Using an object-oriented framework to construct wide-area group communication mechanisms

Richard A. Golding[†]
Vrije Universiteit, Amsterdam, The Netherlands

Darrell D. E. Long[‡]
University of California, Santa Cruz

UCSC–CRL–93–11

March 17, 1993

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

Many wide-area distributed applications, including distributed databases, can be implemented using a group communication mechanism. We have developed a family of weak-consistency group communication mechanisms, based on the timestamped anti-entropy communication protocol, that provides the scalability and fault-tolerance needed by wide-area systems. We discuss an object-oriented framework for constructing this kind of group communication mechanism, and how its components can be selected to take advantage of specific application semantics. We examine several design choices that we made in building two very different wide-area distributed database applications, and how this framework led to simple, efficient implementations in both systems.

[†] Supported in part while at the University of California by the Concurrent Systems Project at Hewlett-Packard Laboratories and by a graduate fellowship from the Santa Cruz Operation.

[‡] Supported in part by the National Science Foundation under Grant NSF CCR-9111220 and by the Office of Naval Research under Grant N00014–92–J–1807.

1 Introduction

Many wide-area systems are being, or have recently been, constructed on the Internet. These services range from *AFS* [Howard88], which provides a generic distributed file service, to highly specific applications such as the *Archie* FTP location service [Emtage92] and the *Indie* distributed indexing tool [Danzig92]. We have developed the *Tattler* reliability monitoring tool [Long92] and the *Refdbms* bibliographic database system [Golding92a]. Other possible services include distributing public cryptographic keys, software distribution, and resource location for mobile hosts.

These applications share several requirements. They all will be used by millions of users in all parts of the world. They must provide these users with a highly available and reliable service, even though the Internet is never without partitions and the hosts on which the service operate are unreliable. At the same time, users expect fast response when using the application – certainly no worse than remote applications on a local-area network provide. Some applications must be able to provide service to users with mobile computers that have been disconnected from the network.

Replication is the only way to meet these requirements. A replicated service can provide better availability than a single host, and it spreads the operation load over many replicas. In a wide-area network, replicas can be placed near users so communication latency is never too large. Replicas can also be placed on the local storage of a mobile computer. Unfortunately this approach leads to large numbers of replicas – we expect hundreds or thousands in a system that places a replica in most geographic regions, and potentially millions if users can place replicas on their disconnectable systems.

Group communication protocols are a convenient way to implement replicated services. We have found that *weak consistency* group protocols, which allow replicas to diverge temporarily from one another, provide good performance for wide-area applications.

While all the application we have mentioned require high degrees of replication, their semantics differ substantially. The Archie service only provides queries, while the Tattler and Refdbms provide both queries and updates. The host reliability database that the Tattler collects has a very different data model than the bibliographic references maintained by Refdbms. These differences allowed us to use different optimizations when implementing the systems.

We have taken a modular, object-oriented approach to constructing weak-consistency group communication mechanisms. This approach uses *frameworks* (§1.2) to build a mechanism that can take advantage of application semantics. In the remainder of this section we define the kinds of systems we are trying to build, and the assumptions we make about processes, hosts, and the network. We also briefly survey related group communication systems and systems that use frameworks. In §2 we present the framework we have developed for wide-area group communication, and introduce the Refdbms and Tattler systems. In §3 through §7 we detail each component, how it was customized for our two applications, and the advantages we derived from the customization. We present our conclusions in §8.

1.1 Model

The replicas in a replicated service coordinate their activities using a *replication* or *group communication* protocol. The two kinds of protocols solve the same problem, but replication protocols are generally written in terms of data stored at multiple replica processes, while group communication protocol are usually expressed in terms of multicast communication between group member processes. We have adopted the language of group communication in our work.

Principals are the entities that participate in group operations. Other terms such as site, replica, process, and server have been used in other systems, but we prefer a term that has little connotation. Principals are persistent: once created, they provide correct operation until explicitly destroyed. During their existence, however, there may be periods when the principal appears to stop communicating. Traditional processes, unlike principals, can fail, perhaps because the host on which they execute fails. A persistent principal can be closely approximated on a real host by a combination of stable storage and a mechanism to restart a process whenever it fails. Many Unix network services, such as name services or mail routing, behave in just this way: they are created afresh from data on disk every time a host recovers. A Refdbms principal, for example, consists of a set of programs and data files on disk, and an entry in the host's inetd configuration table.

