REPLICATION

11.1 Introduction
11.2 Basic architectural model
11.3 Consistency and request ordering
11.4 The gossip architecture
11.5 Process groups and ISIS
11.6 Summary

Replication is a key to providing good performance, high availability and fault tolerance in distributed systems. In this chapter we describe design issues concerning the replication of data and other resources. The main problem tackled is that of applying operations from clients to multiple replicas in a consistent way, while maintaining reasonable response times and system throughput rates.

The chapter describes two approaches to replication: the gossip approach and the process group approach. In the gossip architecture, updates are lazily propagated between replicas. In the process group approach, updates are multicast to all replicas. Process groups are also more widely applicable, beyond replication. The ISIS programming environment, based on the notions of process groups and virtually synchronous execution, is described.

The chapter deals primarily with the order in which updates are applied, and identifies the main ordering types as causal, total and sync-ordered. Algorithms are given for achieving causal and total ordering, for achieving atomic multicast, and for ordering multicasts with respect to changes in process groups.
11.1 Introduction

Replication is the maintenance of on-line copies of data and other resources. It is an important tool to the effectiveness of distributed systems, in that it can provide enhanced performance, high availability and fault tolerance. Replication is used widely. For example, the USENET system maintains replications of items posted to electronic bulletin boards across the Internet, the replications being held within or close to the various organizations that provide access to it. The DNS naming service, described in Chapter 9, maintains copies of name-to-address mappings for computers and other resources, and is relied on for day-to-day access to services across the Internet. The caching of file and other data in client computers is another form of replication, since the data held in caches and at servers are replications of one another. In addition, replication is intrinsic to distributed shared memory implementations discussed in Chapter 17.

The motivations for replication are as follows:

Performance enhancement: The caching of data at clients is by now familiar as a means of performance enhancement, but replication can be applied equally and at a higher level of servers to enhance service response times. Data that are shared between large client communities should not be held at a single server, since this component will act as a bottleneck that slows down responses and has a limited throughput capacity in terms of requests processed per second. It is preferable to distribute copies of the data to several servers, and to arrange that each provides data to a smaller community of users which is close to it in network terms. If the servers provide read-only access to data then replication is relatively trivial but this is not always useful, since almost all data are subject to at least occasional modification.

Enhanced availability: If service data are replicated at two or more fail-stop independent servers running equivalent software and reachable by independent communication links (where these can fail), then client software can in principle access an alternative server should the default server fail or become unreachable. That is, the percentage of time during which the service is available can be enhanced by replicating servers. If \( n \) servers each have an independent probability \( p \) of failing or becoming unreachable, then the availability of a data item stored with each of these servers is:

\[
1 - \text{probability(all managers failed or unreachable)} = 1 - p^n
\]

For example, if there is a five per cent probability of any individual failure over a given period, and if there are two managers, then the availability is \( 1 - 0.05^2 = 0.9975 \) per cent. An important difference between caching and server replication is that caches do not necessarily hold collections of data items such as files in their entirety. So caching does not necessarily enhance availability. The Coda file system provides an interesting counter-example, which cached copies of whole files are used as replicas during disconnected operation.

Fault tolerance: If each of a collection of servers processes every request from a client in parallel, then it is possible to provide guarantees of correct

processing, even though one (or more) of the servers should fail. This is a stronger statement than that of high availability, because it can include real-time guarantees, and also guarantees against arbitrary (so-called Byzantine) failures rather than simple, so-called fail-stop failures in which a component stops after changing to a state in which its failure can be detected. For example, a space-flight engine's thrust and direction could be calculated within a crucial time interval by an on-board computer that continues to function, even though its peers crashes. Furthermore, if one of several replica computers fails in such a way that it does not stop responding but produces random results, then its results can be discarded if two or more of its peers produce the same, majority results.

The chief requirement when data (or other resources) are replicated is for replication transparency. That is, clients should not normally be aware that multiple physical copies of data exist. As far as clients are concerned, data are organized as individual logical data items (or objects), and there name only one item in each case when they request an operation to be performed. Furthermore, under replication transparency operations return only one set of values. This is despite the fact that operations may be performed upon more than one physical copy in concert.

The other general requirement for replicated data – one which can vary in strength between applications – is that of consistency. Consider that clients make requests, and that requests involve updates as well as reads upon a set of logical data items (there is no consistency problem if the data are read-only). It is not normally acceptable for different clients to obtain differing results when they make the same requests affecting the same logical data items. At least, this is not acceptable if the results lead to detectable and significant inconsistencies between different applications or users, or if the results break a constraint within a single application.

Of course, updates are normally eventually applied to all copies of a data item; but it is the order in which operations occur at a given copy which may affect the results of client operations. Note that the problem of identifying and locating all the copies of a data item is itself an interesting issue, but one for which there is little space in this chapter. In Chapter 8 we describe, for example, how Andrew servers keep track of cached file copies held by clients.

The example of a bulletin board is now introduced, which will be used later in this chapter. Figure 11.1 illustrates the information displayed to a client 'Smith' accessing the os.interesting bulletin board. The items are numbered, and have sender names and
The user can select any item to view, can post new items, and in particular he can respond to a message. In the latter case the subject of the response is automatically given as ‘Re: Microkernels’, for example, if the subject of the item which it refers is ‘Microkernels’. The bulletin board reader program sends new items to a server, and also fetches items from a server. It automatically requests copies of any items in the bulletin board from the server at regular intervals. Once they are downloaded they are stored locally, but they remain stored at the server, of course.

Suppose that clients at several universities share the same bulletin boards, but that the clients are sufficient in number and the universities sufficiently far apart that the bulletin boards are replicated, with a server at each university holding a replica. A simple asynchronous model of replication management is one in which all client requests are processed by the local replica server. The local replica servers communicate updates to all other servers managing replicas of the same data items, in this case bulletin board items. In this asynchronous model each server allows a client operation to return as soon as it has been performed locally, and communicates updates to its peers whenever convenient, or perhaps periodically. Servers process updates as they arrive.

Under this replication scheme, the display shown to a client ‘Jones’ at another university might be as shown in Figure 11.2. Note that the set of items shown is the same except for the item from ‘A.Sahiner’ on Jones’ display, which has arrived at Jones’ local server but has not arrived at Smith’s local server (or has not yet been retrieved from the server). However, the numbering and the ordering of the items differ between the two. Since propagation of updates is asynchronous and since items are posted from different universities, items may not arrive in the same order at any two universities. This is particularly so when a site fails after it has sent an item to some sites but not others, the local servers simply number the items as they arrive, then there is similarity to record why the item numbers should be the same for Smith and Jones. Finally, note in particular items 22 and 24 on Jones’ display. Intuitively, an item referring to another item should appear after it. But the variable propagation delays to Jones’ university have reversed this natural ordering.

These characteristics are similar to those encountered when using the USENET bulletin board system. That system is used by many people, but the inconsistencies are a nuisance. It would be more convenient if users could refer unambiguously in items to, for example, ‘item number 25 on os.interesting’; it would also be more satisfactory if an item whose subject is X could always be read before a response whose subject is re:X. On the other hand, it could be considered a convenient characteristic of this system that a new item marked Re:X can be read, even though the X item cannot be read currently because all links are down to the university at which it was originally posted.

Now consider by contrast a totally synchronous model of replica management, in which all update requests are totally ordered. That is, requests are processed at all replicas in the same order. Only after the single current update request has been processed at all servers holding replicas, is control passed back to the client that requested the update, and the next request (from whatever client handled) this model clearly would maintain strict consistency. In the bulletin board example, all items would be given the same number and therefore appear in the same order. However, it turns out that such a system would give response times and throughput for updates that are worse than that of any single server, because of the extra costs of propagating new items and ordering them.

In general, most replication schemes fall between the simple asynchronous and synchronous models described above. A degree of replica consistency beyond that obtained in the asynchronous model is often required; but the response times provided by the totally synchronous model are sometimes inadequate. Various techniques have been devised to achieve satisfactory trade-offs between consistency, availability and response time. Key among these are two basic approaches.

Quorum-based schemes: Some schemes minimize communication costs and enable a degree of concurrent processing by allowing some but not all managers of replicas to be contacted before a read or update request is allowed to complete.

Causality: There are designs which achieve a degree of concurrent request processing by allowing some requests to be processed in different orders at different replicas – in so far as this cannot produce inconsistencies. An important ordering of this type is causal ordering (see Section 10.3). This is missing from our asynchronous bulletin board example, where an item re:X can appear before the original item about X.

An important general mechanism for keeping replicas consistent is to multicast updates to all processes that manage physical replicas. That is, updates are sent in messages which are delivered to precisely the set of processes that manage replicas of the corresponding data item. Process groups are logical destinations for multicast messages, whose membership can be kept transparent to the message sender. The ISIS system [Birman 1993], described below, provides process groups and ordered multicast on top of existing operating systems, particularly UNIX.

Finally, what effect does replication have on performance: the time that elapses between a client issuing a request and values being returned to it, and the number of requests that can be processed per second? If the replicated data are read-only or if consistency is not an issue, then the answer to the first question is surely ‘no longer than for a non-replicated implementation’. This is more or less the case for the simple asynchronous model given above (although sending and handling updates in the background would have some impact on response times).

