A weak-consistency architecture for distributed information services

Richard A. Golding

UCSC–CRL–92–31

July 6, 1992

Concurrent Systems Laboratory
Computer and Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064

Services provided on wide-area networks like the Internet present several challenges. The reliability, performance, and scalability expected of such services often requires they be implemented using multiple, replicated servers. One possible architecture implements the replicas as a *weak-consistency process group*. This architecture provides good scalability and availability, handles portable computer systems, and minimizes the effect of users on each other. The key principles in this architecture are component independence, a process group protocol that provides small summaries of database contents, caching database *slices*, and the *quorum multicast* client-to-server communication protocol. A distributed bibliographic database system serves as an example.

Keywords: weak-consistency process group, quorum multicast, component independence, scalability, fault tolerance.

1 Introduction

Several information services have recently been, or will soon be, made available on the Internet. These services provide access to specialized information from any Internet host. The bulk of these systems centralize some parts of their service – either by centralizing the entire system, or by breaking the service into several pieces and implementing each piece as a centralized application.

In this paper I will present an architecture for building distributed information services, drawing examples from the refdbms bibliographic database system. The architecture emphasizes scalability and fault tolerance, so the application can respond gracefully to changes in demand and to site and network failure. It uses weak-consistency replication techniques to build a flexible distributed service. I will start by defining the environment in which this architecture is to operate and its goals. Next I will give an overview of the architecture, followed by sections detailing three components: weak-consistency process groups, quorum multicast protocols, and mechanisms to cache predefined *slices* or subsets of the database.

1.1 Environment

The Internet has several behaviors that must be accounted for when designing an information service. These include the latency required to send messages, which can affect the response time of an application, and communication unreliability, which may require robust communication protocols. Two hosts on an Ethernet can exchange a pair of datagram packets in a few milliseconds, while two hosts on the same continent may require 50–200 milliseconds. Hosts on different continents can require even longer. Packet loss rates of 40% are common, and can go much higher [Golding91b]. The Internet has many single points of failure, and it is usually partitioned into several non-communicating networks. This is a difficult environment for building distributed applications.

The application architecture must also handle the vast number of users that can access a widely-available service. The Internet now includes more than 900 000 hosts[1]; the potential user base is in the millions, and these numbers are expected to increase rapidly. The archie anonymous FTP location service reported on the order of 10 000 queries per day (0.12 queries per second) using two servers in November 1991 [Emtage92]. The archie system is a specialized service with a limited audience, as compared to traditional information services used by the general public, such as newspapers and library card catalogues. Anecdotal evidence points to some current services with nearly 100 queries per second.

Despite this environment, users expect a service to behave as if it were being provided on a local system. Several studies have shown that people work best if response time is under one second for queries presenting new information, and much less for queries that provide additional details [Schatz90]. Furthermore, users expect to be able to make use of the service as long as their local systems are functioning. This is an especially difficult expectation to meet on portable systems, where the system may be disconnected from the network for a long time or may be "semi-connected" by an expensive low-bandwidth connection. Several researchers are investigating file systems that can tolerate disconnection [Kistler91, Alonso90a].

Throughout this paper the term *process* refers to a process, running at some *site.* Sites are processor nodes on the network such as a workstation or file server. Server processes have access to pseudo-stable storage such as disk that will not be affected by a system crash. Sites also have loosely synchronized clocks. Sites and processes fail by crashing; that is, when they fail they do not send invalid messages to other processes and they do not corrupt stable storage. Processes can temporarily fail and recover. Sites have two failure modes: temporary recoverable failures, and permanent removal from service. The network is sufficiently reliable that any two processes can eventually exchange messages, but it need never be free of

[1]This value was provided by Darrell Long, who has been tracking the Internet population as part of a longitudinal reliability study [Long91, Long92].

partitions. *Semi-partitions* are possible, where only a low-bandwidth connection is available between one or more sites and the rest of the network.

1.2 Principles

There are some general principles guiding the solutions presented here. Service *replication* is the general mechanism for meeting availability demands and enabling scalability. The replication is *dynamic* in that new servers can be added or removed to accommodate demand changes. The system is *asynchronous*, and servers are as *independent* as possible; it never requires synchronous cooperation of large numbers of sites. This improves communication- and site-failure tolerance. *Local communication* is almost always faster than long-distance communication, and should be used whenever possible. The solutions use *prefetching* and *caching* where possible to improve response time. The service should be *secure* to the degree appropriate and possible, so that client processes can trust the information provided by the service and service providers can accurately charge for their service if needed. Finally, the architecture should minimize the effect of one user on another.

1.3 The refdbms system

The weak-consistency architecture is being used to implement a distributed bibliographic database system, refdbms. This project aims to evaluate this architecture for convenience and performance. The refdbms system is derived from a system developed at Hewlett-Packard Laboratories over several years [Wilkes91]. That system emphasized sharing bibliography information within a research group. Users could search a database by keywords, use references in TEX, and enter new or changed references.

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

Refdbms stores references in a format similar to that used by refer, as shown in Figure 1. Every reference has a *type*, and a unique, mnemonic *tag* like Golding92l. Since these tags are determined by users and can potentially collide, the system internally uses a unique identifier consisting of a timestamp plus the address of the site that created the reference. References are stored in hashed files, and are indexed both by tag and by keyword. Using location information in the database and an inference engine, the system will determine the best way to provide the user with a copy.

