JV_3@C.6(1)

When any copy hears that a copy has been destroyed, it must look for notices that were signed by the deleted copy after the effective deletion epoch. The deleted copy clears the signing epoch and postmaster fields and forwards such notices to an existing copy. Existing copies re-sign them, incrementing the signature count. This procedure produces a correct final state. The two versions of the second signing of notice N_2 will be eliminated by the duplicate notice rejection rule, as explained in Section 4.6.

	Posting Epochs			Agreement	
Copy	*A*	*B*	*C*	Epoch	Notices
A	**6**	**(4)**	—	**4**	N_1 @A.2(1), iV_2@A.6(2), N_3 @A.6(2)
B	**(4)**	**6**	—	**4**	@A.2, iV_2@B.6(2)

It is possible under this scheme for a notice to be perpetually re-signed should it be so unfortunate as to perpetually arrive at copies that are thereafter retroactively deleted. However, it is felt that this outcome is unlikely enough that the algorithm is still acceptable.

4.5.2 Potential Race Conditions

Changes to the set of copies of a bulletin board as made by the majority vote scheme have the same effect as sequentially making the same changes in order of the write times of the committed proposals. That the history of changes to the copy set in terms of their effective epochs also makes sense as a single sequence of events can be proven.

To get sensible behavior, the computation of the history field must truly match changes to the copy set. A history event must never be created except when a copy is added to or removed from the copy set. Every time a copy is created or destroyed, a corresponding history event must be appended. Furthermore, each effective deletion epoch must never be earlier than the effective epoch of the most recent (re-)creation of the copy. If both creation and deletion occur in the same epoch, the proper interpretation is that the copy was created during the epoch, but deleted before it could accept any notices. This ensures that it is plausible that creations and deletions are interleaved on the epoch time-scale as well.

The majority vote algorithm ensures that multiple deletions must delete different incarnations of the copy. That is, in terms of time-stamp ordering, if a copy on node *A* has been deleted, another copy will be created at node *A* before a motion to delete the copy at *A* can be approved again. This same property is true when the order of events is

determined by their effective epochs.

Consider the sequence of copy deletion, re-creation, and deletion just mentioned. Let *Ediy* J5$_C$, and *E&2* denote their effective epochs and V^i, V_c, and *Vd2* denote their voting epochs. Since posting epochs are incremented every time a proposal is initiated and advanced to match any received proposal, $V_c > Vd$\. The choice of effective epochs forces $E_c > Edi$*

The second deletion must have been drawn up after the second creation passed. Thus, the proposing site knew of the re-creation and must have assigned $Ed2 \geq E_c$. A more formal proof of the plausibility of the ordering of creations and deletions on the epoch time scale is contained in Appendix A.5.4.

4.6 Delayed and Duplicated Messages

It is possible for messages to be arbitrarily delayed or duplicated by the software that delivers them. The only guarantee is of eventual arrival at least once. Furthermore, the scheme for forwarding notices to a postmaster with a copy of the destination bulletin board can cause duplicate postings. It might happen that the initial postmaster attempting to forward a posting will conclude erroneously that the forwarding failed when in fact the response indicating success was delayed. Upon detecting failure, the initial postmaster will attempt to forward the notice to another postmaster.

To minimize the number of times that notices are duplicated within the bulletin board system, notice identifiers are assigned as soon as possible: that is, when the notice reaches any postmaster, even ojie that does not support a copy of the destination bulletin boards. The timing for assigning identifiers also ensures that notices addressed to multiple destinations will bear the same identifier. Thus, user agents can filter out duplicates coming from different bulletin boards.

To avoid storing multiple copies of a notice, a postmaster always checks to make sure that a copy is not alrea4y present by comparing identifiers. If a notice is duplicated, it may well be that the versions will have different signatures. If the initial postmaster doesn't have a copy of the bulletin board, it has no epoch clock to use to assign a signing epoch. Forwarding a notice to multiple copies can result in having it assigned to different epochs. To ensure that all copies keep the same version of the notice, the rest of the signature is compared.

The version with the greater number of signatures is always kept so that notices are truly re-submitted in response to a retroactive deletion. Two notices with the same number of signature attempts are judged first by their signing epoch and then by their signing node. The adopted policy is kbep the one with the lesser epoch number. Because the lesser epoch is chosen, it is not possible to fall into a cycle of perpetually replacing one version with another bearing a later epoch, as might occur if accidentally an infinite loop occurs in the process of forwarding a notice posting.

If both have the same epoch number, then the version with the lesser node identifier will be kept. Any agreed upon ordering of the locations is acceptable. The uniformity of this rule ensures that every copy of the bulletin board will receive and keep the same version of the notice.

Deletion markers are checked so that a single version is kept for them as well. Furthermore, each postmaster checks to ensure that it does not store a notice when a deletion marker for it is already present.

Because of the unreliability of the network, the voting algorithm must also work properly should messages be greatly delayed or duplicated. If a proposal is duplicated and received before the recipient has been notified of whether to commit or abort, the recipient must respond with the same vote. If it voted 'no' or 'pass' before, the voting rules prevent a subsequent 'yes' vote so no record needs to be kept. If it voted 'yes' before, it must vote

'yes' again to avoid confusing the initiating node and to maintain the readiness to commit it promised earlier. Agents must remember their 'yes' votes until they know whether or not the proposals have passed.

Voting can complete before all the votes are in. If five copies are voting and three 'yes' votes have been received, then the voting stage is done. If a vote is delayed until after the decision is made, no harm is done — it will simply be ignored. The voting rules suffice to ensure that no meaningful delayed 'no' vote will be cast after a majority has voted 'yes'. If a vote is duplicated, no harm is done as long as the coordinating node checks to see that it has not already received a copy's vote before tallying the new vote.

Delayed or duplicated abort messages do not cause incorrect behavior. Delay just ties up resources longer. Normal processing of a duplicate abort results in a failure in the attempt to find and expunge the records of the aborted proposal. Delayed or duplicated commit messages are a danger. It is possible to overwrite the results of a later update with the results from an earlier proposal whose commitment is delayed. To avoid this, the current value of the time-stamp for the field is checked before writing.

4.7 Recovery

Recovery of most databases is more complicated than the recovery that needs to be done for Taliesin. For example, databases using locking must check for deadlock[MM79]. The majority voting algorithm aborts transactions whose claims would cause circular dependencies so there is no risk of deadlock. Most databases also treat network partition as an error condition and, when a state of partition ends, they invoke recovery algorithms[ABG84,Dav84,MPM78,PP83,Reu84]. Taliesin's replication algorithms regard network partition as simply a cause of slow communication, not a major disaster.

The reason for invoking recovery mechanisms in Taliesin is the failure of a server or corruption of data structures. Restarting a server involves restoring critical state information from some form of stable storage. How to implement truly stable storage has been discussed in a number of articles in the literature[Lam81]. In this design, the critical state consists of counters used to generate unique identifiers, the objects implemented — bulletin boards, user profiles, and name-bindings — and records relating to the majority vote algorithm. The voting algorithm requires a record of what votes have been cast and all proposed changes that have either not been committed or whose commitment has not been acknowledged by all voting copies. This information is sufficient to re-start the voting procedure and to recover the updated values that have been proposed. A scavenging process can roll state forward for those commits not yet written. Of course, each write should be done atomically. How to produce logs and roll state to the point of failure is described in basic database literature[ABG84,BG84,Dav84,Gra79a,IM79,MPM78].

Unlike the voting algorithm, the notice replication algorithm generates report messages independently. Postmasters do not have to store state remembering what reports have been received, other than by updating the contents of bulletin boards. Occasional failure to send or receive a report has no effect beyond slowing the rate of convergence. In fact, only writes of individual notices and of the collection of control fields need to be done atomically if the algorithm is coded so that the epoch bounds arrays are updated last. Should a postmaster fail in the middle of updating the notice store, it simply will not realize that its copy is more complete than its record indicates. The correctness of the algorithm only requires that a copy have a correct pessimistic belief as to its completeness.

4.8 Handling Other Error Conditions

Not all possible error conditions are as easy to handle as clean processor crashes and duplicate or delayed messages. Two very difficult problems are finding ways to cope with corrupted messages and misbehaving agents. Hopefully, standard techniques such as check sums will ensure that corrupted messages are detected. If the situation warrants, precautionary checks can be made. For instance, an agent can check to see if the read time-stamps of a proposed update are all less than the proposed write time-stamp. The format of messages can be checked in ways such as seeing if the number of copies claimed to be in the copy set is the same as the number of copies actually listed in a description of the copy set.

Recovering from undetected faulty messages and corrupted data structures is more difficult. Logging of updates and periodic check-pointing, as is done for databases, would allow the state of the system to be rolled back until an uncorrupted version is reached. A simpler approach is to periodically compare copies by issuing appropriate queries. If certain simple inconsistencies ajre found, the system can automatically recover. Unexpectedly missing notices can be copied, for instance. A local administrator will probably have to step in to manually fix the nastier forms of corruption.

The other major source of errors is faulty servers. In the present design, all postmasters trust each other to compose reconciliation reports and voting messages according to the rules. An extremely paranoid solution is to use some form of Byzantine agreement algorithm[Ben85,Dol82,LSP82,MSF83,PSL80,TPS85]. It is possible to modify an update protocol to protect against some malicious behavior by incorporating Byzantine agreement at critical points. However, such algorithms are extremely expensive in terms of delay, message traffic, and complexity. They also sidestep the problem of checking that the user agent is behaving property and do not guard against a faulty agent causing all other agents to believe a faulty state.

A more reasonable ajpproach is to implement suspicious servers. The same set of checks that look for corrupted messages will detect some forms of incorrect server behavior. Additional checks can be tnade for plausible claimed actions. For example, the postmaster at *A* can check to make sure that no peer claims to have gotten a reconciliation report from *A* dated later thaii the current epoch at *A*. If reconciliation reports are transferred in stages, using the same query operators as offered to users, postmasters likely to share with users the same view of the state of a faulty server. This does not keep a faulty server from losing or inventing notices. Another way postmasters can check for suspicious behavior is to look for copies whose epochs are advancing unusually fast or are going backward.

Another technique for guarding against faulty servers is to apply the work on verifiable signatures using encryption techniques[DH76a,DH76b,Gif82,NS78,RSA78]. For example, postmasters can guard against falsified indirect reports by having each postmaster include the full text of the report from every other copy: These indirectly transmitted reports can be encoded and signed so that the recipient can verify that the report came from the purported source and has not been tampered with. The recipient can look for incorrect behavior in the form of incorrect epoch claims, missing or invented notices, and incorrect updates to bulletin board descriptors. A less verbose report would give some protection. The epoch bounds, list of signatures, and time-stamps for descriptor fields are the most critical information. A faulty but not malicious server would very likely generate implausible values for one or more of these items.

It is practically impossible to guard against a malicious agent in the voting process. Postmasters can protect themselves by checking the plausibility of time-stamps and the values to be written. Possibly each vote message can be signed so that^ others can verify that the vote was cast as claimed. Commit messages can be signed and include purported signatures of all sites voting for the change so that any postmaster asked to commit can verify that votes were really cast to approve the change.

If these algorithms are implemented without protection against faulty agents, a variety of bad outcomes are possible. A faulty agent can give access to users that lack privileges. It can invent new values for any of the descriptor fields and, if it gives them large time-stamps, they will be adopted by all other copies. The notice replication algorithm can be fouled up by assuring other copies that they have complete knowledge of other copies' actions when in fact they do not. A faulty postmaster can delete, hide, or invent notices. It can also advance the epoch clocks at all copies, possibly causing them to wrap around.

Some forms of faulty behavior are not very destructive. A simple-minded agent that doesn't expunge its deletion markers or sends too much in reports, for example, causes no trouble. An agent that doesn't understand the re-signing procedure can only causes notices it accepted to be lost, should it be deleted. The re-signing procedure guarantees other notices will not be lost. A postmaster that doesn't properly keep a single copy of a notice can present its readers with duplicates, but correct agents will still keep only one version.

4.9 Summary of Replication

Replication of bulletin boards poses some interesting problems in terms of providing enough concurrency for the notice operations while producing correct behavior. In fact, the same challenges arise in any application in which the history of some set is of interest to readers. For example, a software distribution service has similar requirements. The elements of the set would be programs instead of notices. A copy of the set would be kept at each distribution point. Clients of the service need prior viewing records to locate just the new offerings.

The challenge of getting acceptable performance is accomplished by using different algorithms, including two different update mechanisms. To get fast response, the read protocols reference only a single copy. This works because an up-to-date, consistent snapshot is not needed. The notice replication algorithm provides concurrency where it is most needed, while the majority vote algorithm gives consistency when essential. In particular, the replication algorithms:

- Provide fast response times for read operations despite potentially long delays due to frequent partition and slow communication links.
- Never block notice postings or bulletin board reads as long as a single copy of the bulletin board is available.
- Make notices posted to a bulletin board immediately available for reading.
- Support a time-of-read for bulletin boards that is meaningful at any copy yet is compact in its storage requirements.
- Allow dynamic alterations to the set of copies associated with a bulletin board, a user profile, or a name-binding.

Using multiple algorithms as part of a single system has been done before. However, the other previously implemented system, SDD-1, used only protocols based on read/write semantics[BRGP79]. Paper designs have discussed the possibility of using different synchronization schemes based on the semantics of different abstract data types, but the actual implementation of such a combination is new.

Chapter 5

Implementation Options

The design specification states what facilities an advanced bulletin board service should provide. To a certain extent, it must also state how they are provided to ensure that different implementations of the service on separate nodes can cooperate to provide the distributed service. However, the implementor needs freedom to tailor the service for efficiency on a variety of hardware and network configurations. Ways in which the design of the earlier chapters preserves freedom will be discussed in this chapter.

5.1 Autonomy versus Cooperation

To provide replication and access to objects stored on other nodes, there must be cooperation between the agents on different nodes. A significant measure of trust is also required in this design. However, any bulletin board system spanning a network or internet will span administrative boundaries. This means that different entities will demand varying degrees of control over the resources they own. In particular, administrators will probably want to control what gets stored locally and what traffic gets routed through their domains.

On individual computers, administrators control resources by imposing quotas and charging for usage. Quotas may limit how much file storage or CPU time a person may use. In addition, users may be assigned different priorities when running jobs. Administrators will often need to have ways of similarly controlling usage by outside agencies, such as remote postmasters. However, distributed applications such as a bulletin board service break down if too many nodes refuse to provide resources.

Some local quotas can be integrated into a distributed bulletin board service very cleanly. For example, the design can be trivially extended to create a new copy automatically while honoring the storage allocation policy at the node that is to maintain the copy The protocol for servicing a create-copy request would include a check to see if the new copy's node is willing to support the copy. Only if the node is willing would the copy be created. In this manner, each administrator can enforce local policies for granting resources to a newly crekted (copy of an) object.

A local administrator may still need to limit the resources allocated to the object after its creation. However, that can interfere with the need to maintain consistency between copies. For example, 0ne copy of a bulletin board can be flooded with notices. The administrator for a node with a second copy may not want to permit the bulletin board to use the storage it needs to hold all the notices, but the copy would not be consistent if storage was denied.

If the storage or processing costs for a copy grow too large, the local administrator is free to initiate a proposal to delete the copy and so reclaim resources under his control. It is difficult under this design to restrict resources in a less draconian way. For user profiles,

this harsh control suffices. A user profile is naturally associated with a user and so is apt to be replicated and used at only those nodes associated with the user. Users must respect wishes of their local administrators anyway.

The name space is replicated across the entire bulletin board system. This could cause complaints from administrators about costs imposed by others. However, features of the name space design were included to minimize this problem. The attributes SITE and ORGANIZATION were specifically adopted so that the name space can be partitioned along administrative boundaries. Local policies can then be drawn up as desired to provide control over local names. Such policies can prevent outsiders from being granted names from an organization's name space and so masquerading as being part of the organization.

Network capacity is the resource that is really difficult to place under local control without disrupting the necessary cooperation between nodes. Adoption of a fair message routing algorithm will help. Possibly a system of cost accounting could be added to the design so that a system of paying for network usage could be set up.

5.2 Message Propagation

A variety of messages are sent between nodes. Some are generated by the majority vote and notice replication algorithms. Others are requests being forwarded to nodes having a copy of the objects to be manipulated. To minimize communication costs, these messages should be routed and, where possible, batched, for maximum efficiency. Three ways to reduce communication costs are to choose the right granularity of interaction between servers, use smart underlying network multicast facilities, and make use of network connectivity information to determine which copies are 'nearest'.

5.2,1 Server Interactions

Implementations can trade off processing and storage costs to reduce communication costs associated with reconciliation. For example, a copy can use its expunging epoch to determine which notices have been seen by every copy. One reconciliation report suitable for any other copy can be derived using those notices. At the price of additional computation, a smaller report can be prepared that is suitable for a particular destination. The composer can use the agreement epoch bound for the destination to select notices that might not have been seen at that copy. Notices that a third copy might not have seen are not included.

Another way to limit the size of reconciliation reports is to break the operation down into multiple steps. On the first pass, the postmaster issuing the report would inform the other site of its current epochs and inquire as to the current agreement epoch of the intended recipient. The size of the report is thus limited by the actual agreement epoch, not an older agreement epoch bound. In the next pass, the report issuer can offer a list of the identifiers of notices and deletion markers. The recipient would then ask for the full body of only those items not seen before. Finally, the recipient must store the post and agreement epoch bounds sent earlier. This multi-step reconciliation process trades off the cost of transmitting more data against the overhead of multiple interactions.

Storage costs can be reduced by reclaiming unnecessary storage as soon as possible. Frequent reconciliation helps keep the expunging epoch close to the posting epoch. In turn, that means that fewer deletion markers need to be retained. More frequent reports mean greater processing overhead in preparing to compose reports and more communication overhead in setting up connections. If it is costly to set up a connection between a pair of nodes, reports should be generated less often. If the data transfer rate along a link is high and fairly cheap or the delay along the link is high, it is probably worth while to send the entire report at once.

5.2.2 Message Routing

The underlying network provides routing for messages exchanges between agents on different nodes. As long as the messages are to be delivered to a single destination, many networks will do a reasonable job of routing. A lot of work has already been done on methods to produce reasonably smart routing algorithms that do not overburden particular nodes and take advantage of under-utilized communication links[JM81,JS84,Joh83,MS83]. Routing decisions are usually based on the desirability of using a particular path. That is often measured in terms of minimal hop count. Other alternatives are maximum excess bandwidth (unused capacity) and minimum delay.

For reconciliation reports and vote proposal delivery, however, a facility to efficiently deliver the same message to multiple destinations would be very useful. If the underlying network does not provide such a *multicasting* facility, it may well pay to include such a facility in the mail transport agent. A variety of techniques have been designed for broadcasting and multicasting[Agu84,Bog83,Dal77,DM78,CD85].

5.2.3 Choice of a Nearest Copy

The Taliesin design allows agents to pick to which copy they wish to direct messages in many cases. This is certainly so when one agent is trying to forward a request to some other agent that has a copy of the bulletin board, user profile, or name-binding needed. It is also true for daisy chaining majority vote proposals and circulating reconciliation reports.

Hopefully, the underlying network will support queries to determine which copy is closest. If not, a smait mail transport agent may gather its own information on the configuration of the internet to decide which node would give the fastest response. The same sort of routing algorithms usable for networks can be applied here.

5*3 Choosing the Number of Copies

The performance of the bulletin board system is affected by how far copies are from those who use them. In general, the more copies, the quicker the response and the higher the availability. On the other hand, the more copies, the longer the time needed to reach agreement among them in the reconciliation algorithms. The trade-off can be quantified. Coffman, Gelenbe, and Plateau computed the optimum number of copies for one replication algorithm[CGP81]. The tradeoff for Taliesin's replication algorithms will be estimated. To avoid the complexities of analyzing the effects of link failures in random network configurations, it will be assumed that the only failure mode is isolation of nodes. Node isolation is assumed not to change network connectivity. These assumptions are optimistic.

Let the chance of any node being isolated at any time be *p*. Isolation of nodes is assumed to occur independently. Let c be the number of copies of the object. Then the chance that at least one copy is available when a read request is issued is $1 - p^c$. Figure 5.1 shows the increase in availability as a function of the number of copies for four values of p.

Assume that the bulletin board, system has learned where the copies of an object are. If a copy is available, it can be read in the time to deliver two messages by broadcasting the request to all copies — one message for the request and one for the reply. If successive copies are tried until an available one is found, then $E(M_R)$, the average number of messages needed to read or conclude no copy is available, can be computed as follows.

One message is sent to each copy in turn until one responds or until all copies have been tried. The probability that the fc-th copy is available but not any of the earlier ones i_s $(i - p)p^{k-1}$ - *k* messages are sent by the time the fc-th copy is tried. If none of the copies

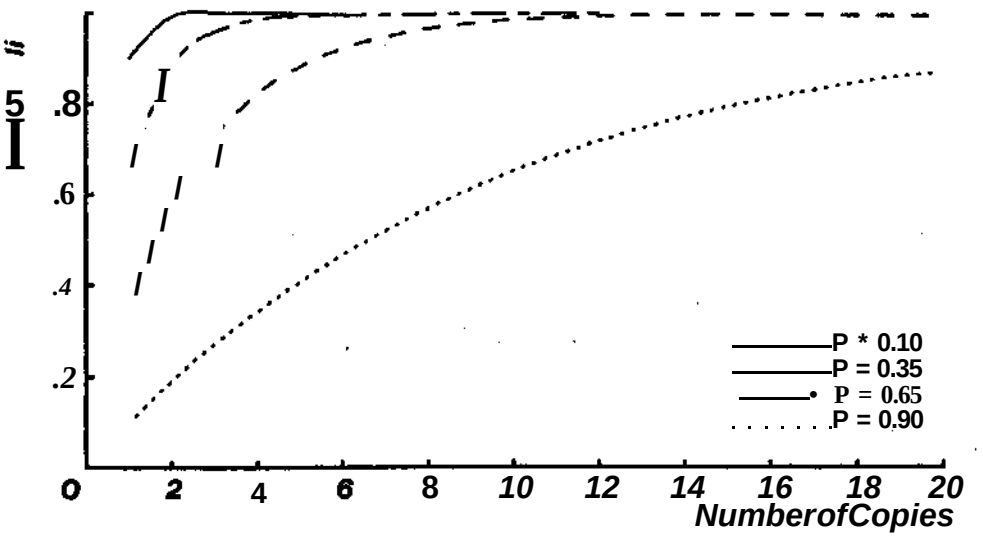

Figure 5.1: Availability and the Number of Copies

are available, one message per copy is still sent. So, the number of messages sent is the mean number of request messages plus, if a copy is available, one more message for the value read.

$$E(M_R) = \sum_{k=1}^{c} k(1-p)p^{k-1} + (1-p^c) + cp^c$$

This formula can be reduced to closed form by plugging in an easily verified value for the sum.

$$\sum_{k=1}^{c} kp^{k-1} = \frac{1-(c+1)p^c + cp^{c+1}}{(i-p)^2}$$

$$E(M_R) = 1 - p^c + \frac{1-p^c}{1-p}$$

Figure 5.2 shows how JB(MH), the average number of messages to perform a read, changes as a function of p and c.

The time needed to advance the agreement epoch and to complete the majority voting process will be estimated in terms of *rounds.* Each round, all nodes that are up receive unseen messages sent during previous rounds, do any necessary computations, and respond. A message may be either a normal message directed to one node or a multicast message.

The problem of determining how long either replication algorithm takes can be reduced to the problem of determining how many rounds pass until enough nodes come up to complete the process. The probability in any one round that a copy is totally isolated and so cannot send is p. The probability that a copy takes more than k rounds to attempt to get its reconciliation report out is p^k. Let $X\{$ be the random variable whose value is the number of rounds taken by the i-th copy to come out of isolation. The $X\{$ are

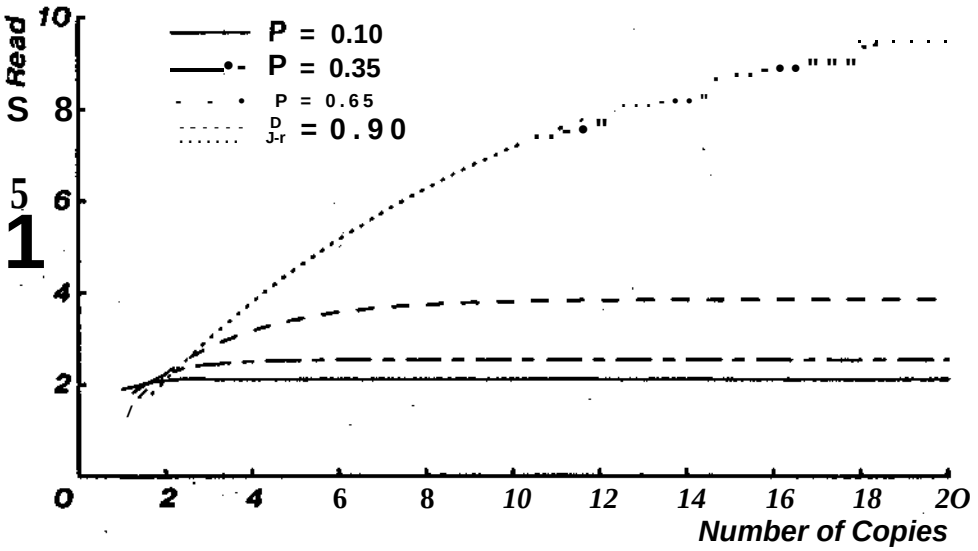

Figure 5.2: Number of Messages to Read

independent, identically distributed random variables with a geometric distribution: mean $(1-p)^{-1}$ [DeG75]. The average number of rounds until n copies have come up for at least one round is the mean of the n-th largest of the X^*: that is, the mean of the n-th order statistic, $X_{(n)}$.

The means of the order statistics is easier to estimate by looking at the problem from a different slant. Pick any fixed ordering of the copies. Scan the copies in that order for however many rounds it takes to find, the first copy that is up. The ceiling of the number of scans divided by the number of copies yields the value of $X_{(1)}$. Continue the scan until another copy comes out of isolation. The additional number of scans divided by the number of remaining copies is approximately the difference between the $X_{(i)}$ and $X_{(2)}$. In general, the i-th order statistic can be approximated as a linear combination of independent, geometrically distributed random variables.

$$\begin{aligned} Y_i \quad & \text{iid} \quad f(y) = (1-p)p^{y-1}, \; y = 1,2,3,\ldots \\ X_{(i)} \quad & \approx \quad \sum_{k=1}^{k=i} \frac{Y_k}{c+1-k} \\ E(X_{(i)}) \quad & \approx \quad (1-p)^{-1}(H_c - H_{c-i}) \end{aligned}$$

H_n denotes the n-th harmonic number[Knu73]. A known approximation of the values of harmonic numbers is used below. 7 is Euler's constant: approximately 0.577.

$$\begin{aligned} H_n \quad &= \quad \ln n + \gamma + o(\frac{1}{n}) \\ E(X_{(i)}) \quad &\approx \quad \begin{cases} (1-p)^{-1}(\ln c - \ln(c-i)) & 1 \le i < c \\ (1-p)^{-1}(\ln c + \gamma) & i = c \end{cases} \end{aligned}$$

Consider the situation in which all copies have agreement epoch E but none have sent out reconciliation reports covering any epoch after E. Starting with round one, all copies

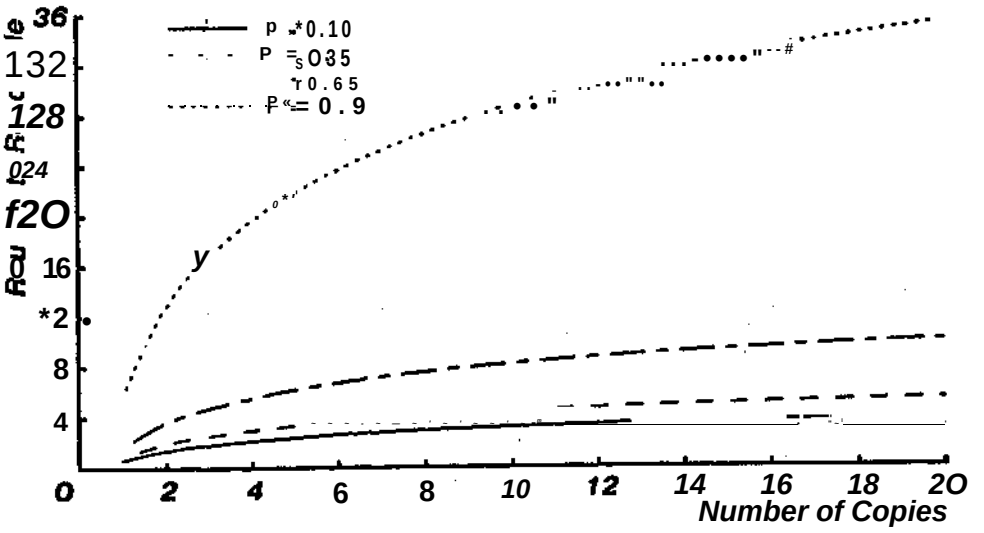

Figure 5.3: Reconciliation Time and the Number of Copies

whose nodes are up send out reconciliation reports. No copy can advance its agreement epoch until every copy has been up at least one round. So, the number of rounds needed to increment the agreement epoch is at least $X(_{c}y_{1}$ The mean number of rounds needed to achieve reconciliation, then, is at least $(1 - p)^{-1}(\ln c + 7)$. Reconciliation time grows at legist as fast as the log of the number of copies. Figure 5.3 shows the increase for selected values of p.