The ratio of read operations to updates performed upon replicated data is an important consideration, especially when designing systems in which the number of
replicas is relatively high. The number of replicas varies from the thousands – for example, USENET or the DNS – where the read/write ratio is high, to the few replicas used in fault-tolerant systems with relatively small read/write ratios.

11.2 Basic architectural model

We now give a basic architectural model for the management of replicated data. For the sake of generality, we describe components by their roles, and do not mean to imply that they are necessarily implemented by distinct processes (or hardware). The model involves replicas of data items held by distinct replica managers, which are processes that contain the replicas and perform operations upon them directly. Note that we avoid the term ‘replica server’ and use ‘replica manager’ instead, because this general model may be applied to any application rather than a service implementation, and client processes can in that case act as replica managers.

Typically, each replica manager maintains a physical copy of every logical data item, but there is no reason why data items should not be maintained by different sets of replica managers. For example, some data items might be needed mostly by clients on one LAN rather than another, and so there is little to be gained by replicating them managers on the other LAN.

The general model is shown in Figure 11.3. Clients each make a series of requests. Each request in general involves a combination of reads of logical data items and updates to logical data items. Requests that involve no updates are called read-only requests; requests that update at least one data item are called update requests (although they may also involve reads of data items).

Each client’s requests are first handled by a component called a front end. The role of the front end is at least to communicate by message passing with one or more of the replica managers, rather than forcing the client to do this itself explicitly. It is the vehicle for implementing replication transparency. A front end may be implemented a user package executed in each client, or it can be a separate process. The choice involves a trade-off between, on the one hand, the greater efficiency of calling an process procedure over communicating with another process; and, on the other hand, the possible advantages to sharing information such as server addresses in the front end on behalf of all clients at the same site.

Figure 11.3 A basic architectural model for the management of replicated data.

![Diagram of a basic architectural model for the management of replicated data.](image1)

In different models the front end communicates either with only a single replica manager per operation or with several or all of them. Furthermore, the front end may collate the results of requests performed at more than one replica manager. For example, if a fault tolerant service is being provided, the front end may be used to request values from all replica managers, and to select values returned by a majority to be passed back to the client, in case one or more of them is faulty (see Section 15.3).

If the clients and front ends are separate processes, the model approximates what we shall call the gossip architecture for highly available data [Ladin et al. 1992]. We choose this name because the replica managers exchange ‘gossip’ messages periodically in order to convey the updates (the ‘news’) they have each received (Figure 11.4). In the gossip architecture, the front end normally communicates with an individual replica manager for each operation (hence the need for exchanging gossip messages); alternatively, it can communicate directly with more than one replica manager per operation. The gossip architecture does not define whether front ends are separate to client processes. Section 11.4 describes the gossip architecture in full.

Figure 11.4 The gossip architecture.

![Diagram of the gossip architecture.](image2)

In the primary copy model of data replication for enhanced availability (Figure 11.5), all front ends communicate with the same ‘primary’ server when updating a particular data item. The primary server propagates the updates to the other servers, called ‘slaves’. Front ends may read the item from a slave. If the primary server fails,

Figure 11.5 The primary copy model for replicated data.

![Diagram of the primary copy model.](image3)
one of the slaves can be promoted to act as the primary. The primary copy model is used in the Harp replicated file system [Liskov et al. 1991]. In the Sun Network Information Service (formerly Yellow Pages), information such as password entries which changed relatively infrequently is updated at a master server and propagated from the master to slave servers. Front ends may communicate with either a master or a slave server to retrieve information. In this case, however, front ends may not request updates; updates are made to the master’s files.

At the other extreme, an architecture for a shared editor is shown in Figure 11.6. Users at different computers can use the editor to, for example, design a document. (This and other forms of software for multi-user collaboration, called groupware, is discussed by Ellis et al. [1991].) Each executing instance of the editor program holds a copy of the entire document state. There is only one class of process, and it has the responsibility of performing the roles of client and replica manager. Additionally, a front end module may be used to hide replication from other modules within the process that access the shared document state.

In between these cases, there lies an architecture in which front ends and replica managers are combined, but are separate from clients. In distributed shared virtual memory implementations, for example, clients read and write data values from and to local variables, and the local kernel acts as a combined front end and replica manager. Processes programmed using the Orca distributed programming language [Bal et al. 1990] access shared objects. The Orca run-time system, one copy of which executes on each computer, plays the combined roles of front-end and replica manager.

The role of propagating updates to all replica managers is divided differently across architectures. The choice of one pattern of communication rather than another is determined by the requirements for availability, consistency and response times. If high availability is to be achieved, the front end cannot rely upon a particular replica manager since it might fail. For good response times, the front end should contact as few replica managers as possible to ensure consistency before returning a result to the client. In general, the most suitable architecture has to be chosen on the basis of the application requirements.

11.3 Consistency and request ordering

Section 11.1 described a simple asynchronous communication model in which updates are eventually processed at all replicas, but in no particular order. It also gave a simple synchronous model in which updates are processed at all replica managers, and are processed everywhere in the same, total order. This section motivates some application-level consistency requirements and shows how they lead to ordering constraints for processing requests at different replica managers. The chief orderings considered are total and causal orderings.

It is assumed that a replica manager processes requests one at a time. If it is multi-threaded then it can carry out requests concurrently, but nonetheless we require that their combined execution is equivalent to processing one at a time. (Section 12.4 discusses this concept of serial equivalence.) Moreover, each replica manager is assumed to be a state machine. That is, the data items managed by a replica have values which are only a function of their initial values and the operations applied to them. Other stimuli, such as the passage of time or the readings of attached sensors, have no bearing on these values [Schneider 1990].

The order in which requests are processed at different replicas is an important issue, not only because it is often necessary that particular ordering constraints are obeyed for correctness, but also because meeting ordering requirements carries certain expenses. First, the processing of a request may be delayed because a ‘prior’ request has yet to be processed. Secondly, as will be described below, protocols designed to guarantee a particular ordering can be expensive to implement in terms of the number of rounds of messages that have to be transmitted.

Therefore it is advisable to avoid request ordering wherever possible. In general, not all pairs of requests have to be processed in a particular order. Requests $R_1$ and $R_2$ whose effect when consecutively processed in the opposite orders $R_1; R_2$ and $R_2; R_1$ is the same are said to commute. For example, any two read-only operations commute; and any two operations that do not perform reads but write distinct data commutes. A system for managing replicated data may be able to use knowledge of commutativity in order to avoid the expense of request ordering.

**Total and causal ordering requirements**

A requirement for total ordering is exemplified by the bulletin board system, where it would be convenient if replica managers could label items with the same numbers so that users could unambiguously refer to them. The cost of achieving total ordering is liable to be prohibitive, however, over a wide area network.

As a second example, consider a multi-user document editor, in which users at different computers may attempt to select an object on the screen such as a word at more or less the same time, in order to change it. The global state of the application is stored and displayed at each computer. The requirement when selecting an object is for total ordering, since one of the conflicting users must be consistently chosen to have selected the item before the other.

Formally, under totally ordered request processing, if $R_1$ and $R_2$ are requests then either $R_1$ is processed before $R_2$ at all replica managers or $R_2$ is processed before $R_1$ at all replica managers. Total ordering is a general relation over events in a distributed system, which Section 4.4 introduced in the case of multicast message delivery.
Figure 11.7  Totally and causally ordered request processing.

Notice the consistent ordering of t1 and t2, the consistent ordering of the causally related operations c1 and c2, and the arbitrary relative ordering of c2 and c3, which are unrelated.

Causal ordering for request processing is defined according to the general notion of causal ordering—also known as happened-before ordering (see Section 10.3). Recall that this ordering captures a potential causal relationship between two events: one event happened-before another if information flowed, in part as messages and in part as sequential actions of individual processes, between the two events. Causal ordering was first applied to request processing by Birman and Joseph [1987]. The bulletin board system exemplifies the requirement for causal ordering, since when an item refers to another item, the posting of the latter clearly happened-before the posting of the former and so should appear before the other. Formally, under causal ordering, if \( r_1 \) and \( r_2 \) are requests and \( r_1 \) happened-before \( r_2 \), then \( r_1 \) is processed before \( r_2 \) at all replica managers.

Figure 11.7 illustrates the processing at three replica managers RM1, RM2, and RM3 of totally ordered requests t1 and t2 and causally ordered requests c1, c2 and c3 originating at two front ends FE1 and FE2. Because \( c_1 \) happened-before \( c_2 \), corresponding requests are processed in the same order; but \( c_3 \) is unrelated to either \( c_1 \) or \( c_2 \), and their relative orderings are mixed.

Note that total ordering is not necessarily also causal ordering. For example, unrelated items could be consistently ordered across all bulletin board replicas, but in the opposite order to their causal relationship.

Stronger ordering requirements: Consider a system in which some requests are totally ordered and others are causally ordered. It is sometimes convenient to employ a stronger level of ordering which we shall call sync-ordering. A sync-ordering preserves the order of requests processed at replica managers to be 'in sync', in sense that every other request is consistently processed before it or after it at all of them. Sync-ordering is needed because a causally ordered request and a totally ordered request can be processed in an arbitrary order, unless they are causally related. A sync-ordered request effectively flushes any outstanding requests that have been issued but not yet processed everywhere, so that they are processed before it; all later requests are processed after it. It thus draws a conceptual line across the system, dividing all request processing consistently into a 'past' and a 'future'. Formally, if \( r_1 \) and \( r_2 \) are requests and \( r_1 \) is sync-ordered, then either \( r_1 \) is processed before \( r_2 \) at all replica managers or \( r_2 \) is processed before \( r_1 \) at all replica managers regardless of the declared ordering of \( r_2 \) (causal, total or sync).