1.4 The architecture

Replication is the cornerstone of this architecture. A set of replicated servers cooperate to provide the service, as shown in Figure 2. Client processes use the service by contacting some server process. Server processes in turn communicate amongst themselves to propagate update information.

The use of multiple servers can improve communication locality. If all clients must communicate with the same server, some of them will have to use long-distance communication. If there several servers, clients can communicate with the one closest to them. This both reduces communication latency and decreases the load each communication imposes on the Internet. One approach is to place one server in each geographic region or organization. Of course, clients must be able to identify nearby servers and maintain performance when nearby sites fail; this problem is discussed in Section 3.

%z TechReport *(the type)*
%K Golding92l *(the tag)*
%A Richard A. Golding
%A Kim Taylor
%T Group membership in the epidemic style
%R UCSC–CRL–92–13
%p CISBD., UCSC.
%D 22 Apr. 1992
%l FTP postscript from ftp.cse.ucsc.edu pub/tr/ucsc-crl-92-13.ps.Z
%x We present a new lightweight group membership
%x mechanism that allows temporary inconsistencies in membership
%x views. This mechanism uses epidemic communication techniques to
%x ensure that all group members eventually converge to a consistent
%x view of the membership. Members can join or leave groups, and we
%x show that the mechanism is resilient to $k \leq n - 2$ members
%x failing by crashing, where n is the number of members in the group.
%k distributed systems, weak-consistency replication
%k lightweight group membership, process groups

FIGURE 1: An example reference.

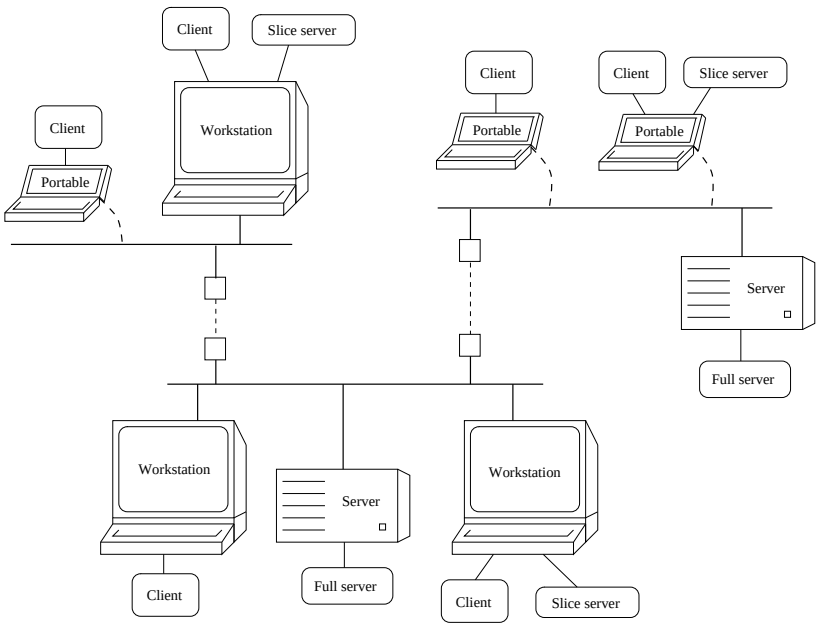

FIGURE 2: Overall system architecture. Some local-area networks will have a nearby server, while others must communicate with more distant servers. Portable systems may include a slice server that maintains a copy of part of the database.

Replicated servers also help meet the goal of a highly available and reliable service. The service is available as long as clients remain connected to at least one server, and that server is functioning. A recent study of workstation reliability [Long91] shows that most hosts are available better than 90% of the time, with a mean time-to-failure (MTTF) between two and three weeks. Another study has found that hosts within North America respond when polled about 90% of the time [Golding91b], indicating that long-term network failure is probably uncommon. This same study showed that communications were more reliable the closer two sites were. This architecture can therefore be expected to provide nearly complete availability.

Each server maintains a copy of the database. For simplicity, every database entry is assumed to have a unique key. Some servers will maintain a copy of the full database. Others will maintain *caches* or *slices* of the information, as discussed in Section 4. A cache is an arbitrary collection of recently-used database entries, while a slice is a subset of the database defined by a query, similar to a relational database view. In refdbms, for example, a slice might maintain a copy of all entries on marsupials.

The servers are organized into a *process group*. Servers use a *group communication* protocol to multicast a message to the group when they need to perform a database operation. Client processes send request messages to just one server, which forwards the request if needed to other servers in a multicast. When servers are added or removed, they follow a *group membership* protocol to inform other servers and to obtain a copy of the database.

The servers must coordinate their operation so they provide *consistent* service: the answers provided by one server should not contradict those provided by another. Consistency is controlled by the group communication protocol, since the state of a server is determined by the messages it has received. This topic is discussed further in Section 2.1.

Eventually or *weakly consistent* communication protocols do not perform synchronous updates. Instead, messages are first delivered to one site, then propagated asynchronously to others. The answer a server gives to a client query depends on whether that server has observed the update yet. Eventually, every server will observe the update. Many existing information systems, such as Usenet [Quarterman86] and the Xerox Grapevine system [Schroeder84], use similar techniques. Users of a bibliographic database are unlikely to be worried if an update takes a few hours to propagate to every server, as long as their updates are available right away at their server.