The number of rounds needed to commit using the voting process can be similarly approximated. Assume that all copies have exactly one vote. At least half the copies must become available for an update to be committed. During round one, the coordinator sends out the vote proposal. Each voter sends back its vote on the first round that it is available and capable of casting a vote. An immediate reply may not be possible due to conflicting transactions, so $-X''(c/2)$ is a lower bound on how many rounds are spent doing the voting.

$$E(X_{(\frac{c}{2})}) \approx (1-p)^{-1}(\ln c - \ln(\frac{c}{2}))$$

$$E(X(\%)) \approx (1-p)^{-1}(\ln 2)$$

The number of rounds taken for voting is not dependent on the number of copies, except insofar as that with more copies there may be more conflicting transactions. The time needed to ensure that all copies get a copy of the commit/abort decision is the same as the lower bound on reconciliation. Of course, this assumes no conflicting transactions.

The preceding analysis assumes a fully connected network. In practice, the cost of implementing a fully connected network in hardware is prohibitive. Full connection is simulated through a store-and-forward mechanism. Consequently, the delay in sending a message between an arbitrary pair of nodes will increase as the number of nodes increases because more hops will be needed. This affects the analysis in two ways. First of all, the time spent in a round will increase. A round is roughly the time needed to carry out a send/receive/reply sequence. If network routing is based on a spanning tree, the average number of hops between pairs of nodes will probably grow roughly as the log of the number of nodes. The duration of a round will have to grow at roughly the same rate.

The other effect is that the effective probability that two nodes are in communication during a round may change. If the routing algorithm quickly finds alternative routes to bypass failed links, the odds of a route being found during a round will increase. The probability will decrease if the routing is fixed over a significant interval because more than one link must be traversed.

The number of copies, then, affects availability and the time needed to advance the computed epochs of the notice replication algorithm. More copies buys higher availability and better response, although beyond the first few, each additional copy provides minimal improvement in availability. The delay in advancing epochs, however, increases as the log of the number of copies. Interestingly enough, the time needed to complete a vote is only indirectly influenced by the number of copies. It only increases if the copies are more widely scattered or the concurrent submission of updates causes greater delays in deciding a vote. By plugging in typical delays and failure rates, the quantitative formulas presented in this section can be used to decide how many copies to create.

5.4 Tailoring to a Network Configuration

The replication algorithms were specifically designed to allow implementations to pick policies that will improve performance. The options discussed earlier, such as picking the number of copies and a granularity of interaction between agents do not exhaust the possibilities. Response can be improved by judicious choice of copy placement. The time taken to complete a vote can be influenced by the way votes are assigned. Perhaps most interestingly, the system can dynamically change these decisions to adapt to changes in the network configuration or user demands.

5.4.1 Copy Placement

A major policy decision is to decide where the copies of each bulletin board, user profile, and name-binding replication group are to be located. This placement determines both the availability and the access time for all operations. The "closer" the copies are to where they are used, the faster the response to reads and the greater the availability in general. Note that in this context, a copy is *close* if it is kept on the local node or a remote node (preferably with idle processing power) connected by a high speed, reliable communication link. On the other hand, the more spread out the copies on many network configurations, the more hops needed to propagate messages for replication. Convergence to a consistent state will take more time.

Given storage costs, node and communication link accessibility patterns, a pattern of update traffic, and an update routing policy, the cost of a storage placement can be evaluated. Quite a bit of work has been done on the related issue of file storage placement to minimize costs. Much of the work has focused on choosing an initial placement without consideration of increasing delays based on loading of the processor storing the copies. For example, a paper by Casey[Cas72] examines placement and its resemblance to the classic problem of placing warehouses.

Many other approaches are based on deriving formulas for the costs and formulas describing the desired performance requirements. These formulas and constraints are then massaged into the form that can be handled by linear 0-1 programming methods. One formulation given constraints on the minimum number of desired copies and fixed communication and storage capacities was formulated by Chu[Chu79]. Another formulation that allowed for choice of communication links as well was developed by Mahmoud and Riordon[MR79]. Morgan and Levin explore copy placement considering that there may a difference in the costs of accesses, depending on whether the access constitutes a read or write[ML79].

Others have considered processor load and other queuing delays. For a good summary of the less recent work on copy placement, see a paper by Dowdy and Foster[DF82]. More recently, Bay and Thomasian developed heuristics for incremental copy placement to give good response in the face of queuing delays [BT85].

5.4.2 Choice of Vote Assignment

Not all portions of a network and not all nodes are equally reliable. There may be dramatic speed differences as well. The majority vote algorithm enables implementations to take these differences into account. Copies can be assigned arbitrary weightings in terms of their votes[Gif81]. In every implementation, votes can be assigned so that more reliable or more quickly responding copies carry more weight. A consensus of only two reliable copies, for example, might suffice, while allowing the existence of another twenty less reliable copies.

5.4.3 Automatic Load Adjustments

Most of the work related to copy placement is oriented toward pre-computation of copy placements. However, even an optimal placement will soon cease to be so as the network configuration changes and as the readership associated with individual bulletin boards evolves. Since the replication algorithms allow the set of copies to change, it would be interesting to try an adaptive scheme that starts with a single copy and initiates changes to the placement based on local computations using a record of traffic. Traffic statistics should include type of access, source of the request, and possibly the routing of the request. No literature was found covering such an idea for replicated files. The problem does have some similarities with the problem of paging, but the delays and sizes of bulletin boards most closely resemble files. Several researchers have examined the problem of migrating files between disk storage and longer term storage media such as tape[LRB82,Str77]. The research on distributed file systems considers only adaptive placement of single copies of files[PT83,Por82].

Another way in which adaptive measures might be useful is to automate the handling of inaccessible copies. The notice replication algorithm requires that every copy be able to send reconciliation reports to every other copy, although the routing may deliver them indirectly. If a copy is inaccessible for an extended period of time, the storage and communication costs rise. The copy set can be changed to get rid of offending copies. To reduce the burden on the maintainers of the bulletin board system, it would be handy if the bulletin board system itself were to detect and delete inaccessible copies. Selection of policies for deciding when to delete and when to readmit copies is an unsolved problem.

5.5 Storage Reclamation

The bulletin board system uses information that must be kept around until it is no longer wanted or needed. To keep delayed or duplicated messages from doing harm, the majority voting algorithm must record the state of partially processed update proposals. This state information must be expunged eventually, but it can be done at varying times. The routines implementing the voting can be written to make various assumptions if the record for a proposal is not found. Depending upon the decision, the routines can delete different records before a vote has been committed or aborted at all sites. For example, suppose the presumption is always made that if no record for a proposal can be found, the proposal must have been aborted. This rule alk>ws records for aborted transactions to be deleted immediately. A more complete discussion of when it is safe to free up records related to the voting process can be found in a paper by Mohan and Lindsay[ML85].

Over time, it becomes unnecessary to store other information as well. For example, notices become candidates for deletion when they expire or meet the automatic deletion conditions of a bulletin board. The history event and deletion marker records stored in bulletin boards also expire. The design does not force any particular timing for the evaluation of what information can be expunged. The implementor is free to choose between lazy evaluation methods and more immediate checks. For example, lazy evaluation of notice deletion might be encoded as part of the reading or reconciliation operations. Lazy evaluation leaves storage unclaimed for longer periods of time, but requires less processing power. Periodic evaluation can be done by clean-up daemons instead. The choice should be made on the basis of the relative costs of storage and processing.

5*6 Review of Choices

Taliesin is intended to be a reasonable design for a bulletin board service distributed over a large internet. Accordingly, it mustfc>eadaptable to a wide variety of network configurations. This design goal is met, largely because the replication algorithms are guided by policies. The choice of a particular policy — for example, for choosing copy votes — tailors the implementation to a particular situation. Other such policies include choice of the number arid placement of copies of an object.

The real key to Taljesin's adaptability is that the decision as to when and where to send messages is a matter of policy, not a wired-in part of the replication algorithms. For instance, on circuit-switched networks with a high cost of setting up connections, reconciliation reports can be batched. Over satellite links with their extremely high data bandwidth and high delay, it pays to send entire reconciliation reports at once. The summary/query approach wins, however, over links such as phone lines in which data transmission is slow or expensive, but it is easy and cheap to change send/receiver roles on the link. All these possibilities permit the design to be efficiently implemented on a wide variety of network configurations.

Chapter 6

A Prototype Implementation

To test the feasibility of the proposed design, a prototype of Taliesin was implemented. The prototype runs on Sun workstations under an experimental version of the V-System[1]. The V-System is a distributed operating system designed for potentially diskless workstations connected by local networks. Under the V-System, workstations may optionally act as components of a system rather than as purely independent units. Accordingly, the programming environment has many features that support distributed applications. The kernel of the V-System is message-based and provides applications with network-transparent inter-process communication[Ber]. Using the IPC facility, any program can access files maintained by any server obeying the V I/O Protocol. Furthermore, users can remotely execute programs on other workstations or on mainframes that participate by running V servers to translate standard V-System requests into operations under other operating systems. Because of these facilities, the V-System is an excellent testbed for developing distributed applications.

The organization of the prototype implementation will be presented first in this chapter. Then some of the experiences with it will be related. The original plan was to observe user behavior to determine the palatability of Taliesin. Unfortunately, the combination of several factors made it infeasible to gather such data. The performance, however, was measured. The timing measurements in this chapter are made in terms of elapsed time, to simulate delays that would be seen by users.

6.1 Physical Agents

The logical design calls for a collection of agents: user agents, authentication agents, name servers, postmasters, and mail transport agents. However, this division is purely from a abstract point of view. Implementors are free to further subdivide or combine functions. The choice made can produce some interesting interactions between the different agents.

The organization of this implementation is geared toward the needs of a prototype, as opposed to a production, system. For example, the early versions were expected to have many bugs. The anticipation of bugs prompted a decision to monitor the message traffic between agents. Similarly, to test the replication code, it is preferable to simulate network partition rather than to cause actual hardware partitioning. A new agent, the Nexus, was designed to provide these and other helpful facilities.

The prototype implements user agents as distinct entities. This gives users the flexibility of picking user agents that satisfy their current needs. It also allows user agents to come and go as users start and end sessions. Finally, it enables workstations to run only the user agent, instead of a copy of the whole service.

[x]The V-System is completely unrelated to UNIX System V

Distinct agents also provide general purpose naming facilities. A number of reasons led to the adoption of a general purpose naming service over a special purpose one. From the start, it was clear that Taliesin was going to have to handle more than just bulletin board names. Until very recently, users had accounts on multiple machines rather than a network-wide identity in the prototype's environment. Multiple identities for a single user would create all sorts of headaches, so the prototype needed to provide its own naming of users.

The names of other named objects, such as user profiles, might not share the syntax of bulletin board names. In fact, the very syntax of bulletin board names was initially undecided. The format of the information the names should bind to was even more fuzzy. All of these factors made it desirable to defer the commitment to a particular naming scheme until a later stage in the design. A general purpose naming service promised the flexibility to experiment whereas a specialized service would require more re-writting each time a revision was made.

Differences in replication policies were another, more theoretical motivation. Replication of names needs to be done on a different basis than the replication of bulletin boards. For example, consider a bulletin board whose readers are thinly scattered over the entire country. They might be willing to wait once for a remote handling of a request to read the bulletin board, but they might be too impatient to double the delay by also having the name-binding done remotely. Given that a bulletin board is apt to consume more storage than the name-binding, storage costs would be minimized by having a central or regional copy of the bulletin board while replicating the name-binding at each of the reader's nodes. More generally, the entire name space must be accessible to each Taliesin agent, but because complete replication at every server would impose too great a storage cost, the name space needs to be partitioned into smaller replication groups. The more wide-spread demand for name resolution suggests that separate policies for placing name-binding and bulletin board copies might be appropriate. Since the data structures of name-binding are at <1> unlike those of bulletin boards, it seemed reasonable to employ a separate name service. Use of a general purpose name service seemed more reasonable given the separation.

The last motivation was simply curiousity. The entire issue of whether it is possible to design a truly general purpose name service that would handle the names of arbitrary new types of objects without modifying the name service was personally interesting. The naming agents were designed with this goal in mind.

In the prototype implementation, postmasters and mail transport agents are merged into a single agent. The reason why mail transport agents are considered logically distinct from postmasters is because they hide the messy details of transferring data across a network. The V-System provides network transparent IPC, rendering it as simple to send a request directly over the network as it is to send it to a local mail transport agent.

6.2 The Organization and Function of Agents

The internal organization of the prototype's agents is heavily influenced by the execution environment provided by the V-System. Creation of a new address space is a relatively heavyweight operation. However, groups of processes may share an address space and creation of processes within an address space is a relatively fast operation. Accordingly, each agent is organized as a collection of processes sharing one address space. This organization allows easy sharing of data between processes. It also simplifies the implementation of concurrency within agents. The operating system provides multiple threads of control through the separate scheduling of processes.

The separation of the address spaces of different agents provides a firewall isolating the damage done by a faulty agent. However, it also means that a lot of the code for basic functions is duplicated. Due to physical memory limitations, the version of the prototype

released for general use places several agents within a single address space so that the code can be shared.

6.2.1 The Nexus

The Nexus is implemented as a single process. It intercepts message traffic between agents. If so instructed, it will log the messages it sees. The Nexus defines Taliesin pseudo-hosts and enforces simulated partitioning. Multiple pseudo-hosts can run on one Sun workstation. The existence of pseudo-hosts forces all messages sent between Taliesin agents to be prefixed by both V-System process identifiers and the Taliesin pseudo-host names of the source and destination agents.

The Nexus also provides a very primitive name-binding service. It records the process identifiers of services on the simulated hosts and answers queries concerning them. Agents need only locate the Nexus via V-System primitives in order to locate Taliesin services.

6.2.2 User Agents

Two user agents were implemented. The first satisfies the need to have some means of testing Taliesin. It recognizes all the commands to all the agents: postmaster, name service, and Nexus. This *maintainer's agent* never uses default values in filling out requests. Because it requires the user to enter every option every time, it can test all the variations of all the commands. However, it is very tedious to use.

The maintainer's agent was designed to be quick to implement so that it could be used during the development of the prototype. For this reason, it offers only minimal line-editing facilities and keeps minimal state between commands. Again, these features lead to a high degree of user-hostility. Its one saving grace is that it does have help facilities to inform the user of his possible responses.

The other user agent was designed for the general user community. It makes use of the window and menu facilities available under V[LN84,Now85]. Commands are selected with a mouse from menus. The mouse is also used to select the groups of notices or bulletin boards that the commands are to be applied to.

This *friendly user agent* locates a postmaster by querying the Nexus. It prompts for an account and password then starts a session with that postmaster. Next, it reads the user's profile and the envelopes of the new notices in all the interesting bulletin boards. It presents one window holding the main menu and a second listing the interesting bulletin boards. Bulletin boards with new notices are flagged. The user may select bulletin boards for further inspection. In response, the user agent creates a window holding summaries of the new notices. The user may select notices to read using the mouse. Each notice is read in full when selected and is displayed in a notice-display window. When the user indicates he wishes to exit, the friendly user agent composes and sends requests to update the user profile.

6.2.3 Name Service

In the implementation of Taliesin, all name-bindings are managed by a universal directory service or UDS[LEH85]. It stores name-bindings for bulletin boards, user profiles, distribution lists, and users. The name-binding for a user maps to a user identifier generated by the UDS and a list of identifiers for the groups to which the user belongs. For the other object types, each name-binding maps to an identifier generated by the UDS and a list of Taliesin hosts supporting a copy of the object.

Each UDS agent consists of several processes. The main process registers itself with the Nexus and accepts requests for service. The requests are forwarded to auxiliary processes

that are created as needed. Auxiliary processes are also created to handle each incoming voting message. Since name-bindings for bulletin board descriptors, user profiles, and distribution lists are only changed when a copy is created or destroyed, Taliesin only causes the spawning of voting processes when a copy set changes. The UDS has a resource arbitration process and a timer process. The timer process periodically re-transmits messages relating to the voting process. This ensures that voting messages get through eventually.

The UDS name space is hierarchical. UDS names are a concatenation of path components separated by slashes. Absolute names are prefixed by a percent sign. The name space is divided into pieces called replication groups that are replicated in full at every name server authorized to maintain the group. Each name in a replication group starts with a common prefix. Names are mapped to catalog entries designed to provide exactly enough information to locate the server managing the object named and to identify the object to its manager.

The UDS has three main data structures. It has a collection of files, one per directory, holding catalog entries. To keep track of where each replication group is located, the UDS also has a table with one entry per replication group. Actually, this table only has entries for replication groups that are either maintained locally or whose root directory's catalog entry is kept locally. The other major data structure is a table storing additional information on servers: how to contact them and what object manipulation protocols they understand. The server table is implemented as a separate data structure because of an expectation that it will be read and modified more often than the corresponding directory entries. In particular, it will be read every time a client wants to learn how to contact the manager of an object whose catalog entry it just looked up. Of course, each UDS server has other data structures, including a boot file, an error log file, and a log of updates being voted upon.

6.2.4 Postmasters

The main postmaster process registers itself with the Nexus and accepts requests. It, however, does not service requests. Instead, it creates helper processes, or *clerks* to do so. A clerk rejects any initial request that is neither a forwarded request from another postmaster nor a request to start a session for a user. If the request is to start a session, the clerk will await further requests from the same user agent process. The ability to send messages purportedly from the user agent process is considered proof of the client's identity in subsequent requests. When the session is ended or the death of the user agent detected, the clerk releases its resources and commits suicide.

Each postmaster agent includes a number of other processes. For example, there is a resource allocation process that synchronizes local access to shared objects, primarily files, and generates unique identifiers as requested. In addition, one daemon process refreshes the name caches kept by postmasters to speed up the binding of names while other daemons periodically wake up and generate replication messages to ensure that the algorithms make progress. Additional helper processes are created as needed to process messages related to either of the replication algorithms.

The interaction between the postmasters and UDS agents is of particular interest. Postmasters use the hierarchical name space offered by the UDS to support an attribute-oriented, non-positional bulletin board name space. In order to use the name service, postmasters must transform Taliesin names into a canonical, hierarchical form.

The rules for translating into a hierarchical UDS name mask attribute ordering distinctions and allow an inverse mapping into a Taliesin name. The canonical ordering of (attribute, value) pairs is attained by first sorting the pairs on attribute then on value. The attribute ordering is predefined to be SITE, then ORGANIZATION, and finally TOPIC. This order wajs chosen to break up the name space into pieces with a clear sense of locality: The values for each attribute are strings and are sorted alphabetically.

Once the (attribute, value) pairs have been ordered, a hierarchical string name is constructed. The basic form of the string is **%$G/$P/$T**. After each attribute's tag, the value strings are inserted in sorted order. The string **/.** is added as a prefix to each value. Consider the attribute-oriented name:

SITE:	USA/Iowa
ORGANIZATION:	(none given)
TOPIC:	taxes, alcohol

%$G/.USA/Iowa/$P/$T/.alcohol/.taxes is the equivalent UDS name. The tags **$G**, **$P**, and **$T** were chosen as abreviations for the types of keywords: *geographical*, *political*, or *topical*. Because of their use as tags, the strings **$G**, **$P**, and **$T** cannot be used as attribute values. Furthermore, value strings must not start with a period.

This canonical ordering adapts well to a form of wildcard search. Sub-topics within a particular attribute add keywords. Their corresponding canonical UDS names can be found by searching for path names to insert after the attribute tag and after each of the attribute values. For example, **%$G/$P/$T/.for-sale/.housing** is a sub-topic of **%$G/$P/$T/.housing**. The new path component **for-sale** is added after the attribute tag **$T**.

The initial dot before a value marks the beginning of distinct keywords. For example, **%$G/.USA/.Iowa/$P/$T/.alcohol/.taxes** is the canonical form of the name below. Note the addition of a period before the keyword **Iowa**.

SITE:	USA, Iowa
ORGANIZATION:	(none given)
TOPIC:	taxes, alcohol

%$G/.Iowa/$P/$T/.alcohol/.taxes would be considered in a search for super-topics of the above name. It would not be considered in a search based on the name **%$G/.USA/Iowa/$P/$T/.alcohol/.taxes**. The absence of a period implies that the keyword **Iowa** is only meaningful as a qualifier to the keyword **USA** in the latter name.

6.3 Experiences and Performance

The prototype was intended to demonstrate that the design of Taliesin is feasible to implement. Because it was not coded with efficiency in mind, the response times in this section are only lower bounds on how fast the system can run. A production version of the system using more sophisticated techniques such as indices into files would have far better response times.

All the performance measurements are made in terms of elapsed time, to simulate the delays seen by a human being using Taliesin. The underlying operating system, the V-System, provides fast message transfer but, at present, slow file access. Table 6.1 shows the delays incurred in a Taliesin message exchange. A message exchange consists of a client sending a request, a server receiving the request and then replying.

A Taliesin message exchange is more than a V-System message exchange. A Taliesin message exchange includes two V-System messages exchanges, two allocations of buffers from the heap, and two data copies, possibly across the ethernet connecting the Sun workstations. The message passing routines are further slowed by their paranoid coding. For example, they check to determine whether a message's format follows the protocol

Sender/Receiver Locations	Sun-2	Sun-3
Same workstation	70094.0U40	
Different workstations	.0280	.0120

Table 6.1: Message Passing Delays (Seconds)

definitions and whether the sender/receiver identifiers returned by the V-System agree with redundant identification in the Taliesin message body. However, despite all this cumbersome mechanism, message passing is still fast enough that it is not a cause for concern in the final performance figures.

The figures shown are for nearly idle workstations. The results of the IPC timing test depend significantly upon how heavily a workstation's CPU is being used by other programs. Delays about 25% greater were seen for workstations under moderate usage. Minor variations in CPU utilization swamped the differences in delays due to differences in sizes of the messages.

Table 6.2 shows the delays incurred for a pattern of file access that is typical of the agents in the prototype. Each trial consists of opening a file, reading the stated number of bytes of data, rewinding the file, writing the same amount of data, and then closing the file. Only times for those file servers that could handle the demands of the later timing tests are included in Table 6.2. The delays were sampled at various times of the day when no Taliesin timing tests were being run. The times are slow, largely because the file service was provided by a guest level implementation of V using underlying UNIX facilities. The added overhead of going through UNIX increases the access times for files. The native V-System file servers are faster, but were not wholly reliable at the time these tests were run.

File Server	0 KByte	1 KByte	2 KByte	3 KByte	4 KByte
VAX 11/750	.12 - .20	.25 - .44	.63 - .88	.80 - 1.25	1.19 - 2.00
VAX 11/780	.062 - .22	.16 - .33	.35 - 1.16	.59 - .92	.79 - 1.17

Table 6.2: File 10 Delays (Seconds)

In the initial implementation, agents explicitly provided their own multiple threads of control. This meant state information had to be frequently saved and recovered from a file. The long delays due to file access rendered that approach hopelessly slow. The prototype was re-written to create separate clerk processes, which keep their state on their stacks. This change simplified the implementation and improved the performance dramatically.

The dominant cause of delay in almost every service offered, by any of the agents is file access. The type and location of workstation running the various agents produce little difference in the performance measures. In practice, the timing runs for replication were run on Sun-2's, with one agent per workstation. The Nexus and UDS were run on a Sun-3. The other timing tests ran all the agents on a single Sun-3.

6.3.1 The Nexus

The decision to introduce the Nexus turned out to be a good idea. Being able to monitor all inter-agent messages greatly eased the process of debugging. It also helped minimize the impact of changes made to the V-System. In particular, the naming facilities of the Nexus proved to be useful in isolating the effect of experimentation with methods of locating processes.

The fact that the Nexus acts as a registry for the other services enables multiple servers

to be tested on a single Sun. Given the difficulty of finding multiple idle Suns, particularly ones near one another, that feature was almost essential. The extra level of indirection in defining host names means that Taliesin host names are unrelated to V-System workstation names so agents from different hosts can be placed on one workstation.

The Nexus does introduce some overhead into the message passing routines. The overhead is the difference between the cost of a message exchange through the Nexus and one bypassing it. Comparison of the times given in Tables 6.1 and 6.3 shows that sending messages through the Nexus roughly doubles the delay. However, the cost per message is still a small fraction of a second. Since most requests require only a single message exchange, Nexus overhead is dwarfed by the delays imposed by file access.

Locations of Agents	Sun-2	Sun-3
Common workstation	.017	.0075
Client remote	.045	.019
Different workstations	.050	

Table 6.3: Message Passing Delays Through the Nexus (seconds)

6.3.2 UDS Experiences and Performance

The design of the UDS was considerably improved because of its use to support a bulletin board system. The demands of Taliesin pointed out hidden dependencies on the unwanted assumption that all objects behave like files. In addition, the demands of Taliesin prompted special features to be added to the UDS that are useful, but not essential, for a general purpose name service. These features are user groups and a wildcarding facility that accommodates attribute-oriented name spaces.

The wildcarding facility is of particular interest because it is used to locate the bulletin boards covering the sub-topics of a given topic. To perform such a search, a postmaster must be able to scan directories to locate new keywords differentiating the sub-topics from their parent topic. In the first version of Taliesin, this need was met by using a wildcard token that matches any string within a single directory. Figure 6.1 shows some times to create, read, and destroy catalog entries in the prototype. The quoted times for reading are actually those for scanning a small directory, much as a postmaster might do in locating sub-topics.

Early versions of the postmasters had to scan at least one directory per keyword each time they tried to locate the sub-topics of a bulletin board. The cumulative cost grows rapidly with the number of keywords in the name because the overhead of UDS invocation must be paid for every directory. It turned out to be faster to define another wildcard token, **, that matches any sequence of path components in zero or more directories.

The ** search enables present postmasters to ask for all the sub-topics of a bulletin board with UDS name **%$G/$P/$T/.taxes** by asking for all bulletin board names matching the pattern %$G/$P/$T/**/*taxes/**- Any keyword further qualifying the topic must appear after the token $T, either before or after the path component for the keyword **taxes,** so it will be returned as part of the search.