Principals are connected by a network that allows any pair of principals to communicate as long as there are no network failures. The network can partition, dividing the principals into two or more non-communicating subsets. Gateway failure and disconnected mobile computers are two common sources of network partitioning. A principal can also be partitioned from the others when the host on which it executes has crashed. We assume all partitions are repaired in a finite but unbounded time, meaning that no host remains permanently disconnected from the network

and no network fault lasts forever. The Internet closely approximates this behavior.

The combination of eventual repair and persistent principals implies that any principal can eventually pass a message to any other principal. In pathological cases, as when two mobile computers are never connected at the same time, it may be necessary to communicate through other principals. These conditions are sufficient to make distributed consensus possible [Turek92].

1.2 Frameworks

A *framework* is an object-oriented description of the components that make up a system and how they are connected. It generalizes concepts such as layered design, often used in specifying network protocols, and structured design. It is related to the Object-Oriented Design methodology [Rumbaugh91]. A framework is useful both as a tool to design components, and as a method for sharing design and coding effort between applications.

A framework composes a number of component objects to form a complete subsystem. While it is possible to represent frameworks as run-time objects, we will only consider them as design entities in this paper.

An abstract framework defines a set of components, the interfaces they must provide, and how they will be connected. Each component is an instance of some (possibly abstract) class. A concrete framework is derived from an abstract one by specifying a concrete implementation of each component. When a concrete framework is to be instantiated, an instance of each class is created and connected. The framework therefore serves as a way to organize the classes.

The *Choices* object-oriented operating system uses frameworks to structure the implementation of process management, virtual memory, storage, and other services [Campbell92]. The x*-kernel* used a similar idea to combine components form a fast and efficient interprocess communication subsystem [Peterson90]. This mechanism has been used to construct a modular system for building consistent group communication protocols [Mishra92]. The *Synthesis* operating system uses run-time code synthesis to combine protocol objects at run-time [Massalin89].

1.3 Group communication

A group communication mechanism organizes a set of principals, acting as *group members,* into a distributed group. The mechanism provides a group multicast protocol, by which each member can send a message to all group members.

Group communication can be used to implement a replicated service using the *state machine approach* [Schneider90]. Each group member maintains a copy of the state being replicated. Operations on this state are multicast to the group members, causing each member to transform its copy of the state in the same way as other members.

Several group communication protocols have been developed. Many of these protocols provide strong consistency guarantees, usually providing one-copy serializability (1SR) so that the group members behave as if they were a single logical principal. Some protocols such as the Isis ABCAST protocol [Birman87] ensure that every principal receives every update message, so that there is never any difference between group members. Others allow some divergence, as long as it does not violate 1SR: the Isis CBCAST protocol [Birman90] and the Psync protocol [Mishra89] provide *causal* consistency [Lamport78], while voting protocols [Gifford79] ensure that a majority of the members perform the update.

Protocols that provide 1SR cannot scale to the large numbers of group members that are required for wide-area systems. When a client sends an operation request message to the group, the group cannot reply until at least a large fraction of the group has committed the operation. These protocols allow some classes of operations to involve fewer members than others, but some operation in every system must require the synchronous participation of at least $\sqrt{n}$ members from a membership of size n. Synchronous operations are not possible when the network can partition or when systems can be disconnected. They are infeasible when communication latency is long: an exchange of messages on the Internet can require as much as two seconds.

Protocols that relax their serializability guarantees can scale to large groups. *Weak consistency* protocols provide no serializability guarantees [Golding92a]. These protocols allow members to differ for a while, as long as every member eventually receives and acts on every update message. Any client request can be processed by any single member, and the resulting update message propagated asynchronously to other group members. This approach was first formalized in the *epidemic replication* protocols used in the Xerox Clearinghouse name service [Demers88]. The Lazy Replication system [Ladin91] provided a combination of weak and causal consistency.