Delayed propagation means that clients do not wait for distant sites to be updated, and the fault-tolerance of the service does not depend on client behavior. It also allows messages to be transferred using bulk communication protocols, which provide the best efficiency on high-bandwidth high-latency networks. These transfers can occur at off-peak times. Servers can be disconnected from the network for a period of time, and will be updated after they are reconnected. On the other hand, clients must be able to tolerate inconsistency, and the service may need to provide a mechanism for reconciling conflicting operations. In refdbms, updates take the form of differences to the text of a reference, and all updates are applied in the same order at every site. One update may occasionally be superseded by another, but collisions are unlikely.

The group of processes communicating this way can be organized into a *weak-consistency process group*. Propagating updates from one server to another form a logical, asynchronous group multicast operation that is immune to temporary crashes. I have developed protocols for weak-consistency group communication and for adding and removing servers from the group. These are detailed in Section 2.

2 Weak-consistency process groups

Replicated services can be implemented as a process group. Members of the group use *group communication* protocols to communicate amongst themselves, and *group membership* protocols to determine what processes are in the group. The membership and communication protocols are closely related: the membership protocol usually uses some form of the communication protocol to send membership information to

processes, and the communication protocol uses membership information to identify what processes should receive messages.

Weak consistency protocols guarantee that messages are delivered to all members but do not guarantee when. In this section I will discuss how weak consistency compares to other kinds of consistency, and detail protocols for weak-consistency group communication and membership.

2.1 Kinds of consistency

The service provided by a process depends on the messages it has received, so application-level consistency depends on communication consistency. Communication protocols can provide guarantees on:

1. Message delivery. Messages can either be delivered *reliably,* in which case they are guaranteed to arrive, or with *best effort,* meaning the system will make an attempt to deliver the message but it is not guaranteed.

2. Delivery ordering. Messages will be delivered to processes in some order, perhaps different from the order in which they are received. A *total* ordering means that all processes will see the same messages in the same order, though that order will not necessarily be the order messages were sent. *Causal* ordering implies that any messages with a potential causal relation will be delivered in the same order at all replicas [Lamport78, Ladin91]. Messages with no causal relation, however, can be delivered in different orders at different processes. Messages can also be delivered so that the database at one site never differs from the correct global value by more than a constant [Pu91, Barbará90]. Weaker orderings include a *per-process* or *FIFO channel* ordering, where the messages from any particular process are delivered in order, but the streams of messages from different processes may be interleaved arbitrarily. Finally, there is the possibility of guaranteeing no particular order.

3. Time of delivery. The communication protocol can deliver messages *synchronously,* within a *bounded* time, or *eventually* in a finite but unbounded time.

In general, strong guarantees require multiphase synchronous protocols while weaker guarantees allow efficient asynchronous protocols.

The weak consistency used in this architecture provides reliable delivery, and can be modified to produce several delivery orderings, but it only guarantees eventual message delivery. In particular, there is a non-zero probability that two processes have received all the same messages, and all processes are guaranteed to agree in finite but unbounded time if no further messages are sent.

Grapevine [Schroeder84] was one of the first wide-area systems to use weak consistency. In that system, replicated data was updated first at one site, then the results were propagated to other sites in the background. Updates were propagated three ways. A site might first use *direct mail,* an unreliable multicast, to get the update to as many sites as possible. Then it would use *rumor mongery* to propagate recent updates from one site to another. Finally, pairs of sites would periodically exchange all known updates in an *anti-entropy session* until they were mutually consistent. Of the three methods, only anti-entropy guaranteed delivery to all sites.

The *tattler* [Long92] is another system that uses weak-consistency groups. It uses group communication to coordinate a group of processes that periodically retrieve uptime statistics from Internet hosts. The list of hosts to be polled and the experimental results are propagated using the group communication protocols outlined in the next section.

2.2 Group communication

I have developed a new group communication protocol that provides reliable, eventual delivery, called *timestamped anti-entropy* [Golding91a]. Since the protocol is fault tolerant, messages will be delivered to every process in the group even if processes temporarily fail or are disconnected from the network.

To send a message to the group, a process appends some timestamp information and writes it to a log on stable storage.[2] From time to time each site selects a partner site, and the two exchange logs in an *anti-entropy* session. In addition, processes maintain a summary of the timestamps on messages they have received. These summaries are exchanged as the first step of anti-entropy sessions, and allow each process to send only those messages the other has not yet received. An unreliable multicast can be used to propagate a message quickly while anti-entropy sessions ensure the message is delivered to sites that miss the multicast.

This protocol meets many architectural goals. It provides an asynchronous communication mechanism that allows replicas to be mostly independent. Anti-entropy sessions only involve two replicas, so the mechanism can scale to large, dynamically changing groups. Sites can tend to select nearby partners for anti-entropy, minimizing long-distance communication. The protocol also handles disconnected and failed sites well. While a replica is unavailable, messages accumulate in other replica's logs, and are transmitted to the replica when it becomes available again. Summaries provide a compact way for portable systems to measure how far out of date their information has become; this measure can be used to prompt the user to plug their machine into the network for fresher information.