However, the procedure first used for doing wildcard expansion was slow. Given the pattern %**/B/C and a name %A/B/C, the obvious definition of ** used by an early version of the UDS required it to match ** against **A/B/C** and A/B as well as just A. Because the expansion was so slow as to be annoying, the interpretation of the ** token was changed so that it matches zero or more path components, but treats the path component after ** as a sentinel. The present UDS will not try to match the sentinel to

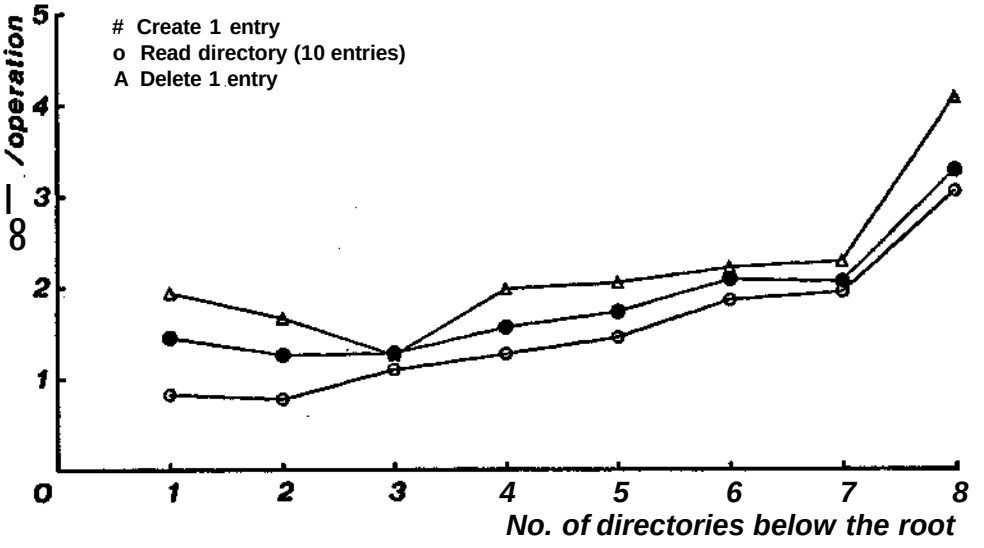

Figure 6.1: Elapsed Times for UDS Operations

**. In the situation just mentioned, the UDS matches ** against A but notes in the next step that the sentinel B matches the next path name. The UDS does not try either of the two losing searches. Eliminating many losing searches reduces the search time. The larger the name space, the greater the savings tend to be.

The cost of using the ** wildcard facility was measured by sampling the elapsed time seen by a client for searches using varying numbers of keywords, each separated by **. Specifically, %** is the pattern with no keywords while %**/A/**/B/** is a pattern with two keywords. The name space was initialized to contain 2^N names, where N is the number of paths given in Figures 6.2 and 6.3. In addition, the root directory always contains a number of standard entries.

Figure 6.2 plots some elapsed times for such wildcard searches. Figure 6.3 shows the same data but with the costs divided by the number of names in the name-space. Note that the times are considerably less when no sentinel is given. The difference is an artifact of the way the UDS prototype is coded. First a pass is made over a directory attempting to continue the parse by matching a path against **. Then, if there is a second path name, a second pass is made over the directory looking for it. The difference is not quite a factor of two because some overhead is accrued on a per-invocation basis and because matching the sentinel eliminates losing searches. Because the second search is necessary to check for the case in which ** matches no path components, the doubling of the time spent parsing would occur even if aU matches where sought: that is, if the next path component were not a sentinel.

This form of wildcarding has enough usefulness that it seems justified as a general purpose facility. For example, it would be handy for finding a directory when the name of the directory can easily be guessed but its location in the hierarchy is unknown. Treating the first path name after the ** token as a sentinel reduces the number of potential matches. However, the slight loss of generality seems minor given the improved performance.

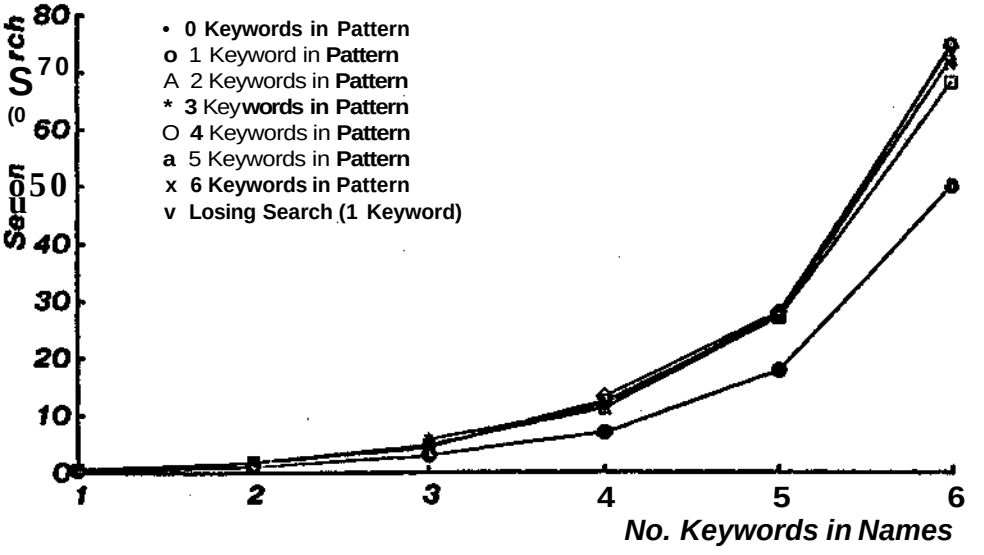

Figure 6.2: Searches using Multi-path Wildcarding

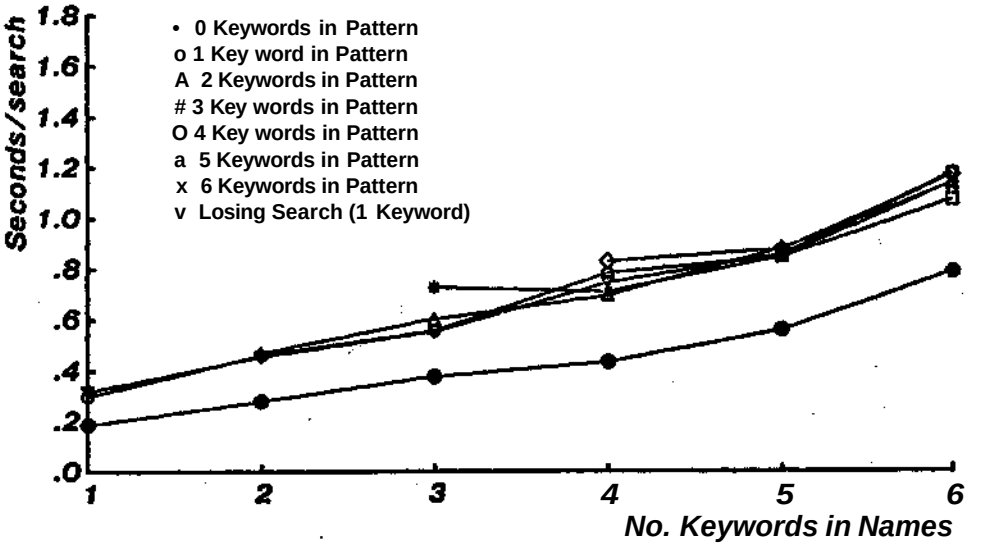

Figure 6.3: Multi-path Wildcarding Scaled by the Number of Names

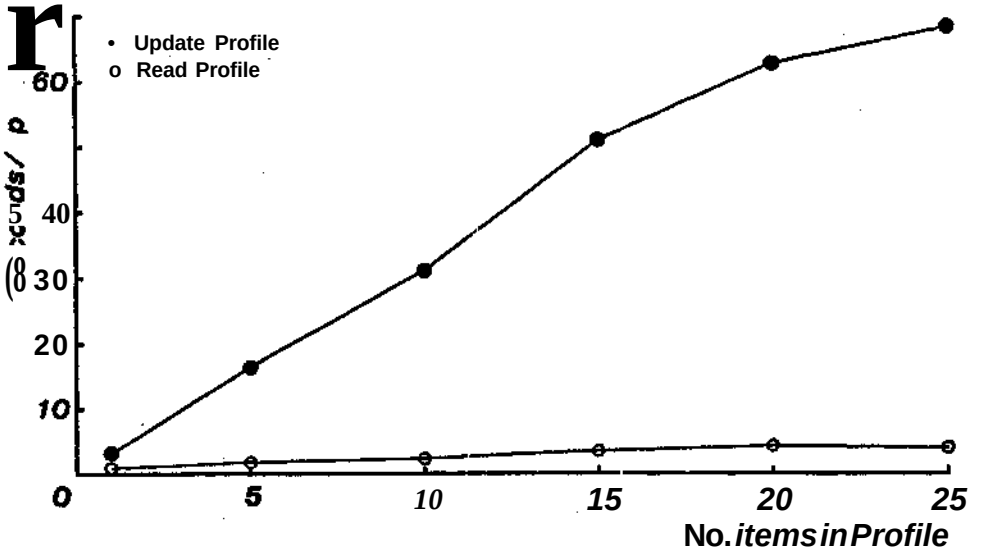

Figure 6.4: Time for User Profile Operations

6.3.3 Performance of User Operations

The response times for those operations invoked by human users are particularly critical. Any delay is especially obnoxious when a user is probing the name space to determine where to send a notice. The search must be fast or users will not attempt it, resulting in more notices being sent to the wrong bulletin boards. Even with the addition of the ** wildcarding facility, going through the UDS takes time. Accordingly, postmasters cache names to avoid unnecessary name look-ups. One cache keeps a list mapping all bulletin board names to internal identifiers. The internal identifier can be used to locate the files for the local copy of a bulletin board. For other bulletin boards whose only copy is remote and for user profiles, a second cache stores mappings from names to copy-set locations. The second cache keeps records only for those names that have been used recently.

The first action a user agent takes is to read a user's profile. Figure 6.4 shows the delay experienced by a user aigent reading profiles of varying sizes. It also shows the time taken to update items within a profile, assuming no voting needs to be done. The prototype provides barely adequate response. Comparison of the number of file accesses with the data in Table 6.2 reveals that the delay is almost entirely due to file operations.

The other operations typically performed by users are notice operations. Figure 6.5 shows the response times for reading a bulletin board. Analysis of traffic to a handful of bulletin boards and distribution lists indicated that most notices are small: typically around one kilobyte in size. So, Taliesin can be expected to read a bulletin board in ten to twenty seconds. A clever user agent can disguise some of this delay by pre-fetching notices from the first bulletin board while displaying relevant information about what bulletin boards a user will nornially read. However, the start up delay is still worrisome.

The other notice operations, posting and deleting, can be performed in the background. Figure 6.6 shows the elapsed time seen by a client posting a notice to a bulletin board. Figure 6.7 similarly shows the delays for deleting notices/Section 6.3.4 will discuss the time needed to propagate word of the postings and deletions to other copies.

Response times were also measured for three other operations. The first, ^searching the bulletin board name space, is apt to be invoked fairly frequently. As shown in Figure 6.8,

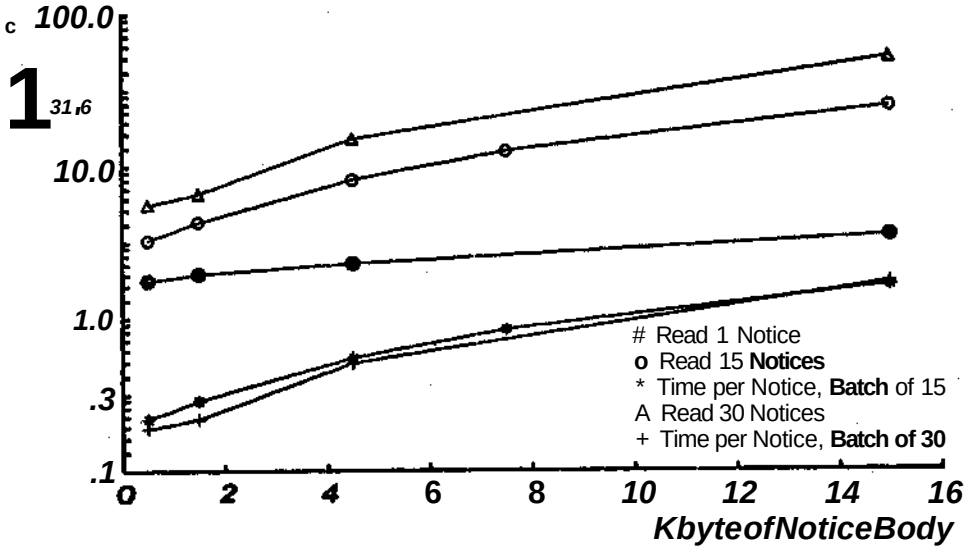

Figure 6.5: Time for Reading Notices

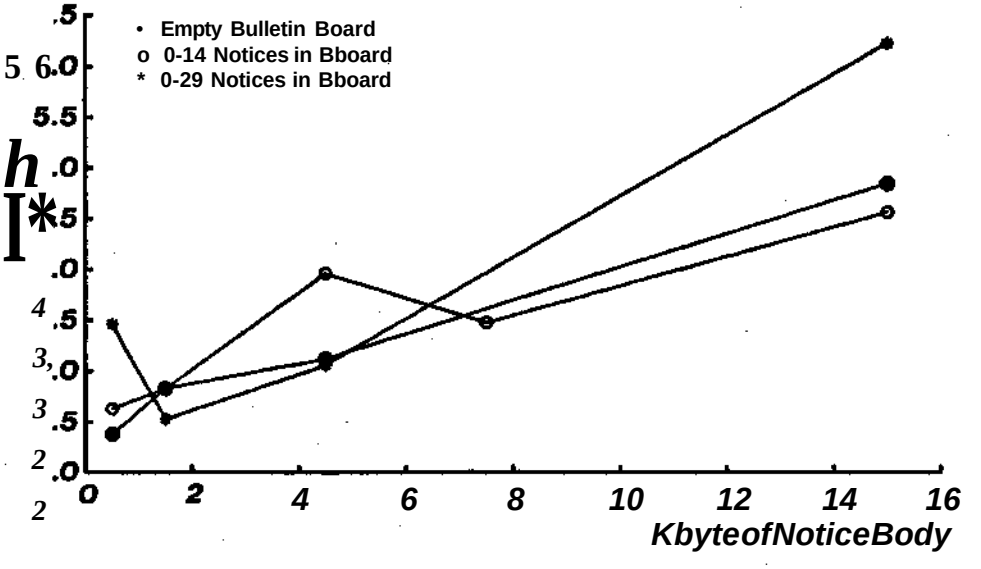

Figure 6.6: Time for Posting Notices

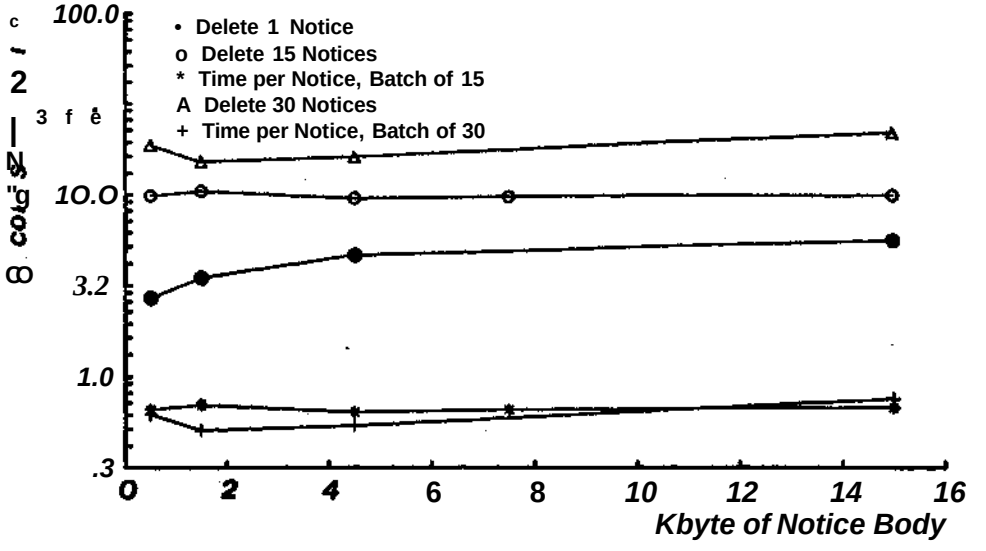

Figure 6.7: Time for Deleting Notices

the response times are last because all names are cached by postmasters in the prototype. The delay is more than adequate, even considering that a user will be waiting for the results.

Finally, the creation and destruction of bulletin boards were timed. Figure 6.9 plots the relationship between the delays seen by a client and the number of key words in bulletin board names. The rapidly increasing delay for creation stems from the need to create all the relevant directories as well as the final name. The more gradual increase for destruction reflects only the increasing time to parse a name. In both cases, the delay is largely due to the number of file operations.

6.3.4 Performance of Replication Operations

Taliesin uses two replication algorithms. Notices axe replicated at the copies of a bulletin board through the regular exchange of reconciliation reports. Figure 6.10 plots the time needed to send out reconciliation reports as a function of the number of copies. In each trial, each copy receives four new notices with 512 bytes of body and four new deletion markers. Furthermore, half of the old notices are deleted. The time needed to send a report to every other copy was measured.

The other replication algorithm used by Taliesin is a form of majority voting. The total elapsed time spent in the voting code was measured for updates to user profiles. Figure 6.11 shows samples of times as a function of the number of copies. The time spent for other object types will vary with the differences in the sizes of the updated values to be written and in the times needed to initialize a new copy should the copy set be changed.

Note that the times grow rapidly. Most of the growth is due to contention at the coordinator for access to the file record for the update. Variation in the file system response is responsible for the oddity of the data point for four copies. The voting algorithm admittedly takes a lot of time. Again, the reason for the long delays is the slowness of file operations. To ensure that recovery is possible, the prototype updates the voting log frequently. The delays are increased by the fact that the timing routine issues bursts of five updates then waits for all to complete.

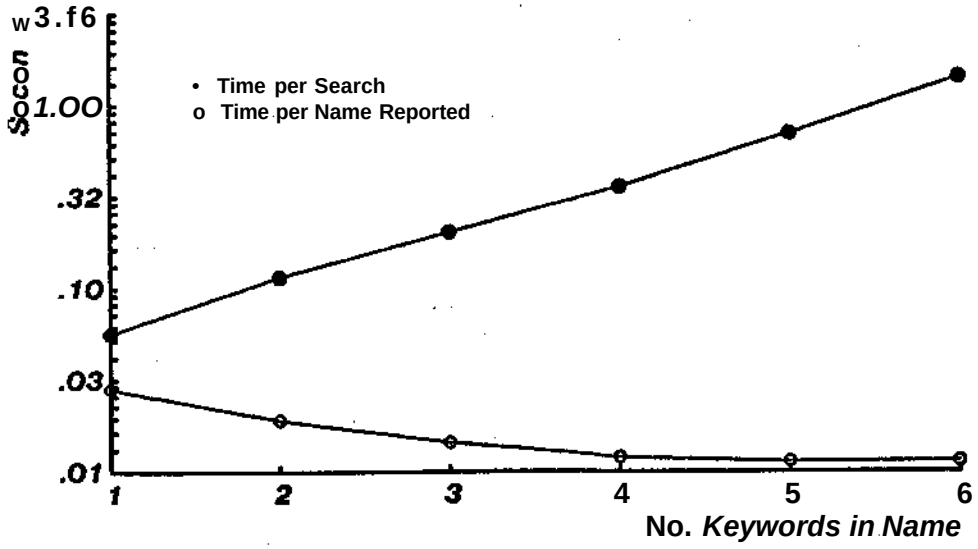

Figure 6.8: Searching the Bulletin Board Name Space

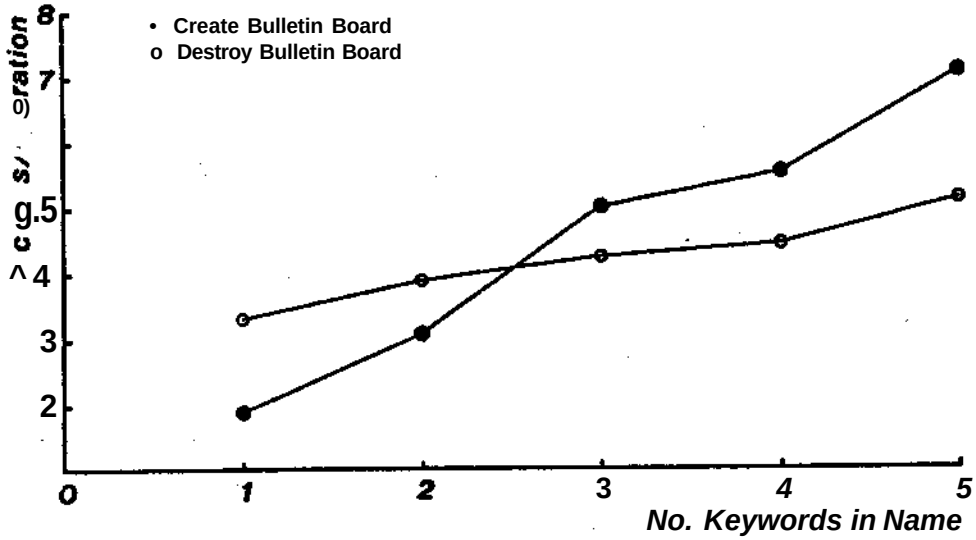

Figure 6.9: Creating and Destroying Bulletin Boards

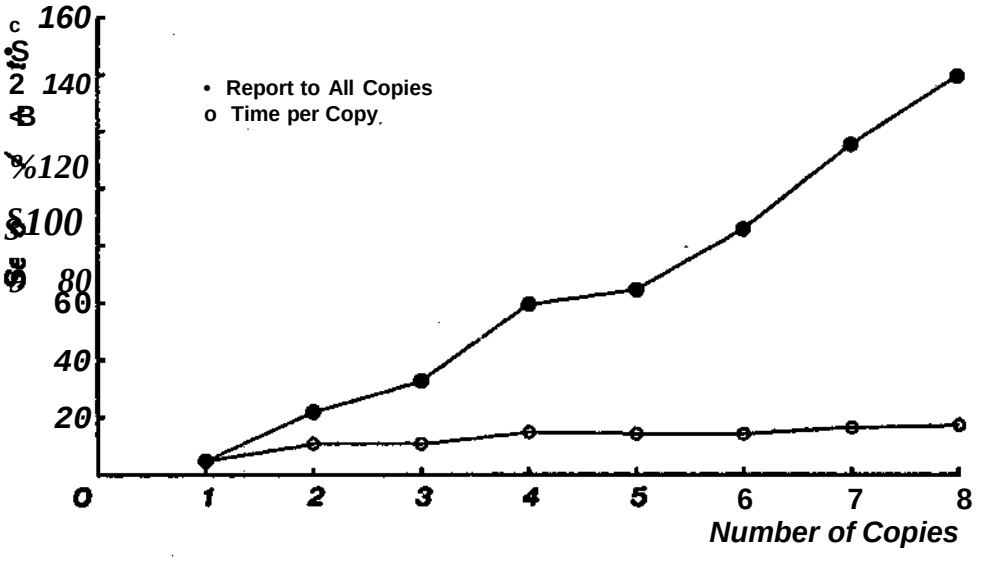

Figure 6.10: Time to Generate Reconciliation Reports

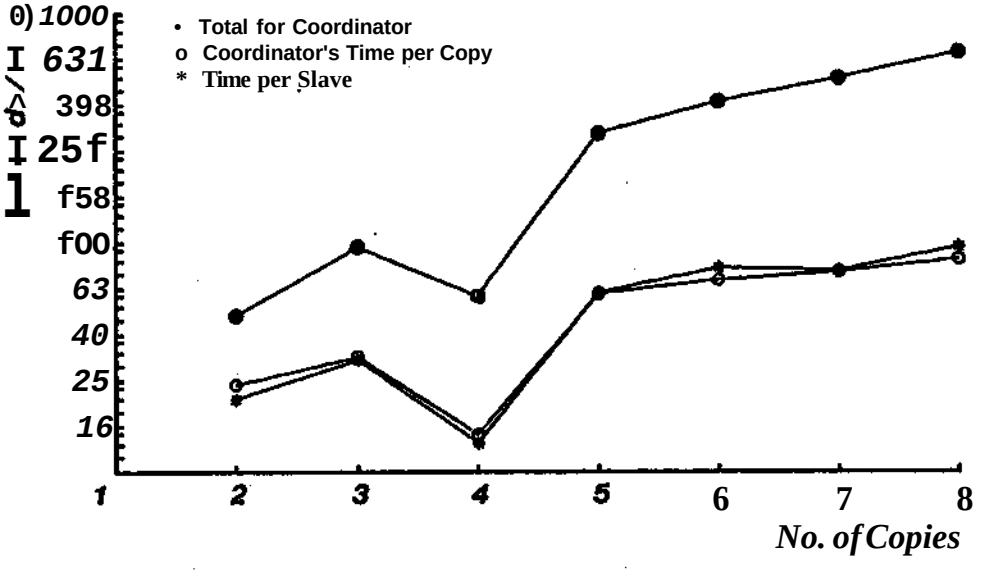

Figure 6.11: Time to Complete Voting

6A Overall Evaluation of the Prototype

The prototype demonstrates the feasibility of many aspects of the design, while pointing out some potential bottlenecks. Experience implementing the design also illustrates many of the considerations and trade-offs that apply to distributed applications in general.

Use of a separate name service works well. Its flexibility made it easier to revise the name space mappings used by Taliesin. It also simplified the implementation by providing uniform, unique naming of users. The UDS and Taliesin work fairly well together.

The decision to introduce the Nexus was a major success. Being able to monitor all inter-agent messages greatly eased the process of debugging. It also reduced the amount of code that had to be modified when new methods for locating services were provided under the V-System.

The major problem was in achieving reasonable performance. A number of initial decisions on the distribution of data structures between memory and disk were incorrect. Considerable time would have been saved in developing the system if the message passing and file access times had been measured early. Those measurements should have been used to guide data structure storage. Furthermore, the early policy for saving state on disk to allow the re-starting of operations killed performance. Estimating the service delays based on measured disk access times would have shown that the policies were bad before they were implemented.

Another bottleneck turned out to be the number of iterations needed between the UDS and postmaster to enumerate the names of sub-topic bulletin boards. This was resolved by the addition of a new search mode to the UDS. Even so, slow file access resulted in barely tolerable response times. To get more acceptable delays, it was necessary for postmasters to cache bulletin board names in core.

Only the ordinary file system services were used to provide stable storage. This works tolerably well. The Nexus keeps no state between incarnations, so it has no trouble whatsoever. Taliesin and the UDS recover most of their state from boot files. Writing and flushing the boot files whenever a unique identifier is generated slows the servers down some, but seems to be unavoidable.

There is a problem keeping the servers up and running. Because they run on workstations that can be claimed by users at any time, the servers are frequently destroyed by rebooting. This exposed a weakness in the original structuring of the data files that sometimes left them in a corrupted state as a result of an interrupted sequence of writes. A new data file format was adopted to minimize the problem.

The use of the V-System did make some aspects of the implementation almost trivial. In particular, the network-transparent IPC greatly simplified the implementation of communication between agents. Agents could all be run on one workstation during debugging, then scattered over different workstations when put into real use with no change. Furthermore, the speed of V-System message passing meant that the cost of using of the Nexus, a tremendous to aid in debugging, was negligible.