We have developed the timestamped anti-entropy (TSAE) protocol, which provides efficient, reliable propagation of update messages throughout the group. Like the protocols used in the Clearinghouse, members periodically contact each other and exchange messages. Unlike other

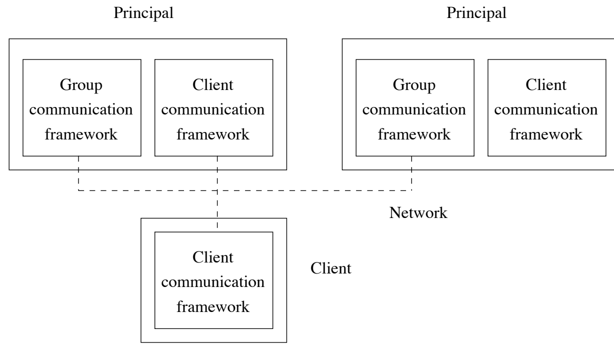

FIGURE 1: Constructing group members and clients using frameworks.

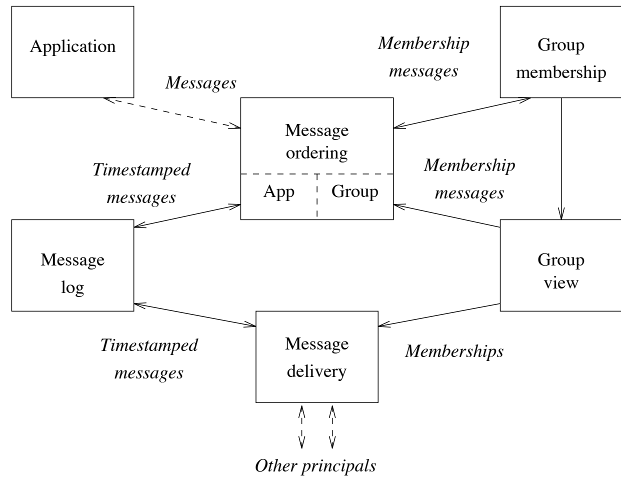

FIGURE 2: A framework for constructing a group communication system. Each principal in the group includes an instance of this framework, in the form of objects instantiated from concrete implementation classes.

weak consistency protocols, TSAE maintains summaries of the messages each principal has received and acknowledged, and uses this information to optimize communication. It ensures that every message is reliably delivered exactly once to each member, and that every member can determine when other members have received the message – allowing message logs to be maintained correctly. Unlike other group communication systems, the TSAE protocol only requires weakly-consistent membership views. We have described the TSAE protocol and its performance in more detail in other work [Golding92a].

2 Framework

We have used an abstract framework to design and implement weak-consistency group communication mechanisms based on the TSAE protocol. This framework is only part of a complete application. A complete group member would also include a framework for communicating with clients. Figure 1 shows how group members and clients might be structured.

The group communication framework provides two interfaces. The first uses a lower-level network protocol to exchange messages with other principals. The other accepts new messages from the application and delivers messages to it. The application applies the operations in delivered messages to update the local copy of group state.

The application maintains the principal's copy of group state. The semantics of the group state determine the guarantees that the group communication mechanism must provide, and hence the implementations that can be used for each component in the mechanism.

The group communication framework has five components, as shown in Figure 2: a message *log;* message *delivery;* message *ordering;* and group *membership* and associated membership *view* components. An instantiated framework includes one instance of each component.

The *message delivery component* implements a multicast communication service that exchanges messages with other principals. It decodes incoming messages and writes them to the message log, from which they will be delivered to the application or group membership component. We assume that this component uses the TSAE weak-consistency protocol.

The *group membership* component maintains the local *view* of the set of the principals that are in the group. When the set changes, perhaps because some principal has joined the group, this component communicates with the group components at other principals by sending group update messages through the message log and the message delivery component.

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

A typical communication might proceed as follows: a group member receives an update request from a client,

and translates the request into a group message. The message is given to the message ordering component, which adds a header containing ordering information and stores the message in the log. Some time later the message delivery component sends the message to other principals.

At another principal, the message is received by that principal's message delivery component and is written to the message log. Eventually, the message ordering component can determine from the log and group state that the message can be delivered to the application. The application component then updates its copy of the state according to the contents of the message.

