Design choices for weak-consistency group communication

Richard A. Golding and Darrell D. E. Long

UCSC–TR–92–45

October 12, 1992

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

Many wide-area distributed applications can be implemented using distributed group communication, a mechanism for coordinating the activities of related processes running at different sites. We have developed a modular architecture for group communication systems that can be used to build a system tailored to application requirements. We focus on *weak consistency* mechanisms for group communication, since these allow highly efficient operation on wide-area internetworks. We examine several design choices that can be made in building a weakly-consistent group communication mechanism, and discuss the decisions we made in building two very different wide-area applications. The architecture accommodates both systems well, and allows several application-specific decisions that increase efficiency and flexibility.

Keywords: group membership, weak consistency replication, mobile computing systems, bibliographic databases, reliability measurements

1 Introduction

Wide-area distributed systems that are fault-tolerant and scalable typically include a large number of processes, all running at different sites. A *group communication* or *distributed process group* mechanism is a convenient basis for constructing such systems. The group communication mechanism provides ways for processes to coordinate their actions by sending messages to the group. The group mechanism automatically maintains the list of processes that make up the group, so applications can be written in terms of an abstract group without having to know exactly what processes make it up.

The member processes can multicast messages to the group using a *group communication* protocol, which guarantees that member processes will have a particular degree of *consistency* in their view of the messages that have been sent. Processes join and leave the group using a separate *group membership protocol*.

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

In contrast, we are exploring the weakest appropriate guarantees and the applications that can use them. We are concentrating on wide-area systems connected by an internetwork, where communication is expensive and unreliable. We are also investigating mobile computer systems, where a system may be disconnected from the internetwork for extended periods, or may be connected only by a low-bandwidth cellular modem or wireless link.

Efficiency is of paramount importance in wide-area and mobile systems. Strong consistency systems require expensive protocols, and perform poorly (or not at all) when communication is unreliable or when the network is partitioned. By contrast, weak consistency systems can use protocols that use fewer network packets, allow caching and delayed operation for mobile systems, and are not affected by many forms of processor and network failure.

We are investigating ways to tailor group communication systems to specific application requirements. To that end we have developed an architecture that we believe will lead to a set of reusable components that can be connected to build custom wide-area group communication systems.

In this section we will overview the architecture. In Section 2 we discuss how we used this architecture to build two different systems: the Refdbms bibliographic database [Golding92a] and the Tattler distributed reliability monitor [Long92]. In Sections 3, 4, and 5 we detail the components of this architecture and the customizations we used for each application. In the Conclusions we discuss the lessons we have learned, and provide some directions for future research.

1.1 Assumptions

We use the term *process* for the principals that participate in group operations. This makes it necessary for us to define a process in a rather strict manner. Other terms such as site, replica, and server may seem appropriate, but are well-defined in other contexts and may have equally confusing connotation.

Processes, as we are using the term, survive temporary failures and host crashes. They have some form of stable storage to record information that must survive failure. Both processes and hosts fail by crashing,

so that spurious data are never transmitted on the network or written to stable storage. In practice a carefully implemented system can closely approximate this ideal. Many Unix network services, such as network file systems, name services, and mail routing behave in just this way: they are created afresh from data on disk every time a host recovers.

We assume that hosts have loosely-synchronized clocks. Hosts can either fail (by crashing) and recover, or are permanently removed from service. We assume that the network is sufficiently reliable that any two processes can eventually exchange messages, but it need never be completely free of partitions.

Many of these assumptions about networks, processes, and hosts are necessary to make consensus possible [Turek92]. The Internet approximates synchronous processors with unbounded communication; when combined with processes that do not fail, or at least are reincarnated on demand, multiple processes can reach a shared decision.

1.2 Group communication architecture

We are developing an architecture for building custom weak-consistency group communication systems. This architecture has four components, as shown in Figure 1: an *application,* message *delivery* and *ordering* components, and a *group membership* component. These components communicate using shared data structures.