The prototype demonstrates that all the design features are amenable to implementation. The response times are barely adequate, but could easily be improved in a production version. For example, a production system could create indices for bulletin boards and cache the access control and copy locations records. The naming and replication algorithms of Taliesin have the potential for working well if more care is taken in the implementation and a file system with a more reasonable access time is used.

Chapter 7

Conclusions

This thesis addressed a number of the problems involved in providing bulletin board service to users scattered over a potentially large network. The design for the Taliesin bulletin board system meets many of the functional and performance goals developed in Chapter 1. To meet them, a novel replication algorithm was developed. The strengths, novelties, and weaknesses of the design will be summarized in this chapter. Taliesin will be shown to solve a number of the problems, although others remain to be dealt with.

7.1 Functional Requirements

The purpose of a bulletin board system is to enable people to speak on a particular topic to those interested in the topic, without knowing exactly who is in the audience. The notices must be distributed to the locations of interested readers and stored until the readers are ready to listen. Section 1.3 listed a number of functional requirements. These fall into three classes. A distributed bulletin board system must provide a method of identifying the recipients of a notic^, it must provide an environment that is easy for users to work with, and it must truly be a distributed service.

7.1.1 Establishing a Rendezvous

The most basic function of a bulletin board system is to deliver notices to interested users. The functional requirements relating to the task of delivering notices demand that a bulletin board system:

1. Name notices by encoding their subject matter.
2. Use intuitive names.
3. Allow users to inquire as to what names are recognized.

In Taliesin, bulletin boards are named by a set of keywords. The keywords are classified as specifying a location, an organization, or a topic. The contents of a bulletin board corresponds to the intersection of the categories identified by the keywords. Since keywords can identify topics, notices are named by encoding their subject matter. The name space is significantly more user friendly than existing name spaces. For instance, keywords can be listed in any order by users, unlike the hierarchical name space used by USENET. Hopefully users will select keywords whose meaning is clear to others, resulting in an intuitive name space. The operation **EriumerateBboardNames** allows users to explore the name space. The ability to inquire about bulletin board topics reduces the need to guess about what keyword combination some other person decided was appropriate for a topic.

The major function of a bulletin board system is to support retrieval of notices based on a specification of topics of interest. The name space used by Taliesin lends itself to the natural and intuitive recognition of sub-topic and super-topics, unlike the flat name-space of Arpanet and Grapevine distribution lists. Adding or removing keywords from a name is equivalent to adding or removing restrictions on a topic. Any operation can be applied to a collection of bulletin boards defined in this way, except for **CreateBBoard**.

The name space and the operators upon it provide a fair degree of expressive power. However, there are a couple of missing features that might prove to be desired by users. The Taliesin name space makes no provision for users to specify what they don't want to see. To keep from inundating readers with unwanted notices, it might be very helpful to let them select the general categories of interest, then exempt specific subtopics.

Another, related missing feature is support to recognize conversations. Users may want to read or avoid reading notices pertaining to a particular conversation. They might also want to avoid reading notices submitted by particular authors. Determining which of these and other selection mechanisms are really useful remains an open question.

7.1.2 User Environment

A second important group of functional requirements that a satisfactory bulletin board system must meet relates to its general user friendliness. The sheer scope of a network implies a complex environment. A good bulletin board system must hide this complexity and generally perform the dirty work automatically, without user intervention. Users must be able to explore the environment and protect themselves from unwanted behaviors of others.

1. Provide a friendly user environment.
2. Minimize the work done by users.

The work done by users is minimized in several ways. The design defined a format for user profiles so that user agents can store information about what bulletin boards each user currently is following, plus a specification of which notices have been seen. Users do not have to tell their user agents what bulletin boards they wish to view every time. They can express interest in the entire collection of bulletin boards related to a topic, thus avoiding the work of entering each bulletin board separately and tracking the creation and destruction of bulletin boards related to the topic.

The design also provides support to minimize the effort of bulletin board maintainers. Normally, bulletin boards act as archives. However, notices can be assigned expiration dates or a bulletin board can be set to flush notices that have been sitting in a copy for longer than some minimum time. The bulletin board system takes responsibility for automatically performing these deletions.

Users can control their interaction with others in part through the access control mechanisms included in the design. Bulletin boards and user profiles are protected so that read, write/post, and delete operations can be limited to exactly those individuals or groups the owner desires. Even ownership can be transferred. While it often would be useful if the originator of a notice were allowed to delete it even when he ordinarily does not have deletion rights, that option is not allowed in the present design.

Finally, user friendliness is supported in the form of providing the ability to query about almost every aspect of the design. In addition to queries probing the name space, commands such as **QueryCopySet** and **QueryAutoDeletion** allow users to find out what the attributes of bulletin boards and user profiles are. Even casual users have control over all attributes of the bulletin boards and user profiles they own. They can set the protection, contents, and attributes such as automatic deletion conditions.

7.1.3 Distributed Service

The whole point of having a bulletin board system spanning a network is to provide access to non-local bulletin boards. To scale well, its services must be distributed so that no node become a critical resource or bottleneck. The system must not tie users down to a single node if it is to increase availability by allowing users to read whatever copy of a bulletin board is currently available. In brief, a distributed bulletin board system must:

1. Provide access to resources anywhere on the network.
2. Not assume that users always will use the same copy of a replicated object.
3. Handle requests for which a user must wait using a minimal number of cooperating locations.
4. Service requests in a distributed manner.

Postmasters cooperate to provide access to non-local bulletin boards and user profiles. However, if the underlying network does not support rapid communication between arbitrary pairs of nodes at all times, reading operations such as ReadNewNotices cannot be performed while users wait. A production implementation based on the design described in this thesis could allow users to set up outstanding requests to do the reads overnight, however.

All read operations are done by referring to any one copy of a bulletin board, user profile, or name-binding. Clearly, no fewer number of copies can be referenced while still satisfying the demand for information. Update operations are either applied to the single copy of an unreplicated object or undergo a voting procedure. An immediate reply is generated, indicating that the voting procedure has begun, so users never need to wait for more than one node with a copy to service their requests.

Because all data structures can be accessed from any node, providing only that there is currently an open channel for shipping the bits, the only possible location dependence would be in the interpretation of the user profile. The notice replication algorithm was designed specifically to ensure that the record of what a user has seen is independent of what copy was actually read.

The cooperation between postmasters allows copies of objects to be placed wherever is convenient. The replication algorithms treat all copies as peers. No node acts as a central authority.

7.2 Performance Requirements

Performance of a distributed bulletin board system can be evaluated in terms of how much it costs in terms of storage, network bandwidth, and CPU usage, and how fairly the costs are distributed over the participating nodes. In the short run, costs are determined by policies, including those governing copy placement and the location of query processing. This thesis did not attempt to experiment with policies to find out what ones lead to the lowest costs. Some basic heuristics and a number of pointers to relevant work in related areas were presented in Chapter 5. For example, it was pointed out that experience with distributed file and database systems includes ideas on how to place copies to achieve acceptable reliability while minimizing networking and storage costs. Research into network multicast facilities can be applied to reduce communications costs.

The replication algorithms used by Taliesin provide a number of possibilities for controlling costs by judicious selection of policies. Notices are replicated when convenient by issuing reconciliation reports. An implementation is free to choose any means of delivering reports to the other copies. Batching of updates in a report is even possible. The policies for determining when and where to send reports allow the system to be tailored to the

delay, data transfer, and partitioning characteristics of a network.

The fact that the copy set can be changed with no more user intervention than issuing a command to do so opens up a whole new area to explore in internally initiated adaptive measures. If sites keep useful performance statistics, the system can alter the copy set to relieve to performance bottlenecks. For example, it is possible to change the weighting of copies so that more reliable copies have greater influence in voting. Copy placement can be altered to match new usage patterns.

Ultimately, the costs are limited by scaling properties produced by user behavior. Chapter 3 pointed out how the choice of a name space for bulletin boards appears to have a major impact on cost scaling. The key to achieving good scaling is to encourage some form of locality of reference. Ideally, small communities of users will follow bulletin boards of very restricted subject matter. While bulletin boards covering more general topics can exist, the number of notices posted to them must be relatively small. Encouraging the creation of many small bulletin boards is a necessary step to get user behavior leading to good scaling.

Taliesin encourages the creation of refined bulletin boards by offering casual users a simple operation that will create a new bulletin board. The introduction of new bulletin boards is partially automated for readers because the recognition of groups of bulletin boards covering a common topic implies that the system will automatically present notices from a new bulletin board to all users interested in a super-topic of it. The **CreateBBoard** operator will need to be restricted in a production system by enforcing local resource allocation policies.

It would have been interesting to measure the scaling properties of Taliesin by observing how people made use of the name space. Unfortunately, the prototype could only be tested on a small user community for a limited period of time. It wasn't possible to gather statistics proving that it scales gracefully. Some suggestive theoretical analysis in Section 3.3 indicates that the storage and communication costs might well grow as the logarithm of the number of users of the bulletin board system. While not the ideal of constant cost, logarithmic growth is slow enough that costs would remain acceptable over a very wide range of network sizes. It is certainly an enormous improvement over the current tendency of bulletin board systems to grow in proportion to the total user community.

The scaling of the communications costs associated with the replication algorithms was examined in Section 5.3 in terms of the impact of increasing the number of copies. The delay in reaching agreement using the majority voting algorithm increases only due to the increased delay in transmitting messages across the network (more hops when there are more nodes), to the added processing needed to send a message to a greater number of destinations, and to the additional processing to handle a greater number of replies. The time needed to reach complete knowledge using the notice replication algorithm grows with these factors, but in addition grows as the logarithm of the number of copies. There is a secondary effect in the size of the messages and in the storage costs because the more copies there are, the longer the record of the set of copies will be.

The response times seen by users are minimally influenced by the size of the network. Users only need to wait for an operation to complete at a single copy. How long that takes is dependent upon the copy placement policy. Generally, more copies buys faster response at the cost of more storage.

7*3 An Original Replication Algorithm

The most important original contribution of this thesis is the development of a replication algorithm particularly well suited to the semantics of operations on a bulletin board system. The replication algorithm was prompted by the need to retain the ability to concurrently

submit new postings from many nodes that characterizes the better current bulletin board systems, such as USENET, while enabling the records of what users have seen to be equally applicable to subsequent reads of any copy. Existing replication algorithms do not meet both needs. In fact, the novel algorithm for replicating notices even works well if nodes are only able to establish periodic rather than continuous communication links. The notice replication algorithm plus a modified version of Thomas' voting algorithm provide good performance in replicating both the notices and administrative information comprising a bulletin board. Together, these algorithms have the following properties:

1. As long as a single copy is available, users may access a bulletin board.
2. The replication algorithms are adapted to the possibility of frequent network partitions of moderate duration.
3. The read-time reported by one copy works equally well on all copies, yet it uses a sequence number to truncate a list of notice identifiers.
4. The number of copies can change dynamically. The changes are handled by the replication algorithms without need for manual intervention by a systems manager, beyond telling the system to create the new copy.

The notice replication algorithm allows each copy of the bulletin board to act independently throughout the processing of all user operations. Newly posted notices are dated using a local logical clock and assigned unique identifiers without any need to refer to the state of other copies. They are visible immediately at the initial copy and become visible at subsequent copies as soon the notices are transmitted in reconciliation reports.

The illusion of agreement on the order of notice postings is provided by reporting not the current real time, but a locally computed logical time as the measure of when a read operation was handled. The computed time is based on knowledge of lower bounds on the local times of all copies. The record of what a user has seen is that logical time plus a list of identifiers for those notices that were presented yet had a later submission time. This representation of what a user has seen is equally applicable to any copy for subsequent reads, is based on reading a single copy, and has much of the storage compactness of a clock value.

The notice replication algorithm has a problem with copies that remain inaccessible for extended periods of time. They prevent the advancement of the computed time returned as a measure of when reads occur. In turn, that means that the the number of notice identifiers in the records of what users have seen grows. This flaw is overcome by using a majority vote algorithm to change the agreed upon set of copies. The majority voting scheme can be adapted to a frequently partitioned environment at the cost of greater delays in reaching agreement. As was explained in Section 4.1.1, moderately long delays are acceptable.

Another potential weakness is that users can see notices more than once under unfavorable conditions. If a single bulletin board is being read, this happens only when a copy is retroactively deleted after it has sent out reconciliation reports. Because copies receiving those reports do not have to wait for the deletion before advancing their epochs, illegally signed notices may be dated before some copy's agreement epoch. A user reading such a copy will see the notice but no individual record of it will be kept in his user profile. Then, when the user reads again, he will see the re-submitted version. This anomaly will remain tolerable if copies that are not isolated are rarely deleted.

The usefulness of the notice replication algorithm is not limited to the context of bulletin boards. It provides replication with a high degrees of concurrency for any object that can be characterized as a set with a history. That is, the object consists of a set whose elements can be independently added or deleted. The readers of a set with history are not particularly interested in the complete, current membership of the set. They are interested in just the changes since the last time they inquired.

These semantics can arise in many ways. For example, consider a library catalog kept on line and a pool of researchers checking out material to learn about new advances in their fields. Entries for new books and journals are added independently. The researchers are normally interested in just the new entries in the areas of their interest. The notice replication algorithm would be applicable to this situation.

The algorithm would also be useful in a software distribution service. The elements of the set would be programs instead of notices. A copy of the set would be kept at each distribution point. Clients of the service need prior viewing records to locate just the new offerings. In general, the notice replication algorithm is often useful when many people or programs periodically poll a database so that they may respond to new events or new information.

7.4 Ideas for Design Enhancements

The design does not provide several potentially useful features related to the notice operations. The notice replication algorithm as it stands does not guarantee that a reply will be dated after the original notice, although it is likely. Consider the case in which the signing epoch of the original notice is no later than the agreement epoch at the copy read. It is a basic property that every copy's posting epoch is greater than every other copy's agreement epoch. Accordingly, the reply's signing epoch, taken from a posting epoch, will be later.

Even if the original notice has been accepted recently, it is likely that the reply will bear a later signing epoch. Suppose the user composing the reply invokes a user agent, reads the original, and composes the reply during the same session. Normally, the user agent will talk to the same postmaster throughout the session. Since postmasters predictably try to locate the closest copy of a bulletin board when posting notices, the reply will most often go to the copy that the original notice was read from. Since posting epochs are incremented in response to read operation, the reply will bear a later signing epoch.

It is possible to provide a guarantee that replies will be dated after the notices they are in reply to. Although it means trusting user agents, this feature could be implemented by allowing the user agent to specify a minimum signing epoch for a newly posted notice. That epoch for a reply would be one more than the signing epoch of the original notice. The accepting postmaster is already allowed to advance its posting epoch at will so adding this feature does not introduce any erroneous behaviors in the notice replication algorithm.

Another notice operation that would sometimes be useful is the ability to modify a notice. Right now, this can only be simulated by deleting the original and posting the new version. However, the replication algorithm can be readily adapted to handling revision of existing notices. To do this, it need only be possible to instruct the system to give the new version of the notice the same identifier as the old and then re-sign it just as would be done if the copy had been deleted. The only potential problem with this is that confusion will result if the revised notice is being forwarded due to copy deletion. A slightly better approach would be to add a revision field to the signature. It would be handled like the number-of-signatures field, except that a later revision supercedes an earlier one with a greater number of signatures.

A final potential improvement in the notice operations would fix a flaw in the existing filtering of duplicates from different bulletin boards. Right now, when a user asks to read all the bulletin boards in a collection, postmasters present one copy of each notice, even if the notice was stored in more than one bulletin board. However, postmasters give the user agent no indication that a duplicate was suppressed so no record of its being seen is placed in the user's profile. If the suppressed duplicate happens to be dated after the bulletin board's agreement epoch, it will be presented the next time the user reads. Ideally, some marker like a deletion marker in form, but indicating that a notice was suppressed, should be returned as part of the read operation.

In the implemented version of the design, the automatic deletion conditions are identical at all copies. A very reasonable alternative that would provide more site autonomy is to make the decision on a copy-by-copy basis. Doing this would provide benefits such as reducing storage costs by having most copies retain only recent notices while still allowing automatic archiving at selected copies. However, the coding of per-copy deletion is complicated by the notice replication algorithm. Every notice must be kept until it is no longer needed for propagation in reconciliation reports, even if it should be deleted according to the local deletion policy. Creating a local-deletion marker that hides a notice from readers but doesn't delete it immediately is one way of implementing this feature.

Another potential enhancement is to define a new class of rights corresponding to the copy-owner. The right of copy ownership provides a way to acknowledge that some administrative authority has control over the resources at a particular location. It is reasonable to grant the copy-owner the right to modify and delete his copy. If per-copy deletion rules are also defined, the copy-owner would also logically be granted that power. This feature would enhance the autonomy of nodes by giving them more control over their storage.

A related feature that would support greater autonomy would be to recognize a new class of maintainers. Each node would give special privileges to those users responsible for managing local resources. The users could be identified as those belonging to the user group **local-superuser.** Each set of nodes under a different administration would have its own **local-superuser** user group.

Another feature that should be added is support for switching from following a bulletin board to just one of its subtopics. It would be nice if the bulletin board system would pick out what read time records from the parent entry are relevant. Using all of the old information will work because the invalidation scheme will eventually cause the irrelevant records to be deleted.

The bulletin board system also needs to do something to get a user started on a bulletin board. If a bulletin board has a large number of messages stored, it is painful to read them all on startup. A variant of the **ReadNewNotices** operation that accepts a real time as a selection parameter instead of an epoch number would be a reasonable solution.

Another related problem is what to do if the number of copies grows large. Some of the problems of distribution list expansion arise. Fortunately, the full expansion only has to be done when sending reconciliation reports and vote proposal messages, instead of every time a notice is posted. A lesser degree of expansion is done every time a name-binding is examined. There the concern is not the time taken to read the whole list, but that needed to pass the whole list from name server to postmaster. One idea for a solution that could be explored is to create *cache copies.* These would have no votes and could not advance epochs. Basically, all they would do is read *official copies,* store their state, and handle posting of notices to the official copies. The notices could be visible locally, of course, pending forwarding to ah official copy.

Updates approved through the majority voting process are in effect carried out using a two-phase commit[Gra79b]. Sending out a proposal asks copies to prepare to commit. Servers voting 'yes' are agreeing that they are prepared. Sending out the commit/abort command is the second phase of the two-phase commit. If network partition delays the completion of the voting on a proposed change to the copy set of a bulletin board, advancement of its agreement epoch is blocked at every copy. Only the coordinator can decide to abort and so release copies from their promise to commit the change if it is approved. It is quite possible that the advancement of the agreement epoch could be held up indefinitely. Three-phase commits are more easily aborted or committed by ordinary voters in the presence of a partition[CK85,Ske81]. If delays in voting are enough of a problem, the voting algorithm could be changed to use a three-phase commit instead.

7.5 Contributions and Open Problems

This thesis on computer bulletin boards makes several contributions to computer science. The most important is the design of a replication algorithm for bulletin boards that has a combination of functional and performance properties needed for this application. Such a combination is not provided by algorithms invented by others. In particular, the notice replication algorithm is interesting because users can read any copy available and remember what they have seen in a fairly compact record. At the same time, notices can be added or removed without need for synchronizing the act across multiple copies. The replication algorithms can be used in other applications with similar semantics.

The replication algorithms are also interesting because they allow the number of copies to change without manual intervention. This leads to the interesting possibility that the bulletin board system may adapt itself to its load and to prolonged network partition through the application of internally initiated changes to the set of copies. This is a novel feature. Experimenting with it to develop good adaptive policies is left as an open problem.

Another noteworthy aspect of the replication algorithms is that two distinct algorithms were used. The notice replication algorithm needs immediate access to only one copy and so ensures good response to time-critical operations. The majority vote algorithm provides enough consistency to ensure that changes to the set of copies meshes properly with the operations on notices. In fact, it even handles copy deletions that occur retroactively on the epoch time-scale. There have been very few attempts at mixing different replication algorithms to combine their strengths.

The other contributions of this thesis lie in its identification of some of the causes of poor scaling of costs with increases in the number of users and in its proposals for solutions. Others have observed the growth, seen some of the more obvious causes, but have been committed to retaining existing software. The difficulty of completely re-writing a distributed bulletin board system is great enough that prior workers chose to try to take remedial measures using the existing software. The thesis examines the problem from a fresh viewpoint and a willingness to try a completely new approach if need be. As a result, some fundamental problems not probed by earlier researchers are identified and solved.

The bulletin board name space and the policies for routing updates are key contributors to the cost growth. A number of ways to control the costs due to routing were suggested. The notice replication algorithm offers implementors a wide latitude in the choices of update message batching, timing, and routing policies. Work by others on copy placement and efficient broadcast and multicast techniques was reviewed, pointing out ways in which the work can be applied.

A novel approach to controlling growth was suggested, namely, selecting a name space that will encourage helpful user behaviors. It is interesting that a keyword-oriented name space, suggested as being good from the user-friendliness point of view, also promises to result in better cost scaling. Keyword-based name spaces have been used or proposed by others[MP85]. However, the recognition of their potential for improving scaling by altering user behavior is an original contribution.

While keyword-based names spaces have been proposed, there seems to be some debate as to how to solve such basic problems as implementing the test for the legality or ambiguity of a name[CCM85,Kil85]. An interesting aspect of this thesis is that the prototype demonstrates that keyword-oriented name spaces can be built on top of an underlying hierarchical ones with tolerable efficiency, if a scan for all matches crossing boundaries in the hierarchy is built into the name service. It is not clear, however, how well this technique will work on a larger network.

This thesis also leaves open a number of problems. The design was not tested on a large network to verify its scaling properties. Doing so would be an interesting test of the many conjectures made. Another useful exercise would be to experiment with adaptive policies for copy placement, update routing, and for choosing to exclude isolated copies.

Finally, the functions offered reflect the traditional, common operations. Recent work in computer conferencing and conversation based mail systems reflect interest in extended functionality. It would be interesting to experiment to find out what extensions users find most useful.

This thesis does not address the numerous issues associated with the design of user agents. There are many problems dealing with interaction modes for expressing commands, presenting information in a convenient fashion, and deciding what additional functions, especially in terms of query processing, are useful. It would be interesting to learn if a common user profile format can support a wide variety of user interfaces.

Bibliography

[ABG84] Rony Attar, Philip A. Bernstein, and Nathan Goodman. Site initialization, recovery, and backup in a distributed database system. *IEEE Trans. on Software Engineering*, SE-10(6):645–649, November 1984.

[Ada85] R. Adams. USENET statistics. USENET messages on mod.newslists, 1985.

[Agu84] Lorenzo Aguilar. Datagram routing for internet multicasting. In *Proc. ACM SigComm '84 Tutorials and Symposium on Communications Architectures and Protocols*, pages 58–63, ACM SigComm, Montreal, Quebec, June 1984.

[Arb86] Arbitron program (written by Brian Reid). Readership of USENET news groups at selected sites. USENET messages on ba.news.ratings, 1985/1986.

[Ben85] Michael Ben-Or. Fast asynchronous Byzantine agreement. In *Proc. 4th ACM Symposium on the Principles of Distributed Computing*, pages 149–151, ACM, Minaki, Ontario, August 1985.

[Ber] E. J. Berglund. An introduction to the V-System. To appear in *IEEE Micro*. Also available from the Distributed Systems Group, Department of Computer Science, Stanford University.

[BG83] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control — theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, December 1983.

[BG84] 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.

[BLNS82] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: an exercise in distributed computing. *CACM*, 25(4):260–274, April 1982.

[Bog83] David Reeves Boggs. *Internet Broadcasting.* Technical Report CSL-83-3, Xerox PARC, October 1983.

[BRGP79] P. A. Bernstein, J. B. Rothnie, Jr., N. Goodman, and C. A. Papadimitriou. The concurrency control mechanism of SDD-1: a system for distributed databases (the fully redundant case). In Wesley W. Chu and Peter P. Chen, editors, *Tutorial: Centralized and Distributed Data Base Systems*, pages 516–530, IEEE Computer Society, 1979.

[BT85] Paul Bay and Alexander Thomasian. Data allocation heuristics for distributed systems. In *Proc. of IEEE INFOCOM 85*, pages 118–129, IEEE Communication Society, Washington, D.C., March 1985.

[Cas72] R. G. Casey. Allocation of copies of a file in an information network. *AFIPS*

Conference Proceedings, 40:617-625, 1972.

[CCM85] Kiat N. Chong, Eng K. Chew, and Ken J. McDonell. Implementation considerations of a name validation function for distributed directory services. In *2nd Int'l Symposium on Computer Message Systems,* pages 89-98, IFIP TC.6, Washington, D.C., September 1985.

[CD85] David R. Cheriton and Stephen E. Deering. Host groups: a multicast extension for datagram internetworks. In *Proc. 9th Data Communications Symposium,* pages 172-179, ACM SigComm, Whistler Mountain, British Columbia, September 1985.

[CGP81] Edward G. Coffman, Jr., Erol Gelenbe, and Brigitte Plateau. Optimization of the number of copies in a distributed data base. *IEEE Trans, on Software Engineering,* SE-7(l):78-84, January 1981.

[CH79] Wesley W. Chu and Paul Hurley. A model for optimal query processing for distributed data bases, In Wesley W. Chu and Peter P. Chen, editors, *Tutorial: Centralized and Distributed Data Base Systems,* pages 617-623, IEEE Computer Society, 1979.

[Chu79] Wesley W. Chu. Optimal file allocation in a multiple computer system. In Wesley W. Chu and Peter P. Chen, editors, *Tutorial: Centralized and Distributed Data Base Systems,* pages 414-418, IEEE Computer Society, 1979.

[CK85] David Cheung and Tiko Kameda. Site optimal termination protocols for a distributed database under network partitioning. In *Proc. J^th ACM Symposium on the Principles of Distributed Computing,* pages 111-119, ACM, Minaki, Ontario, August 1985.

[CP85] Douglas E. Comer and Larry L. Peterson. Conversations. *BYTE,* 10(13):263-272, December 1985.

[Cro82] David H. Crocker. Standard for the format of ARPA internet text messages. Request for Comments 822, August 1982.

[CW83] James Clifford and David S. Warren. Formal semantics for time in databases. *ACM Trans, on Database Systems,* 8(2):214-254, July 1983.

[Dal77] Yogen K. Dalai. *Broadcast Protocols in Packet Switched Computer Networks.* PhD thesis, Stanford University, April 1977.

[Dan81] Charles Daney. The VMSHARE coputer conferencing facility. In Ronald P. Uhlig, editor, *Computer Message Systems,* pages 373-384, North-Holland Publishing Company, Ottawa, Canada, 1981. Proc. IFIP TC-6 Int'l Symposium on Computer Message Systems.

[Dav84] Susan B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Trans, on Database Systems,* 9(2):456-481, September 1984.

[DeG75] Morris H. DeGroot. *Probability and Statistics.* Addison-Wesley Publishing Corp., Reading, Mass., 1975.

[Deu81] Debra Deutsch. Design of a message format standard. In Ronald P. Uhlig, editor, *Computer Message Systems,* pages 199-220, North-Holland Publishing Company, Ottawa, Canada, 1981. Proc. of the IFIP TC-6 Int'l Symposium on Computer Message Systems.

[DF82] Lawrence W. Dowdy and Derrell V. Foster. Comparative models of the file

assignment problem. *ACM Computing Surveys,* 14(2):287-313, June 1982.

[DH76a] Whitfield Diffie and Martin E. Hellman. Multiuser cryptographic techniques. In *AFIPS Conf. Proc. 1976 NCC,* pages 109-112, AFIPS, New York, June 1976.

[DH76b] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans, on Information Theory,* IT-22(6):644-654, November 1976.