Figure 11.8 illustrates these ordering properties at replica managers RM1, RM2, and RM3 for causally ordered requests c1 and c2, totally ordered requests t1 and t2, and sync-ordered request s, issued from front ends FE1 and FE2.

Implementing request ordering: Several algorithms exist for implementing total and causal ordering. Before describing some of these, two remarks about the basic techniques involved are appropriate. The first is that hold-back is used. In terms of our general replication model, a received request is not processed by a replica manager until ordering constraints can be met—that is, it is held back. To give the familiar example, a bulletin board item Re:Microkernels may be held back until an item concerning Microkernels has already appeared.

A request message is said to be stable at a replica manager if all prior requests (defined according to the type of ordering) have been processed—that is, if it is ready to be processed next. Schneider defines this concept in [Schneider 1990]. Figure 11.9 shows incoming requests being placed initially on a hold-back queue, where they remain until satisfied.
Ordering the requests arriving at a replica manager.

Two properties need to be proved, in general, for implementations of request ordering. The safety property is that no message will be delivered out of order by being transferred from the hold-back queue to the delivery queue prematurely. That is, an implementation must guarantee that once a request has (potentially) been processed, it is impossible for a "prior" request to arrive. The liveness property is that no message should wait on the hold-back queue indefinitely. An incorrect implementation might await some "prior" request that will never in fact arrive, and so will never transfer an existing request to the delivery queue.

A second general point concerning the implementation of ordered request processing is that techniques for achieving it are applicable, with appropriate adjustments, to the implementation of ordered multicast. Both the gossip architecture and the ISIS distributed programming toolkit, described below, provide facilities for total and causal request ordering. The basic difference between their approaches is that ISIS provides an ordered multicast facility for delivering requests to replica managers and achieving ordering at the same time, whereas the gossip architecture utilizes unordered point-to-point communication normally, but provides for the replica managers themselves to propagate updates among themselves and to order them in processing. This difference is examined in detail below.

Implementing total ordering Consider the problem of totally ordering requests arriving at 'FE sites', for processing by replica managers at 'RM sites'. The basic approach to implementing total ordering is to assign totally ordered identifiers to requests so that each RM site makes the same ordering decision based on these identifiers.

We discuss two main methods for assigning identifiers to requests. The first is for a request coming from a sequence to be used to assign them. All requests arriving at a replica manager are processed in the order in which they arrive, until their order has been determined; when stable, they are placed on a processing queue, from which they are extracted and processed in FIFO order.

The second method, which will be described in detail, uses a sequence identifier. The identifier increases by one for each new sequence of requests, and an identifier for a request is generated by concatenating the sequence identifier of the request with the identifier of the request itself. The sequence identifier is assigned by the sequencer, which is assumed to be a separate process, and is incremented at each request arrival.

The sequencer places each new request on the sequence of requests that is active at the time of the request. This sequencer is assumed to be able to assign an increasing number of identifiers, and it does not need to be very reliable, as the identifier is used only for ordering and not for security purposes.

The sequecner assigns identifiers to requests arriving at an RM site whenever it receives a request. These identifiers are assigned in the order in which the requests arrive, and are incremented by one at each request arrival.

Ideally, the generation of request identifiers is integrated with the transmission of requests to replica managers, to minimize the total number of messages involved. For example, each request can be sent first to the sequencer, which increments its sequence count, attaches this to the request and forwards it to the replica managers. The obvious problem with such a scheme is that the sequencer may become a bottleneck and is a critical point of failure. Nonetheless, practical algorithms exist that address the problem of failure. Chang and Maxemchuk (1984) first suggested a multicast protocol employing a sequencer (which they called a token site), and the sequencer-based protocol used in the Amoeba system is discussed in detail in Chapter 18.

The second method for achieving totally ordered request processing avoids the need for a sequencer; it achieves distributed agreement in assigning request identifiers. A simple algorithm – similar to one which was originally developed to implement totally ordered multicast delivery for the ISIS toolkit (Birman and Joseph 1987) – is shown in Figure 11.10. RM sites propose identifiers for requests as they arrive and return these to the corresponding FE sites, which use them to generate final identifiers.

Each RM site stores $F_{max}$, the largest final identifier agreed so far and $P_{max}$, its own largest proposed identifier. The algorithm for identifier generation and stability testing proceeds as follows.

1. The FE site sends the request bearing a temporary identifier to all RM sites. The temporary identifier is chosen to be larger than any identifier used previously by the FE site.

2. Each RM site replies; site $i$ responds with a proposal for the request’s final identifier of $Max(F_{max}, P_{max}) + 1 + i/N$, where $N$ is the number of RM sites. Each RM site provisionally assigns the proposed identifier to the request and places it on its hold-back queue, which is ordered with the smallest sequencer identifier at the site.

3. Each RM site sends the request with its final identifier to the sequencer.

4. The sequencer assigns an increasing number of identifiers, and each RM site assigns identifiers based on the sequence identifier of the request. The identifiers are incremented by one at each request arrival.

5. The sequencer assigns a final identifier to each request, and each RM site assigns the identifier to the request.

6. The request is delivered to the requesters, and the RM sites return the identifiers to the FE site.

7. The FE site assigns the identifiers to the requesters, and the RM sites return the identifiers to the requesters.

Figure 11.10 The ISIS algorithm for total ordering.
front. The FE site collects all the proposed identifiers and selects the largest of these as the next agreed identifier. This is guaranteed to be unique, because of the term \( i/N \) in the formula for proposed identifiers.

3. The FE site then notifies all the RM sites of the final identifier. The RM sites then attach the final identifier to the request. The request is then reordered on the hold-back queue at sites where the final identifier differs from the proposed identifier assigned in Step 1. When the request at the front of the hold-back queue has been assigned its final identifier, it is stable and is transferred to the tail of the delivery queue. Requests that have been assigned their final identifier but are not at the head of the hold-back queue are not yet transferred, however.

It is easy to see that the liveness property holds for this algorithm, as long as failures do not occur. Every request will eventually be assigned a final identifier and reach the front of the hold-back queue, to be transferred to the delivery queue. (Recall that new requests are assigned increasing proposed identifiers, so a finalized request cannot continually be pre-empted by candidate requests with smaller identifiers.)

To see that the algorithm has the safety property, assume that a request \( r_2 \) has been assigned a final identifier and has reached the front of the hold-back queue. Let \( r_2 \) be another request on the same queue that has not yet been assigned its final identifier. We have that:

\[
\text{finalId}(r_2) \geq \text{proposedId}(r_2)
\]

by the algorithm just given. Since \( r_2 \) is at the front of the queue:

\[
\text{proposedId}(r_2) > \text{finalId}(r_1).
\]

Therefore:

\[
\text{finalId}(r_2) > \text{finalId}(r_1),
\]

and the safety property is assured.

This algorithm has the advantage that it is relatively straightforward to implement. Unlike a protocol based upon a single sequencer, there is no bottleneck or single point of failure for transmitting messages. Also, the protocol can be adapted so that an RM site fails while it is engaged in delivering a request, one of the RM sites can detect this and take over co-ordination in its place. But the algorithm is expensive. Three messages are sent between the front end and each replica manager before a request can become stable. Even if hardware broadcast is used to transmit the request initially, the cost is prohibitive. It is shown below that a sequencer-based protocol can be made cheaper in terms of the numbers of messages sent.

Note that the total ordering chosen by this algorithm is not also causal: even if the two messages are delivered in an arbitrary total order, which could conflict with potential causality.

For a third approach to implementing total ordering, see [García-Molina and Spauter 1991].

Implementing causal ordering with vector timestamps

Consider the example of a bulletin board system with replica managers implemented at three universities, with front ends usually post and receive items by communicating only with their local replica manager, but they are free to communicate with the others. They might do this, for example, if their local replica manager becomes heavily loaded or unavailable for some reason. The consistency problem to be addressed is that no item that refers to another can be posted before the item to which it refers—that is, that items are to be posted in causal order, despite the out-of-order arrival of items from front ends and from replica managers.

The problem of achieving causal ordering can be solved through the use of suitable ordered identifiers called vector timestamps, which will shortly be defined. The reader may recall the integer logical timestamps introduced in Section 10.3, which are used to timestamp general events in accordance with their causal relationships. At first sight it might seem that these could be used to timestamp the bulletin board replication requests issued by front ends, so as to order request processing causally. Unfortunately these timestamps are unsuitable because no deductions can be made about causal ordering from comparing them. What is required are timestamps (used as identifiers) that carry more information and can be compared to determine causal ordering.

For the sake of simplicity it is assumed that bulletin board items are not removed, and so a bulletin board is updated only by the addition of new items. In other words, however, it is not necessary to consider the details of this application-specific state ordering to compare the versions of timestamps of it held by different replica managers. The timestamp of a bulletin board replica can be represented by counts of the update events that led to the current state. Some update events occur when a front end makes a request directly to the replica manager; other updates are forwarded from its peers. The timestamp of the bulletin board can be represented by a list of counts of update events for each of the replica managers. Such an array of event counts is called a vector timestamp or a multipart timestamp.