The message delivery component implements a multicast communication service that exchanges messages with other processes. It decodes incoming messages and routes them either to the group membership component or to the message log, from which they will be delivered to the application. It may also maintain summary information of the messages sent and received that can be used by the message ordering component.

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

The message ordering component processes incoming messages to ensure they are presented to the application according to some ordering. The ordering that is used depends on the application. This component also processes outgoing messages so that the ordering components at other processes will have enough information to properly order messages.

Different implementations of each component would provide different levels of service, so that they could closely match the needs of an application. The two systems we have implemented are steps toward the goal of building a set of reusable implementations for each component.

2 The application

Distributed process groups are usually used to coordinate operations on data shared among the group members. This shared state has a logical *data model,* whether or not the data are actually stored at each process. The data model consists of the data to be shared, the operations to be performed on that data,

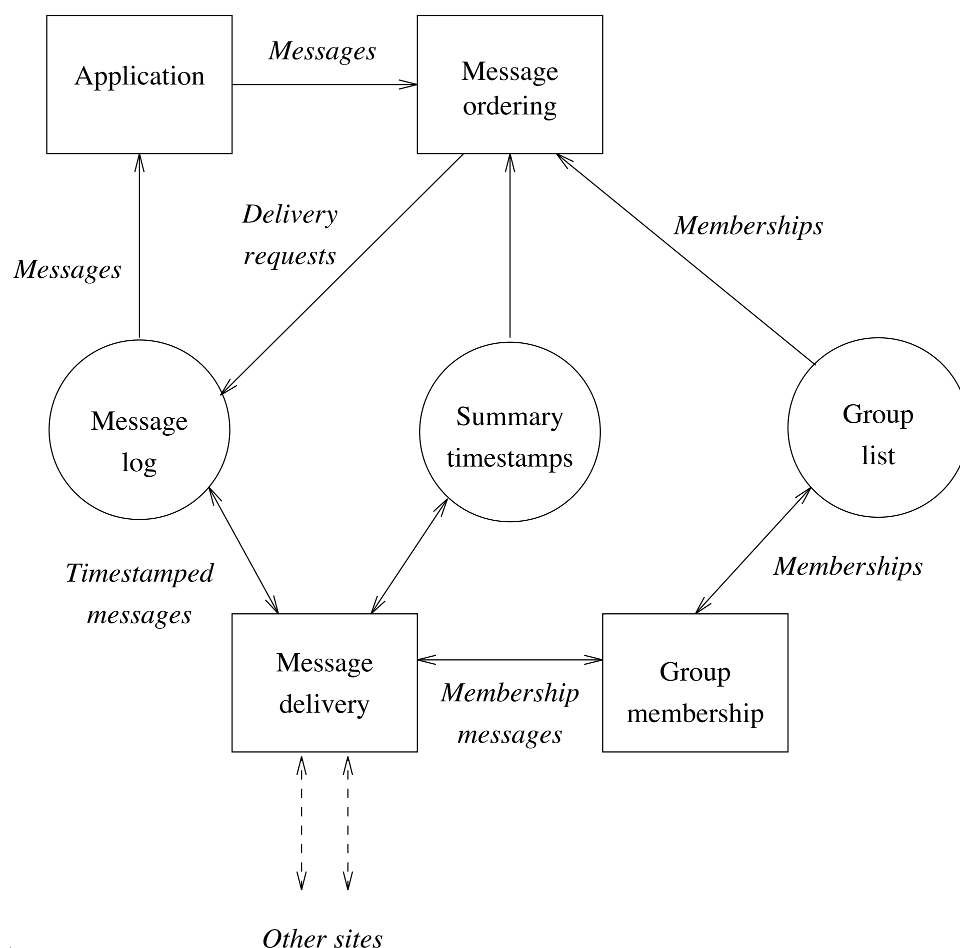

FIGURE 1: The components implementing a weak-consistency group multicast system.

---

and correctness constraints that must be maintained. The guarantees provided by the group communication mechanism are determined by these aspects of the data model.