[DHK83] Peter J. Denning, Anthony Hearn, and C. William Kern. History and overview of CSNET. In *Proc. ACM SigComm '83 Symposium on Communications Architectures and Protocols,* pages 138-145, ACM SigComm, University of Texas at Austin, March 1983.

[DHM*81] N. W. Dawes, S. J. Harris, M. I. Magoon, S. J. Maveety, and D. J. Petty. The design and service impact of COCOS, an electronic office system. In Ronald P. Uhlig, editor, *Computer Message Systems,* pages 373-384, North-Holland Publishing Company, Ottawa, Canada, 1981. Proc. IFIP TC-6 Int'l Symposium on Computer Message Systems.

[DM78] Yogen K. Dalai and Robert M. Metcalfe. Reverse path forwarding of broadcast packets. *CACM,* 21(12):1040-1048, December 1978.

[Dol82] Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms,* 3(l):14-30, March 1982.

[DS83] Dean Daniels and Alfred Z. Spector. *An Algorithm for Replicated Directories.* Technical Report CMU-CS-83-123, Dept. of Computer Science, Carnegie-Mellon University, May 1983.

[Edi86] Judy Lynn Edighoffer. Taliesin reference manual. Tech report in preparation, 1986.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *CACM,* 19(ll):624-633, November 1976.

[FO81] M. Fridrich and W. Older. The FELIX file server. In *Proc. 8th Symposium on Operating Systems Principles,* pages 37-44, ACM, Pacific Grove, CA, December 1981.

[Gar83] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans, on Database Systems,* 8(2):186-213, June 1983.

[Gif81] David K. Gifford. *Informantion Storage in a Decentralized Computer System.* PhD thesis, Stanford University, June 1981.

[Gif82] David K. Gifford. Cryptographic sealing for information secrecy and authentication. C^C'M,25(4):275-286, April 1982.

[Gra79a] J. N. Gray. Overview of recovery management. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course,* chapter 3.F, pages 460-465, Springer-Verlag, 1979.

[Gra79b] J. N. Gray. The two phase commit protocol. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating System: An Advanced Course,* chapter 5, pages 465-472, Springer-Verlag, 1979.

[Gre81] Richard J. Greene. An alternative approach to distributed database updating. *AFIPS Conf. Proceedings,* 50:481-485, 1981.

[Gue77] Lady Charlotte Guest. *The Mabinogion.* Academy Press Limited, Chicago, IL., 1977.

[HM79] D. Austin Henderson and Theodore H. Myer. Issues in message technology. In *6th Data Communications Symposium,* pages 6-1-6-9, IEEE Computer Society, IEEE Communications Society, and ACM, Pacific Grove, California, November 1979.

[H0l8O] Jeffrey Holden. Experiences of an electronic mail vendor. In Kenneth J. Thurber, editor, *Tutorial: Office Automation Systems,* pages 70-74, IEEE Computer Society, 1980.

[Hor83] Mark R. Horton. Standard for interchange of USENET messages. Request for Comments 850, July 1983.

[Hor85] Mark Horton. USENET statistics. Personal communication, February 1985.

[Hor86] Mark R. Horton. UUCP mail interchange format standard. Request for Comments 976, February 1986.

[HT78] Starr Roxanne Hiltz and Murray Turoff. *The Network Nation: Human Communication via Computer.* Addison-Wesley, 1978.

[Hui85] Christian Huitema. The COSAC electronic conferencing experiment. In *2nd Int'l Symposium on Computer Message Systems,* pages 247-254, IFIP TC.6, Washington, D.C., September 1985.

[IM79] Sreekaanth S. Isloor and T. Anthony Marsland. System recovery in distributed databases. In *Proc 3rd Int'l Computer Softward and Applications Conf. (IEEE Compsac 79),* pages 421—426, IEEE, Chicago, Illinois, November 1979.

[JM81] Jeffrey M. Jaffe and Franklin H. Moss. A responsive distributed routing algorithm for computer networks. In *Proc. 2nd Int'l Conf. on Distributed Computing Systems,* pages 348-353, Paris, France, April 1981.

[JN81] Anita K. Jones and Bruce Jay Nelson. *CERRO: A Secure Mail System.* Technical Report CMU-CS-81-120, Dept. of Computer Science, Carnegie-Mellon University, May 1981.

[Joh83] Marjory J. Johnson. Analysis of routing table update activity after resource failure in a distributed computer network. In *Proc. ACM SigComm '88 Symposium on Communications Architectures and Protocols,* pages 14-20, ACM SigComm, University of Texas at Austin, March 1983.

[Jon79] A. K. Jones. Protection. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course,* chapter 3.C, pages 229-251, Springer-Verlag, 1979.

[JS84] Jeffrey M. Jaffe and Adrian Segall. Automatic update of replicated topology data bases. In *Proc. ACM SigComm '84 Tutorials and Symposium on Communications Architectures and Protocols,* pages 142-148, ACM SigComm, Montreal, Quebec, June 1984.

[KC85] Kathleen Kelleher and Thomas B. Cross. *Teleconferencing: Linking People Together Electronically.* Prentice-Hall, Inc, Englewood, Cliff, N.J., 1985.

[Kil85] S. E. Kille. Mapping the global naming tree onto a relational database. In *2nd Int'l Symposium on Computer Message Systems,* pages 99-108, IFIP TC.6, Washington, D.C., September 1985.

[Knu73] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algo-*

rithms. Volume 2, Addison-Wesley Publishing Corp., Reading, Mass., 1973.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM,* 21(7):558-565, July 1978.

[Lam81] Butler Lampson. Atomic transactions. In *Distributed Systems- Architecture and Implementation,* chapter 11, pages 246-265, Springer-Verlag, 1981.

[LEH85] K. A. Lantz, J. L. Edighoffer, and B. L. Hitson. Towards a universal directory **service. In *Proc. J^th ACM Symposium on the Principles of Distributed** Computing,* pages 250-260, ACM, Minaki, Ontario, August 1985.

[LHM84] Carl E. Landwehr, Constance L. Heitmeyer, and John McLean. A security model for military message systems. *ACM Trans, on Computer Systems,* 2(3):198-222, August 1984.

[Lis84] SF-Lovers Distribution List. SF-Lovers Distribution List. Archive at Rutgers, 1980-1984.

[LLNS83] L. Landweber, M. Litzkow, D. Neuhengen, and M. Solomon. Architecture of the **CSNET name server. In *Proc. ACM SigComm '83 Symposium on Communications** Architectures and Protocols,* pages 146-153, ACM SigComm, University of Texas at Austin, March 1983.

[LN84] K. A. Lantz and W. I. Nowicki. Virtual terminal services in workstation-based **distributed systems. In *Proc. 17th Hawaii Int'l Conf. on System Sciences,*** pages 196-205, ACM/IEEE, January 1984.

[LRB82] D. H. Lawrie, J. M. Randal, and R. R. Barton. Experiments with automatic file migration. *Computer,* 15(7):45-55, July 1982.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals **problem. *ACM Trans, on Programming Languages and Systems,* 4(3):382-401,** July 1982.

[Lyn86] William Lynch. Stanford su-bboard posting statistics. Messages to SU-Bboard, 1985/1986.

[MD76] R. H. Myer and D. W. Dodds. Notes on the development of message technology. **In *Proc. Berkeley Workshop on Distributed Data Management and** Computer Networks,* pages 144-154, Berkeley Workshop on Distributed Data Management, May 1976.

[Mee85] Brock N. Meeks. An overview of conferencing systems. *BYTE,* 10(13):169-184, December 1985.

[ML79] Howard L. Morgan and K. Dan Levin. Optimal program and data locations in computer networks. In Wesley W. Chu and Peter P. Chen, editors, *Tutorial: **Centralized and Distributed Data Base Systems,* pages 432-439, IEEE** Computer Society, 1979.

[ML85] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. *ACM Operating Systems Review,* 19(2):40-52, April 1985.

[MM79] Daniel A. Menasce and Richard R. Muntz. Locking and deadlock detection in distributed databases. In Wesley W. Chu and Peter P. Chen, editors, *Tutorial: **Centralized and Distributed Data Base Systems,* pages 95-112, IEEE Computer** Society, 1979.

[MM81] *MM User's Manual* 1981. On-line documentation at SCORE.

[Moc83] P. Mockapetris. Domain names - concepts and facilities. Request for Comments 882, November 1983.

[MP85] P. Mockapetris and J. Postel. A perspective on name system design. In *Proc. IEEE INFO COM 85*, pages 349-355, IEEE Communications Society, Washington, D.C., March 1985.

[MPM78] Daniel A. Menasce, Gerald J. Popek, and Richard R. Muntz. Centralized and hierarchical locking in distributed databases. In Philip A. Bernstein, James B. Rothnie, and David W. Shipman, editors, *Tutorial: Distributed Data Base Management*, pages 178-195, IEEE Computer Society, 1978.

[MR79] Samy Mahmoud and J. S. Riordon. Optimal allocation of resources in distributed information networks. In Wesley W. Chu and Peter P. Chen, editors, *Tutorial: Centralized and Distributed Data Base Systems*, pages 419-432, IEEE Computer Society, 1979.

[MS83] Kiyoshi Maruyamo and David Shorter. Dynamic route selection algorithms for session based communication networks. In *Proc. ACM SigComm '83 Symposium on Communications Architectures and Protocols*, pages 162-169, ACM SigComm, University of Texas at Austin, March 1983.

[MSF83] C. Mohan, H. R. Strong, and S. Findelstein. *Method for Distributed Transaction Commit and Recovery Using Byzantine Agreement Within Clusters of Processors.* Technical Report RJ 3882, IBM Research Lab, San Jose, June 1983.

[Msg75] MsgGroup distribution list. First 100 messages. Log at USC-ECLC: <msggroup>msggroup.0001-0100, 1975.

[Msg80] MsgGroup distribution list. Conversations on distribution lists, MsgGroup archive at USC-ECLC: <msggroup>, January/February 1980.

[Msg84] MsgGroup distribution list. Msggroup archive. MsgGroup archive at USC-ECLC: <msggroup>msggroup.*, 1975-1984.

[Nat83] National Bureau of Standards. Specification for message format for computer based message systems. Request for Comments 841/FIPS Pub 98, January 1983.

[NL79] D. A. Nowitz and M. E. Lesk. A dial-up network of UNIX(tm) systems. In *Unix Programmer's Manual: Supplementary Documents*, pages 557-564, Bell Laboratories, January 1979.

[Now85] W. I. Nowicki. *Partitioning of Function in a Distributed Graphics System.* PhD thesis, Stanford University, 1985.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks. *CACM*, 21(12):993-998, December 1978.

[NSH83] Toshihiko Nakayama, Yoshihiro Shimazu, and Katsuhiko Haruta. A message handling system for public networks. In *Proc. 8th Data Communication Symposium*, pages 103-111, ACM SigComm, North Falmouth, MA, October 1983.

[OD81] Derek C. Oppen and Yogen K. Dalai. *The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment.* Technical Report OPD-T8103, Xerox Office Products Division, October 1981.

[Pal85] Jacob Palme. Conferencing standards. *BYTE*, 10(13): 187-192, December

1985.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. JACJIf,26(4):631-653, October 1979.

[Pet85] Larry Lee Peterson. *Defining and Naming the Fundamental Objects in a Distributed Message System.* PhD thesis, Purdue University, May 1985.

[Pic79] John R. Pickens. Functional distribution of computer based messaging systems. In *6th Data Communications Symposium,* pages 8-17, IEEE Computer Society, IEEE Communications Society, and ACM, Pacific Grove, California, November 1979.

[Por82] J. M. Porcar. *File Migration in Distributed Computer Systems.* PhD thesis, UC Berkeley, 1982.

[Pos82] Jonathan B. Postel. Simple mail transfer protocol. Request for Comments 821, August 1982.

[PP83] D. Stott Parker, Jr. and Gerald Popek et al. Detection of mutual inconsistency in distributed systems. *IEEE Trans, on Software Engineering,* SE-9(3):240-246, May 1983.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *JACM,* 27(2):228-234, April 1980.

[PT83] Jehan-Francois Paris and Walter F. Tichy. STORK: an experimental migrating file system for computer networks. In *Proc. IEEE INFOCOM 88,* pages 168-175, IEEE Communications Society, 1983.

[PWC*81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Fudisin, and G. Thiel. LOCUS: a network transaparent high reliability distributed system. In *Proc. 8th Symposium on Operating Systems Principles,* pages 169-177, ACM, Pacific Grove, CA, December 1981.

[Rei86] Brian Reid. Usenet costs-who's paying for what? Message posted to USENET: net.space etc, March 1986.

[Reu84] Andreas Reuter. Performance analysis of recover techniques. *ACM Trans, on Database Systems,* 9(4):526-559, December 1984.

[Rey84] J. K. Reynolds. Post office protocol. Request for Comments 918, October 1984.

[Ros83] Marshall T. Rose. Proposed standard for message header munging. Request for Comments 886, December 1983.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *CACM,* 21(2):120-126, February 1978.

[SBN84] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: the growth of a distributed system. *ACM Trans, on Computer Systems,* 2(l):3-23, February 1984.

[Sch81] Peter Schicker. Service definitions in a computer based mail environment. In Ronald P. Uhlig, editor, *Computer Message Systems,* pages 159-171, North-Holland Publishing Company, Ottawa, Canada, 1981. Proc. IFIP TC-6 Int'l Symposium on Computer Message Systems.

[SJRN83] Lui Sha, E. Douglas Jensen, Richard F. Rashid, and J. Duane Northcutt. Distributed co-operating processes and transactions. In *Proc. ACM SigComm '88 Symposium on Communications Architectures and Protocols,* pages 188-196,

ACM SigComm, University of Texas at Austin, March 1983.

[Ske81] Dale Skeen. Nonblocking commit protocols. In *ACM SIGMOD 1981 Int'l Conf. on the Management of Data*, pages 133-142, ACM SIGMOD, Ann Arbor, Michigan, April/May 1981.

[SN79] Michael Stonebraker and Erich Neuhold. A distributed data base version of INGRES. In Wesley W. Chu and Peter P. Chen, editors, *Tutorial: Centralized and Distributed Data Base Systems*, pages 381-411, IEEE Computer Society, 1979.

[Sol85] Karen Rosin Sollins. *Distributed Name Management* PhD thesis, Massachusetts Institute of Technology, February 1985.

[SS83] Alfred Z. Spector and Peter M. Schwarz. Transactions: a construct for reliable distibuted computing. *ACM Operating Systems Review*, 17(2):18-35, April 1983.

[SS84] Peter M. Schwarz and Alfred Z. Spector. Synchronizing shared abstract types. *ACM Trans, on Computer Systems*, 2(3):223-250, August 1984.

[Str77] Edwin P. Stritter. *File Migration.* PhD thesis, Stanford, 1977.

[Str85] Carl R. Strathmeyer. Integrating corporate message systems: a practical application of electronic mail standards. In *2nd Int'l Symposium on Computer Message Systems*, pages 255-271, IFIP TC.6, Washington, D.C., September 1985.

[Svo81] Liba Svobodova. A reliable object-oriented data repository for a distributed computer system. In *Proc. 8th Symposium on Operating Systems Principles*, pages 47-58, ACM, Pacific Grove, CA, December 1981.

[Ter83] Douglas B. Terry. *An Analysis of Naming Conventions for Distributed Computer Systems.* Technical Report UCB/CSD 83/156, Computer Science Division, University of California, Berkeley, December 1983.

[TGGL82] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and consistency in distributed database systems. *ACM Trans, on Database Systems*, 7(3):323-342, September 1982.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy data bases. *ACM Trans, on Database Systems*, 4(2):180-209, June 1979.

[TPS85] Same Toueg, Kenneth J. Perry, and T. K. Srikanth. Fast distributed agreement. In *Proc. J^th ACM Symposium on the Principles of Distributed Computing*, pages 87-101, ACM, Minaki, Ontario, August 1985.

[Tsi84] D. Tsichritzis. Message addressing schemes. *ACM Trans, on Office Information Systems*, 2(l):58-77, January 1984.

[Uhl81] Ronald P. Uhlig, editor. *Computer Message Systems*, North-Holland Publishing Company, Ottawa, Canada, 1981. Proc. IFIP TC-6 Int'l Symposium on Computer Message Systems.

[UU82] Jeffrey D. Ullman. Relational algebra. In *Principles of Database Systems*, chapter 5.2, pages 151-156, Computer Science Press, Rockville Maryland, 1982.

[Val84] Jacques Vallee. *Computer Message Systems.* McGraw-Hill Pulications Co, 1221 Ave. of Americas NY, NY, 1984.

[Vit81a] John Vittal. Active message processing: messages as messengers. In Ronald P. Uhlig, editor, *Computer Message Systems*, pages 175–195, North-Holland Publishing Company, Ottawa, Canada, 1981. Proc. IFIP TC-6 Int'l Symposium on Computer Message Systems.

[Vit81b] John Vittal. MSG: a simple message system. In Ronald P. Uhlig, editor, *Computer Message Systems*, pages 329–343, North-Holland Publishing Company, Ottawa, Canada, 1981. Proc. IFIP TC-6 Int'l Symposium on Computer Message Systems.

[Wat81] Richard W. Watson. Unique machine-oriented identifiers. In *Distributed Systems– Architecture and Implementation*, chapter 9, pages 197–203, Springer-Verlag, 1981.

[Wei85] Willaim E. Weihl. Data-dependent concurrency control and recovery. *ACM Operating Systems Review*, 19(1):19–31, January 1985.

[Won79] Eugene Wong. Retrieving dispersed data from SDD-1: a system for distributed databases. In Wesley W. Chu and Peter P. Chen, editors, *Tutorial: Centralized and Distributed Data Base Systems*, pages 598–616, IEEE Computer Society, 1979.

[WPE*83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proc. 9th ACM Symposium on Operating Systems Principles*, pages 49–70, ACM SIGOPS, Bretton Woods, NH, October 1983.

Appendix A

Correctness Proofs

In Chapter 4, the replication algorithms were described with an eye toward showing how they meet the needs of a bulletin board system. This appendix will define them in greater detail and prove the correctness of the novel ones. Correctness for majority vote is determined by whether it achieves serializability and atomicity. That is, it must produce the same effect as sequentially executing the updates in some order on a single copy. Notice replication will be deemed correct if a regular reader sees every notice posted to the bulletin board exactly once.

A.I Notation

When a few key properties characterize the interesting aspects of a function and its full statement would be tedious, the function will be presented as a rule. All other parts of the algorithms will be presented as program fragments in a form of extended Pascal. For example, a record in an array may be accessed by its key through the syntactic construct *arrayName(key Value)*. Iteration over all elements of a set is done by using a *for all* statement. Occasionally, sets are constructed by giving a property that all elements have.

Assertions that are part of the correctness proofs will often state that some relationship holds under particular conditions. This is intended to mean that the asserted relationship holds at those points in time that an external observer sees the conditions holding. Such an external observer can simultaneously view all copies. Local read/write locks are assumed to be used so that the concurrent activities at a single node do not corrupt the data structures.

For the sake of clarity, certain conventions will be observed. Assertions about the algorithms need to express relationships between the values of various fields of replicated objects and the contents of messages exchanged between agents. The value of a field of object *0* will be identified as *O.fieldName* where *fieldName* is drawn from the type definition of *O*. For example, if *0* were an identifier, the value of its *counter* field would be expressed as *0. counter*.

Copies are identified using bold face capital letters. All other constants and objects are written in non-boldface capital letters. The letter *E* is reserved for epoch values while T is reserved for time-stamps. *F* is used to refer to an arbitrary field when replicated objects are viewed as being arrays of fields. Updates to be approved through majority vote are denoted using *P* if their commit/abort status is not specified, *C* if committed, and *A* if aborted. A reconciliation report generated by the notice replication algorithm is denoted byi*.

Indices are used to distinguish between multiple items of the same type. In the case of time-stamps, a specific choice of indices will be made, corresponding to the apparent commit order of the proposals. A single level of indexing of proposals will always order them

according to Rule 12 in Section A.4.3. Thus, $\{P_i : i > 0\}$, $\{C_i : i > 0\}$ and $\{A_i : i > 0\}$ will always denote updates in the defined order. A double level of indexing will be used to specify updates in an arbitrary order. Note, however, that the term *subsequence* will be used to indicate an ordering consistent with the indexing rule.

The notice replication algorithms deal with a particular bulletin board. It is never explicitly identified in the notation used. The value of a field / at copy A will be written as A./. The majority vote algorithm deals with a variety of object types, but since each update in Taliesin affects exactly one object, it is useful to refer to the object as *O. O* is always treated as consisting of an array of fields. The value of field *F* is expressed as *O[F].* Additional qualifiers are added to indicate what version of the value is being talked about. Note that assertions claiming a property is true for all *F* mean the property is true for all *F* defined for the object being discussed.

A particular value of *O* can be expressed in one of two ways. Often, it is most useful to identify it in terms of the value of a field written by a particular update. If *C{* is a committed update that writes field F, *Oi[F]* will be used to denote that value and time-stamp written to field *F* by update *C%.* Note that individual fields are modified, not the entire object, so it is not necessarily true that for any committing update C_t and field *F* there must be a meaningful value O_t[F]. At other times, it is useful to consider the value at a particular instant at a particular copy. The value of field *F* at copy B will be expressed asB[F].

The initial value of *O* will be denoted by O_0- Note that by Assumption 21 all the original copies start with this same initial value. The initial value of a particular field *F* will be expressed as Oo[.F]. When convenient, *Oo* will be treated as having been written by an initial committing update Co.

The difference that makes the correctness of Taliesin's version of majority vote suspect is the possibility that the value of the copy set will be changed. The proofs will often discuss the impact of those updates that alter the copy set or of those based on a particular version ofit.

$\{C^m\}$ will denote the subsequence of committing updates that alter the copy set. Define the mapping *M.* by saying *A4(m) = i* iff $C^m = C\%$. The use of the term *subsequence* implies that *M(m) < M(n)* iff m < n. Also note that $C^\circ = C_o$.

Similarly, let $\{F_j^{TM} : i > 0\}$ denote the subsequence of committing updates such that F^m modifies field F, based upon reading the version of the copy set written by C^m. When needed, the definition may be extended to % = 0 by choosing $F_J^{TM} = C^m$.

[i, *k]* is used to denote the closed interval running from *i* to *k: {j : i ≤ j ≤ k}.* (ż, *k)* denotes the open interval *{j : i < j < k}. (i,k]* and [ż, *k)* denote half open intervals.

A.2 General Assumptions

The interesting aspect of the algorithms is their essential correctness, not whether straightforward definitions of them can be fouled up by server crashes or network glitches. In order to present the algorithms without including gory recovery code, some assumptions will be made.

Assumption 15 All data structures and checkpoint records are kept in stable storage.

Assumption 16 Writing a new local version of any object is an atomic operation.

Assumption 17 Check-pointing and re-start facilities produce the effect of either never running or completely running the routines that respond to requests

and message receptions. The facilities needed include logging and re-delivery of messages.

Assumption 18 Servers correctly execute the replication algorithms.

These assumptions ensure that state is not lost between the time a processor crashes and the time the service is re-started from a check-point. They also ensure that it is unnecessary to worry about disk crashes, garbled data, or faulty/malicious processors.

Network glitches are handled by periodically re-transmitting each message until the destination acknowledges reception. The acknowledgement must not be sent until after the message has been logged in stable storage so that re-delivery can be accomplished as required by Assumption 17. This makes it reasonable to assume that every message will reach its destination. Arbitrary delays and duplication are permitted, but undetected corruption of messages is assumed not to occur.

Assumption 19 All messages arrive at least once.

Assumption 20 No message is corrupted in transmission.

Because delay and duplication are the only types of communication errors that need be dealt with, the algorithms will be structured so that they can wait indefinitely for message arrival. They will respond to message reception by either calling an idempotent operation or by verifying that a message is not a duplicate of an earlier one before acting on it.

Many of the correctness proofs have an inductive character. The algorithms can be shown to preserve *consistent* state. Consistency is defined to mean that the replicated data contained in all copies is identical and that any control information associated with a particular copy (e.g., identifier of the local copy) is correct. Proper behavior requires initial consistency.

Assumption 21 Initially, all copies are consistent.

Assumption 22 Each time a copy is created, it starts out consistent with some already existing copy.

Initial agreement is usually obtained in Taliesin by starting with one copy. Consistency of new copies is achieved by duplicating an existing copy and modifying the control information as needed. At various points, the features of the algorithms that ensure the validity of Assumptions 21 and 22 will be pointed out.

A.3 Identifiers and Time-Stamps

The replication algorithms require unique identifiers and times that appear to come from a global clock. These are produced by standard techniques[Wat81,Lam78]. Unique identifiers are generated by concatenating a local counter with a unique server identifier. It is assumed that the counter runs over a great enough range that it will not wrap around during the lifetime of the bulletin board system.

Assumption 23 Each server has a unique identifier.

Assumption 24 Each counter runs over a range large enough that it will never wrap around.

The uniqueness of identifiers falls out easily given these assumptions and the definition of **Generateldentifier.** The counter increases each time so that no server generates the same identifier twice. Because the server identifier is unique, the whole identifier must be. Unique server identifiers may be assigned by some central, human authority.

Program Fragment 1

```
type
    Identifier = record
        counter: CounterRange;
        madeAt: ServerIdentifier;
        end;
    EpochType = CounterRange;
    IdentifierClass = (UPpATEJDJTYPE, TIME-STAMPJTPE, EPOCH.TYPE, ...);
    TimeStamp = Identifier;
    TimeOrder = (EARLIER, SIMULTANEOUS, LATER);

var
    LocalServerId: ServerIdentifier;
    LastIdGenerated: array[IdentifierClass] of CounterRange;

function GenerateIdentifier(idType: IdentifierClass): Identifier;
    begin
    GenerateIdentifier.madeAt := LocalServerId;
    LastIdGenerated[idType] := LastIdGenerated[idType] + 1;
    GenerateIdentifier.counter := LastIdGeneratedJidType];
    end;

function GenerateEpoch: EpochType;
    begin
    GenerateEpoch := LastIdGenerated[EPOCHJTYPE];
    end;

procedure AdvanceEpoch;
    begin
    LastIdGenerated[EPOCH.TYPE] := LastIdGenerated[EPOCHJTYPE] + 1;
    end;

function CompareTime(timel, time2: TimeStamp): TimeOrder;
    begin
    if (timel.counter = time2.counter) and (timel.madeAt = time2.madeAt) then
        CompareTime := SIMULTANEOUS
    else if Ttimel.counter > time2.counter) or
            ((timel.counter = time2.counter) and (timel.madeAt > time2.madeAt)) then
        CompareTime := LATER
    else CompareTime := EARLIER;
    end;
```

Observation 20 Identifiers are unique.

Observation 21 Each server generates a series of strictly increasing time-stamp and epoch values.

The ordering is normally on the counter values, but in the case of ties, the server identifiers are used. Any ordering over server identifiers may be used as long as it distinguishes between different servers. For convenience, the notation $T\backslash > T_2$ will be used when **CompareTime(Ti,T_2)** = *LATER* or, equivalently, **CompareTime(T_2,** T_t) = *EARLIER:*

Use of a technique developed by Lamport can make time-stamps to appear to be drawn from a logical, global clock[Lam78], Rule 1 states the additional property that must be enforced. Note that while epochs are similarly extended, they have been defined without the requirement of uniqueness. Certain aspects of the notice replication algorithm are more easily stated in the absence of uniqueness.