Vector timestamps are similar to the Coda version vectors used in the management of replicated data in the Coda file system (see Section 8.4). They were originally used in timestamped database concurrency control [Fischer and Michael 1982, Wu and Bernstein 1984] and the file system of the Locus distributed operating system [Walker et al. 1983]. More recently, their use in event ordering was developed by Fidge [1988], Mattern [1988] and Schiper et al. [1989]. Their role in the grid architecture and in the ISIS causal multicast protocol is described in the next sections.

For example, suppose that, as shown in Figure 11.11, replica manager \( RM_1 \) has processed three new items sent to it directly by front ends, four items forwarded to \( RM_2 \) and four from \( RM_3 \). The state of the replica at \( RM_1 \) can be represented by a vector timestamp \((3,4)\). Similarly, the replica held by \( RM_2 \) in the figure has timestamp \((2,5,6)\) and that of \( RM_3 \) has timestamp \((3,7)\).

In order to maintain causal ordering it is necessary to ensure that each front end reads from a version of the bulletin board that is at least as advanced as the version to which it last read (which may have been from a different replica manager); further, a new item should be added to a replica only when the replica already reflects the previously created state of the bulletin board. These two goals can be met if the replica managers maintain the following causal order constraints:
own timestamp representing their version of the bulletin board, and if the front end does not maintain vector timestamps reflecting the latest version of the bulletin board they have read.

Let us suppose that the front end in our example has a timestamp (2,3,4), and that it reads the latest items from RM1. The timestamp of RM1 is (3,4,4), and the front end should be updated to reflect the latest items seen, that is, to (3,4,4). This is the case of merging vector timestamps: of choosing the largest values from the two vector components-wise. The front end should not be returned data from RM2 at this stage since by comparing timestamps it can be seen that RM2 has not seen one of the updates accepted at RM1 and seen by the front end. Similarly, an update from this front end should not be incorporated at RM3 at this stage, since the front end has seen four updates originally accepted at RM2, whereas RM3 contains only the first three.

The use of vector timestamps will be fully described in the context of the gossip architecture in the next section. In preparation for that some more formal definitions are now given. Let the set of processes (for example, replica managers) be considered be \( p_1, p_2, \ldots, p_n \). We shall define, for each \( p_i \), a vector clock whose timestamp, reading we denote by \( VT_i \). Each \( VT_i \) is a vector of integer values, of length \( n \). The \( VT_i[k] \) represents a count of events (for example, updates to replicas) that have occurred at \( p_i \) and that are known at \( p_i \); either because they originate there or because their existence is known about through message passing.

The following is the vector clock update algorithm:

1. All processes \( p_i \) initialize \( VT_i \) to zeroes.
2. When \( p_i \) generates a new event (for example, a new version of its data) it increments \( VT_i[i] \) by 1; it piggybacks the value \( v = VT_i \) on the message it sends.
3. When \( p_i \) processes a request bearing a timestamp \( vt \), it updates its vector clock as follows:

\[
VT_j := \text{merge}(VT_j, vt).
\]

The \textit{merge} operation is defined for any pair of vector timestamps \( u, v \) as:

\[
\text{merge}(u, v)[k] = \max(u[k], v[k]), \quad \text{for } k = 1, 2, \ldots, n.
\]

Partial orders ‘\( \leq \)’ and ‘\( < \)’ on vector timestamps \( u, v \) are defined as follows:

\[
u \leq v \text{ iff } u[k] \leq v[k], \quad \text{for } k = 1, 2, \ldots, n.\]

\[
u < v \text{ iff } u[k] < v[k] \text{ for some } k, \quad \text{and } u \neq v.
\]

If \( e, f \) are events and \( u, v \) are their timestamps, it can be shown that \( u < v \) if and only if \( e \) happened-before \( f \). The reader is referred to Raynal [1992] for a general treatment of vector timestamps.

### 11.4 The gossip architecture

Ladin et al. [1992] developed what we have termed the gossip architecture (see Figure 11.4) as a framework for making highly available service implementations through use of replication. It is based upon earlier work on databases by Fischer and Michael [1982] and Wuu and Bernstein [1984]. It has been applied to such areas as distributed garbage detection and deadlock detection, and could be used, for example, to create a highly available electronic mail service or bulletin board service. The presentation given here omits certain aspects of the gossip architecture, in the interests of clarity.

High availability implies that a service remains accessible despite some computer or network failures; it is also desirable to maintain reasonable response times despite heavy load. In the gossip architecture, clients request service operations which are processed initially by a front end (there is a front end for every computer with clients). The front end normally communicates with only a single replica manager at a time, although they are free to communicate with others. In particular, a front end will communicate with a different replica manager when the one it normally uses fails or becomes unreachable, and it may try one or more others if the normal manager is heavily loaded. The replica managers update one another by exchanging \textit{gossip messages} which contain the most recent updates they have received. They are said to update one another in a \textit{laziness} fashion, in that gossip messages may be exchanged only occasionally, after several updates have been collected, or when a replica manager finds out that it is missing an update sent to one of its peers, which it needs to process a request.

The gossip architecture supports three different strengths of update ordering: \textit{causal}, \textit{forced} (total and causal) and \textit{immediate} (sync-ordered). The choice of which to use is left to the application designer, and reflects a trade-off between consistency and operation costs. Causal updates are considerably less costly than the others, and are expected to be used whenever possible.

**Operations and ordering** One of the three orderings causal, forced or immediate has to be specified for each type of update operation, and the system automatically enforces it. Updates are normally applied in a causal (but not total) order. This is the least expensive ordering in terms of latency, and it allows operations to be applied to a replica that is not the most up to date, as long as causality is respected. Forced ordering is causal and total. Immediate ordering is what we have described as sync-ordering: it
forces an operation to be applied in a consistent order relative to any other operation on all replica managers, whether the other operation be specified as causal, forced, or immediate. Note that queries, which can be satisfied by any single replica manager, are always executed in causal order with respect to other operations.

In the electronic bulletin board example, causal ordering could be used for posting items. Forced ordering would be used for adding a new subscriber to a bulletin board so that if several uncoordinated clients attempt to add the user at more or less the same time, only one will succeed (instead of several entries being created with the same name). Immediate ordering would be used for subtracting a user from a subscription list so that messages could not be retrieved by that user via some tardy replica manager before the deletion operation had returned.

The front end in a replicated service is application-specific, since it has to handle client operations. In general, client operations can either read the replicated state or modify it or both. Inside a gossip implementation, however, only two basic types of operation are recognized: queries are read-only operations, and updates modify but do not read the state (the latter is a more restricted definition than the one we have been using). The front end can convert an operation that both reads and modifies a replicated state into a separate update and query.

Clients are blocked on query operations, since they must wait for a value to be returned to the front end by a replica manager (see Figure 11.12). The default arrangement for update operations, on the other hand, is to return to the client as soon as the operation has been passed to the front end; the front end then propagates the operation in the background. Alternatively, for increased reliability, clients may be prevented from continuing until the update has been delivered to k+1 replica managers and so will be delivered everywhere despite up to k failures.

In order to control the ordering of operation processing, each front end keeps track of the latest values observed by the front end, along with a vector timestamp (and therefore accessed by the client). This timestamp, denoted prev in Figure 11.12, is sent in every request message from the front end to a replica manager, together with a description of the query or update operation itself. When a value is returned as a result of a query operation, a new timestamp (new in Figure 11.12) is also returned, since the replica may have been updated since the last call. Similarly, an update operation returns a vector timestamp (update id in Figure 11.12) which is unique to the update. Each returned timestamp is merged with the front end’s previous timestamp, to record the version of the replicated data that has been seen by the client.

Clients exchange information in two ways. The first is by accessing the same replicated services. Systems based on the gossip architecture are expected to test for the happened-before relationship between operations and respect the ordering implied by this. Secondly, clients are assumed to communicate directly (or via other services). Since this latter communication can also lead to causal relationships between operations applied to the replicated service, it must occur via the clients’ front ends. That way, the front ends can piggy-back their vector timestamps on client messages. The recipients merge them with their own timestamps in order that causal relationships can be correctly inferred. The situation is shown in Figure 11.13.

**Replica manager state** Regardless of the application, a replica manager contains the following main state components (Figure 11.14), which are kept in main memory:

- **Value**: This is the value of the application state as maintained by the replica manager. Each replica manager is a state machine, which begins with a specified initial value, which is thereafter solely the result of applying update operations to that state.

- **Value timestamp**: This is the timestamp that represents the updates that are reflected in the value. It is updated whenever an update operation is applied to the value.
A gossip replica manager, showing its main state components.

**Update log:** All update operations are recorded in this log as soon as they are received. A replica manager keeps updates in a log for one of two reasons. The first is that the replica manager cannot yet apply the update because it is not yet stable: that is, it must be held back and not processed yet. The second is that, even though an update has become stable and has been applied to the value, the replica manager has not received confirmation that this update has been received at all other replica managers. In the interests of high availability, it can in the meantime supply the update to those peers that require it.

**Replica timestamp:** This timestamp represents those updates that have been accepted by the replica manager — that is, placed in the manager’s log. It differs from the value timestamp in general, of course, because not all updates in the log are stable.