The tradeoffs are that the protocol is blocking, that replicas must maintain fault-tolerant logs, and that timestamps must be appended to every message. If an operation must be coordinated with the entire group, perhaps so total consistency is preserved, it must be delayed until the request message can be received and acknowledged by every process in the group. Until that time, the request message must be stored on disk so it is not affected by failure and recovery.

The timestamps appended to each message can be used to generate a variety of different message delivery orderings, including total (but not causal), per-process, or no ordering. Causal orderings are possible if process clocks meet Lamport's happens-before condition [Lamport78].

To execute the protocol, each process must maintain three data structures: a *message log* and two *timestamp vectors* [Mattern88]. These must all be maintained on stable storage, so they are not corrupted when the site or process crashes. Each site must also maintain a clock that is loosely synchronized with other processes.[3]

The *message log* contains messages that have been received by a process:

$$\text{Log} = \text{list of (sender id, timestamp, message)}.$$

Timestamped messages are entered into the log upon receipt, and removed when all other processes have also received it. The sender identification and timestamp can both be on the order of four bytes each. Messages are eventually *delivered* from the log and applied to the database.

Processes maintain a *summary timestamp vector* to record what updates they have observed:

$$\text{Summary vector} = \text{list of (process id, timestamp)}.$$

Process A records a timestamp t for process B, if A has received all messages generated at B up to time t. Each process maintains one such timestamp in its timestamp vector for every process in the group. The vector provides a fast mechanism for transmitting summary information about the state of a process.

[2] This paper is written in terms of a log. However, if update information can be retrieved from database contents a log is not technically necessary. Grapevine [Demers88] used this technique.

[3] I have also developed a similar protocol that requires $O(n^2)$ state per process rather than $O(n)$, but allows unsynchronized clocks. This alternate protocol was discovered independently by Agrawal and Malpani [Agrawal91].

Each process also maintains an *acknowledgment timestamp vector* to record what messages have been acknowledged by other processes:

$$\text{Acknowledgment vector} = \text{list of (process id, timestamp)}.$$

If process A holds a timestamp t for process B, A knows that B has received every message from any sender with timestamp less than or equal to t. Process B periodically sets its entry in its acknowledgment vector to the minimum timestamp recorded in its summary vector. This mechanism makes progress as long as process clocks are loosely synchronized and the acknowledgment vector is updated regularly. A process can determine that every other group member process has observed a particular message by looking at its local acknowledgment vector.

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

At the end of a successful session, both processes have received the same set of messages. Processes A and B set their summary and acknowledgment vectors to the elementwise maximum of their current vector and the one received from the other process.

After anti-entropy sessions have completed, update messages can be delivered from the log to the database, and unneeded log entries can be purged. If the system guarantees that all processes will observe messages in the same order, messages whose timestamp is less than the minimum timestamp in the summary vector can be delivered. If per-process or weaker orderings are allowed, messages can be delivered immediately upon receipt. A log entry can be purged when every other process has observed it. This is true when the minimum timestamp in the acknowledgment vector is greater than the timestamp on the log entry.

The reliable delivery guarantee is not met in one important case: when a process permanently fails and loses data. No weak consistency communication scheme can be free from this, since a window of vulnerability must exist while the data is being sent to other processes. In practice the duration can often be reduced by disseminating the new updates rapidly, but real networks do not allow complete certainty.

2.3 Group membership

The group membership protocol provides mechanisms for listing group membership, creating a new group, joining and leaving a group, and recovering from member failure. The group communication protocol uses this information to identify what sites should receive multicast messages. This section sketches a weak-consistency group membership protocol; details and proofs of correctness are reported elsewhere [Golding92c].

Each process maintains an eventually-consistent *view* of the membership, indicating the status of each member:

$$\text{View} = \text{list of (process id, status, timestamp)}.$$

Views are updated during anti-entropy sessions, and eventually all processes can reach agreement if the membership stops changing.

Inconsistent group views can make a system vulnerable to failure. Since a process can only contact processes in its view, the transitive closure of all views must be kept equal to the group membership. The knows-about graph formed by membership views can become incorrect if the only process to know about

another fails. To ensure the knows-about graph stays correct after up to k failures, the graph connectivity must be $k + 1$ or greater.

To *initialize* a new group, a process p creates a new view with only itself. To *join* a group, p finds $k + 1$ sponsor processes in the group. These processes insert p into their views, and send the resulting view back to p. To *leave* a group, p marks its status as leaving, then waits for every other process to observe the status change. While waiting it performs anti-entropy sessions but does not originate any messages. When p fails, some outside mechanism will inform a functioning member process. This process marks p as failed in its view. This information propagates to other processes, re-establishing $k + 1$-connectivity along the way.

In contrast to this system, previous group membership mechanisms ensure greater consistency of group views at the expense of latency and communication overhead. Both the Isis system [Birman87, Birman91] and a group membership mechanism by Cristian [Cristian89] are built on top of synchronous atomic broadcast protocols, and hence provide each process with the same sequence of group views. The Arjuna system [Little90] maintains a logically centralized group view via atomic transactions.

2.4 Performance

Three disadvantages of weak-consistency process groups were pointed out in the last section: some operations must be delayed until request messages have been observed throughout the group; they require on-disk message logs; and messages can be lost when many sites fail simultaneously.