The operations allowed on the data can dictate a particular message ordering. If all operations are idempotent, that is, if they can be applied in any order with the same net result, message delivery order is unimportant. It is more likely though that operations will be order-dependent, in which case a total message order will ensure that every process computes the same result for each operation.

If operations are order-dependent, the application will need to provide mechanisms for detecting and resolving conflicting updates. 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, processes make optimistic updates that must be checked before they are applied to the database. Delivering messages in a total order can provide a simple solution for consistent conflict detection.

Some applications require that the data have unique identifiers. Unique identifiers are a common source of update collisions, and the collisions can be especially difficult to resolve. 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 databases.

The shared data may have 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.

2.1 The Refdbms system

The Refdbms 3.0 system implements a distributed bibliographic database. It is based on the version 1 system that has been under development for several years at Hewlett-Packard Laboratories for sharing bibliographic information among a research group. Users can search databases by keywords, use references in TEX, locate copies of papers, and add, change, or delete references. We have extended it into a distributed, replicated database.

A Refdbms database consists of a set of references, each with an internal *unique identifier* and a *tag* like **Golding91** that humans can use to name a reference. At all times the internal identifier is guaranteed to be unique. The tag *should* be unique, but this is not guaranteed for newly-added references until all sites holding a replica of the database can come to consensus. The references are indexed by the tag and by an inverted index of content keywords.

Refdbms is implemented as a set of programs that communicate over the Internet using TCP (see Figure 2.) Users can submit operations to an update log. From time to time an anti-entropy program (Section 3.1) propagates these messages to another replica by connecting to a daemon there, which in turn writes the update message to its log. The anti-entropy program and daemon together form the message delivery and group membership components. The message ordering component is contained in a posting program that periodically determines what updates can be delivered to the database.

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 for it to a different replica process. The basic mechanism for handling conflicts is to process update messages in the same order at every process – that is, the message ordering component imposes total delivery order. We will discuss how conflicts are resolved in more detail in Section 4.

2.2 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. A record of the form $\langle$*host, poll method, poll interval*$\rangle$ is kept for each host. The client interface allows hosts to be added or deleted from this list. The recorded statistics are stored in a database. This database stores

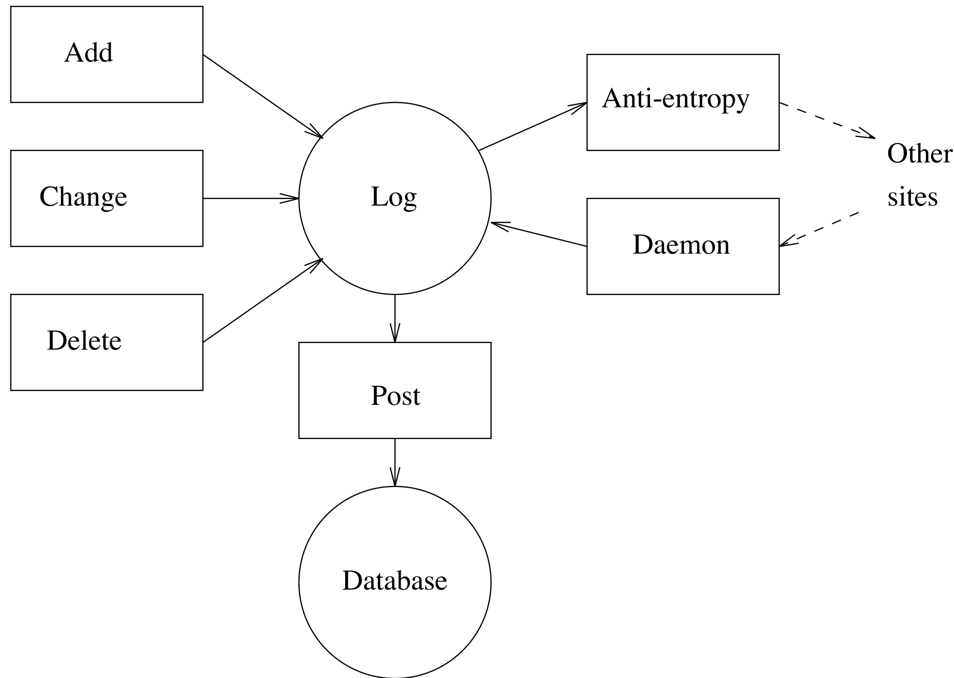

FIGURE 2: Structure of a Refdbms replica process.