3 The application

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

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

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

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

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

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

3.1 The Refdbms application

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

Three operations can update the database: adding, changing, and deleting references. The update operations are neither commutative nor idempotent, meaning that every update operation must be performed exactly once, and in exactly the same order by every principal, if the databases are to reach agreement. This suggests that a message delivery component should deliver update messages in a total order, and that messages should be delivered reliably.

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

Refdbms is implemented as a set of programs that communicate over the Internet using TCP (see Figure 3). Users can submit operations, which are written as messages to a log. From time to time the message delivery program propagates these messages to another replica by connecting to a daemon there, which in turn writes the update message to its log. Group membership changes are exchanged at the same time. The message delivery program and daemon together form the message delivery and group

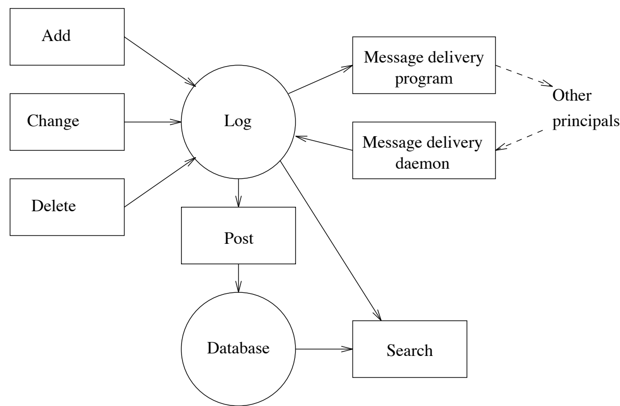

FIGURE 3: Structure of a Refdbms principal. The system uses reliable eventual delivery, implemented in the message delivery program and daemon, and total message ordering, implemented in the posting program.

membership components. The message ordering component is contained in a posting program that periodically determines what updates can be delivered to the database.

Users at different sites can submit conflicting updates. There are three sources of conflict: adding two different references with the same tag; changing one reference in two different ways; or deleting a reference then submitting another operation on it to a different principal. Different techniques are used to detect, resolve, and avoid each kind of conflict. All of the techniques make use of messages being delivered in the same order at every principal and reliable, exactly-once message delivery.

3.2 The Tattler system

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

Each measurement site runs a *tattler,* which samples host uptimes and shares these measurements with other tattlers. Collectively the tattlers maintain a list of hosts to monitor and collect statistics on them. The client interface allows hosts to be added or deleted from this list. The recorded statistics consist of a sequence of tuples of the form $\langle$*host address, boot time, sample time*$\rangle$ for each host being monitored. Each tuple represents one interval the host was known to be available.

Only one operation updates a Tattler database: merging a set of samples. A sample that is being merged into a

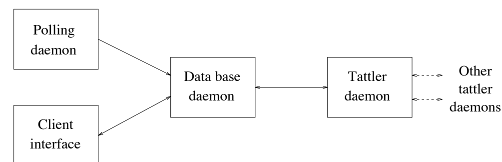

FIGURE 4: Structure of a Tattler.

database will either be disjoint from every other sample recorded for the same host, or it will overlap with another sample. If it overlaps, the two samples are combined. Otherwise, the host has been rebooted and a new interval has begun.

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

Each tattler is composed of four parts: a *client interface,* a *polling daemon,* a *data base daemon,* and a *tattler daemon*. Figure 4 shows this structure. The *polling daemon* produces sample observations. It takes samples at a specified rate, and can be requested to start or stop sampling using the *client interface*. The *data base daemon* provides stable storage for sample observations (from the polling daemon), and meta-data from the client interface and the tattler daemon. All of the group communication components are implemented in the *tattler daemon,* which exchanges samples, host lists, and membership information between tattler sites using a reliable, eventual delivery protocol.

4 Message delivery

The message delivery component fills a function similar to the transport layer in the ISO layered network model, in that it exchanges messages with other principals without interpreting message contents. In our group communication framework, it retrieves messages entered into a message log by other components and transmits them to other principals.

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

TABLE 1: Possible message delivery reliability guarantees, from strongest to weakest.