The magnitude of each of these problems depend on how fast messages are propagated through the group, which can be determined by simulation. A system of n server processes can be modeled as a Markov system of $O(n^2)$ states, where each state is labeled with the number of processes available and the number that have observed the update.

Data loss due to permanent site failure appears to be negligible in systems like the Internet, where sites stay in service for several years. The probability of losing a message depends on the ratio ρ of the rate at which sites perform anti-entropy to the rate of permanent site failure. If sites perform anti-entropy hourly and sites remain in service for a few years, ρ is more than 10 000 and the probability of failure is less than 1 in 10 000. The usual approximations to stable storage, such as delayed writeback from volatile storage, also have negligible effect.

The size of logs is a function of the time required to propagate a message to every group member. The time required increases approximately as the log of the number of processes, and under reasonable update and read rates information is likely to propagate to most sites before it is needed. Figure 3 shows the distribution of time required to reach consistency for different numbers of processes.

2.5 Presenting inconsistent information to users

Inconsistent information can be confusing to users. If one refdbms user finds a newly-added reference, then sends a message to another user discussing the reference, the second user may not be able to find the reference because it has not propagated to the right servers. This problem can be remedied by only making information available to users after it has propagated everywhere, although this may make the information unavailable for quite some time.

The refdbms system provides a hybrid solution that allows users immediate access to new information. It maintains a *pending* copy of inconsistent database entries. Users can access the pending copies by appending a .pending suffix to reference tags. The consistent copy is available using the unmodified tag. Each server attempts to apply changes to the pending copy as they are received, but the changes may be applied in an order different from the order they are applied to the final consistent copy.

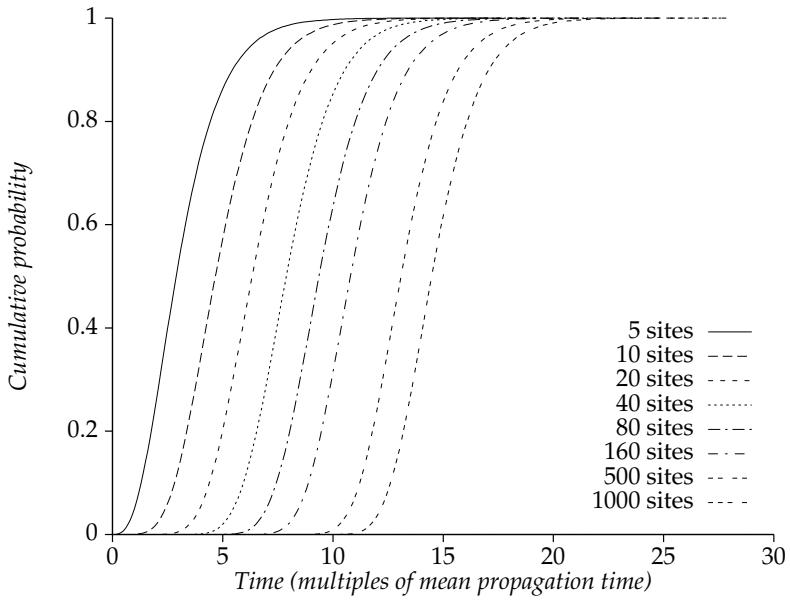

FIGURE 3: Time required to propagate a message to all processes.

---

3 Using nearby servers

If a replicated service is to make communications local, clients must be able to locate and use the closest servers. Servers can use a similar mechanism to bias their partner selection to favor other nearby servers. I have investigated *quorum multicast* protocols that will use preferred sites [Golding91b, Golding92b]. These protocols use an ordering on m sites, and attempt to communicate with the best n of them. Sites can be ordered using predictions of communication latency, failure, and bandwidth.

3.1 Using quorum multicast to select sites

Quorum multicast protocols allow clients to generally communicate with nearby sites, falling back to more distant sites when the nearby ones have failed. While these protocols were originally developed for implementing majority voting replication protocols, they provide exactly the communication locality and fault tolerance needed for communication with a single server.

The semantics of quorum multicast define the interface:

quorum-multicast(message, sites, reply count) $\rightarrow$ replies
Exceptions: *reply count not met.*

The message is sent to at least a reply count of the sites. If at least that many responses are received, the operation succeeds; otherwise, it fails. Either way it returns the set of responses received. Quorum multicasts can be used for client-server communication by setting the sites to the list of servers and the reply count to one. If the servers are ordered from nearest to farthest, the protocol will select nearby servers over distant ones.

There are several variations on the protocol, each of which uses a different policy to handle site and communication failure. Most can be tuned to declare possible failure of a nearby site early, improving communication latency when the site is down at the expense of extra messages. Other variations provide