tuples of the form ⟨*host*, *boot time*, *sample time*⟩. The host identifier is used as a key, and is assumed to be unique across all Tattlers since it is derived from an external unique host address.

Each tattler is composed of four parts: a *client interface,* a *polling daemon,* a *data base daemon,* and a *tattler daemon*. Figure 3 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 the timestamped anti-entropy protocol (Section 3.1).

Unlike Refdbms, the Tattler does not explicitly implement a message log. Samples represent idempotent operations on the host uptime database, and that database contains timestamp information. Two tattlers can exchange portions of the uptime database as if they were update messages and merge the information to obtain a database with better coverage of the monitored hosts. They are merged based on overlapping intervals. A sample reaches back some distance in time. If that point is contained in the previous interval, then that interval is extended. Otherwise the machine has been rebooted and a new interval begins.

3 Message delivery

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

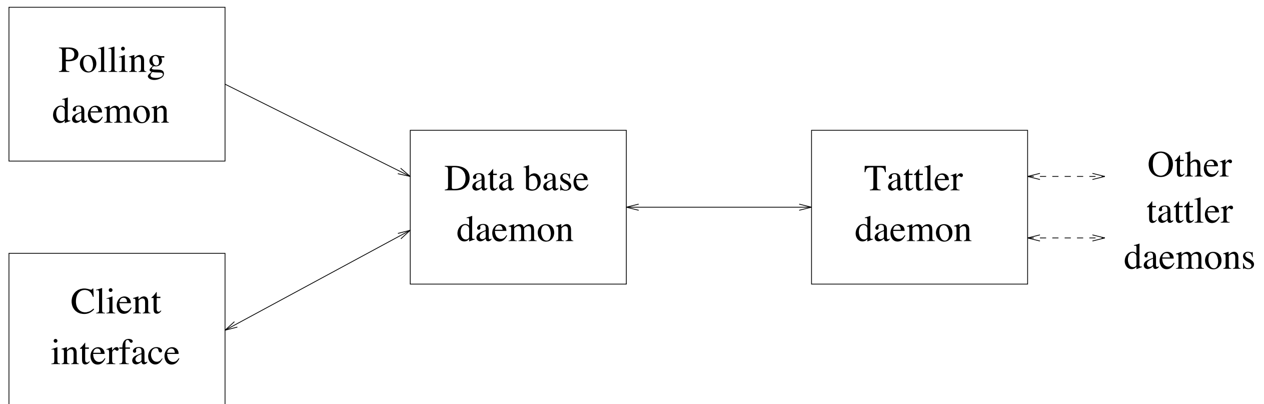

FIGURE 3: Structure of a Tattler.

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.
Reliable	Delivered to every functioning group member or to none, but failed members need not receive the message.
Quorum	Delivered to at least some fraction of the membership.
Best effort	Delivery attempted to every member, but none are guaranteed to receive the message.

contents. In our system, it retrieves messages entered into a message log by other components and transmits them to other processes.

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

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

We used reliable delivery for both Refdbms and the Tattler. Reliable delivery is essential for Refdbms, because even a single lost message can cause some replica processes to 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. However, reliability costs little and makes the system easier for users to understand.

TABLE 2: Possible message delivery latency guarantees.

Kind	Guarantee
Synchronous	Delivery happens immediately, and completes within a bounded time.
Interactive	Delivery happens 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.

Message *latency* complements reliability: it determines how long processes may have to wait if a message is delivered to them. Table 2 lists latency guarantees that span the range of possibilities, from synchronous to eventual delivery. Other guarantees can be used that fall between the ones listed.