**Identifiers of executed calls:** The same update potentially can arrive at a given replica manager from a front end and in gossip messages from other replica managers. To prevent an update from being performed twice, a list is kept of the identifiers of updates that have been applied to the value. The replica managers check this list before executing a stable update.

The replica managers are numbered 1, 2, 3, … and the $i$th element of a vector timestamp held by replica manager $i$ corresponds to the number of updates received from front ends by $i$; and the $j$th component ($j \neq i$) equals the number of updates received by $j$ and propagated to $i$ in gossip messages. So, for example, in a three-manager gossip system a value timestamp of $(2, 4, 5)$ at manager $1$ would represent the fact that the value there reflects the first two updates accepted from front ends at manager 1, the first four at manager 2 and the first five at manager 3. The following looks in more detail at how the timestamps are used to effect ordering.

**Processing query operations** The simplest operation to consider is that of a query. Recall that a query request $q$ contains a description of the operation and a timestamp $q.\text{prev}$ sent by the front end. The latter reflects the latest version of the value that the front end has read or submitted as an update. Therefore the task of the replica manager is to return a value that is at least as recent as this. If $\text{valueTS}$ is the replica’s value timestamp, then $q$ can be applied to the replica’s value if:

$$q.\text{prev} \leq \text{valueTS}.$$  

The replica manager keeps $q$ on a list of pending query operations (that is, a hold-back queue) until this condition is fulfilled. It can either await the missing updates, which should eventually arrive in gossip messages; or, it can request the updates from the replica managers concerned. For example, if $\text{valueTS}$ is $(2, 5, 5)$ and $q.\text{prev}$ is $(2, 4, 6)$, it can be seen that just one update is missing – from replica manager 3. (The front end that submitted $q$ must have contacted a different replica manager previously for it to have seen this update, which the replica manager has not seen.)

Once the query can be applied, the replica manager returns $\text{valueTS}$ to the front end as the timestamp $\text{new}$ shown in Figure 11.12. The front end then merges this with its timestamp: $\text{frontEndTS} := \text{merge(frontEndTS, new)}$. The update at replica manager 2 that can be seen missing in $q.\text{prev}$ in the example is reflected in the value returned, and this is shown in the update to $\text{frontEndTS}$.

**Processing update operations in causal order** When a replica manager receives an update request it increments its own element in its replica timestamp by one, to keep count of the number of updates it has received directly from front ends. Then the update request $u$ is assigned a unique identifier (a vector timestamp whose derivation is given shortly) and a record for the update is placed in the replica manager’s log. If $u.\text{op}$ is the update operation, $u.\text{prev}$ is the timestamp sent with it by the front end and $\text{TS}$ is the unique identifying timestamp assigned to the update, then the log record is constructed as the following tuple:

$$\text{logRecord} := \langle \text{TS}, u.\text{op}, u.\text{prev} \rangle.$$  

The identifier $\text{TS}$ is derived from $u.\text{prev}$ by replacing its $i$th element by that of the replica timestamp, where $i$ is the replica manager that receives the update from the front end. This action makes $\text{TS}$ unique. The remaining elements in $\text{TS}$ are copied from $u.\text{prev}$, since it is these values sent by the front end which must be used to determine when the update is stable. $\text{TS}$ is immediately passed back to the front end, which merges it with its existing timestamp. Note that a front end can submit its update to several replica managers, and receive different unique identifier timestamps in return, all of which have to be merged into its timestamp.

The stability condition for an update $u$ is similar to that for queries:

$$u.\text{prev} \leq \text{valueTS}.$$  

Expressed informally, this condition states that all the updates on which this update depends have already been applied to the value. If this condition is not met at the time the update is submitted, it will be checked again when gossip messages arrive. When the condition has been met for an update record \( r \), the replica manager checks to see whether the call identifier \( r.cid \) (contained in \( r.op \)) appears already in the list of identifiers of executed calls. If not, the update is applied to the value and its timestamp and the executed call list \( executed \) are updated:

\[
\begin{align*}
\text{value} & := \text{apply}(\text{value}, r.op) \\
\text{valueTS} & := \text{merge}(\text{valueTS}, r.ts) \\
\text{executed} & := \text{executed} \cup \{r.cid\}
\end{align*}
\]

The first of these three statements represents the application of the update to the value. In the second statement, the update’s timestamp is merged with that of the value. In the third, the update’s call identifier is added to the set of identifiers of calls that have been executed — which is used to check for repeated operation requests.

**Gossip messages**

Gossip messages are sent by replica managers to assist other replica managers to bring their state up to date. Gossip messages normally contain information concerning several updates. The gossip architecture does not specify in general when gossip messages are exchanged. Since gossip travels in both directions between pairs of replica managers there is no need for each message to be separately acknowledged unless timeliness constraints require that missed updates are detected within a certain interval.

A gossip message \( m \) consists of two items sent by a replica manager: its log \( m.log \) and its replica timestamp \( m.ts \) (see Figure 11.14). The replica manager that receives it has three main tasks:

- to merge the arriving log with its own (it may contain updates not seen by the receiver before);
- to apply any updates that have become stable and have not been executed before (stable updates in the arrived log may in turn make pending updates become stable);
- to eliminate records from the log and entries in the list of executed call identifiers when it is known that the updates have been applied everywhere, and for which therefore there is no danger of repeats. Clearing redundant entries from the log and from the list of executed call identifiers is an important task since they would otherwise grow without limit, but there is insufficient space to describe here how this is achieved. Full details are given by Ladin et al. [1992].

Merging the log contained in an arrived gossip message with the receiver’s log is straightforward. Let \( valueTS \) denote the recipient’s value timestamp. A record \( r \) in \( m.log \) is added to the receiver’s log unless either it is already in it or \( r.ts \leq valueTS \). If \( r.ts \leq valueTS \), the update is already reflected in the value but the original record has been discarded from the log. The replica manager merges the timestamp of the incoming gossip message with its own replica timestamp \( replicaTS \), so that it corresponds to the additions to the log:

\[
\text{value} := \text{apply}(\text{value}, r.op) \\
\text{valueTS} := \text{merge}(\text{valueTS}, r.ts) \\
\text{executed} := \text{executed} \cup \{r.cid\}
\]

When new update records have been merged into the log, the replica manager collects the set \( S \) of any updates in the log that are now stable. These can be applied to the value, but care must be taken over the order in which they are applied, so that the happened-before relation is observed. That is, each \( r \in S \) is applied only when there is no \( s \in S \) such that \( s.prev < r.prev \). All updates in the set are applied while following this rule (the set can first be ordered according to the partial order “\( \leq \)”).

**Processing forced and immediate update operations**

Forced and immediate updates require special treatment. Recall that forced updates are totally as well as causally ordered. The basic method for ordering forced updates is for a unique sequence number to be appended to the timestamps associated with them, and to process them in order of this sequence number. As explained above, a general method for generating sequence numbers is to use a single sequencer process. But reliance upon a single process is of course inadequate in the context of a highly available service. The solution is to designate a so-called primary replica manager as the sequencer at any one time, but to ensure that another replica manager can be elected to take over consistently as the sequencer, should the primary fail. What is required is for a majority of replica managers (including the primary) to record which update is next in sequence before the operation can be applied. Then, as long as a majority of replica managers survive failure, this ordering decision will be honoured by a new primary elected from among the surviving replica managers.

Immediate updates are sync-ordered. They can easily be ordered with respect to forced updates by using the primary replica manager to order them in this sequence. The primary also determines which causal updates are deemed to have preceded an immediate update. It does this by communicating and synchronizing with the other replica managers in order to reach agreement on this. There is insufficient space to cover the details of implementing forced and immediate updates here.

**Some optimizations**

Optimizations can be applied to gossip traffic to lessen the size and number of messages. For example, replica managers can note which updates are recorded as having been received by their peers, so as not to send them updates that they have already received.

In addition, communication between the front end and a replica manager can be made more efficient by using a stream connection instead of a request-reply protocol, as long as the replica manager is the one always used by the front end (except when a failure occurs). There is then no need for the front end to receive the identifiers of the updates it sends to the replica manager; the latter merely has to record these for the client. Moreover, unless communication from other clients takes place, there is no need for the front end to send its timestamp to the replica manager. Successive updates can be sent over the stream without blocking, and they can be batched before transmission, in order to save on the number of messages used.

**Discussion**

The gossip architecture is aimed at achieving high availability for services. High availability can only be achieved if updates are propagated to more than one failure-independent computer in a timely fashion. For example, a client may cease to be able to obtain the bulletin board service because it depends on an update (that is, a
11.5 Process groups and ISIS

A process group is a collection of processes that co-operate towards a common goal or that consume one or more common streams of information. Groups can be used to implement replicated services, to collect together clients that subscribe to information published by a service, or to implement self-contained distributed applications. Although the members of a group do not necessarily manage replicated data, the members of a group act together primarily by receiving and processing the same set of messages. Process groups, in other words, are destinations for multicast communication, and transmitting a message to all members of a process group is also known as group communication. Chapters 2 and 4 have already discussed multicast communication.

In addition to providing multicast communication, a group service allows process groups to be created and their membership to be changed dynamically. The V system [Cheriton and Zwaenepoel 1985] was the first system to include support for process groups. Currently, the principal example of a group service implementation is ISIS [Birman and Joseph 1987, Birman et al. 1991, Birman 1993], which provides a programming interface for process groups on top of UNIX. The following describes the common types of group structure and a simple programming interface for process groups; it goes on to examine the design issues concerning the relationship between multicast and group membership, before describing ISIS.

