

REPLICATION

- 18.1 Introduction
- 18.2 System model and the role of group communication
- 18.3 Fault-tolerant services
- 18.4 Case studies of highly available services:
The gossip architecture, Bayou and Coda
- 18.5 Transactions with replicated data
- 18.6 Summary

Replication is a key to providing high availability and fault tolerance in distributed systems. High availability is of increasing interest with the tendency towards mobile computing and consequently disconnected operation. Fault tolerance is an abiding concern for services provided in safety-critical and other important systems.

The first part of this chapter considers systems that apply a single operation at a time to collections of replicated objects. It begins with a description of architectural components and a system model for services that employ replication. We describe the implementation of group membership management as part of group communication, which is particularly important for fault-tolerant services.

The chapter then describes approaches to achieving fault tolerance. It introduces the correctness criteria of linearizability and sequential consistency, then explores two approaches: passive (primary-backup) replication, in which clients communicate with a distinguished replica; and active replication, in which clients communicate by multicast with all replicas.

Case studies of three systems for highly available services are considered. In the gossip and Bayou architectures, updates are propagated lazily between replicas of shared data. In Bayou, the technique of operational transformation is used to enforce consistency. Coda is an example of a highly available file service.

The chapter ends by considering transactions – sequences of operations – upon replicated objects. It considers the architectures of replicated transactional systems and how these systems handle server failures and network partitions.

18.1 Introduction

In this chapter, we study the replication of data: the maintenance of copies of data at multiple computers. Replication is a key 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 caching of resources from web servers in browsers and web proxy servers is a form of replication, since the data held in caches and at servers are replicas of one another. The DNS naming service, described in Chapter 13, maintains copies of name-to-attribute mappings for computers and is relied on for day-to-day access to services across the Internet.

Replication is a technique for enhancing services. The motivations for replication include:

Performance enhancement: The caching of data at clients and servers is by now familiar as a means of performance enhancement. For example, Chapter 2 pointed out that browsers and proxy servers cache copies of web resources to avoid the latency of fetching resources from the originating server. Furthermore, data are sometimes replicated transparently between several originating servers in the same domain. The workload is shared between the servers by binding all the server IP addresses to the site's DNS name, say *www.aWebSite.org*. A DNS lookup of *www.aWebSite.org* results in one of the several servers' IP addresses being returned, in a round-robin fashion (see Section 13.2.3). More sophisticated load-balancing strategies are required for more complex services based on data replicated between thousands of servers. As an example, Dilley *et al.* [2002] describe the approach to DNS name resolution adopted in the Akamai content distribution network.

Replication of immutable data is trivial: it increases performance with little cost to the system. Replication of changing data, such as that of the Web, incurs overheads in the form of protocols designed to ensure that clients receive up-to-date data (see Section 2.3.1). Thus there are limits to the effectiveness of replication as a performance-enhancement technique.

Increased availability: Users require services to be highly available. That is, the proportion of time for which a service is accessible with reasonable response times should be close to 100%. Apart from delays due to pessimistic concurrency control conflicts (due to data locking), the factors that are relevant to high availability are:

- server failures;
- network partitions and disconnected operation (communication disconnections that are often unplanned and are a side effect of user mobility).

To take the first of these, replication is a technique for automatically maintaining the availability of data despite server failures. If data are replicated at two or more failure-independent servers, then client software may be able to access data at 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 server data. If each of n servers has an independent probability p of failing

or becoming unreachable, then the availability of an object stored at each of these servers is:

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

For example, if there is a 5% probability of any individual server failing over a given time period and if there are two servers, then the availability is $1 - 0.05^2 = 1 - 0.0025 = 99.75\%$. An important difference between caching systems and server replication is that caches do not necessarily hold collections of objects such as files in their entirety. So caching does not necessarily enhance availability at the application level – a user may have one needed file but not another.

Network partitions (see Section 15.1) and disconnected operation are the second factor that militate against high availability. Mobile users may deliberately disconnect their computers or become unintentionally disconnected from a wireless network as they move around. For example, a user on a train with a laptop may have no access to networking (wireless networking may be interrupted, or they may have no such capability). In order to be able to work in these circumstances – so-called *disconnected working* or *disconnected operation* – the user will often prepare by copying heavily used data, such as the contents of a shared diary, from their usual environment to the laptop. But there is often a trade-off to availability during such a period of disconnection: when the user consults or updates the diary, they risk reading data that someone else has altered in the meantime. For example, they may make an appointment in a slot that has since been occupied. Disconnected working is only feasible if the user (or the application, on the user's behalf) can cope with stale data and can later resolve any conflicts that arise.

Fault tolerance: Highly available data is not necessarily strictly correct data. It may be out of date, for example; or two users on opposite sides of a network partition may make updates that conflict and need to be resolved. A fault-tolerant service, by contrast, always guarantees strictly correct behaviour despite a certain number and type of faults. The correctness concerns the freshness of data supplied to the client and the effects of the client's operations upon the data. Correctness sometimes also concerns the timeliness of the service's responses – such as, for example, in the case of a system for air traffic control, where correct data are needed on short timescales.