Rule 1 Every majority vote message has associated with it a time-stamp. If an incoming message bears a time-stamp T, advance the counter used to generate time to match *T.counter:*

```
Last!dGenerated[TIME-STAMP.TYPE] :=
    maximum(T.counter, LastIdGenerated[TIME^TAMPJTYPE]);
```

Every notice replication message has an associated epoch. If an incoming message bears epoch J5, advance the counter used to generate epochs to match *E:*

```
LastIdGenerated[EPOCH.TYPE] :=
    maximum(E, LastIdGeneratedjEPOCH-TYPE]);
```

When this rule is obeyed, the times of events appear to have been drawn from a global clock. Time flows forward because the counters perpetually increase. If an server hears of an event £i, then the counters are advanced so that any subsequent local event $£_2$ receives a later time-stamp.

Observation 22 Suppose a server has received a message describing an event occurring at time Ti. For every time-stamp T_2 generated by that server thereafter, $T_2 > T\backslash$. Similarly, if a server has received a message describing an event occurring at epoch *Ei,* then for every epoch E_2 generated by that server thereafter, $E_2 \geq E\backslash$.

Note that Observation 20 ensures that the granularity of time-stamps is sufficient to ensure that no two events occur simultaneously. These properties of time-stamps and epochs will be used throughout the rest of this appendix, often without being explicitly cited.

A.4 Majority Vote Algorithm

Taliesin uses majority vote to verify the legality of updates to user profiles, name-bindings, and the administrative information about bulletin boards, including the record of what copies exist. The majority vote algorithm is not new and so its correctness ordinarily would not need be established[Tho79]. Correctness of the original algorithm stems from the requirement that each change must be approved by agents handling copies with a majority of the outstanding votes. Any two majorities out of the same voting pool have a common member. Such common members ensure that updates produce a single new version from every old one.

Because Taliesin changes the information majority vote relies upon, correctness must

Program Fragment 2

```
type
   ObjectType = (BBOARD-CODE, PROFILE-CODE, NAMINGXODE);
   ObjectName = union of (BboardName, ProfileName, BindingName);

   BboardFields = (COPY^ET, COPY-HISTORY, ACCESS-CONTROL, DELETEJUJLES);
   ProfileFields = (COPY^ET, ACCESS-CONTROL, MISCJNFO, BBS-OFJNTEREST);
   NamingFields = (COPY.SET, NAME-BINDING);
   FieldCode = union of (BboardFields, ProfileFields, NamingFields);

   FieldType = record
      written: TimeStamp;
      data: (* depends on the field *);
      end;
   CopyData = record
      location: ServerIdentifier;
      votes: 0 .. MAXJNTEGER;
      end;
   CopyFieldType = record (* data for a COPY^ET field: *)
      numCopies: integer;
      totalVotes: integer;
      copies: array of CopyData;
      end;
   B board Descriptor = array[BboardFields] of FieldType;
   UserProfile = array[ProfileFields] of FieldType;
   NameBinding = arrayJNamingFields] of FieldType;
   ObjectValue = union of (BboardDescriptor, UserProfile, NameBinding);
```

be shown during the transition between different versions of the set of copies. The initial subsections of this section describe the particular variation of majority voting used by Taliesin. The final one proves its correctness, assuming the correctness of Thomas' majority vote algorithm for a class of particularly simple updates.

A.4.1 Format of Objects

The objects — bulletin board descriptors, user profiles, and name-bindings — are replicated in full. Each server that keeps a copy of any part of an object keeps a copy of the whole. The majority voting algorithm uses the *COPY,SET* field. The *COPY,SET* lists the locations of the copies of an object, together with the number of votes assigned to each copy. Each of the objects has a variety of other fields as well. Except for the *COPYJSET,* only a field's time-stamp is used by the majority vote algorithm. Program Fragment 2 defines the format of objects.

Taliesin allows the fields to be independently modified. The earlier assumption that all copies start off being consistent is refined to require that the initial version be dated with a time-stamp of To. Every agent with an initial copy must generate time-stamps greater than T_o after the object has been created.

Rule 2 VC 6 0_0***[COPYJSET\, VF C[F].wriUen* = T_o and VT generated** by C after the creation of 0, $T > T_o$.

An update proposal must state which object is being changed and what the new value is to be. The proposal lists the fields being modified in its *write set* To make sure that updates are properly synchronized, each proposal also must identify the information used in computing the new values. The *read set* lists the fields used while the *read times* identify

Program Fragment 3

```
type
    UpdateStatus == (VOTING, COMMITTED, ABORTED);
    UpdateProposal = record
        updateId: Identifier;
        proposer: ServerIdentifier;
        status: UpdateStatus;
        class: ObjectType;
        name: ObjectName;
        readSet: set of FieldCode;
        readTime: arrayfFieldCode] of TimeStamp;
        writeSet: set of FieldCode;
        writeTime: TimeStamp;
        newValue: array[FieldCode] of FieldType;
        end;
    VoteType = (UNDECIDED, YES, NO, PASS);
    VoteCopyData = record
        voter: ServerIdentifier;
        numVotes: 0 .. MAXJNTEGER;
        voteCast: VoteType;
        end;
    CoordinatorRecord = record
        proposal: UpdateType
        votesTotal,
        votesFor,
        votesPass,
        numVoters: integer;
        voters: array of VoteCopyData;
        end;
```

what versions were read. Program Fragment 3 details how this information is represented.

Taliesin generates a restricted class of update proposals. In particular, every update proposal both reads and writes the same object. Only one object is ever read or written by any one update proposal. This means that only a single object needs to be identified in the proposal. Updates to different objects proceed independently, other than possibly for collisions in naming. Naming collisions are avoided by forcing voters reject attempts to create multiple objects with the same name.

Rule 3 Any update P reads and modifies exactly one bulletin board, user profile, or naming binding group.

Another property resulting from the way Taliesin agents carry out operations is that the new value of any field is based upon its old value, as well as parameters supplied as part of a request. Read-only operations bypass the majority vote algorithm altogether, so it can be assumed that at least one field is written.

Rule 4 *VP P. writeSetc P.readSet*

Rule 5 VP P.*writeS'et*^<*j*>.

The intent of the *writeTime* field is to represent when a proposal is to take effect: that is, when it will officially write out new values. The uniqueness of time-stamps ensures that each proposal has a unique *writeTime.* The strictly increasing nature of time-stamps ensures that the *writeTime* is greater than any of the *readTimes.*

Rule 6 VP VF 6 *P.writeSet, P.newValue[F].written = P.writeTime.*

Rule 7 $\forall P \, \forall F \in P.readSet$, $P.readTime[F] < P.writeTime$.

These characteristics simplify the statement of the algorithms. They also make a number of the proofs simpler because many degenerate cases cannot arise. Other observations can be made concerning the *COPY_SET* field. Every update proposal identifies the set of copies associated with the object being updated. The *COPY_SET* field is used in determining which servers are to vote on the proposal. Accordingly, it is included in the *readSet* for every update proposal.

Rule 8 $\forall P$ $COPY\_SET \in P.readSet$.

Because the majority vote process requires current knowledge of the *COPY_SET* field, any change to the *COPY_SET* field interacts with modifications to other fields of the same object. Updates might be forgotten if copies are deleted or lose influence. To prevent this, the *writeSet* is required to include all fields if it includes the *COPY_SET*. Normally, the previous values of fields will be re-written.

Rule 9 $\forall P$ if $COPY\_SET \in P.writeSet$, then $\forall F \; F \in P.writeSet$.

Other conventions also produce correct behavior. However, they must all ensure that an update modifying the *COPY_SET* interferes with all other updates. Rule 9 is simple to implement, clarifies what initial value should be assigned to a copy, and buys as much concurrency as can be obtained while ensuring correctness. It has the drawback that if objects are large, the update proposal will be as well.

Notice that if a new copy is being created, it starts off with the version of the object used by the coordinator when proposing the creation. This shows that the majority vote algorithm satisfies Assumption 22 .

Observation 23 If C_i creates copy **A**, then $\forall F \; F \in C_i.writeSet$ and $\mathbf{A}[F]$ starts off with $\mathbf{A}[F] = C_i.newValue[F]$.

A.4.2 The Voting Algorithm

The majority vote algorithm as used by Taliesin differentiates between the roles of the *coordinator*, the server who initiates a proposal, and the ordinary *voters*, all other servers having a copy of the object being modified. The data structures used by the coordinator are initialized as per Program Fragment 4.

The coordinator fills in the update proposal proper in accordance with the procedure in Program Fragment 5. The coordinator uses the current values for the object from its local version to compute the new values for those fields to be modified. The new version plus information identifying what version was read are stored in the update proposal.

Inspection of Program Fragment 5 shows that a number of the earlier rules are obeyed. In particular, the *written* value of each new field is equal to the update's *writeTime*. The *writeSet* is truly a subset of its *readSet*. The *COPY_SET* field is in the *readSet* of every update proposal.

The proposal is circulated to all voters. A voter is started up when a message regarding an unknown proposal is received. Because messages may not be received in the order of sending, the voter must wait until it receives the proposal message. It then casts its vote and waits to be told of the outcome of the election. Should a voter be asked to vote a second time on the proposal, it must remember and report the vote it first cast.

Both the voters and the coordinator cast their votes in accordance with procedure **ComputeVote** in Program Fragment 6. The voting process is designed to ensure that exactly one new version of any object's field is created by modifying an older version.

Should a voter find itself out of date, it tries to bring itself up to date by querying the coordinator of the proposal. To avoid producing the effects of non-atomic writes, such

Program Fragment 4

```
function InitiateVote(objectType: ObjectType;
                      name: ObjectName): CoordinatorRecord;
  var
    aCopy: integer;
    value: ObjectValue;

  begin
  with InitiateVote do begin
    value := FindObject(name, objectType);
    proposal := InitProposal(objectType, name, value);
    numVoters := valuefCOPYJJETj.data.numCopies;
    votesTotal := valuefCOPYJSETj.data.totalVotes;
    votesPass := 0;
    votesFor := 0;
    for aCopy := 0 to numVoters do
      with value[COPY.SET].data.copies[aCopy]| voters[aCopy] do begin
        numVotes := votes;
        voter := location;
        voteCast := UNDECIDED;
        end;
    end;
  end;
```

queries must return the full state of the object, not just values for particular fields.

Rule 10 **If** a server responds to a query about the current value of an object, it will return the value of the entire object.

The voting rules require each voter to determine whether or not it has'cast a vote on another *conflicting* update. A conflicting proposal is one that touches a field touched by the new update or would cause a naming conflict.

Normally, it is also necessary to check to see if the respective *writeSets* have a non-empty intersection. Rule 4 allows this test to be omitted. If the *writeSets* have a non-empty intersection, so will either *readSet* with the other *writeSet* since a *writeSet* is a subset of the corresponding *readSet.*

The votes are tallied by the coordinator. As each vote is received, the coordinator records it and checks to see if it can decide whether to abort or commit the update. If a majority approves the update, the coordinator commits the update. If a single node has valid reason for rejecting the update, the coordinator aborts. Deadlocks are avoided by also aborting updates that cannot garner a majority of *YES* votes.

When the coordinator reaches a decision, it informs all of the voters. Each voter follows the coordinator's instructions as to whether to abort or commit the update. After committing or aborting, a voter may forget about the proposal and its vote.

Time-stamps are used as the measure of when an update logically takes place. The rules for voting and for writing committed updates ensure that versions are created with progressively later time-stamps. As shown in Program Fragment 8, a new value is only written over a previous version, where *previous version* is defined in terms of time-stamps.

The value of any field of an object may be modified only in response to an update proposal approved through the majority vote algorithm. Hence, the only value an object can take on is either its original one or one proposed by a committed update.

Rule 11 VF **VA** $3t \geq 0$ such that $A[F] = d.newValue[F]$.

Program Fragment 5

```
function InitProposal(objectClass: ObjectType; name: ObjectName): UpdateProposal;
    var
        aField: FieldCode;
        value: ObjectValue;

    begin
    value := FindObject(name, objectClass);
    with InitProposal do begin
        updateId := GenerateIdentifler(UPDATEJD.TYPE);
        proposer := LocalServerId;
        status := VOTING;
        writeTime := GenerateIdentifier(TIME«STAMP.TYPE);
        class := objectClass;
        name := name;
        readSet := { COPYJSET };
        readTime[COPYJET] := value[COPY-SET].written;
        writeSet:=<f>;
        for all aField do
            if WantToChangeField(aField) then begin
                readSet := readSet + f aField };
                readTime[aField] := value[aFieldl.written;
                writeSet := writeSet +.{ aField };
                newValuefaFieldl.written ;= writeTime;
                newVa!ue[aField].data := ComputeNewValue;
                end
            else if UsedFieldValue(aField) then begin
                readSet := readSet + l aField };
                readTime[aField]. := value[aField].written;
                end;
        end;
    end;
```

Program Fragment 6

```
function ComputeVote(update: UpdateProposal): VoteType;
    var
        aField: FieldCode;
        otherUpdate: UpdateProposal;
        value: ObjectValue;

    begin
    for all aField do begin
        value := FindObject(update.name, update.dass);
        with update, valuefaField] do
            if aField in readSet then begin
                if readTime[aField] < written then begin
                    ComputeVote := NO;
                    return;
                    end
                else if readTime[aField] > written then begin
                    QueryCoordinatorValue(class, name, proposer);
                    ComputeVote := UNDECIDED;
                    return;
                    end;
                end;
        end; (* end for all *)

(* Time-stamps are okay. Look for conflicts: *)

    ComputeVote := YES;
    for all otherUpdate do
        if ConflictTest(update, otherUpdate) then begin
            if update.writeTime < otherUpdate.writeTime then begin
                ComputeVote := PASS;
                return;
                end
            else if update.writeTime > otherUpdate.writeTime then
                ComputeVote := UNDECIDED;
            end; (* end if conflict *)
    end;
```

Program Fragment 7

```
function ConflictTest(considering, other: UpdateProposal): boolean;
    begin
    with considering do
        if (updateId = other.updateId) or (class <> other.class) or
                (name <> other.name) or (other.status <> VOTING) or
                (OwnVote(other) <> YES) then
            ConflictTest := false
        else if (readSet * other.writeSet <> 4>) or (writeSet * other.readSet <> <t>) then
            ConflictTest := true
        else if WouldCauseNameConflict(considering, other) then
            ConflictTest := true
        else ConflictTest :=• false;
    end;
```

Program Fragment 8

```
procedure LogVoteCast(var voteRecord: CoordinatorRecord;
                          voteGiven: VoteType; voterId: ServerIdentifier);
    begin
    with voteRecord, voteRecord.voters(voterId) do begin
        if (voteGiven = UNDECIDED) or (voteCast <> UNDECIDED) then
            return;
        voteCast := voteGiven;
        case voteGiven of
YES:        begin
            votesFor := votesFor + numVotes;
            if 2 * votesFor > totalVotes then
                proposal.status := COMMITTED;
            end;
NO:         proposal.status := ABORTED;
PASS:       begin
            votesPass := votesPass + numVotes;
            if 2 * votesPass >= totalVotes then
                proposal.status := ABORTED;
            end;
        end; (* case *)
        end;
    end;

procedure WriteUpdateValue(committedUpdate: UpdateProposal);
    var
        aField: FieldCode;
        value: ObjectValue;

    begin
    with committedUpdate do begin
        value := FindObjectfname, class);
        for all aField in writeSet do
            if (CreateSelffcommittedUpdate) or (value[aField].written > writeTime)) then
                value[aField] := newValue[aField];
        end;
    end;
```

Rule 1 requires that the local counter used to generate time-stamps be advanced to correspond to message receptions. The only messages that describe events are update proposals and query responses. The time-stamp associated with an update is the *writeTime* of the proposal. The time-stamp associated with a query response is the maximum *written* value of any field of the object described. Coordinators and voters are responsible for obeying the rule when proposals and query responses are received.

A.4.3 Correctness of Majority Vote

Taliesin uses what is in many ways a simplified version of Thomas' algorithm. Some of the correctness properties of the basic algorithm are not affected by changing the copy set. These will be assumed to hold and will be used to prove that the act of changing the set of copies preserves correct behavior. The first property that will be taken for granted is progress.

Theorem 24 Every update eventually commits or aborts, but not both.

Majority voting is correct in the sense that executing its legal schedules is equivalent to applying the committing updates in increasing order of their time-stamps to a single copy, provided the set of copies remains unchanged. Accordingly, the choice of indices is based on time-stamp orderings. Rule 12 constrains the choice of indices. Any total ordering satisfying this rule will do. An example of a suitable ordering would be to order all updates in the order of increasing *writeTimes.*

Rule 12 If *Pi.writeTime* < *Pj.writeTime,* then choose $i < j$ if either *Pi.writeSet* n *Pj.readSet* ^ <£ or *Pi.readSet D Pj.writeSet* ^ <£.

The uniqueness of time-stamps means that it is possible to determine if two copies have the same version of **a** field by simply comparing their *written* values. Observation 25 will be used frequently, often without explicit citation.

Observation 25 *A[F]* and B[F] are derived from the commitment of the same update iff *A[F].written = 3[F].written.*

Proof If *A[F].written* = To, then by Rule 2 and Rule 11, it must be that both versions are the original version 0o- Both are derived from update *Co.*

Otherwise, Rule 11 requires there to be committing updates $C_{\text{Ío}}$ and Ci_b that wrote A[F] and B[F], respectively. The procedure for initializing proposals in Program Fragment 5 specifies that each proposal is tagged with a newly generated time-stamp. Observation 20 then implies that Ci_a*.writeTime* = Ci_b*.writeTime* iff $C\{_a$ and $C.i_b$ are the same update. Rule 6 extends the requirement that the updates be the same to the case when *A[F].written = 3[F].written.* •

The equivalence of legal sequences of committing updates to serial execution in time-stamp order is used in several forms. The following results characterize properties that will be used to extend the correctness to include the effects of changes to the set of copies.

Lemma 26 VF VA, let {A*[JF] : $k > 0$} be the sequence of versions seen successively at A: that is, A*[F] is replaced by A*+I[JF]. Let *A* be the mapping defined by *A(k)* =? *i* if A_k*[F] = d.newValue[F].* Then $k < I$ implies *A(k) < A(l).*

Lemma 27 If *F* € *Cj.readSet, 3i* such that $i <$ j, *F* € *Ci.writeSet,* and the coordinator of *Cj* and all copies that vote *YES* on *Cj* have the field value O;[F] at the time they initiate/vote on *Cj,*

Lemma 28 VF Vm Vn, let ^(m, n) be the set of updates writing field *F*

based upon reading $O_n[F]$ and $O_m[COPYSET)$. Then at most one P *e T{m, n)* ever commits.

Lemma 29 *\/F* Vm Vi, if *F?* and JFJJk exist:

$$J\,F \wedge .readrirae[F] = F^{TM}_{\cdot}.writeTime$$

Lemma 29 characterizes one-copy serializability, but is constrained to the case in which no changes are made to the set of copies. The goal is to show that it extends across changes to the copy set. The first step is to verify that the updates changing the *COPYJSET* field serialize.

Lemma 30 $\mathbf{Vm \geq 0\ C^{m\wedge 1}.readTime[COPYJSET] = C^m.writeTime.}$

Proof Since Rule 8 says that Vm > 0 *COPYJSET* € C^m*.readSet,* it is meaningful to talk of C^m*. read Time/COP YSETJ.* The result will be proved by induction. The inductive hypothesis is that the claim holds for all $m \leq N$. Keep in mind that *M* was defined so that $m < n$ iff *M,{pi) < ftA(n).*

Case 1: $N = 0$.

Lemma 27 implies C^1 must have been drawn up using some version *Oj[C0PYJSET* with $j < M\{\backslash)$. The definition of C^1 says the only committing update writing the *COPYJSET* field with a lesser index is C_o. So, $j = 0 = M(0)$ as desired.

Case 2: Vm € [0, *N)* the claim is true.

The goal here is to show that the hypothesis is true for m = *N*. By Lemma 27, C^{N+1} must be based on version *Oj[COPYJSET]* for some choice of $j \leq M(N + 1)$. Since the only updates that modify the *COPYJSET* field are those in $\{C^n\}$, $j = M(l)$ for some /.

Both C^{N+1} and C^{l+1} are committing updates writing the *COP YJSET* field. Both read the version of the *COPY-SET* written by C_j. Since only one can commit according to Lemma 28, they must be the same proposal: i.e., / = *N*. The inductive hypothesis extends to all m € [0, JV + 1). D

Knowing that changes to the copy set serialize makes it possible to attack the problem of how other fields behave. All other updates read the *COPYJSET* field. Lemma 31 shows that any update reading a particular version of the *COPYJSET* field will have an index falling between the update that wrote that version and the next update writing the ***COPYSET* field.**

Lemma 31 For any j, *Brrtj* such that $M\{rrij) < j \leq M\{rrij + 1)$ and $C_j.readTime[COPY\_SE \backslash = C^{m_j}.writeTime.$

Proof Lemma 27 ensures that $C^{m>}$ exists and that $M(rrtj) < j$. Trivially, If $M(mj + 1) = j$, $j \leq M(mj + 1)$. Suppose, then, that j ^ $M(mj + 1)$.

Lemma 30 says that the coordinator of $C^{m>+1}$ reads $C^{m}*[COPYJSET\backslash$. So, the same set of servers, with the same copy weightings, approved both *Cj* and £my+i. There must be some copy V that voted *YES* on both. Rule 8 says that *COPYSET e Cj.readSet* Hence, *Cj* and $C^{m>+1}$ are in conflict as defined in Program Fragment 7. V is not permitted to vote on the second proposal received until it knows that the first has committed.

Consider what happens if V sees $C^{m>+1}$ first. It defers voting on *Cj* until $C^{m>+1}$ **commits.** ***VfCOPYSETj.written > Cj.readTimefCOPYJSET]*** **at that** point, so Program Fragment 6 says V must vote *NO* on *Cj*. V actually votes

YES so it must commit *Cj* before voting *YES* on C^{m+1}.

Cj.writeTime $\leq$ $C^{m>+1}$*.readTime[COPYJIET]* since V commits Cj before voting YES on $C^{m>+1}$. According to Rule 5, *3F* G *Cj.writeSet* Rule 9 says that *F £ C^m{^1,WT%teSet* as well. The relation between *readTime* and *writeTime* values given in Rule 7 and the index choice of Rule 12 force *j < A4(rrij* + 1). •

At this point, it has been verified that the *{C^m}* appear as if serially applied to a single copy of the *COPY-SET* field. The indices of updates reading a particular version of the copy set fall in the interval between the index of the update that wrote that version and that of the next modification to the copy set. These facts are pulled together in Lemma 33 to prove that the serializability of the ordering extends across changes to the copy set. A useful relationship among indices is derived first in Lemma 32.

Lemma 32 VF, let *{F{}* denote the subsequence of updates writing *F*. Define the mapping *V* by choosing *V{%)* = (m,j) iff i** = FJ^71. Consider any *V(i)* = (m, *k)* and *V(j)* = (n, *I).* Then *i < j* iff either m < *n* or m = *n* and *k < I.*

Proof Note that the sequence $\{F_t\}$ is just some re-ordering of the sequences *{FT}.* Consider *T>{%)* = (m,fc) and *V(j)* = (n,/), where *n* > m. Lemma 31 says *M(m) < i* $\leq$ *M(m* + 1) and *M(n) < j* $\leq$ *M(n* + 1). Lemma 30 gives *M{m) < M(n).* So, *i < j .*

Consider *V(i)* = (m,fc) and *V(j)* = (m,/) with *k < I.* Lemma 29 and the choice of indices force *i < j* again. In one direction, the result is true.

Suppose *i < j.* Were *m > n,* the claim just shown would require *j < i.* Similarly, were m = *n* and *k* > /, the claim would require *j < i.* The transactions *F{* and *Fj* are distinct and so cannot both be based on the same versions of the fields according to Lemma 28 . Thus, the only possibilities are the either *m <n* or that *m = n* and *k < I.* •

Theorem 33 VF, let $\{F_t\}$ denote the subsequence of updates writing *F. Then ViFi.writeTime = Fi+i.readTime[F].*

Proof Use the mapping *T>* defined in Lemma 32. Let *T>{%)* == **(m,jf)** and *D{%* + 1) = (rc, *k).* Lemma 32 allows only two possible relationships between *mjUjj^* and *k.* The first is that *m* = n and *j < k.* Lemma 29 applies, yielding *Fi+\.readTime[F) = Fi.wrUeTime.*

The other possibility is that *m < n.* If n ^ m + 1, C^{m+1} would write *F* and by Lemma 31 *M(m* + 1) < *M(n).* However, the choice of indices for *F>* and JFVn means there is no intermediate update writing *F.* Similar reasoning prevents *k* ^ 1 and *Fi* ^ C^{m+1}. Consider what *F^.readTime[F]* might be. The coordinator of *F?*^1* has written the committed value, *O^m[C0PYSET\.* By the atomicity of writes (due to commit or query), the coordinator must also have written the value *O^m[F].* So, *F^writeTime = F_{i+1}.readTime[F].* D

Since all updates to all fields appear to have been executed in the same order and since writes are atomic, the majority vote algorithm produces the same effect as a serial execution of the committing updates on a single copy.

A.5 Notice Replication

The correctness of the notice replication algorithm is not based on the one-copy serializability of its updates. Instead, its behavior will be deemed correct if the bulletin board

presents the appearance of having a single copy to the user. The illusion of one-copy equivalence requires that the following properties be present:

1. A regular reader sees every undeleted notice posted to the bulletin board.
2. A regular reader never sees a notice more than once.

Normally, it would also be required that replies be dated after the notices replied to. However, a notice specifies what notice it is in reply to. That means a user agent can sort the notices read to present the original notices first.

Taliesin does not provide completely correct behavior in the sense defined above. The notice replication of Taliesin provides correct behavior under most circumstances, but can present notices twice when copies are deleted. It can also hide notices if copies are perpetually being deleted at a rapid rate.

A.5.1 Bulletin Board Structure

Notice replication uses several attributes of a bulletin board, defined in Program Fragment 9. Naturally, it uses the store of notices and deletion markers. The three epochs values and their bounds are likewise updated as part of the reconciliation process. The *copySet* field is used to determine what other copies exist. Events on the time-stamp and time-scales are kept in step through the use of the *history* and *pendingVotes* fields.

A single starting copy of a bulletin board is created when the bulletin board is created. Program Fragment 10 indicates how that copy is set up. Since a single copy is created, Assumption 21 is trivially satisfied.

Rule 13 A single initial copy of a bulletin board is created.

Note that postmasters are responsible for keeping the *pendingVotes* field properly defined at all times. The field must be modified whenever certain update proposals are coordinated, received, committed, or aborted. Rule 14 indicates which proposals are kept in this field.

Rule 14 Suppose P has $COPY\_HISTORY \in P.writeSet$ and that P was either coordinated by **C** or received by **C**. Then $P \in \mathbf{C}.pendingVotes$ iff all messages seen by **C** indicate $P.status = VOTING$.

Additional copies are created at the command of clients. The new copies are initialized using values taken from the coordinating copy, but with some modifications to the control information, as shown in **NewBBoardCopy**. The end result is that Assumption 22 is satisfied. It also ensures that copies do not act until after their official creation time.

Rule 15 If **C** is created effective epoch E, $\mathbf{C}.postEpoch \geq E$.

The procedure for creating the new *copySet* and *history* fields is shown in Program Fragment 11. Note that the *postingEpoch* is incremented as part of the procedure. This behavior helps ensure that notice replication makes progress. It also ensures that the *postEpoch* is at least as great as any epoch issued in a message by the coordinator.

Observation 34 A server handling one copy of a bulletin board may make at most one change in the set of copies before advancing its posting epoch.

Another important property of this procedure is that the events added to the *history* field really do correspond to changes in the *copySet*, as required by Rules 16 and 17.