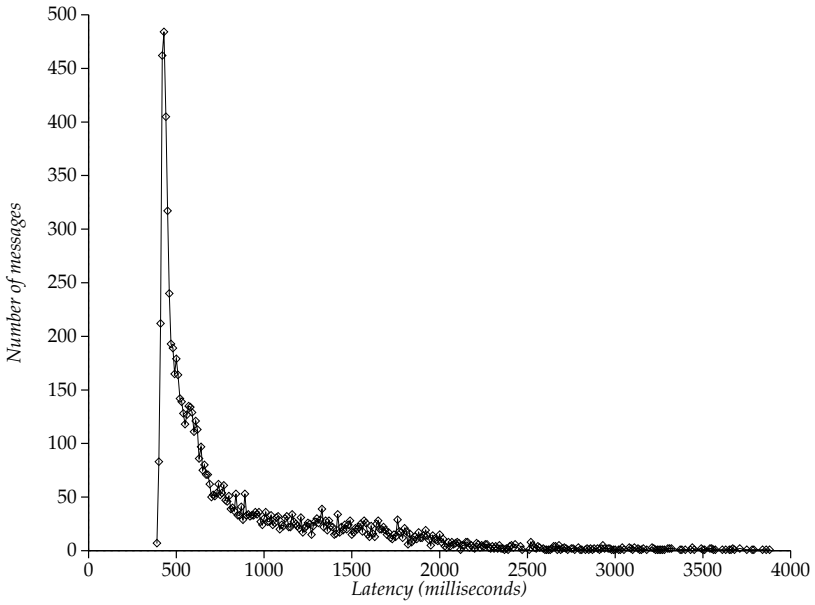

FIGURE 4: Distribution of communication latency. Measured between **maple.ucsc.edu** and **cana.sci.kun.nl;** average latency 938 milliseconds.

---

different policies for retrying communication with nearby sites that may have failed.

3.2 The performance prediction problem

A client site must be able to rank servers by expected communication performance if quorum multicast is to work. Expected performance is based on a prediction of communication latency, failure, and bandwidth. If an operation requires that only a small amount of information be moved between sites, message and processing latency will dominate performance. If large amounts of information must be transferred, then bandwidth will dominate. The prediction should be biased by the probability that the client can communicate with the server. A detailed examination of this problem is available [Golding92a].

Predictions can be derived statically from the topology of the network, or dynamically using performance samples. The topology of the Internet is quite complex, and no detailed topological models are available. Approximations of topological information, such as hop counts, have been shown to be poor performance predictors [Golding91b]. Dynamic prediction is generally more accurate.

Communication latency is often predicted by a *moving average* of recent samples. The moving average at time t of a sequence a_i is defined as $\overline{a_t} = wa_t + (1 - w)\overline{a_{t-1}}$, with $\overline{a_0} = 0$. The estimator can be biased to weight recent or older samples more heavily by adjusting the parameter w. This method is used in most implementations of TCP [Jacobson88]. That work assumes that latency is normally distributed, and computes an estimate of the variance to determine failure timeouts. The actual distribution is generally similar to that in Figure 4. While it is not normally distributed, it is predictable.

Latency predictions should be biased by the probability that a site will respond. A site may not respond either because a message did not get through, or because it has crashed. A moving average of the probability of message failure $\overline{f_t}$, $0 \leq \overline{f_t} \leq 1$, can be combined with the packet timeout p_t and expected latency $\overline{l_t}$ to give an overall expectation o_t:

$$o_t = \overline{f_t}p_t + (1 - \overline{f_t})\overline{l_t}.$$

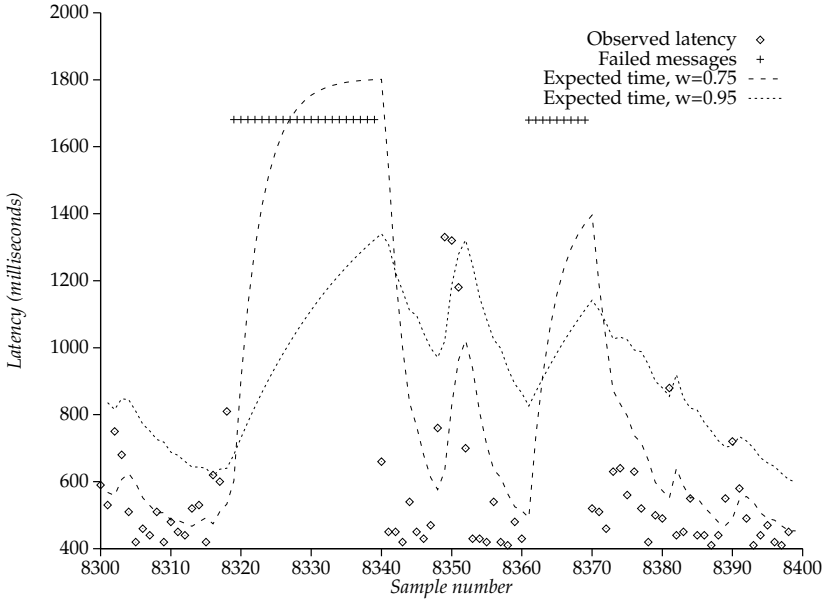

FIGURE 5: Overall communication latency. 100 samples from a longer trace, measured between **maple.ucsc.edu** and **cana.sci.kun.nl.** The packet timeout is reported for failed messages. The graph includes two estimation curves, showing the effect of different weights on the moving average estimator.

Figure 5 shows how this estimation responds to a sequence of samples. Experience has shown this is a good overall estimator.

Sometimes bandwidth must be considered to rank servers – for example, when the refdbms system is selecting a site to retrieve a copy of a paper. Unfortunately bandwidth is not as predictable as latency. Figure 6 shows a typical bandwidth distribution. The distribution is nearly uniform and consequently has a high variance. I have evaluated a number of prediction methods for bandwidth, and a moving average prediction appears to work well.