We used eventual delivery in both our 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 spend a substantial fraction of the time disconnected.

While interactive delivery is not necessary, both systems are most convenient when updates propagate quickly. The Tattler performs extra anti-entropy sessions on observing changes to group membership or the list of monitored hosts. This propagates important changes quickly, while ordinary updates are not propagated until the next anti-entropy session.

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.

3.1 The timestamped anti-entropy protocol

We have developed the *timestamped anti-entropy* protocol for reliable, eventual message delivery in our systems. Details can be found in other papers [Golding92d, Golding92b, Golding93]. It maintains a summary of the messages and acknowledgments it has received, and periodically exchanges batches of messages between pairs of processes.

As long as every process periodically performs these exchanges, every message will eventually be delivered to every process – reliable eventual delivery. It masks transient failures by periodically retrying message exchanges, making it ideal for for the Internet and mobile computing.

The protocol maintains two *summary timestamp* vectors. One records which messages have been received, while the other tracks acknowledgments from other processes. The vectors contain one timestamp for every process in the group. A process has received every message sent by another process up to the time

recorded for that other process in the local vector. The acknowledgment vector tracks acknowledgment messages in the same way.

The summary vectors provide a compact way to determine how far out of date one process is with respect to another. This allows processes to exchange exactly those messages that need to be sent, so that every message is sent exactly once to every process. The acknowledgment vectors form implicit acknowledgment of receipt, providing an efficient way to detect when a message has been delivered to every group member. For mobile systems communicating over low-bandwidth wireless networks the summaries can be used to determine how far out of date information on the mobile system is, perhaps so that the application can prompt the user to provide a higher-bandwidth connection.

From time to time two processes contact each other and exchange messages. They first exchange their timestamp vectors. Each can then determine what messages the other process has not yet received, and reliably transmit those missing messages. At the end of this exchange both processes adjust their timestamp vectors to reflect the messages just received. Acknowledgments are transmitted implicitly by exchanging acknowledgment vectors.

Our performance evaluations have shown that messages propagate much faster when anti-entropy is coupled with an unreliable best effort multicast [Golding92c]. In addition, different policies can be used to select partners for anti-entropy sessions, and these policies affect the time required to propagate messages and the network traffic generated by the application.

This protocol requires $\Theta(n)$ state for the timestamp vectors, and the clocks at every host must be synchronized to within some ϵ. We have developed a similar protocol, also developed independently by Agrawal and Malpani [Agrawal91], that uses $O(n^2)$ size timestamp arrays but allows unsynchronized clocks. We have found that most Internet hosts are synchronized to within a few minutes.

The Grapevine distributed database used similar mechanisms [Birrell82, Demers88]. 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 using a combination of best effort multicast, unreliable message exchange, and a form of anti-entropy session. Only anti-entropy provided reliable delivery. The Grapevine anti-entropy protocol did not use summary timestamp vectors; instead, it directly exchanged database contents, using checksums to determine when enough entries had been exchanged to make both processes mutually consistent. Lacking summary vectors, messages could not be ordered, information deletion could not occur reliably, and a process might receive an update more than once. The protocol therefore could not be used for applications like Refdbms that need stronger guarantees.

3.2 Propagating logs versus databases

There are two models for storing updates: either each update can be entered into a message log and later applied to the database or sent to another process, or it can be immediately applied to the log and its *effects* can be transmitted to other processes. Refdbms uses a message log, while the Tattler operates from the database.

Message logs are simple. Every update operation produces one update message, which is then sent to every group member process. After the update message arrives at other processes, it can be applied to the

database. The messages can be tagged with timestamp information so that any ordering is possible. The database need not maintain any extra information to ensure that messages are applied in the right order.