Group structure

Group structures are defined according to the pattern of communication in which the members of a group are involved. In a peer group, all communication is directed from processes within the group to the group. This structure is suitable for a collection of processes implementing a multi-user editor, for example (see Figure 11.6). The processes use the group service to multicast the updates made by users, and to include themselves in the group as users join the editing session. In some peer group models, multicast messages are not delivered to the sender. This is the case, for example, in the model of group communication assumed by the Psyic protocol [Peterson et al. 1989]. However, in the editor example, delivering a multicast message to the sender may be necessary in order for it to observe the order in which multicasts are delivered.

There are several choices for the group structure when client processes are to make requests that are serviced by a collection of server processes. The simplest method is for the client to make an RPC to some member of a peer group, which uses group communication in implementing the service with its peers, and then replies to the RPC. Alternatively, the client can multicast its requests to a server group. All members of the server group receive the request. In simple cases, only one need reply. The process that replies can be chosen on the basis of information known to all members of the group. For example, the oldest surviving member of the group can reply if they are equivalent. Or, if service information is partitioned between the servers, only the server that contains the requisite data processes the request and replies. Alternatively, to achieve rapid response in case a server fails, two or more processes could each process and reply to each request.
Yet another variation on this model, used in ISIS and shown in Figure 11.15, is the client-server group. Here, requests from clients are multicast to all members of a server group, but the server that processes the request from a particular client multicasts the reply to the client-server group, which contains all the server group members plus the client itself. There is one client-server group for each client, so that ISIS has to support many overlapping groups for each service. In this structure, the other servers are able to update their state according to the results returned. In the virtually synchronous execution environment provided by ISIS (described below), if the server that processes requests fails before it can reply, then the other servers in the group will be informed of this and can elect one of their members to reply instead.

A subscription group is a group of processes that are sent the same information from an information source. The members of a subscription group do not reply to the messages they receive and they process the published information in their own independent, application-specific ways. For example, brokers dealing on a trading floor require access to financial information as it arrives from several sources. To realize this, the broker’s workstation executes one or more processes that are members of the subscription groups corresponding to the different sources of information required. Central computers gather the data and multicast them to these groups. A set of related servers may also be included in a subscription group, because of dependencies between the information that they each publish. Subscription groups are called diffusion groups in ISIS.

To avoid the overheads associated with the management of very large groups, groups can be composed into hierarchical groups. For example, a large group can be divided into sub-groups, and one member of each sub-group joins a root group. A process whose information must reach all members multicasts it to the root group, and the members of this group then multicast the information to the sub-groups. This principle can be extended to sub-groups of sub-groups. The advantages gained by hierarchical groups are the smaller vector timestamps needed for causal ordering, and the fact that communication within separate sub-groups can take place in parallel. Again, this is the increase in latency introduced by causing multicasts to traverse two or more groups.

Finally, group organizations differ as to whether group membership is visible to their members. In a subscription group, group members normally do not need to be aware of their peers. A member process in a server group or peer group, however, often needs to be aware of the membership of the group. For example, if the service provided by a group is reliable, then existing members may need to create a further member when the membership drops below a specified minimum number.

**Group services**

A system-wide service provides support for process groups. The members of a process group can be distributed across any collection of computers, interconnected by WANs or LANs. As shown in Figure 11.16, an implementation of process groups provides, in general, three main, strongly inter-related, group services: the management of group membership, group address expansion and multicast communication.

**Group membership management** This service provides operations to create and change the membership of groups. Groups, it is assumed, are named by globally unique group identifiers. A group identifier is returned when a new group is created; and the group identifier must be supplied when a process joins or leaves a group. In their simplest form, these group operations are as in Figure 11.17.

Processes may also be removed from a group’s membership as a result of failures. The group membership service monitors the group’s current membership, recording changes
Caused either by the above operations or through fail-stop failure. The membership service supplies on request a list of current members for a given group. It also sends messages to interested processes when group membership changes occur, giving them an up-to-date membership list. For example, the members of a group are often interested in changes in their membership.

A process can belong to several groups simultaneously. In a variation on the model described here, processes do not themselves belong to groups. Rather, they possess ports which they insert individually in groups. It is convenient to manage the reception of messages sent to distinct groups, by placing different ports in different groups. The Chorus distributed operating system provides port groups [Rozier et al. 1988], although it provides only low-level group communication.

Group address expansion □ A process sending a message to the members of a group does not supply a list of the processes involved. Instead, it supplies a group identifier, which is the same as the group identifier returned by groupCreate. The group identifier is mapped onto a current membership list as part of the group communication implementation, which in turn uses the group membership service. This facility hides the group's internal structure from processes that do not need to know it.

The use of group identifiers enables the implementation to synchronize message delivery with changes in the group membership. The expansion of a group identifier into a list of current members could be implemented as a simple request to the group membership service. The resultant list could be passed to the multicast service to transmit the message. But what would happen if a membership change occurs while a multicast is in progress? As the description of virtual synchrony explains below, some applications require that messages are delivered to group members in a consistent order relative to messages informing them about changes in the group state.

If, instead of using a group identifier, processes transmitted messages to a group by explicitly listing the members, then communication would be required with all group senders whenever the membership changed, and synchronization with group membership change notifications would be very awkward to achieve.

Multicast communication □ As explained in Chapter 4, multicast communication semantics vary according to reliability guarantees and also according to message ordering guarantees. To recap, the main reliability guarantees that are found are:

- **Unreliable multicast**: An attempt is made to transmit the message to all members without acknowledgement.
- **Reliable multicast**: One that makes a best effort; however, the message might be delivered to some but not all group members.
- **Atomic multicast**: A reliable multicast which guarantees that either all operational members of the group receive a message, or none of them do.

Reliability and atomicity do not of themselves imply any particular ordering of multicast message delivery. The four ordering semantics that are found in group implementations are: unordered, totally ordered, causally ordered and sync-ordered. Unordered multicast is self-explanatory. Total, causal and sync-ordering are defined similarly to the description of ordered request processing in Section 11.3 above.

Let $G_1$ and $G_2$ be groups (not necessarily distinct) with overlapping members (that is, $G_1 \cap G_2$ is non-empty), and let $m_1$ be a multicast message to $G_1$ and $m_2$ be multicast to $G_2$. Multicast delivery orderings are defined as follows:

- **Total ordering**: Either $m_1$ is delivered before $m_2$ at all members of $G_1 \cap G_2$, or $m_2$ is delivered before $m_1$ at all members of $G_1 \cap G_2$.
- **Causal ordering**: If the multicast of $m_1$ happened-before the multicast of $m_2$, then $m_1$ is delivered before $m_2$ at all members of $G_1 \cap G_2$.
- **Sync-ordering**: If $m_1$ is sent with a sync-ordered multicast primitive and $m_2$ is sent with any ordered multicast primitive, then either $m_1$ is delivered before $m_2$ at all members of $G_1 \cap G_2$, or $m_2$ is delivered before $m_1$ at all members of $G_1 \cap G_2$.

A simple primitive for atomically multicasting a message to a group is as follows. Its first parameter $order$ is used to declare the type of ordering required, and is given as UNORDERED, CAUSAL, TOTAL or SYNC. As its second argument this procedure takes the identifier group of the group to which the message $m$ is to be delivered. The sender declares the number of replies $nReplies$ that are to be received from members of the group. This can be set to zero, allowing the implementation to multicast the message asynchronously. If it is non-zero, then the implementation collects the stated number of replies and places them in the array of messages buffer replies.

```prolog
PROCEDURE multicast(order: orderType, group: groupld, msg: m:msg, nReplies: int, replies: ARRAY of msg)
multicast a message and optionally obtains replies.
```

The value of $nReplies$ would sensibly be zero when sending to a subscription group, since they will not reply to the message. When a client multicasts to a server group, it would normally be one. Use of a larger value of $nReplies$ implies knowledge of the size of the group membership. This might be used, for example, in a peer group, in which a member collects information from its peers. A larger value might also be used where the main goal is fault tolerance. A front end could obtain replies from each of a group of replica managers and choose a value returned by a majority (if such exists), in case some of the replica managers are faulty and can send spurious replies.
ISIS and virtual synchrony

The ISIS system [Birman and Joseph 1987, Birman 1993] is a framework for reliable distributed computing based upon process groups, which has been under development at Cornell University since 1983. ISIS has been fully implemented as a set of library calls on top of UNIX, and is being implemented in a form suitable for integration with Mach and Chorus communication. It is commercially available.

ISIS is a programming toolkit whose most basic facilities consist of process group management calls and ordered multicast primitives for communicating with the members of process groups.

**Multicast facilities:** ISIS provides unordered multicast (FBCAST), causally ordered multicast (CBCAST), totally ordered multicast (ABCCAST) and synchronously ordered multicast (GBCAST). All multicasts are reliable; ideally, they would be atomic, but they are not strictly atomic as currently implemented (see the discussion of CBCAST below). A process does not have to be a member of a group in order to communicate with it, and all the types of group structure described above are supported. Indeed, they have come about largely as a result of experience with ISIS.