Prediction methods using moving averages must have several recent samples to be accurate. Every time a site communicates with a server it can log communication statistics to a database, and the sites on a local-area network can pool their results to increase the number of samples. However, quorum multicast techniques will cause most of the samples to come from nearby sites. Periodically dropping sites from the database will ensure accurate prediction for all potential servers, as well as keep the database size small. If a database of samples is available at every local organization, portable systems can find a database wherever they connect to the Internet.

4 Caching and prefetching

While multiple servers bring information closer to clients, they do not necessarily make the information local. Most clients, particularly portable systems, will not have the disk space to store the entire database. However, clients on disconnected portable systems can only operate if the information is local. Other systems perform better when information is local or on the same network. Caching and prefetching information to personal or organization-wide servers can meet this need.

Cache and slice servers play different roles. Cache servers maintain copies of recently-accessed database

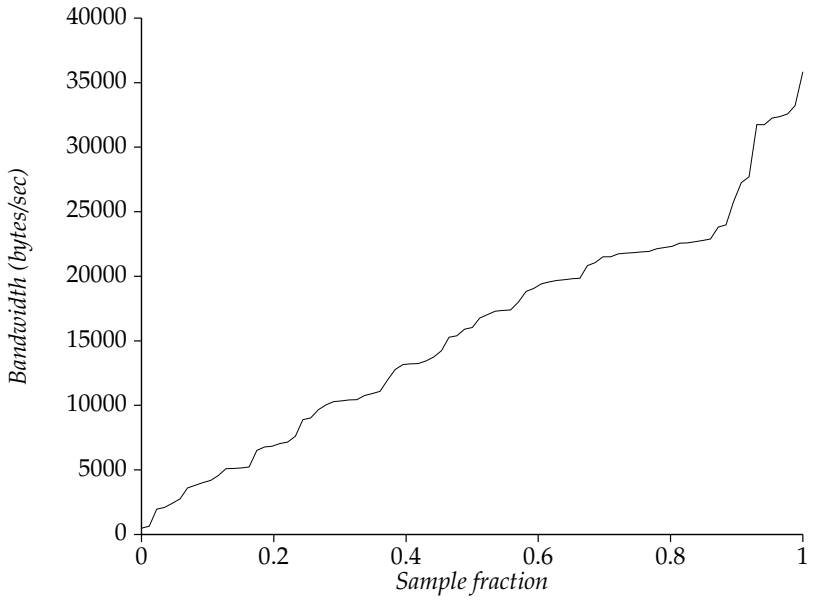

FIGURE 6: Typical bandwidth distribution. Measured between **beowulf.ucsd.edu** and **lcs.mit.edu.**

---

entries, which can improve performance if one site or organization repeatedly accesses a small set of database entries. Slice servers prefetch database entries that are likely to be used in the near future according to user-specified interests. Slices provide a way to group information that has not yet been accessed with entries that have. Caches and slices differ in their handling of new database entries: slice servers will store a copy of a new entry if it matches some predicate; cache servers will not.

Alonso, Barbará, and Garcia-Molina have researched these issues for systems that use bounded inconsistency [Alonso90b]. They point out that slices (which they call quasi-copies) are similar to materialized views in a relational database. As with views, the entries in a slice are determined by evaluating an expression that has the same form as a query on the database. In their system, each slice also has a coherency condition that specifies how far out of date the entries in a slice can be.

A weak-consistency cache server stores a random subset of database entries. It is similar to servers that store a full database copy, in that it maintains a message log and summary and acknowledgment vectors. It periodically perform anti-entropy sessions with full servers, propagating any updates it originated to other servers and receiving updates to the entries it has cached. The cache server will be unable to answer some client queries, such as keyword searches in the reference database, because it does not have a full database copy. These queries must be performed at a full or slice server.

Slice servers in a weak-consistency architecture also act much like ordinary servers, except that they store the selection condition in addition to the database, log, and timestamp vectors. The selection condition is a predicate on database entries. For simplicity assume the predicate is in disjunctive normal form; that is, it has the form

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

When a client queries the slice server, the server can determine whether it can satisfy the query if the query is equivalent to a subset of the slice predicates. For example, a refdbms slice server could answer a query on marsupials if it stored references on marsupials $\vee$ australian animals. If the server cannot satisfy the query, the query must be processed by some other server. As in the Domain Name Service, the

forwarding can be recursive, where the server forwards the query to another sever, or iterative, where it informs the client of other servers that might answer the query. Recursive forwarding makes for simple clients, but increases the dependence of clients on server correctness.

The slice server conducts anti-entropy sessions to maintain its information. These sessions are similar to ordinary ones, except that the selection predicate is passed to the partner and less information flows between the two. The slice server can only perform sessions with full servers. The full server sends updates for entries of interest to the slice server, while the slice server only sends updates it has originated.

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

The selection predicates will need to be changed from time to time to reflect changing user interest. When a slice server needs to add to its slice predicate, it potentially increases the information it maintains. The server computes the difference between the old and new predicates, then performs a special anti-entropy session to both become consistent with another server and retrieve database entries matching the difference predicate. To remove something from its predicate, potentially narrowing the server's scope, the server can discard consistent database entries without communicating with other servers.

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

4.1 Using slices for resource discovery

Many information services split their information into several separate databases, so users can have private copies and to reflect different administrative domains. When multiple databases exist, there is the separate problem of finding out about databases as they are added.