Propagating effects rather than updates is more complex, but it can be a more robust and efficient solution when eventual delivery is allowable. Since there are no messages, the database must maintain ordering or timestamp information for each entry. In the Tattler each entry already contained a sample timestamp. When updates are to be propagated from one process to another, database entries rather than messages are exchanged. In the Tattler, the database entry timestamp is used just as a message timestamp would be. The value of the database entry may reflect the effects of more than one update operation, so there can be fewer messages sent between processes than update operations. Some systems that use database exchange can also tolerate some lost "messages" because the database value can be obtained from different process in a later update exchange.

Unfortunately, database exchange cannot be used for many applications. It is impossible to construct global orderings on updates before they are applied to the database because updates are always applied immediately. In some distributed systems, such as Refdbms, update conflicts cannot be resolved without global message orderings.

Deleting database entries requires special consideration when message logs are not used. Deletion should be a stable property: once an entry has been deleted, it should remain so forever. The entry should not spontaneously reappear, though of course a new entry 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 processes so that no process can miss the operation and re-introduce the entry to other processes. In Grapevine these records were called *death certificates* [Demers88], while the Bloch-Daniels-Spector distributed dictionary algorithm [Bloch87] places timestamps on the gaps between entries as well as on the entries themselves. The Tattler uses the death certificate approach to track hosts that should no longer be polled.

4 Message ordering

This 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 some of the most common message orderings. Some of these ensure that every process delivers messages in the same order. An application can use this property to ensure that updates occur in the same order everywhere. Total causal ordering, for example, is provided by the Isis ABCAST protocol [Birman90]. Other orderings respect potential causality [Lamport78]. That is, if there is any possibility that the contents of one message were caused by another message, the other message will be delivered first at every process.

Message ordering guarantees can be limited just to message senders, to the process group, or among all processes anywhere in the network. The FIFO guarantee is limited to message senders, and can be useful when each process is sending out an independent stream of updates. Group consistency is more common, but it is insufficient when the group must interact with other systems. Ladin's Lazy Replication mechanism

TABLE 3: Some popular message ordering guarantees.

Kind	Guarantee
Total, causal	The strongest ordering. Messages are delivered in the same order at every process, and that order respects potential causal relations between messages.
Total, noncausal	Messages are delivered in the same order at every process, 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 process. If they are not, they may be delivered in different orders.
FIFO	Messages from each process will be delivered in order, but the messages from different processes may be interleaved in any order.
Unordered	Messages are delivered without regard for order.

[Ladin91] provides ways to order messages by any potential causal relation that can be detected by a process, even those caused by activities outside the group.

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

4.1 Using message ordering

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

The operations on a Refdbms database, on the other hand, are order-dependent. The value of a reference is the value of the last update applied to it. For two processes to record the same value for a reference, they must apply the same updates in the same order. For Refdbms, each update message was tagged with a timestamp from its originator's clock. Messages were then applied to the database in timestamp order.

This simple ordering is total, but not causal. Furthermore, it is biased so that messages from processes whose clocks run slow will always be applied before those with faster-running clocks. As long as clocks are loosely synchronized to within some ϵ and the mean time between updates to a reference is larger than ϵ this bias has no effect.

Message ordering can require delaying updates for extended periods. Users, on the other hand, may need to use the results of an update immediately. Refdbms resolves this by making recent database changes available in a *pending image* of a reference. If there are conflicting updates, the contents of the pending image are only an approximation of the final reference. The pending image is removed when there are no update operations pending for the reference. The pending can be retrieved by providing a tag of the form **Golding92.pending.** This allows citations to be embedded in a LaTeX document or sent to another user by electronic mail.

4.2 Handling update collisions

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

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

Two newly-added references conflict if they have the same *tag*. Recall that tags are assigned by users and are supposed to be unique within a database, but this cannot be guaranteed when users at different sites add references independently. This kind of conflict is detected when the second add message is delivered to the application at each process. The first reference will already have been added to the database. When Refdbms finds that the tag has already been used, it computes a new tag for the reference by adjusting a suffix on the tag: **Golding90** would become **Golding90a**, and **Long90b** would be changed to **Long90c**.[1] The update message can then be re-processed using the new tag.