**Group view maintenance:** ISIS provides primitives to create, join and leave process groups, which can be referred to by character-string names. Furthermore, ISIS monitors group members so that their fail-stop failure or unreachability is detected. A member is deemed to have left the group through failure when either its local operating system has detected its abnormal termination, or when it fails to respond after repeated attempts are made to communicate with it. ISIS maintains for each group a group view, which is a list of the current membership of the group, identified by the members’ unique process identifiers. The list is kept ordered according to the order in which the members joined the group. A new group view differs from the last group view by the addition or deletion of one process.

**State transfer:** Consider a multi-user application such as a shared editor implemented as a peer group of processes, one per user. If a user’s workstation crashes, then it ought to be possible for that user to go to a neighbouring workstation, start up the application and recommence work in the same editing session. The state of the application might not be the same as when the user’s computer crashed, since the other users may have made updates in the meantime. But the state should at least be consistent with that of the other users, even though they may attempt to update the state while it is being transferred. ISIS provides facilities for the application implementor to arrange that all relevant components of the application state are automatically sent to a newly joining group member; and moreover it guarantees that the state is consistent between all members of the group from the point of joining.

The members of an ISIS group are each conceived of as generating and processing a series of events. Processes generate events visible to other processes when they join or leave groups and when they issue multicasts. The two major types of event that group members process (as opposed to generate) are the delivery of a multicast message and the notification of a new group view. Group view notifications can be thought of as the delivery of messages containing the new group view (and indeed they are implemented as a special type of multicast messages).

An ISIS program is constructed as a collection of routines that are declared at runtime to correspond with specific types of incoming events, and which are thereafter called automatically by the ISIS run-time as these events occur. ISIS can be thought of primarily as managing the order of these events on each process’s delivery queue and thus scheduling their processing.

Given that multicasts are atomic, all the members of a process group are guaranteed to process the same set of events. The designers of ISIS considered the most useful and efficient relative orderings of event processing between different group members. Two degrees of coupling or synchrony between different members are distinguishable.

**Closely synchronous** execution is similar to the totally synchronous model introduced in Section 11.1. Under closely synchronous execution, all members processing a common set of events (which may pertain to different but overlapping groups) do so in the same order. Moreover:

- this order is consistent with the happened-before relation (that is, it is total and causal);
- it respects failure: it is guaranteed that no multicast message will be delivered after a view change notifying its sender’s failure – since this could lead to inconsistencies;
- it incorporates consistent state transfer. A newly joining process receives state marshalled from the variables of some extant member at the well-defined instant at which the member joins. In other words, the set of incoming events that are subsequently processed on the basis of this state is the same at the new member as it is at the extant members.

The closely synchronous execution model has the advantage of conceptual simplicity, since it automatically enforces complete consistency between members processing the same set of events. An important property is that when the group view changes, the individual members can react consistently without the need for further communication. For example, consider a group of processes that has divided some data equally between the members. They process the data in parallel, and periodically checkpoint completed results by multicasting them to their peers. If one of their number fails, then its work can be re-allocated to one or more extant members. They can all agree at the point of failure notification as to which work has been completed, and the re-assignment can be consistently made without further consultation according to the new group view.

The problems with closely synchronous execution, as was pointed out for the totally synchronous replication model introduced above, are that it does not allow for parallel operation between the members; and it is expensive to implement since it involves totally and causally ordering all events.

**Virtually synchronous** execution is conceived of in an attempt to retain the virtues of close synchrony while relaxing some of the ordering requirements as far as application consistency constraints will allow. Specifically, the advantageous features retained are the sync-ordered and failure-respecting nature of group view changes, and
the associated consistent transfer of state. However, multicast messages do not always have to be totally ordered in their delivery. Causal ordering is cheaper to implement; it allows unrelated messages to be processed in different orders at different group members, thus allowing for a degree of asynchronous operation; and yet it allows consistency constraints to be met in many cases.

Virtual synchrony does not specify the relative ordering of event processing, but it introduces an application-specific choice of causal and total ordering into a framework of sync-ordered group view changes and state transfers. A virtually synchronous execution is equivalent to some closely synchronous execution.

The ISIS CBCAST Protocol  ISIS uses vector timestamps to implement causally ordered multicaasts between the members of a peer group. The method is straightforwardly related to the technique described above for ordering requests in the gossip architecture, and the reader who feels comfortable with the presentation there may safely skip the following description.

It is assumed that all messages are multicast to all members of the group (including the sender). ISIS uses UDP/IP as its basic transport facility, and sends acknowledgments and retransmits packets as necessary to achieve reliability. Messages from a given member are sequenced and delivered in order. There is no assumption that hardware support for broadcast or multicast exists. If IP multicast [Deering and Cheriton 1990] is implemented, then ISIS can exploit it to send a single UDP packet to the appropriate multicast address; IP multicast takes advantage of hardware (for example, Ethernet) multicast facilities. Otherwise packets are sent point-to-point to the individual group members.

Let the members of the peer group be \( p_1, p_2, \ldots, p_n \). Once more we shall define, for each \( p_i \), a vector timestamp denoted by \( VT_i \) which is used to order multicast delivery. It will turn out that \( VT_j[i] \) is the count of multicast messages sent by \( p_j \) that causally lead up to the latest message delivered to \( p_j \). The following is the vector timestamp update algorithm:

1. All processes \( p_i \) initialize \( VT_i \) to zeroes.
2. When \( p_i \) multicasts a new message, it first increments \( VT_i[i] \) by 1; it piggybacks the value \( v_t = VT_i \) on the message.
3. When a message bearing a timestamp \( v_t \) is delivered to \( p_j \), \( p_j \)'s timestamp is updated as: \( VT_j[i] = \text{merge}(VT_j[i], v_t) \).

Every multicast message can be delivered to its sender immediately, since it is by definition in causal order with respect to messages already delivered to it. A multicast message arriving at \( p_j \)'s site from another \( p_i \) must be placed on the hold-back queue until it can be delivered in causal order. The incoming message’s timestamp \( v_t \) is examined, and the following criteria are used for transferring the message to \( p_j \)'s delivery queue:

- The message must be the next in sequence expected from \( p_i \), that is, \( v_t[i] = VT_i[i] + 1 \).
- All causally prior messages that have been delivered to \( p_j \) when it sent the message, should have been delivered to \( p_j \), that is, \( VT_j[k] \geq v_t[k] \) for \( k \neq i \).

It is straightforward to show that these criteria are necessary and sufficient to satisfy the safety property of causal ordering being assured. Liveness can only be established if:

- all members of the group are destinations for every message, and
- multicast delivery is atomic.

For otherwise, a message delivered to \( p_j \) might never be delivered to \( p_i \) and so some messages might indefinitely fail the second of the above criteria.

Atomicity and virtually synchronous group view changes  Successive group views \( \text{view}_0, \text{view}_1, \text{view}_2 \) and so on differ from their immediate predecessors either by the addition of a single new process, or by the removal of a single process that has failed or left the group voluntarily. Messages bearing the view’s identifier are delivered to all operational members of that view.

When a view change is to occur, ISIS sends a notification to all member sites, which then must reach a distributed agreement so as to sync-order the view change before passing the notification to the group members. Recall that, under virtually synchronous execution, view changes are delivered to group members consistently with respect to message delivery. This means that all messages sent during a view \( \text{view}_k \) are guaranteed to be delivered to all operational members of \( \text{view}_k \) before ISIS delivers notification of \( \text{view}_{k+1} \). It should be noted, however, that there is an intimate relation between the questions of when a view change is deemed to occur and of how and in which order messages are sent in view.

If some process \( p \) has joined the group to produce \( \text{view}_{k+1} \), then no message originally sent during \( \text{view}_k \) will be delivered to \( p \); but all messages sent by the members of \( \text{view}_{k+1} \) after ISIS has delivered notification of \( \text{view}_{k+1} \) to them will be delivered to the entire new group, that is, including \( p \).

ISIS multicaasts are not strictly atomic. Chapter 18 describes a version of the Amoeba multicast protocol that provides guarantees of atomicity despite up to \( r \) computer failures. This is a two-phase protocol, and the ISIS designers chose to avoid this expense. Instead, the ISIS guarantee is: as long as at least one member site that stores a message survives, then the message will be delivered to all surviving members despite the failure of the sender and any other sites.

Received messages are stored at each site until they are known to have been delivered to all members of the current view (information about which messages have been received is piggybacked on other traffic in the usual way). We shall call messages that have been delivered to all members of the current view group-stable (these are not to be confused with stable messages, described in Section 11.3). If ISIS finds that the sender of a message has failed, then some other site that stores the message — if such exists — is elected to send it to all sites not yet known to store it. Such a site can be chosen on a simple basis, for example it could be the oldest surviving member in the group view (which is ordered by age) that holds a copy of the message. Duplicates are of course detected.

If some member \( q \) of \( \text{view}_k \) has failed, producing \( \text{view}_{k+1} \), then the question arises of whether \( q \)'s application received all messages delivered during \( \text{view}_k \); and also whether \( q \) sent a message just before it failed which was received at \( q \) and perhaps other failed sites. These possibilities may seem awkward, but they are unlikely to lead to inconsistencies since \( q \) can no longer communicate and ISIS automatically deletes any
The ISIS algorithm for view agreement and message delivery.