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

There are several possible message reliability levels, ranging from *atomic* to *best effort,* as listed in Table 1. Reliable mechanisms generally require extra state at each principal and induce more message traffic than unreliable ones. They require the sender to retain a copy of the message in its message log so the message can be retransmitted if necessary, and they require receivers to acknowledge incoming messages. Best effort mechanisms need not keep a copy of the message.

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

Message *latency* complements reliability: it determines how long principals may have to wait to receive a message if it is delivered to them. There are two aspects to latency: when the delivery process begins, and when it ends. The process can either begin immediately, or messages can be queued for later delivery. Once started, delivery can complete in either a bounded time, or eventually. The four combinations are listed in Table 2. Other guarantees can be used that fall between the ones listed.

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

TABLE 2: Possible message delivery latency guarantees.

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

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

While both Refdbms and the Tattler only provide eventual delivery, both systems are most convenient when updates propagate quickly. The Tattler takes steps to increase the propagation rate on observing changes to group membership or the list of monitored hosts. This propagates important changes quickly, while ordinary updates are propagated normally.

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

We have developed the *timestamped anti-entropy* protocol as one implementation of the message delivery component. It provides reliable, eventual, exactly-once message delivery in wide-area distributed systems. We have discussed this protocol in detail in other works [Golding92a]. It maintains a summary of the messages and acknowledgments it has received, and periodically exchanges batches of messages between pairs of principals. The summaries

make the exchange efficient by allowing each principal to send only the messages the other has not yet received. It masks transient failures by periodically retrying message exchanges, which makes it ideal for the Internet and for mobile computing.

5 Message log

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

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

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

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

Unfortunately, many applications cannot use state exchange. It is impossible to construct global orderings on updates before they are applied to the database because updates are always applied immediately. In the Tattler, neither the message log nor the message ordering component exist. In some distributed systems, such as Refdbms, update conflicts cannot be resolved without global message orderings. Other applications simply cannot maintain the necessary information in their group state.

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

6 Message ordering

The message ordering component is responsible for ensuring that messages are delivered to the application in a well-defined order. This order may be different from the order in which messages are received. For example, an application should receive updates to a database record after the message creating the record. Even if the messages were sent in the right order, they may be rearranged in transit and arrive at their destination in a different order.

Table 3 lists the four most common message orderings. Some of these ensure that messages are delivered to every principal in the same order. Other orderings respect potential causality: if there is any possibility that the contents of one message depend on the effects of another message, the ordering component guarantees that the other message will be delivered first.

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

The operations on a Refdbms database, on the other hand, are order-dependent. The value of a reference is the value of the last update applied to it. For two principals to record the same value for a reference, they must apply the same updates in the same order. In Refdbms, the ordering component tags each update message with a timestamp from its originator's clock. Messages are then applied to the database in timestamp order. This technique correctly

TABLE 3: Four popular message ordering guarantees.

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

produces a total (but not causal) message order as long as every principal has access to a local clock that is loosely synchronized with other clocks. This scheme is biased so that messages from principals whose clocks lag behind others will always be applied before those with faster-running clocks. Clocks on Internet hosts are generally synchronized to within a few minutes. The mean interval between concurrent updates is generally much larger, so this bias has little effect.

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

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

Our performance evaluations [Golding93] have shown that the simple total ordering used in Refdbms does not substantially delay message delivery on average. Messages are delayed at most by the maximum difference between clocks, plus the delay between receiving a message and receiving a greater or equal timestamp from every other group member.

7 Group membership

This component is responsible for maintaining the local *view* of the principals that make up the group. As with updates to application state, changes to membership views are propagated eventually from one group member to another, which allows the operations of adding and removing group members to scale to very large groups. This approach can be contrasted to other group membership systems such as Isis or Psync, which maintain consistent group views and require synchronous operations to change the group membership. Persistent group members further simplify group management because failure of group members need not be considered.

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

Principals join and leave the group by executing **join** and **leave** protocols. To join the group, the principal contacts an existing member, which acts as the new member's *sponsor.* The sponsor first adds the new member to its local group view, sends an update message to the group, then sends a copy of its application state, message log, and group view to the new member. A principal is considered to be a member when it and its sponsor commit their their changed group views. To leave a group, a member sends an update message to the group indicating its intent to leave. When at least one other member has acknowledged the message, the member can destroy its state. Between sending the message and receiving the acknowledgment the member enters a special state where it cannot orig-

inate messages, but does exchange messages with other members. We have proven that this approach is correct when composed with the TSAE message delivery protocol [Golding92b, Golding92a]. We have also developed a variant on these protocols that allow two groups to merge.