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

Conflicting change operations in Refdbms are more complex. They are not explicitly detected or resolved; instead, change operations are simply applied in the same order by every process. However, change operation messages only carry the *difference* the change is supposed to apply to the reference. In this way if one user corrects the spelling of an author's name while another user at a different process independently adds keywords, both changes will eventually appear in the reference. Fields can be grouped together, and a separate policy is used for each group of fields. For example, a change to any author-related

[1]There is a limit of up to ten suffix characters, but is most unlikely that there will be more than 26^{10} references from one author in one year.

field will result in all author-related fields being overwritten, while location lines can be inserted or deleted individually. This technique reduces the probability that two change operations will conflict, even if they apply to the same reference.

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

5 Group membership

This component is responsible for maintaining the *view* of what processes make up the group. The group components at different processes exchange messages among themselves separate from the normal application messages. In some systems these group operation messages are considered by the message ordering component so that group changes are consistent with application messages. This way every member can observe, say, a process joining the group at the same point in the message sequence. In the two systems we built, however, this sort of consistency is not important and so group messages are delivered independent of application messages.

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

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

Group state management is an essential part of the join and leave protocols. When a process finishes executing the join protocol, it must have received a copy of the group state. This copy will be "consistent" in the sense that the new process will not miss the effects of any messages and will not compromise message reliability, latency, and ordering guarantees. Group membership mechanisms that allow groups to merge must also provide a way to merge their state.

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

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

5.1 Using group membership

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

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

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

6 Conclusions

We have built the Refdbms and Tattler applications on the Internet. These are two of the many wide-area applications that are likely to become available in the next several years. Both applications were constructed as a collection of processes organized into a weak-consistency process group.

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

We have developed an architecture for constructing weak-consistency group communication mechanisms. 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. We find it a valuable alternative to ad hoc application construction.

Authentication is perhaps the most difficult problem we have not yet addressed. All the authentication mechanisms we know use trusted centralized servers. We have not yet discovered a good model that matches the fault tolerance and distribution of weak-consistency group communication.

Some modular architecture of this sort is necessary if wide-area distributed applications are to become common, efficient, and easy to construct. Building programming language translators was once an expensive process, requiring many years of programmer effort; the separation of compilation into a distinct set of phases and the introduction of interoperable tools for each phase has made compiler-writing a subject for one-semester undergraduate courses. We believe that our approach to structuring wide-area applications will yield similar results for wide-area applications.

Acknowledgments

Refdbms development has been assisted by the Computer Systems Research Group and by the Mammoth Project, both at the University of California at Berkeley. This paper was written using Refdbms.

Richard Golding was supported in part by a fellowship from the Santa Cruz Operation, and by the Concurrent Systems Project at Hewlett-Packard Laboratories.

Darrell Long was 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.

References

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

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

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

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

[Birrell82] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: an exercise in distributed computing. *Communications of the ACM*, **25**(4):260–74 (April 1982).

[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).

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

[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).

[Golding92a] Richard Golding. A weak-consistency architecture for distributed information services. *Computing Systems* (1992). Usenix Association. To appear.

[Golding92b] Richard A. Golding. The timestamped anti-entropy weak-consistency group communication protocol. Technical report UCSC–CRL–92–29 (July 1992). Computer and Information Sciences Board, University of California at Santa Cruz.

[Golding92c] Richard A. Golding and Darrell D. E. Long. The performance of weak-consistency replication protocols. Technical report UCSC–CRL-92–30 (June 1992). Computer and Information Sciences Board, University of California at Santa Cruz.

[Golding92d] 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.

[Golding93] Richard A. Golding and Darrell D. E. Long. Simulation modeling of weak-consistency protocols. *Proceedings of International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (January 1993). To appear.

[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).

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

[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 (10–12 October 1989). IEEE Computer Society Press, catalog number 88CH2807-6.

[Tanenbaum81] A. S. Tanenbaum. *Computer networks* (1981). Prentice-Hall.

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