**The ISIS ABCAST protocol** The ABCAST (causally and totally ordered multicast) protocol uses a sequencer, which is a member site designated as holding the ABCAST token. In each new group view a token-holder is elected. ABCAST messages carry CBCAST timestamps, but are marked as ABCAST. The token-holder attaches a sequence number to all ABCAST messages it delivers. It causally orders and delivers ABCAST messages, and sends so-called sets-order messages, which contain the sequence numbers of one or more identified ABCAST messages it has received. Other destination sites delay the delivery of a received ABCAST message until:

- they have received any prior ABCAST messages referred to in the sets-order message;
- they have delivered any causally prior CBCAST messages.

The token can be transferred between sites. The main advantage of this is that one site that is the sole source of ABCAST messages can be designated as the token holder; separate sets-order messages are unnecessary in this case.

**Discussion of the ISIS and gossip approaches**

ISIS is a mature product that has been used for a substantial number of real-world applications. It could be used to program the front ends and the group of replica...
managers in our general replication architecture. It is also applicable outside the domain of replicated data, for example in the propagation of financial and other information to varied clients. The gossip approach has no concept of process groups, is strictly applicable to replicated data and is relatively untested by comparison. But it offers a potentially more lightweight alternative to the use of ISIS for managing replicated data at least for certain applications. There are two major differences between these approaches: the use of process groups, and the use of multicast as opposed to gossip messages.

Process groups □ Ladin et al. [1992] have stated that replica managers can recover from failure, but how this is achieved is not described. In principle, it would involve a consistent transfer of the value, log, timestamps and other administrative information. Moreover, limited attention has been paid to the problem of incorporating a new replica manager into a running gossip-based system. ISIS, by contrast, provides a general approach to the integration of new process group members through its state transfer and group view maintenance mechanisms. But a process that is deemed to have failed can recover: a new process must be created, which must join the group afresh.

In the original ISIS implementation, if a network partitioning occurs then only majority of group members in a single partition is allowed to continue operation; the others are automatically killed. This is to obviate the possibility of several concurrent processes each continuing under the mistaken assumption that all their peers have failed – a situation that would clearly lead to inconsistencies. In the gossip approach, the replica manager is killed off when a network partitioning occurs. All replica managers continue to accept updates and satisfy any queries as long they have processed sufficient updates. Replica managers in a majority partition can execute updates. When the partition is repaired, gossip messages and information about forced and immediate operation are exchanged in order to bring all replicas up to date.

To refer once more to our bulletin board example, the difference betweenPartition approaches amounts to this. If a university became partitioned from the rest of the country, then under the basic ISIS approach the bulletin board service would be withdrawn altogether from that university until the partition was repaired. Under the gossip design, the users at that university could continue to read items locally, but a set of items would become stale compared to the items posted by those in the core (majority) partition. If posting is implemented as a forced operation, then users at the partitioned university could not continue to post – although users in the major partition could post. But if posting is a causal operation, then users at the partitioned university could continue to post items. However, as was pointed out for the case of very low rates of gossip exchange, two separate and inconsistent states might take place about an issue known to users in both partitions.

More recently, the original ISIS restriction on partitioned groups is viewed as tolerable only in the case of a LAN – where partitions are relatively rare. ISIS now incorporates a separate architecture for coping gracefully with WAN partitions – which are more common – and is similar to the gossip approach to partitioning [Birnbaum, private communication, 1993].

Multicast versus gossip □ The choice of ISIS multicast as opposed to the lazy gossip approach to update propagation depends upon the application: it involves a trade-off between the amount of communication and the timeliness of update delivery. ISIS, however, is not always the best choice. Multicasting updates to all replica managers immediately; in a gossip-based system, updates are propagated at whatever rate gossip messages are made to occur.

For large numbers of replica managers, an important factor affecting a system’s ability to scale is the number of messages that have to be transmitted and processed. The number of messages and bandwidth utilization for various operations are compared in Figure 11.19. The expressions for the gossip architecture are those that were deduced in Section 11.4 above. ISIS can utilize hardware multicast or broadcast over a single LAN that supports this, so entries are given both for the LAN case and for a case in which point-to-point messages must be used. It is very important to note that the figures for the number of messages indicate the worst case. They assume that no piggybacking takes place, and that each message travels alone in its network packet or packets.

As with the gossip design, a read-only (that is, query) operation can be carried out under ISIS by any single replica manager; this is realized as a multicast to a singleton sub-group of the replica group, and takes just two messages. An update under ISIS requires either a single multicast message or R point-to-point messages (where R is the number of replica managers), followed by R acknowledgements. ISIS directly supports operations that both read and modify the state, by multicasting requests to the replica group; in the gossip design, such operations have to be realized as two requests: a query and an update.

Of course, the total amount of update processing at each replica manager is the same in either type of system. The expressions given for the number of messages are intended for illustration. They are at best a crude guide to actual performance – that is, client response times and system throughput. This is in part because some ISIS-related messages are received in parallel; and in part because various administrative overheads have not been taken into account. Also, as stated above, the expressions derived are worst cases for the number of packets sent. Both ISIS and the gossip architecture combine multiple requests and acknowledgements in the same network packet wherever possible. For example, a more realistic expression for the number of packets sent per update or query-and-update in the ISIS case is \((1 + R/p)\), where \(p\), the piggy-back factor, has been estimated as lying in the range 3–15 in some applications involving small

<table>
<thead>
<tr>
<th>Number of messages per query</th>
<th>ISIS (hardware multicast)</th>
<th>ISIS (point-to-point)</th>
<th>Gossip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth (large query Q)</td>
<td>–</td>
<td>2</td>
<td>Q</td>
</tr>
<tr>
<td>Bandwidth (large update U)</td>
<td>(1 + R)</td>
<td>(2R)</td>
<td>(2 + (R-1)/G)</td>
</tr>
<tr>
<td>Number of messages per query-and-update</td>
<td>(1 + R)</td>
<td>(2R)</td>
<td>(4 + (R-1)/G)</td>
</tr>
</tbody>
</table>

The number of messages and bandwidth utilization for various operations. \(R = \) no. of replica managers; \(G = \) no. of updates in gossip message.
requests and using IP multicast [Birman, personal communication]. A piggy-back feature, with similar values can be expected in the gossip architecture.

**Timestamp sizes** Another factor affecting a system’s ability to scale is the size of timestamps. Section 11.4 described the suggestion of using a vector timestamp, which reduces the size of each timestamp to a constant. The size of a timestamp in a gossip-based system is proportional to the number of managers that handle updates directly from clients.

In an ISIS system, by contrast, requests carry timestamps of size

\[
\text{number of clients} + \text{number of replica managers}
\]

These are potentially considerably larger than those of a gossip-based system. To understand why there is a timestamp entry for each client, consider that an individual replica manager must order multicast requests arriving from two or more clients that may be communicating with one another. In a gossip-based system, a front end does not need to communicate from one client to another if necessary, until timestamps for all sending client’s outstanding updates have been returned by replica managers. The fronts end propagate and merge these timestamps to ensure that ordering is obeyed. But in ISIS system, the run-time at the replica managers cannot feed back back timestamps information to clients in this way (client requests reach replica managers at more or the same time). Requests must therefore bear timestamps which count the messages between the clients in order that the ISIS run-time can order them. (See the CBCAST protocol, described above.)

In favour of ISIS, client-to-client communication is not delayed by up acknowledgement, as it may be in a gossip-based system. Several techniques minimizing the amount of transmitted timestamp information in ISIS are described in Birman et al. [1991]. A particularly interesting issue addressed there is the problem of dealing with many overlapping groups.

11.6 Summary

Replicating data and other resources is an important means of achieving performance, high availability and fault tolerance in a distributed system. Clients access local copies instead of vying for the same resources; and clients can access alternative copies if one copy fails. The attendant problems to be tackled are: achievement of replication transparency, consistency and good response times and system throughput.

Maintaining consistency amounts to ensuring that all replicas process all updates from clients, and that they process these updates in consistent orders. The three types of ordering identified are causal, total and sync-ordered. The choice of which to use depends upon the application semantics.

This chapter has described a general replicated system architecture consisting of clients, front ends and replica managers. Front ends – which may be separate processes or libraries linked into clients – provide replication transparency, and communicate with replica managers on behalf of clients. The chapter has focused on the gossip architecture and the process group model, represented by ISIS.

In the gossip architecture, front ends communicate with a single replica manager. Replica managers communicate updates lazily, in so-called gossip messages. This architecture provides primarily for causal update operations, but also provides forced (total and causal) and immediate (sync-ordered) operation orderings.

Process groups are targets for multicast communication. ISIS provides a programming toolkit on top of UNIX for process groups. Its central design concept is the virtual synchrony. Processes use CBCAST (causal), ABCAST (total/causal) or GBCAST (sync-ordered) atomic multicasts; ISIS allows processes to leave or join groups, and detects process failures. It manages group views, and sync-orders group view notifications with respect to arriving multicasts. Process groups can be used to implement replica managers. They can also be used, for example, for parallel programs or for publishing information streams.

The process group approach seems preferable where timeliness of update propagation is essential. The gossip approach is lighter in the communication load it imposes, and is an alternative to process groups for applications with less stringent timeliness guarantees.

Replication as applied to fault tolerance is dealt with in Chapter 15. Replication is also relevant to transaction-based systems, which are described in Chapter 14.