The usual solution is to have a metadatabase or database location service. This can be built using this architecture as well. A user can specify what databases they want to use by specifying a selection condition on the metadatabase entries. This condition can be used to build a slice of the metadatabase, and user queries can be routed to those databases. As new databases become available, they will be added to the slice and thus become available to the user. The user can install an agent to automatically create a slice of each new database using the user's selection condition.

5 Conclusions

In the Introduction, several principles were put forward as good ideas. The weak-consistency architecture adheres to them.

The architecture uses weak-consistency process groups for *replication.* Having multiple servers provides fault tolerance and allows the service to scale to very large user populations. The weak consistency protocols are expressly designed to allow servers to be added or removed without disturbing normal operation, meeting the goal of a *dynamic* server group. The protocols also allow servers to operate asynchronously and independently. They ensure that servers can continue to function after several other servers have failed.

The quorum multicast mechanism enables local communication. Clients using quorum multicast protocols will make use of dynamically-determined performance predictions to communicate with nearby

servers. The performance prediction information can be packaged so that portable systems can find an accurate prediction database no matter where they are connected to the Internet.

Slicing allows local sites to store a small, often-used subset of the larger database. A slice server will prefetch information based on user's interests. Portable systems can use a local slice server when they are disconnected from the Internet.

5.1 Continuing work

Several parts of the weak-consistency architecture presented here represent work in progress. At the time of writing, the refdbms system uses the basic weak-consistency group communication and membership protocols, but it does not provide slice or cache servers. The metadatabase mechanism has not yet been finalized.

The performance prediction mechanisms are another subject of ongoing research. I will be conducting a long-term performance study of the Internet to improve the analysis of prediction methods. There has been some discussion of a performance prediction service, but this has not yet been implemented.

Refdbms servers can take on different *roles* to control access to the databases. Some servers will allow both queries and updates, while others allow only queries. This paper has not addressed security and authentication problems, but they have not been ignored in the actual implementation. It appears that a new model of authentication is required for weak-consistency systems, so that a central authentication or key server does not become a bottleneck.

Acknowledgments

I have been supported in part by a fellowship from the Santa Cruz Operation, and by the Concurrent Systems Project at Hewlett-Packard Laboratories. Darrell Long, Kim Taylor, and George Neville-Neil provided helpful comments. Peter Danzig prompted my investigation of the resource discovery problem.

The refdbms system is being developed in conjunction with John Wilkes at Hewlett-Packard Laboratories, and has received assistance from Computer Systems Research Group and the Mammoth Project at UC Berkeley. The refdbms system can be obtained by writing the author at **golding@cis.ucsc.edu.**

References

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

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

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

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

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

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

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

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

[Emtage92] Alan Emtage and Peter Deutsch. archie – an electronic directory service for the Internet. *Proceedings of Winter 1992 Usenix Conference* (San Francisco, 24–24 January 1992), pages 93–110 (January 1992).

[Golding91a] Richard A. Golding. Distributed epidemic algorithms for replicated tuple spaces. Technical report HPL–CSP–91–15 (28 June 1991). Concurrent Systems Project, Hewlett-Packard Laboratories.

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

[Golding92a] Richard Golding. End-to-end performance prediction for the Internet – progress report. Technical report UCSC–CRL–92–26 (June 1992). Computer and Information Sciences Board, University of California at Santa Cruz.

[Golding92b] Richard A. Golding and Darrell D. E. Long. Quorum-oriented multicast protocols for data replication. *Proceedings of 8th International Conference on Data Engineering* (Tempe, Arizona, February 1992), pages 490–7 (February 1992). IEEE Computer Society Press.

[Golding92c] Richard A. Golding and Kim Taylor. Group membership in the epidemic style. Technical report UCSC–CRL–92–13 (22 April 1992). Computer and Information Sciences Board, University of California at Santa Cruz.

[Jacobson88] Van Jacobson. Congestion avoidance and control. *Proceedings of SIGCOMM '88*, pages 314–29 (1988).

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

[Ladin91] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: exploiting the semantics of distributed services. *Position paper for 4th ACM-SIGOPS European Workshop* (Bologna, 3–5 September 1990). Published as *Operating Systems Review*, **25**(1):49–55 (January 1991).

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

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

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

[Long92] Darrell D. E. Long. A replicated monitoring tool. Technical report UCSC–CRL–92–14 (April 1992). Computer and Information Sciences Board, University of California at Santa Cruz.

[Mattern88] Friedemann Mattern. Virtual time and global states of distributed systems. *Proceedings of International Workshop on Parallel Algorithms* (Chateau de Bonas, France, October 1988), pages 215–26, M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors (1989). Elsivier Science Publishers, North-Holland.

[Pu91] Calton Pu and Avraham Leff. Replica control in distributed systems: an asynchronous approach. Technical report CUCS–053–090 (8 January 1991). Department of Computer Science, Columbia University.

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

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

[Schroeder84] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: the growth of a distributed system. *ACM Transactions on Computer Systems*, **2**(1):3–23 (February 1984).

[Wilkes91] John Wilkes. The refdbms bibliography database user guide and reference manual. Technical report HPL–CSP–91–11 (20 May 1991). Hewlett-Packard Laboratories.