Rule 16 **A** will not propose to create copy **B** unless $\mathbf{B} \notin \mathbf{A}.copySet$. Similarly, **A** will not propose to delete **B** unless $\mathbf{B} \in \mathbf{A}.copySet$.

Program Fragment 9

```
type
    EnvelopType = record
        noticela: IdentifierType;
        signEpoch: EpochType;
        signer: ServerIdentifier;
        numSignings: integer;
        end;
    NoticeType = record
        signature: EnvelopType;
        body: Uninterpreted;
        end;
    EventType = (CREATION, DELETION);
    EventRecordType = record
        copyId: ServerIdentifier;
        voteEpoch: EpochType;
        effectEpoch: EpochType;
        eventType: EventType;
        end;
    HistoryFieldType = record
        written: TimeStamp;
        numEvents: integer;
        events: array of EventRecordType;
        end;
    BBoardType = record
        expungeEpoch,
        agreehpoch,
        postEpoch: EpochType;
        postBounds,
        agreeBounds: array of EpochType;
        notices: array of NoticeType;
        markers: array of EnvelopType;
        copySet: FieldType;
        history: HistoryFieldType;
        pendingVotes: array of UpdateProposal;
        (* other, irrelevant, fields *)
        end;
```

Program Fragment 10

```
procedure FirstBBoardCopy(var BBoard: BBoardType; suppliedValues: BBoardType);
    begin
    BBoard := suppliedValues;
    with BBoard do begin
        postEpoch := 1;
        agreebpoch := 0;
        expungeEpoch := 0;
        agreeBounds(LocalServerId) := 0;
        postBounds( LocalServerId) := 1;
        notices := <f>\
        markers := <\;
        copySet.written := GenerateIdentifier(TIMEJSTAMP.TYPE);
        (.* all fields set  .written := copySet.written *)
        history.numEvents = 1;
        history.events[0].eventType := CREATION;
        history.events[0].copyId := LocalServerId;
        history.events[0].voteEpoch := 0;
        history.events[0].effectEpoch := 1;
        pendingVotes := <f>;
        end;
    end;

function NewBBoardCopy(parentBBoard: BBoardType,
                               parentId: ServerIdentifier): BboardType;
    var
        createEvent: EventRecordType;

    begin
    NewBBoardCopy := parentBboard;
    with NewBBoardCopy do begin
        createEvent := history.events[history.numEvents];
        postEpoch := createtvent.efrectEpoch;
        postBounds(LocalServerId) := createEvent.effectEpoch;
        pendingVotes := <j>\
        end;
    end;
```

Program Fragment 11

```
(* Assume that the copy set, time-stamps are already set *)

procedure AlterExistingCopies(bbName: BBoardName,
                              var newBBoard, oldBBoard: BBoardType,
                              action: EventType; alterCopyId: ServerIdentifier);
   var
      newEvent: EventRecordType;

   begin
   with newBBoard, newEvent do begin
      eventType := action;
      copyId := alterCopyId;
      voteEpoch := postEpoch;
      case (action) of
CREATION:
         if alterCopyId in copySet.data.copies then
            AbortWithError?'Copy already exists');
         efFectEpoch := posttpoch •+ 1;
         postBounds{alterCopyId} := postEpoch;
         readBounds(alterCopyId) := readEpoch;
DELETION:
         if not alterCopyId in copySet.data.copies then
            AbortWithError('Copy does not exist');
         if not VerifyPriorIncarnationDead(bbName, alterCopyId) then
            AbortWithError('Prior incarnation may exist');
         efFectEpoch := maximum(agreementEpoch + 1,
               CopyCreateEpoch(oldBboard, alterCopyId));
      end;
      history.numEvents := history.numEvents.+ 1;
      history.events[history.numEvents] := newEvent;
      end;
   with oldBBoard do begin
      postEpoch := posttpoch + 1;
      postBounds(LocalServerId) := postBounds(LocalServerId) + 1;
      end;
   end;
```

Rule 17 *COPY-HISTORY* € *P.writeSet* iff *COPYJSET* G *P.writeSet* and the set of copies voting on *P* is not the same as the set of copies in *P.newValuefCOPYSETj.*

The proposed new value for the bulletin board must be approved through the majority voting algorithm. Notices, deletion markers, epochs, and epoch bounds are not included in proposals. However, since epochs are clock values, to obey Rule 1, each recipient of a proposal to modify the history field must advance its *postEpoch* to match the latest epoch mentioned in the new version of the *history* field.

Rule 18 If copy A receives *P* and *COPYJSISTORY* £ P.writeSet, A must advance both *A.postEpoch* and *A.postBounds(A)* to at least *max{H.voteEpoch,H.effectEpock* : *H* G *P.newValuefCOPYJIISTORYj}.*

When changes to the *history* field are committed, some actions must be taken. The coordinator of the update is responsible for seeing to the initialization of any newly created copies. The initial values are based upon the version of the bulletin board returned by **AlterCopySet.** When copies are deleted, notices may have to be re-submitted. Discussion of the re-submission procedure will be deferred until after basic correctness has been proven.

A.5.2 Notice Operations

When a notice is posted to a bulletin board, the postmaster receiving the notice from the user agent assigns it unique identifier and clears the *signature* field. The notice is initially stored in the *notices* field of a single copy. The postmaster implementing that copy fills in the *signature* to indicate when and where the notice was first received as per Program Fragment 12. If an unsigned notice is received as part of any other operation, the same signing procedure is followed.

After the signature is filled in, the postmaster must check to make sure that a copy of the notice or deletion marker has not already been stored. The test given in Program Fragment 14 is based on an ordering of the signatures. Of those versions belonging to the same signing number, the earliest is kept. However, one with a later signing number supercedes earlier ones.

Using these rules for signing notices and avoiding storing duplicates, the procedure for posting and deleting notices can be defined/Program Fragment 12 explains the procedure for posting a notice. *isFirstSubmission* is true if the notice came directly from a user agent.

Deleting notices is handled much like posting notices. The first postmaster with a copy of the bulletin board creates a *deletion marker* to record the act of deletion. This is stored in the bulletin board in much the same fashion as a notice. However, it has the side effect of deleting all occurrences of the notice.

The operation of reading new notices is based upon selecting those notices that were received since the previous time the user read the bulletin board. The *agreeEpoch* is returned as the time of the read. Anew *postEpoch* must be generated at the copy read to ensure that postings after the read can be distinguished from those occurring before it. This procedure for handling a read notice operation is explained in Program Fragment 15.

The usefulness of the *agreeEpoch as* a measure what a user has seen depends critically upon the fact that the *postEpoch* is incremented. Observation 35 points out that the increment makes it possible to distinguish between notices received before and after the read.

Observation 35 **If** a user reads A when *A.postEpoch = E,* thereafter no notice, deletion marker, or history event is dated by copy A with an epoch less than *E* + 1. That is, if *signature* is such a signature, *signature.signEpoch > E.* If *event* is a new event added to the history, *eventvoteEpoch > E.*

Program Fragment 12

```
function InitSignature(var signature: EnvelopType; neverSigned: boolean);
    begin
    with signature do begin
        signEpoch := UNDEFINED-EPOCH;
        signer := UNDEFINED-NODE;
        if neverSigned then
            numSignings := 0;
        end;
    end;

procedure SignIfNeeded(var signature: EnvelopType; BBoard: BBoardType);
    begin
    with signature, BBoard do
        if (signEpoch = UNDEFINED-EPOCH) or
                UlegalSigned(signature, BBoard) then begin
            signEpoch := postEpoch;
            signer := LocalServerId;
            numSignings := numSignings + 1;
            end;
    end;

procedure PostNotice(newNotice: NoticeType; BBoard: BBoardType;
                        isFirstSubmission: boolean);
    begin
    with (newNotice, BBoard) do begin
        if isFirstSubmission then begin
            noticeId := GenerateIdentifier(NOTICEJD-TYPE);
            InitSignature(signature, true);
            end;
        if not HaveCopy(BBoard) then
            ForwardToCopy(notice, BBoard)
        else begin
            SignIfNeeded(signature, BBoard);
            if not (noticeId in BBoard.markers or
                    Superceded(signature, BBoard, true)) then begin
                notices := notices - { noticeId };
                notices := notices + { newNotice };
                end;
            end;
        end;
    end;
```

Program Fragment 13

```
procedure Delete!Motice(toDelete: EnvelopType; BBoard: BBoardType;
                        firstSubmission: boolean);
    begin
    with toDelete, BBoard do begin
        if firstSubmission then
            InitSignature(toDelete, true);
        if not HaveCopy((BBoard) then
            ForwardToCopy(toDelete, BBoard)
        else begin
            SignIfNeeded(toDelete, BBoard);
            if not Superceded(toDelete, BBoard, false) then begin
                markers := markers - { noticeId };
                notices := notices - { noticeId };
                markers := markers + { toDelete };
                end;
            end;
        end;
    end;
```

Program Fragment 14

```
function RejectTest(given, other: EnvelopType): boolean;
    begin
    with given do
        if noticeId <> other.noticeId then
            RejectTest := false
        else if numSignings < other.numSignings then
            RejectTest := true
        else if numSignings > other.numSignings then
            RejectTest := false
        else if signEpoch > other.signEpoch then
            RejectTest := true
        else if signEpoch < other.signEpoch then
            RejectTest := false
        else if signer >= other.signer then
            RejectTest := true
        else RejectTest := false;
    end;

function Superceded(signature: EnvelopType; BBoard: BBoardType;
                    testNotices: boolean): boolean;
    var
        marker: EnvelopType;
        notice: Notice Type;

    begin
    Superseded \= false;
    if testNotices then
        for all notice in BBoard.notices do
            Superceded := Superceded or RejectTest(signature, notice.signature);
    else for all marker in BBoard.markers do
        Superceded := Superceded or RejectTest(signature, marker);
    end;
```

Program Fragment 15

```
type
    ReadReplyType = record
        readEpoch: EpochType;
        readNotices: array of NoticeType;
        end;
    Read Record = record
        readTime: EpochType;
        wantToSee,
        seen Before: array of IdentifierType;
        end;

procedure ReadNewNotices(var BBoard: BBoardType; priorRead: ReadRecord);
    var
        read Reply: ReadReplyType;
        N: NoticeType;

    begin
    with BBoard, readReply, priorRead, N do begin
        readEpoch := agreeEpoch;
        postEpoch := postEpoch + 1;
        postBounds(LocalServerId) := postEpoch;
        readNotices := { N 6 notices: (noticeId € wantToSee) or
                (noticeId ^ seen Before and signature.signEpoch > readTime };
        SendReadReply(read Reply);
        end;
    end;
```

Actually, Observation 35 depends for its validity on more than just the increment of the *postEpoch* field. It also depends on the fact that *postEpoch* never goes backward.

A.5.3 Reconciliation

Copies of a bulletin board converge by periodically sending reports of changes to all other copies. These *reconciliation reports,* denned in Program Fragment 16, include information on notice postings, known lower bounds for the *postEpochs* and *agreeEpochs* of all copies, and the fields replicated using majority vote.

If B sends A a reconciliation report, A will be said to have *directly* received a report from B. This notion is extended inductively. If C has directly or indirectly received a report from B and then sends a report to A, A is said to have *indirectly* received a report from B.

The process of filling out a reconciliation report is quite simple. The composer just copies values from the bulletin board to the report, as shown in Program Fragment 16. A postmaster includes all of the notices and deletion markers that any other copy might not have seen yet. The report is forwarded to every other postmaster handling a copy of the bulletin board, although it need not be sent directly there.

If A composes report JJ, *R.othersPost(A)* is the epoch associated with the message *R.* Again, the practice of incrementing the *postEpoch* field after a report has been generated ensures that any subsequent actions timed on the epoch time-scale occur after the report has been sent. In particular, new history events are dated with an epoch later than that given in any report. Similarly, notices and deletion markers are signed at later epochs. Observation 36 states these behaviors more precisely.

Program Fragment 16

```
type
    ReconcileReport = record
        bbName: BBoardName;
        newNotices: array of NoticeType;
        newDeletes: array of SignatureType;
        othersPost: array of EpochType;
        othersAgree: array of EpochType;
        newCopySet: FieldType;
        newHistory: HistoryFieldType;
        knownVotes: array of TransactionType;
        (* all other fields of a bulletin board *)
        end;

function MakeReport(name: BBoardName): ReconcileReport;
    var
        BBoard: BBoardType;
        N: NoticeType;
        D: EnvelopType;

    begin
    BBoard := FindBBoard(name);
    with MakeReport, BBoard, N.signature do begin
        bbName := name;
        newNotices := { N in notices: signEpoch > expungeEpoch};
        ExpungeMarkers(markers);
        newDeletions := { D in markers: D.signEpoch > expungeEpoch};
        othersPost := postBounds;
        othersAgree := agreeBounds;
        postEpoch := postEpoch + 1;
        postBounds(LocalServerId) := postEpoch;
        newCopySet := copySet;
        knownVotes := pendingVotes;
        if (newHistory.written > history.written) then
            CommitHistory(newHistory, BBoard, false);
        (* copy all other fields into the report *)
        end;
    end;
```

Observation 36 If **A** sends report *R* with *R.othersPost(A)* = E, then:

1. Any history event *H* created thereafter by A has *H.voteEpoch* > *E.*
2. Any report *R* sent out thereafter by A has *R.othersPost(A)* > *E.*
3. Any notice or deletion marker signature *S* signed thereafter by A has ***S.signEpoch*** > ***E.***

Note that the act of creating and initializing a new copy cannot be distinguished from creating the copy and sending it a reconciliation report based on the state of the coordinator at the time the proposal to create was composed. Accordingly, when subsequent claims assert that some copy A has received a reconciliation report from B, either B did send a report or B created **A.**

Observation 37 Suppose A coordinates a proposal to create B, the proposal commits, and A initializes B. There is no distinction in the state of B and what it would be if A had also sent B a reconciliation report containing the new version of the *copySet* and *history* fields and plus all the other state of A when ***A.postEpoch = IZ.postBounds(A).***

Processing a reconciliation report from another copy is more complicated than composing one. It is necessary to factor in the effects of remote events and to update the epochs and epoch bounds. Program Fragment 17 describes what must be done.

Note that the implementation of Program Fragment 17 together with Rule 18 ensures that the *postEpoch* field satisfies the condition specified in Rule 1. It never lags behind the largest epoch value reported by any other copy. The fact that a copy increments the *postEpoch* when composing a new history event likewise guarantees that it is at least as large as any epoch mentioned in a message it generates.

If copies are to be effectively brought up-to-date by indirectly received reconciliation reports, a report really must encapsulate the information received by all reports previously received by its composer. Lemma 38 shows that this is truly so. In subsequent proofs, this result will normally be used without being explicitly cited.

Lemma 38 Consider the two scenarios: B and C send A a reconciliation report versus B sends C a report then C sends A a report. Suppose that B and C do not learn of the commitment of any pending updates during the time these scenarios occur. Then the final state of A is the same in both cases.

Proof Because both courses of action are considered possible, it must be that A,C € *A.copySet* and A G C.*copySet* Copy deletion procedures will not be invoked.

Consider any field *F*. From the report composing and processing procedures, it is clear that in either situation the final version of *A[F]* will be whichever of B[F], C[F], and the original version of *A[F]* have the most recent *written* value.

The pending votes known to the final version of A include all of those known to B and C.

Consider any notice or deletion marker. If it is present in only one of B and C, A will wind up with a copy either way. If different versions are present in both, C wiU make the same decision as to which to keep as A would given both. A has the same notices, too. •

The fact that message orderings and delays can vary means that the timing and routing of reconciliation reports can affect when votes are cast. This may influence the outcome and the timing of when committed values are written. However, such variations are inherent due to delays in the majority vote messages. If a pending vote is committed, subsequent reports include its effects in the values of the bulletin boara fields rather than as a proposal.

Program Fragment 17

```
procedure StoreReport(report: ReconciieReport);
    var
        BBoard: BBoardType;
        self Dele ted: boolean;
        notice: NoticeType;
        marker: EnvelopType;

    begin
    BBoard := FindBBoard(report.bbName);
    with (report, BBoard) do begin
        if copySet.written < newCopySet.written then
            copySet := newCopySet;
        if history.written < newHistory.written then
            WriteHistory(newHistory, BBoard, selfDeleted);
        (* Similarly update all other bboard fields *)
        for all updates in knownVotes
            RecordAndVote(update, BBoard, selfDeleted);
        if selfDeleted then
            return;
        for all notice in newNotices do
            PostNotice(notice, BBoard, false);
        for all marker in newDeletes do
            DeleteNoticef,marker, BBoard, false)
        for copy <> LocalServerId do begin
            postBounds(copy) := maximum(othersPost(copy), postBounds(copy));
            agreeBounds(copy) := maximum(othersAgree(copy), agreeBounds(copy));
            postEpoch := maximum(postEpoch, othersPost(copy));
            end;
        CalcAgreeEpoch(bbName, BBoard);
        Ca(cExpungeEpoch( BBoard);
        end;
    end;
```

Program Fragment 18

```
function NoAddPending(BBoard: BBoardType; atEpoch: EpochType): boolean;
    var
        aVote: UpdateType;
        event: HistoryFieldType;

    begin
    NoAddPending := true;
    with aVote, BBoard, event do
        for all aVote in pendingVotes do
            for all event in newValue[COPYJHlSTORY] - history do
                if (eventType = CREATION) and (effectEpoch <= atEpoch) then
                    NoAddPending := false;
    end;

procedure CalcAgreeEpoch(var BBoard: BBoard);
    var
        advanceOK: boolean;
        copies: set of ServerIdentifier;
        aCopy: ServerIdentifier;

    begin
    with BBoard do begin
        repeat
            advanceOK := NoAddPending(BBoard, agreeEpoch + 1);
            copies := CopySet(BBoard, agreeEpoch + 1);
            for all aCopy in copies do
                advanceOK := advanceOK and (postBounds(aCopy) >= agreeEpoch + 1);
            if advanceOK then
                agreeEpoch := agreeEpoch + 1;
        until not advanceOK;
        agreeBounds(LocalServerId) := agreeEpoch;
        end;
    end;
```

The two computed epochs, *agreeEpoch* and *expungeEpoch,* intuitively can be thought of as measuring the progress of the slowest copy. They are computed as the minimal values of arrays of known lower bounds on the epochs at other copies. The implementations in Program Fragments 18 and 19 shows that these epochs are computed as minimal values, but in a special sense. Potential changes to the copy set are taken into account. This ensures that the copies retain correct knowledge of the copies existing at any epoch. The key property providing the assurance is given in Observation 39.

Observation 39 **VA,** let E == *A.agreeEpoch.* Consider any B such that either B € *A.copySet* or *3P* € *A.pending Votes* proposing to create B on or before epoch 25 + 1: i.e., the new event H_p has H_p.*effectEpoch* $\leq E + 1$. Then ***A.agreeEpoch* advances to $E + 1$ only if *A.agreeBounds(B)*** $\geq E + 1$.

The correctness of notice replication follows in part from relationships between epoch values preserved when reconciliation reports are processed. In particular, the intended ordering between the three computed epochs and their bounds is achieved. The orderings are formally stated and proved in Lemma 40.

Program Fragment 19

```
procedure CalcExpungeEpoch(var BBoard: BBoard)
    var
        advanceOK: boolean;
        copies: set of ServerIdentifier;
        aCopy: ServerIdentifier;

    begin
    with BBoard do begin
        advanceOK := true;
        repeat
            copies := CopySet(BBoard, expungeEpoch + 1);
            for all aCopy in copies do
                advanceOK := advanceOK and (agreeBounds(aCopy) >= expungeEpoch + 1);
            if advanceOK then
                expungeEpoch := expungeEpoch + 1;
        until not advanceOK;
        ExpungeUselessInfo(BBoard);
        end;
    end;
```

Lemma 40 **VA** V **B** the following relationships always hold:

1. ***A.postBounds(B)*** $\leq$ ***A.postEpoch***
2. ***A.postEpoch.•= A.postBounds(A)***
3. ***A.agreeEpoch*** $\leq$ ***A.postBounds(J£)***
4. ***A.agreeEpoch*** = **A.*agreeBounds(*A)**
5. ***A.expungeEpoch*** $\leq$ ***A.agreeBounds(IZ)***

Proof The relationships are proved inductively over actions that modify epochs: reading notices, generating a report, altering the copy set, and receiving a report.

Initially, they hold since the first copy starts out choosing epochs according to Program Fragment 10 . The procedure for creating a new copy in Program Fragments 10 and 11 ensures that a copy created effective epoch *E* starts off satisfying these conditions providing they are true for the parent copy as of epoch ***E-l.***

Reading notices, altering the copy set, and generating reports only increment *A.postEpoch* and A.*postBounds(*A) simultaneously. Thus, they preserve the relationships. Inspection of the procedures in Program Fragments 17, 18, and 19 reveals that the relationships are also preserved when a report is received. •

The intuitive interpretation for the arrays of bounds is that they express knowledge of other copies' *postEpochs* and *agreeEpochs.* Lemma 41 proves that the procedure for handling reconciliation reports guarantees that a copy's *postBounds* are meaningful bounds on when other copies have sent reports in terms of their *postEpochs.*

Lemma 41 **VA** ^ **B, A** has directly or indirectly received **a** report from **B composed when *S.postEpoch = A.postBounds(B).***

Proof This result will be proved inductively over the events of copy creation and reconciliation report reception.

The claim is trivially true initially because only one copy is created.

When a new copy is created, its state is directly derived from the state of the coordinating copy. The inductive hypothesis and inspection of the code in Program Fragments 10 and 11 show that the claim is effectively true for a newly created copy in the light of Observation 37.

Suppose C sends A a report *R* and that the claim is true initially for both C and A. Let *E = C.postEpoch* at the time that C started to compose *R.*

If C = B, Program Fragment 16 forces *R.othersPost(B) .=. E.* The rules for updating *A.postBounds(B)* takes the maximum. Observation 36 implies that since the claim was true for A before, *A.postBounds(B)* will take on the value *E. A* directly received *R* from B = C.

If C 7^ B, the final value of *A.postBounds(H)* will be the maximum of its previous value and *E.* If it is unchanged, by the inductive hypothesis, the claim is still true. If not, the fact that A has received *R* means that by definition A has indirectly received any reconciliation report received by C when *R* was composed. The inductive hypothesis applies, saying that C had directly or indirectly received a report from B. The claim must be true after *R* is received. •

A similar result can be derived for the *agreeEpoch* and *agreeBounds* fields. The proof is the same except that different epoch fields are used.

Lemma 42 VA =^ B, A has directly or indirectly received a report from **B composed when *H.agreeEpoch = A.agreeBounds(B).***

A.5.4 Basic Correctness of Notice Replication

The description of the algorithm for notice replication is now complete enough that some fundamental correctness results can be shown. The first property to be demonstrated is that copies converge to correct knowledge of the copy set on the epoch time-scale. The following terminology is used in stating claims concerning the set of copies.

Whenever a copy A is mentioned, it should be assumed that the copy believes it exists: that is, A € *A.copySet* A copy A will be said to *exist* if every copy B with ***B.copySetwritten*** ≥ ***A.copySetwritten*** **has A € *H.copySetcopies.* Copy A will be said to** *exist at epoch E* if it exists and its most recent effective creation epoch is no later than *E. A potentially exists (at epoch E)* if there is some uncommitted, unaborted proposal *P* such that *P* would create A (at an epoch no later than *E).*

A copy A has *superset* knowledge of the copy set if for every other copy B, ***A.copySetwritten*** > **B.*copySetwritten* or *B.copySet copies* C *A.copySetcopies.* Similarly,** A would have super-set knowledge of all potentially existing copies if it has super-set knowledge of the copy set and for every *P* that would create a new copy, *P* € *A.pendingVotes.* Super-set knowledge of all copies (potentially) existing at a particular epoch just means that A only needs to know about those copies created on or before that epoch.

Computation of the *agreeEpoch* and *expungeEpoch* requires a copy to be able to determine what copies exist at any epoch after the current *expungeEpoch.* The *history* was designed to provide the knowledge for doing so, but as matters stand, there is no guarantee that the sequence of events make any sort of sense. Conceivably, a deletion might be preceded by a deletion on the epoch time-scale. Theorem 43 shows that successive changes to the set of copies on the time-stamp scale are ordered in the same fashion in the epoch scale.

Theorem 43 Let $\{C_{ak} : \mathbf{fc\text{-}} \geq \mathbf{0}\}$ be the subsequence of committed proposals creating or deleting copy A. Let H_{ak} denote the history event added by proposal C_{ak} pertaining to copy A. Then Vfc > 0:

1. H_{ak}.*effectEpoch* ≤ H_{ak}^.*effectEpoch.* **The inequality is strict if** H_{ak}. *event Type* = *DELETION.*
2. **If** *k* **is even,** H_{ak}.*eveniType* = *CREATION.*
3. **If** *k* **is odd,** H_{ak}.*eventType* = *DELETION.*

Proof Note that the choice of indices means that increasing index values correspond to increasing values of C_{ak}**writeTime.* Rule 17 guarantees that each proposal creating or destroying a copy writes the *copyHistory* field by adding an event to it.

Let B_{afc+1} denote the copy acting as coordinator for C_{ttfc+1}. By the serializability proved in Theorem 33, B_{aAf+1} has committed a version of the *history* field derived from that written by C_{ak}. Rule 18 requires B_{afc+1} to advance *Rak+i-postEpoch* to at least H_{ah}.*effectEpoch* when that version of the history field was coordinated/written. Observation 21 says the *postEpoch* field never rolls back. B_{ajt+1}.*postEpoch* ≥ H_{ak}.*effectEpoch* at the time B_{afc+1} filled in H_{ak}^$_t$ so H_{ak}^$_1$.*voteEpoch*>_ H_{ak}-*effectEpoch.*

Rule 16 forces the first event to be the initial creation of A.

Suppose H_{ak}. *event Type* = *CREATION.* Since A exists, Rule 16 forces the majority vote algorithm to approve only a deletion request next. So, *Hak.eventType* = *DELETION.* The procedure in Program Fragment 11 requires **that** H_{ak}^.*effectEpoch*≥H_{ak}.*effectEpoch.*

Suppose *Hi.eventType*^ *DELETION.* Then, since A does not exist, Rule 16 forces *Hak.eventType* = *CREATION.* The procedure for initializing history events requires that *Hak.effectEpoch* > H_{ak+1}.*voteEpoch.* Combining this inequality with the previously mentioned lower bound on H_{ak}^$_1$.*voteEpoch* yields J_$_{ak+1}$.*effectEpoch* > H_{ak}. *effectEpoch.* **D**

Theorem 43 shows that a copy can compute the set of copies existing at any particular epoch. The proper interpretation of the situation in which a creation and a deletion of a copy fall into the same epoch is that the copy was created then destroyed before it could take any actions.

A serious potential problem stemming from a copy's possible ignorance of its own deletion is that it might effectively create copies after its demise. While the effective epoch of a change might indeed be after the effective epoch of the deletion, word still reaches all copies. Lemma 44 verifies that the majority vote algorithm guarantees this.

Lemma 44 Consider a copy D that is created by C_{C1}, deleted by *C** and either never subsequently re-created or re-created by C_{C2}. Choose these committing proposals so that there are no other creations or deletions of D between C_{Cl} and *Cd* in time-stamp order. If D coordinates C* where ci < fc ≤ C2, then *k*_≤*d.*

Proof Theorem 33 says that the majority vote algorithm sequences proposals so that the effect is equivalent to a 1-copy serial schedule. Since D € D.*copySet* when D coordinates C*, *C*\ < *k*_≤ *d.* •

Theorem 43 and Lemma 44 show that changes to the copy set reach all copies, even if initiated by a retroactively deleted copy, and occur in a rational fashion on the epoch time scale. With such a guarantee, it is possible to prove a useful inductive hypothesis. Lemma 45 shows that super-set knowledge of the copy set is preserved as the agreement epoch is advanced.