The same basic technique used for high availability – that of replicating data and functionality between computers – is also applicable for achieving fault tolerance. If up to f of $f + 1$ servers crash, then in principle at least one remains to supply the service. And if up to f servers can exhibit Byzantine failures, then in principle a group of $2f + 1$ servers can provide a correct service, by having the correct servers outvote the failed servers (who may supply spurious values). But fault tolerance is subtler than this simple description makes it seem. The system must manage the coordination of its components precisely to maintain the correctness guarantees in the face of failures, which may occur at any time.

A common requirement when data are replicated is for *replication transparency*. That is, clients should not normally have to be aware that multiple *physical* copies of data exist. As far as clients are concerned, data are organized as individual *logical* objects and they identify only one item in each case when they request an operation to be performed. Furthermore, clients expect operations to 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 that can vary in strength between applications – is that of consistency. This concerns whether the operations performed upon a collection of replicated objects produce results that meet the specification of correctness for those objects.

We saw in the example of the diary that during disconnected operation data may be allowed to become inconsistent, at least temporarily. But when clients remain connected it is often not acceptable for different clients (using different physical copies of data) to obtain inconsistent results when they make requests affecting the same logical objects. That is, it is not acceptable if the results break the application's correctness criteria.

We now examine in more detail the design issues raised when we replicate data to achieve highly available and fault-tolerant services. We also examine some standard solutions and techniques for dealing with those issues. First, Sections 18.2 to 18.4 cover the case where clients make individual invocations upon shared data. Section 18.2 presents a general architecture for managing replicated data and introduces group communication as an important tool. Group communication is particularly useful for achieving fault tolerance, which is the subject of Section 18.3. Section 18.4 describes techniques for high availability, including disconnected operation. It includes case studies of the gossip architecture, Bayou and the Coda file system. Section 18.5 examines how to support transactions on replicated data. As Chapters 16 and 17 explained, transactions are made up of sequences of operations, rather than single operations.

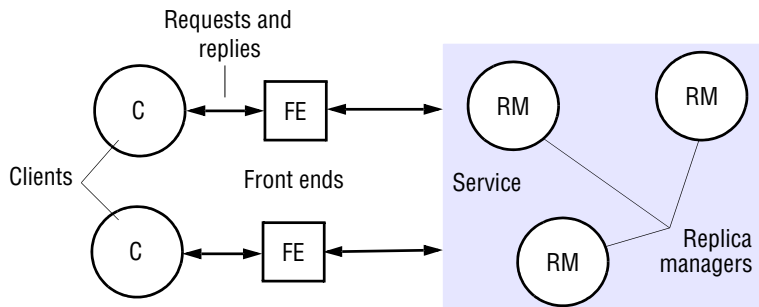
18.2 System model and the role of group communication

The data in our system consist of a collection of items that we shall call objects. An 'object' could be a file, say, or a Java object. But each such *logical* object is implemented by a collection of *physical* copies called *replicas*. The replicas are physical objects, each stored at a single computer, with data and behaviour that are tied to some degree of consistency by the system's operation. The 'replicas' of a given object are not necessarily identical, at least not at any particular point in time. Some replicas may have received updates that others have not received.

In this section, we provide a general system model for managing replicas and then describe the role of group communication systems in achieving fault tolerance through replication, highlighting the importance of view-synchronous group communication.

18.2.1 System model

We assume an asynchronous system in which processes may fail only by crashing. Our default assumption is that network partitions may not occur, but we shall sometimes consider what happens if they do occur. Network partitions make it harder to build failure detectors, which we use to achieve reliable and totally ordered multicast.

Figure 18.1 A basic architectural model for the management of replicated data

For the sake of generality, we describe architectural components by their roles and do not mean to imply that they are necessarily implemented by distinct processes (or hardware). The model involves replicas held by distinct *replica managers* (see Figure 18.1), which are components that contain the replicas on a given computer and perform operations upon them directly. This general model may be applied in a client-server environment, in which case a replica manager is a server. We shall sometimes simply call them servers instead. Equally, it may be applied to an application and application processes can in that case act as both clients and replica managers. For example, the user's laptop on a train may contain an application that acts as a replica manager for their diary.

We shall always require that a replica manager applies operations to its replicas recoverably. This allows us to assume that an operation at a replica manager does not leave inconsistent results if it fails part way through. We sometimes require each replica manager to be a *state machine* [Lamport 1978, Schneider 1990]. Such a replica manager applies operations to its replicas atomically (indivisibly), so that its execution is equivalent to performing operations in some strict sequence. Moreover, the state of its replicas is a deterministic function of their initial states and the sequence of operations that it applies to them. Other stimuli, such as the reading on a clock or an attached sensor, have no bearing on these state values. Without this assumption, consistency guarantees between replica managers that accept update operations independently could not be made. The system can only determine which operations to apply at all replica managers and in what order – it cannot reproduce non-deterministic effects. The assumption implies that it may not be possible, depending upon the threading architecture, for the servers to be multi-threaded.