Our experience with this group membership mechanism in Refdbms is generally positive. It uses the join-leave implementation because there is no sensible way to merge two databases. The Tattler uses the implementation that allows group merges because its sampling operation is based on merging sample results.

In some systems these group view update messages are processed by the message ordering component so that group changes are ordered with respect to application messages. For example, every member can observe a principal joining the group at the same point in the message sequence. In the Refdbms and Tattler systems, however, this sort of consistency is not important because none of the operations on group state depend on the membership. Therefore group messages are delivered independent of application update messages.

8 Conclusions

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

We have developed a framework for constructing group communication mechanisms. The framework consists of an application, which defines the semantics of the state shared among the group; a message delivery component, which communicates messages from one member to another; a message log, which stores the messages until they have been delivered to the group; a message ordering component, which assembles the incoming stream of messages into a coherent order and delivers them to the application; and a group membership component, which maintains a view of the membership.

We constructed custom group communication for both applications using our framework. This allowed us to match the group semantics to the application, enabling us to identify which components required strong guarantees, and which could be implemented using relaxed guarantees for better performance. In Refdbms, for example, the message ordering component must delay messages so they can be totally ordered, while the ordering component in the Tattler delivers messages as soon as they are received.

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

Acknowledgments

John Wilkes, of the Concurrent Systems Project at Hewlett-Packard Laboratories, developed the original Refdbms system and has assisted us in developing the distributed version. Refdbms development has also been assisted by the Computer Systems Research Group and by the Mammoth Project, both at the University of California at Berkeley. Kim Taylor, of UC Santa Cruz, assisted the initial development of the timestamped anti-entropy protocols.

References

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

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

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

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

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

[Danzig92] Peter B. Danzig, Shih-Hao Li, and Katia Obraczka. Distributed indexing of autonomous Internet services. *Computing Systems*, **5**(4):433–59. USENIX Association, Fall 1992.

[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 directroy service for the Internet. *Proceedings of the Winter Conference* (San Francisco), pages 93–110. Usenix Association, January 1992.

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

[Golding92a] Richard A. Golding. *Weak-consistency group communication and membership*. PhD thesis, published as Technical report UCSC–CRL–92–52. Computer and Information Sciences Board, University of California at Santa Cruz, December 1992.

[Golding92b] Richard A. Golding and Kim Taylor. Group membership in the epidemic style. Technical report UCSC–CRL–92–13. Computer and Information Sciences Board, University of California at Santa Cruz, 22 April 1992.

[Golding93] Richard A. Golding and Darrell D. E. Long. Modeling replica divergence in a weak-consistency protocol for global-scale distributed data bases. Technical report UCSC–CRL–93–09. Computer and Information Sciences Board, University of California at Santa Cruz, February 1993.

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, **6**(1):51–81, February 1988.

[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.

[Long92] Darrell D. E. Long. A replicated monitoring tool. *Proceedings of 2nd Workshop on the Management of Replicated Data*, pages 96–9, November 1992.

[Massalin89] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. *Proceedings of 12th ACM Symposium on Operating Systems Principles* (Litchfield Park, AZ, 3–6 December 1989). Published as *Operating Systems Review*, **23**(5):191–201, December 1989.

[Mishra89] Shikavant Mishra, Larry L. Peterson, and Richard D. Schlichting. Implementing fault-tolerant replicated objects using Psync. *Proceedings of 8th Symposium on Reliable Distributed Systems* (Seattle, WA), pages 42–52. IEEE Computer Society Press, catalog number 88CH2807-6, 10–12 October 1989.

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

[Peterson90] Larry Peterson, Norman Hutchinson, Sean O'Malley, and Herman Rao. The *x*-kernel: a platform for accessing Internet resources. *IEEE Computer*, **23**(5):23–33, May 1990.

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

[Schneider90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, **22**(4):299–319, December 1990.

[Turek92] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, **25**(6):8–17, June 1992.