Lemma 45 Let A be any copy and let *E* = *A.agreeEpoch.* If every copy has super-set knowledge of every copy existing at epochs up through J£, A will

likewise have super-set knowledge when *A.agreeEpoch* — **E + 1.**

Proof Suppose *A.agreeEpoch* == *E* +1. Consider a copy B existing at epoch *E* + 1. If it existed at epoch 22, copy A must know of its existence according to the hypothesis. So, consider any copy B that didn't exist at epoch *E* and isn't in *A.copySet.* By Observation 39 and Program Fragment 11, the history event *Hb* marking the creation of B must have *H^effectEpoch* = *E* + 1 and *Hb'VoteEpoch* = *E.* The value of *Hb.voteEpoch* indicates that either A received a reconciliation report from the coordinator of *Hb* generated after *Hb* was drawn up (Observation 36 and Lemma 41) or that the coordinator, C, of *Hb* had been deleted on or before epoch *E.*

Consider the consequences if C were not to exist at epoch *E* but still coordinated a committing update with vote epoch *E.* The voting algorithm requires that C believed it existed at epoch *E.* So, it must have once existed but been deleted at some epoch on or before epoch *E. A* must have known of the creation of C. All is well if A received a reconciliation report from C covering epoch *E.* If A did not get such a report yet advanced its *agreeEpoch* to *E, A* must have learned of the deletion of C while *A.agreeEpoch* < *E.* Lemma 44 says that the effects of any changes made to the copy set by C must precede the deletion of C. Again, A knows what changes C made, including the creation of B.

Suppose C exists at epoch *E.* Then A knows of the pending proposal to create B. A may only advance *A.agreeEpoch* to E7 + 1 if A knows the attempt to create B was aborted or if A has received a reconciliation report from B. If B sends A a report, the report will reflect the creation of B. Either way, A now has super-set knowledge of all copies existing at epoch *E* + 1 and earlier. •

Lemma 45 comes very close to proving the desired result. Its claim can easily be extended to arbitrary epochs by induction. Theorem 46 states that each copy has super-set knowledge of the copy set. Its trivial proof is omitted.

Theorem 46 Every copy A has super-set knowledge of all copies existing up through and including *A.agreeEpoch.*

At this point, it has been shown that if any copy believes it exists, it has correct super-set knowledge of all copies existing as of its *agreeEpoch* or earlier. This result makes it simple to show that notice replication produces agreement on what has been posted, up through the agreement epoch, in the sense defined in Lemma 47.

Lemma 47 Suppose history events are not expunged and all notices and markers are included in reconciliation reports, not just those dated after the composing copy's expunging epoch. Then VA ^ B any notice, deletion marker, or history event signed or dated *(signEpoch* or *voteEpoch* fields) by copy B at epoch *E* ≤ *A.agreeEpoch* is stored at **A.**

Proof Theorem 46 assures that A knows of all copies that have existed up through epoch *E.* Lemma 41 says that since *E* ≤ *A.agreeEpoch, A* has received a report composed by B when *H.postEpoch* ≥ *E.* Lemma 38 and the assumption that reconciliation reports contain all notices and deletion markers mean that A is told of any notice/marker signed by B at epoch *E.* Furthermore, B can never again date anything with epoch *E* according to Observation 36. D

Lemma 47 proves that in the absence of expunging actions, the agreement epoch does indeed indicate what portions of a copy's store of notices and deletion markers are complete. Lemma 48 shows in a like manner that the expunging epoch truly identifies those actions that every copy already knows about.

Lemma 48 Suppose history events are not expunged and all notices and

markers are included in reconciliation reports, not just those dated after the composing copy's expunging epoch. Then, $\forall \mathbf{A} \neq \mathbf{B}$ any notice, deletion marker, or history event signed/dated (*signEpoch/voteEpoch*) at epoch $E \leq \mathbf{A}.expungeEpoch$ by **B** is stored at every $\mathbf{C} \in \mathbf{A}.copySet$.

Proof Theorem 46 says that **A** knows of all copies that have existed up through epoch E. Lemma 40 shows that $E \leq \mathbf{A}.agreeBounds(\mathbf{C})$. In turn, Lemma 42 says that **A** must have received a reconciliation report composed by **C** when $\mathbf{C}.agreeEpoch \geq E$. By Lemma 47 , **C** must know of all the notices, markers, and history events signed by **B** at epoch E. □

Lemmas 47 and 48 demonstrate correctness in the sense that copies have full knowledge of all actions up through the epochs expected. However, there is no guarantee of convergence unless the *agreeEpoch* at each copy advances. To make progress, copies must try to reconcile their differences. The attempt must be made periodically, with the definition of periodicity being not only be in terms of bounded time, but also in terms of a bounded number of actions taken between reports.

Rule 19 Every copy will periodically generate reconciliation reports.

Copies do not converge just because reports are generated. The reports must be delivered, directly or indirectly, to every copy. Assumption 19 guarantees eventual delivery, but the policy for choosing report destinations must assure complete coverage.

Rule 20 If **A** and **B** are any two copies, either **A** must send reconciliation reports to **B** directly or there must be a set of copies $\{\mathbf{C}_i : 1 \leq i \leq n\}$ such that **A** sends to $\mathbf{C}_1$, $\mathbf{C}_i$ to $\mathbf{C}_{i+1}$, and $\mathbf{C}_n$ to **B**.

Because reconciliation reports are generated regularly, the *postEpoch* field of every copy advances because it is incremented in accordance with the procedure of Program Fragment 16 each time a report is generated.

Observation 49 $\forall \mathbf{C} \forall E$, eventually either $\mathbf{C}.postEpoch \geq E$ or **C** ceases to exist at epoch E.

To prove that the *agreeEpoch* advances, it is necessary to show that deleted copies send out reconciliation reports up through their effective deletion epoch. This is accomplished through a combination of two policies. The choice of effective deletion epoch insures that the coordinator has gotten a report. Rules 21 and 22 below force every copy that updates the copy set field to learn of these actions. Lemma 50 shows that the policies work.

Rule 21 Suppose **A** learns of a proposed version of the *history* field H more recent than $\mathbf{A}.history$ through the voting process. If there is a copy deletion event $D \in H.events - \mathbf{A}.history.events$ with $D.effectEpoch > \mathbf{A}.postBounds(D.copyId)$, **A** must ask the composer of the proposal to send **A** a reconciliation report. Furthermore, **A** will not write H until after receiving and storing such a report.

Rule 22 No copy will coordinate a proposal to delete itself.

Lemma 50 For every copy **A** and every event $H \in \mathbf{A}.history.events$, if H deletes copy **B** at epoch E, **A** has received a reconciliation report composed by **B** when $\mathbf{B}.postEpoch \geq E - 1$.

Proof Let **C** be the coordinator of the proposal deleting copy **B**. Program Fragment 11 requires **C** to set the effective deletion epoch to be the later of $\mathbf{C}.agreeEpoch + 1$ and the effective creation epoch of **B**. If the former choice is used, **C** must know of all actions taken by **B** prior to the deletion epoch by Lemma 47 . If the latter choice is used, **B** did not exist before the deletion epoch. If **B** never existed, there is nothing to know. Induction over deletion events can

be used to show that C already knows of all actions taken by B in previous incarnations.

Suppose C and every other copy know of the actions taken by all previous incarnations of B. C knows of the actions taken by B during this incarnation. Any other copy A that hears of the proposed copy set change also hears of the proposed history update by the atomicity of writes and query responses. It can hear of the proposal in three ways. A can hear of it as a proposal or in the response to a query prompted by a proposal. In these cases, before the new value was written, A either already knew of the actions taken by B or Rule 21 forced it to ask for and process a reconciliation report from C, telling A of them. Rule 22 ensures that C or some copy it reconciled with is still around to issue such a report. A might also learn of the deletion of B through a reconciliation report. Induction over reconciliation reports will show that the composer of the report must directly or indirectly have gotten a report from C or a copy that committed the change in the voting process. Either way, every A must know of every action taken by B in its current incarnation, up to the effective deletion epoch. •

Now the convergence of copies can be shown. Previously, Lemma 47 showed that copies are complete at all epochs up through their *agreeEpoch.* The results just shown together with the advancement of the *postEpoch* imply that the *agreeEpoch* advances. Copies progressively acquire more complete knowledge of the actions taken at other locations.

Theorem 51 VA V22, eventually either $A.agreeEpoch \geq E$ or A ceases to exist at epoch *E.*

Proof The result can be proved inductively. Suppose all copies advance to having an *agreeEpoch* of *E* or are deleted effective *E* or earlier. Consider a copy A with $A.agreeEpoch = E$. By Theorem 46, A learns of all copies that might exist at epoch $E + l$. Suppose C is a copy that A thinks might exist at epoch $E + l$. If C is deleted after epoch $E + 1$, A will get a reconciliation report from C covering epoch $E + l$ according to Lemma 50. If C is deleted at epoch JS+1, A doesn't have to wait for a report. All existing copies will send reconciliation reports covering epoch $E + 1$ according to Rule 19. All these reports will eventually arrive at every copy in accordance with Rule 20 and Assumption 19, given the super-set knowledge of all copies existing at epoch $E + l$.

The *postEpoch* fields of all copies were noted to advance in Observation 49. The only reason why *A.agreeEpoch* might not advance is that it is held up by a pending vote. Theorem 24 states that the majority vote algorithm for modifying copies eventually commits or aborts every proposal. The outcome eventually gets through according to Assumption 19. Unless one of these proposals deletes copy A effective epoch $E + 1$ or earlier, A must advance *A.agreeEpoch* to $E + 1$ at that time, D

The rules for calculating *A.expungeEpoch* are just like the rules for calculating *A.agreeEpoch* except that they are based on the values of the *agreeEpochs* at other copies rather than the *postEpochs.* So, it is hardly surprising that at every copy, the *expungeEpoch* advances over time.

Theorem 52 VA VJE7, eventually either $A.expungeEpoch \geq E$ or A ceases to exist at epoch *E.*

Proof Pick any epoch *E* and copy A in existence at epoch *E.* Theorem 51 guarantees that the *agreeEpoch* fields of all copies advance. The combination of Theorem 46 and the fact that reconciliation reports are periodically sent and eventually received means that every copy in existence eventually will get reconciliation reports from every other copy B generated when $B.agreeEpoch \geq E$. The procedure for computing epoch bounds and the *expungeEpoch* will then force

A.expungeEpoch$\geq E$. •

Together, the results shown in this section prove that the copies of a bulletin board converge. Notice reconciliation makes progress in the sense that all three epochs advance. This, in turn, produces progress in the sense that eventually all copies acquire all notices posted by other existing copies.

A.5.5 Handling Copy Deletion

It is altogether possible for a copy to be ignorant of its deletion. Fortunately, the actions taken by a copy while in such a state of limbo are not harmful. A copy ignorant of its deletion still meets the conditions stated in Lemma 47 so it will correctly present users with all notices signed up through its *agreeEpoch*. Lemma 44 showed that the majority vote algorithm ensures that if an ignorant copy changes the copy set, the change must really have gone into effect everywhere before the deletion. The only remaining problem is to take care of the notices signed by a copy after its effective deletion epoch. This is handled by re-posting them after clearing the *signer* and *signEpoch* fields as described in Program Fragments 20 and 21.

This procedure for re-submitting illegally signed notices and deletion markers is intended to guarantee that none are lost when a copy is deleted. Inspection of the algorithms reveals that they are not.

Lemma 53 No notices or deletion markers are lost due to the deletion of a copy.

Proof Rule 17 requires the addition of a new event to the event history whenever a copy is created or deleted. This means that the procedure for writing out a new value to the *history* field in Program Fragment 20 is called each time some copy is deleted.

If the caller finds that it has been deleted, it carries out the instructions in Program Fragment 21. Every notice and deletion marker it signed on or after its effective deletion time is forwarded to another copy that still exists. No notice or marker is lost. D

To speed the process up, every copy looks for invalidly notices whenever it hears of the deletion of another copy. It re-posts any illegally signed notices by simply re-signing them itself. Note that Lemma 53 ensures that notices aren't lost even without this optimization.

One problem with the re-signing process is that it may behave poorly if the set of copies of a bulletin board changes too rapidly. A notice may chase after a real copy, but each time wind up in a copy that doesn't yet know of its deletion. The process of forwarding a notice to an existing copy may not terminate.

Observation 54 The process of re-signing notices following the deletion of a copy is not guaranteed to terminate.

The odds that the forwarding process will not terminate or take an excessive amount of time to terminate is a function of the rate of change in the copy-set, how rapidly epochs advance, and how long it takes to achieve reconciliation. There is a limit to how many copies can be deleted — the number of copies at the start. For perpetual forwarding to occur, then, both creation and deletion must occur at comparable rates. Let C be the average number of copies, Tc the time between changes to the copy-set, and Ty the average time to complete a vote.

For any progress to be made, the name-bindings must be updated rapidly enough that some copy of the bulletin board can be found. If this is not true, no operation will work. Suppose a name-binding lists the locations of all copies. Then, the probability that looking up a name-binding will not return at least one valid location is roughly:

Program Fragment 20

```
procedure WriteHistory(newHistory: array of HistoryRecordType;
                       var BBoard: BBoardType; var selfDeleted: boolean);
  begin
  seltDeleted := false;
  with event, BBoard do begin
    ExpungeHistory(newHistory, BBoard.expungeEpoch);
    for all event in newHistory - history do
      case eventType of
DELETION:  if copyId = LocalServerId then begin
             selfDeleted := true;
             DeleteSelf(event, BBoard);
             return;
             end
           else CleanUpInvalidf event, BBoard);
CREATION:  if copyId = LocalServerId then
             MakeOwnCopy(event, BBoard);
      end;
    history := newHistory;
    end;
  end;

procedure SignIf1nvalid(event: HistoryEventType; var signature: EnvelopType;
                        BBoard: BBoardType);
  begin
  with event, signature do
    if (signEpoch >= efFectEpoch) and (signer = copyId) then begin
      InitSignature(signature, false);
      SignIfNeeded(signature, BBoard);
      end;
  end;

procedure CleanUpInvalid(event: HistoryEventType; var BBoard: BBoardType);
  var
    signature: EnvelopType;

  begin
  for all signature in (BBoard.notices.signature + BBoard.markers) do
    SignIfInvalid(event, signature, BBoard);
  end;
```

Program Fragment 21

```
procedure DeleteSelf(event: HistoryEventType; var BBoard: BBoardType);
    var
        notice: NoticeType;
        marker: EnvelopType;

    begin
    for all notice in BBoard.notices do
        with notice.signature do
            if (signer = LocalServerId) and (signEpoch >= event.effectEpoch) then begin
                InitSignature(notice.signature, false);
                ForwardToCopy(notice, BBoard);
                end;
    for all marker in BBoard.markers do
        with marker do
            if (signer = LocalServerId) and (signEpoch >== event.effectEpoch) then begin
                InitSignature(marker, false);
                ForwardToCopy(marker, BBoard);
                end;
    GetRidOfCopy(BBoard);
    end;
```

$$\text{Odds of failing to locate copy} \approx \frac{\Delta L}{2CTc}$$

Once a copy has been located and the notice forwarded to it, there is a chance that the copy will be deleted before the notice's signing epoch. The odds that a notice will need to be re-signed are a function of the lifetime of a copy and the time needed for reconciliation. Let TR be the time needed for one round of reconciliation, and *Tpj* be the average time between calls to read or issue a reconciliation report for a bulletin board for the most heavily used copy. On the average, the posting epoch will advance, then, every *Tpj* time units. The entire copy-set is changed every *2uTc* time units, so the lifetime of a single copy in terms of epochs is approximately *2CTC/TN.* The deletion epoch is apt to be about one reconciliation time behind the current posting epoch of the coordinating copy. The copy to be deleted will also advance its epoch until told of the deletion. Altogether, the number of invalidated epochs is approximately (Tv + *TR* In C)/2V, using the reconciliation time computed in Section 5.3. Let *p* denote the probability that a copy is isolated during a particular round of sending reconciliation reports. Then, the odds that a notice will need to be re-signed after one particular attempted delivery are approximately:

$$\text{Odds of re-signing} \approx \frac{p)^{-1}TR(\ln C + \gamma)}{2CT_C}$$

A.5.6 Correctness of Expunging Procedures

The final version of the notice replication algorithm includes a number of optimizations based on the fact that all copies know of every action taken up through the expunging epoch of any copy. For example, a deletion marker *D* can be expunged from the copy at A if *D.signEpoch* ≤ *A.expungeEpoch.* These measures and their correctness is shown in the following, easily proved lemma.

Lemma 55 Lemma 47 is also true if any copy A takes the following actions:

1. **If *N* G A.*notices* and *N.signEpoch* ≤ *A. expungeEpoch*^ *N* is not included in** reconciliation reports.
2. **If *M* 6 *A.markers* and *M.signEpoch* ≤ *A.expungeEpoch, M* is expunged** rather than being included in a reconciliation report.

Proof Consider a notice *N* or deletion marker *M* with the hypothesized properties. By Lemma 48, every other copy already knows of it. No information is lost if they are omitted from subsequent reconciliation reports.

The only function served by *M* is to declare that some notice has been deleted. Lemma 38 proved that a copy may expunge the original notice upon receipt of the marker. Since every copy will already have deleted the notice, *M* has no further use and can be expunged. •

The other optimization concerns the *history* field. As stated, re-computing either the *agreeEpoch* or the *expungeEpoch* involves computing the set of copies existing at an epoch. The *history* can be used to do so, but it turns out that it is not necessary to worry about any copy that isn't a member of the current copy set.

Lemma 56 If there are copies A and B such that B *$ A.copySetbut* the *history* field of A indicates that B existed at some epoch, then all the previous claims hold true even if A advances its *agreeEpoch* and *expungeEpoch* without waiting for a reconciliation report from **B.**

Proof Because B *$ A.copySet, A* will never send a report to B. Any copy that A sends a report to will also learn of the deletion of B and likewise cease to send B reports. So, nothing from the current version of A ever propogates to B via the reconciliation process. The uses of the *expungeEpoch* all stem from the property shown in Lemma 48: na,mely, every copy getting a reconciliation report from A has already seen everything dated after *A.expungeEpoch.* If B has been xleleted from the set of copies, this property is preserved even if the expunging epoch is advanced without consideration as to whether B existed at some earlier epoch.

The use of the *agreeEpoch* is two-fold. It is used to compute the *expungeEpoch.* Since the *expungeEpoch* needn't await reports from B, that purpose is satisfied equally well if the advancement of the *agreeEpoch* also ignores B. The other purpose is to guarantee that A has received a reconciliation report from every copy. Lemma 50 showed that since A committed the deletion of B, it must also know of all actions taken by B prior to the effective deletion epoch. It is safe to advance the *agreeEpoch* without considering B. D

Because history events are not needed to advance the computed epochs, they can be deleted under the conditions listed in Lemma 57.

Lemma 57 For any copy A, consider history event *H* 6 *A.history.events* **with *H.voteEpoch* ≤ *A.expungeEpoch.* If *H* is expunged from *A.history.events*** (without changing *A.history, written),* all previous correctness results remainunchanged.

Proof Lemma 48 indicates that *H* has been seen by every other copy. *H* only needs to be stored if it is needed for local operations.

A history event is used by the reconciliation algorithm to compute the set of copies to determine what reconciliation reports must be received before the *agreeEpoch* and *expungeEpoch* can be advanced. Lemma 56 shows that history events are not needed to compute agreement epochs.

Another use for history events is to determine the effective epoch of deletion

events. Suppose *H* re-creates a copy and is used by A in proposing to delete a copy. Because *H.voteEpoch* + 1 = *H.effectEpoch*, *H* only forces A to pick an effective deletion epoch of at least 1 + *A.expungeEpoch.* Every copy's *agreeEpoch* is at least as great as its *expungeEpoch.* Since the earliest deletion epoch A might generate is 1 + *A.agreeEpoch,* A's decision would be unchanged if *H* had been expunged.

Finally, history events are used to identify notices that need to be re-signed. This is a two part problem. When A learns of a new notice or deletion marker, it must decide whether it was signed by a deleted copy and needs re-signing. Notices are received in two ways. They can be original postings. Unsigned notices do not need to have previous signatures invalidated. The only other way to receive a new notice or marker is as part of a reconciliation report. If *A.history* indicates that a notice is illegally signed, it must be that the copy, B, issuing the report has less recent knowledge of the *history* field. Lemma 48 guarantees that B knows of every event *H* with *H.voteEpoch* $\leq$ *A.expungeEpoch.* So, the event *H* that A would need to know to decide if a new notice is illegally signed does not qualify for being expunged. Any re-creation of the copy would have an even later *voteEpoch* so would also be available for use.

The other part of the problem of determining which notices to re-sign is to decide if a history event describing a deletion has been seen before. Only new deletion events trigger the re-signing of notices already stored. By Lemma 47, **A knows of all *H* with *H.voteEpoch* $\leq$ *A.agreeEpoch.* The choices of *voteEpoch*** and *effectEpoch* for a deletion event guarantee that if *H* reflects the deletion of **a copy and *H.effectEpoch* $\leq$ *A.expungeEpoch* then *H.voteEpoch* $\leq$ *A.agreeEpoch.*** *A* knows that it has seen any deletion event that qualifying for expunging. •

A.5,7 The Reader's Point of View

The definition of correctness for notice replication is the behavior of the bulletin board system from the viewpoint of a user. The convergence of all copies' stores of notices is guaranteed by Theorem 51 and Lemma 47. Notices are not lost just because they were signed by a copy ignorant of its deletion, either, according to Lemma 53. So, all posted notices eventually reach all copies, except in the pathological case that copies are so rapidly deleted that some notices never find an undeleted copy.

The correctness criterion requiring that users never miss notices is almost met. The notice replication algorithm does guarantee that if a notice is ever legally signed, it will be seen by every regular reader. A user's record of what he has seen consists of the latest *agreeEpoch* of any copy read and a list of identifiers for all notices read but signed after the *agreeEpoch.* Lemma 47 shows that the read has covered every notice dated up through the *agreeEpoch.* The reader, then, has enough information to insure that he will see all future notices.

Corollary 58 If ReadNewNotices is called upon to read all notices posted after epoch 0 and returns a *readEpoch* of *E,* no read asking for notices with epochs 0 through *E* will return any notice that was not returned in the first read unless the first copy knew of the notice's deletion.

It is possible for fewer notices to be returned the next time. Notices can be deleted or re-signed and accepted with a later epoch. Re-signing is only done to notices accepted by a copy ignorant of its deletion. Because re-signing increases a notice's *signEpoch,* users can see notices more than once. They do not, however, see duplicates under normal operation.

Lemma 59 If a user asks only for what he has not seen before, he will only be presented with new notices and notices that were re-signed during epochs lying between the *agreeEpoch* of the copy read and the *agreeEpochs* of copies proposing

the deletions.

Proof **If** a notice is dated after the *agreeEpoch* of the copy read, its will not be read again because postmasters will check its identifier. The rule for deciding which version of a notice to keep only lets a notice with a later *signEpoch* replace one with an earlier version if the number of signatures is greater. The number of signatures is only incremented as part of the initial posting and when a notice re-signed in response to the deletion of a copy. In fact, only those notices signed after the effective deletion epoch are re-signed. That is, if a notice is seen twice, it must have been signed by a copy ignorant of its deletion with an epoch later than the agreement epoch of the coordinator and at least the agreement epoch of the copy read. •

The observations of this section demonstrate that the notice replication of Taliesin is correct under most circumstances, although it can fail if copies are created and deleted rapidly. Good administrative policies can keep the deletion rate low enough that pathological behavior will not arise. Occasionally, some users will see notices twice, but that mis-feature should not be annoying if the frequency is low. This approximation to correct behavior should be good enough to provide satisfactory service.

Appendix B

Glossary

agent: an active entity: **a** user or a program

agreement epoch: computed epoch — a copy has complete knowledge of all events dated up through its agreement epoch

authentication service: service used to verify that an agent is who it claims to be

bulletin board: a collection of notices, particularly a collection pertaining to a particular topic

client: an agent acting in the role of one receiving services

communication link: a means of transmitting data between nodes

computer-based message systems: facilities on computers for assisting communications between users

computer bulletin board system: computer facility enabling a user to send messages to anyone expressing interest in a topic

computer conferencing: computer based message system providing support for different roles in peer group interactions

computer mail: computer facility for delivering communications to specified individuals

coordinator: the agent that composes an update proposal and supervises the voting upon it

deletion marker: a skeletal data structure indicating that a particular notice is to be deleted

delivery protocol: protocol for passing a notice from a postmaster to a user agent

distribution list: list of individuals that commonly receive a copy of the same notice

effective epoch: time that **a** change to the copy set of a bulletin board takes effect, as measured by epochs

envelope: data structure describing a notice used to pass handling and delivery information between agents

epoch: time unit for a logical clock defined for each copy of a bulletin board

exploder: see distribution list

expunging epoch: computed epoch — every copy knows of every event dated on or before the expunging epoch of any copy

internal consistency: updates to a copy have been applied in a order that preserves its

semantic properties

internet: a collection of interconnected networks

IPC: inter-process communication

mail transport agent: the agent that transports notices over network communication links

mail transport protocol: protocol defining the interface between mail transport agents

message: a logical unit of communication transmitted over a network communication link

multicasting: facility for delivering a message to a specified list of agents or nodes

mutual consistency: all copies have undergone the same series of changes in the same order

name **service:** service mapping string names to lower level identifying information

network: collection of nodes connected by communication paths, normally using common protocols for communicating among themselves

network partition: state in which one collection of nodes are unable to communicate with nodes **in** another collection

node: a hardware unit providing processing power: a computer or workstation

notice: the unit of communication supported by a computer based message system — composed by one user

notice body: the part of a notice composed by a user containing the information he/she wishes to communicate

order statistic: in a sample of random variables, the n-th order statistic is the n-th largest value in the sample

originator: user composing a notice

posting epoch: current time at a bulletin board on the epoch time-scale: the time used in dating notices and changes to the copy set

posting protocol: protocol defining the procedure for asking a postmaster to deliver a notice to one or more bulletin boards

postmaster: an agent supervising access to bulletin boards

pure communication: transmission of a communication geographically without storing it for retrieval at a later time

pure storage: storage of a communication for later retrieval without geographical transmission

query: specification of what notices a user wishes to see

read epoch: epoch reported by a copy as its current time as part of a **ReadNewNotices** operation

recipients: users that read a notice

reconciliation report: a message transmitted between postmasters to exchange information on events related to a bulletin board

replication group: a division of the name space: replicated as a unit

replication group table: a data structure listing the agents that have a copy of each piece of the name space

root replication group: the division of the name space in which the parsing of absolute names begins

schism: a state in which agents hold contradictory beliefs about the history of changes to the set of copies of a bulletin board

serializability: a property of an interleaving of concurrent updates: do they produce the same effect as sequential execution?

server: an agent acting in a role of providing services to others

signature: the part of an envelope containing a notice's identifier and delivery date

signing epoch: official arrival time of a notice on the epoch time-scale

store-and-forward network: a network in which nodes cooperate to relay messages, storing them temporarily if communication paths are temporarily unusable

UDS: the Universal Directory Service: a name service

universal bulletin board: a bulletin board read by a constant fraction of the user community

user: a human being

user agent: a program acting on behalf of a user: translating user input into requests to servers and displaying responses

user profile: a data structure storing information about a user's identity, preferences, and past activities

voter: an agent that is asked to approve or disapprove an update in the voting process

voting epoch: the time at which a change to a copy set is proposed on the epoch time-scale — as opposed to the time at which the change takes effect