Often each replica manager maintains a replica of every object, and we assume this is so unless we state otherwise. However, the replicas of different objects may be maintained by different sets of replica managers. For example, one object may be needed mostly by clients on one network and another by clients on another network. There is little to be gained by replicating them at managers on the other network.

The set of replica managers may be static or dynamic. In a dynamic system, new replica managers may appear (for example, if a second secretary copies a diary onto their laptop); this is not allowed in a static system. In a dynamic system, replica managers

may crash, and they are then deemed to have left the system (although they may be replaced). In a static system, replica managers do not crash (crashing implies *never* executing another step), but they may cease operating for an indefinite period. We return to the issue of failure in Section 18.4.2.

The general model of replica management is shown in Figure 18.1. A collection of replica managers provides a service to clients. The clients see a service that gives them access to objects (for example, diaries or bank accounts), which in fact are replicated at the managers. Each client requests a series of operations – invocations upon one or more of the objects. An operation may involve a combination of reads of objects and updates to objects. Requested operations that involve no updates are called *read-only requests*; requested operations that update an object are called *update requests* (these may also involve reads).

Each client's requests are first handled by a component called a *front end*. The role of the front end is 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 making replication transparent. A front end may be implemented in the client's address space, or it may be a separate process.

In general, five phases are involved in the performance of a single request upon the replicated objects [Wiesmann *et al.* 2000]. The actions in each phase vary according to the type of system, as will become clear in the next two sections. For example, a service that supports disconnected operation behaves differently from one that provides a fault-tolerant service. The phases are as follows:

Request: The front end issues the request to one or more replica managers:

- *either* the front end communicates with a single replica manager, which in turn communicates with other replica managers;
- *or* the front end multicasts the request to the replica managers.

Coordination: The replica managers coordinate in preparation for executing the request consistently. They agree, if necessary at this stage, on whether the request is to be applied (it might not be applied at all if failures occur at this stage). They also decide on the ordering of this request relative to others. All of the types of ordering defined for multicast in Section 15.4.3 also apply to request handling and we define those orders again for this context:

FIFO ordering: If a front end issues request r and then request r' , any correct replica manager that handles r' handles r before it.

Causal ordering: If the issue of request r happened-before the issue of request r' , then any correct replica manager that handles r' handles r before it.

Total ordering: If a correct replica manager handles r before request r' , then any correct replica manager that handles r' handles r before it.

Most applications require FIFO ordering. We discuss the requirements for causal and total ordering – and the hybrid orderings that are both FIFO and total, or both causal and total – in the next two sections.

Execution: The replica managers execute the request – perhaps *tentatively*: that is, in such a way that they can undo its effects later.

Agreement: The replica managers reach consensus on the effect of the request – if any – that will be committed. For example, in a transactional system the replica managers may collectively agree to abort or commit the transaction at this stage.

Response: One or more replica managers responds to the front end. In some systems, one replica manager sends the response. In others, the front end receives responses from a collection of replica managers and selects or synthesizes a single response to pass back to the client. For example, it could pass back the first response to arrive, if high availability is the goal. If tolerance of Byzantine failures is the goal, then it could give the client the response that a majority of the replica managers provides.

Different systems may make different choices about the ordering of the phases, as well as their contents. For example, in a system that supports disconnected operation, it is important to give the client (the application on the user’s laptop, say) as early a response as possible. The user does not want to wait until the replica manager on the laptop and the replica manager back in the office can coordinate. By contrast, in a fault-tolerant system the client is not given the response until the end, when the correctness of the result can be guaranteed.

18.2.2 The role of group communication

Chapter 6 introduced the concept of group communication and Section 15.4 expanded on this discussion by covering algorithms for reliability and ordering of message delivery in group communication systems. In this chapter, we look at the role of groups in managing replicated data. The discussion in Section 15.4 took the membership of groups to be statically defined, although group members may crash. In replication, and indeed in many other practical circumstances, there is a strong requirement for dynamic membership, in which processes join and leave the group as the system executes. In a service that manages replicated data, for example, users may add or withdraw a replica manager, or a replica manager may crash and thus need to be withdrawn from the system’s operation. Group membership management, which was introduced in Section 6.2.2, is therefore particularly important in this context.

Systems that can adapt as processes join, leave and crash – fault-tolerant systems, in particular – require the more advanced features of failure detection and notification of membership changes. A full group membership service maintains *group views*, which are lists of the current group members, identified by their unique process identifiers. The list is ordered, for example, according to the sequence in which the members joined the group. A new group view is generated each time that a process is added or excluded.

It is important to understand that a group membership service may exclude a process from a group because it is *Suspected*, even though it may not have crashed. A communication failure may have made the process unreachable, while it continues to execute normally. A membership service is always free to exclude such a process. The effect of exclusion is that no messages will be delivered to that process henceforth. Moreover, in the case of a closed group, if that process becomes connected again, any messages it attempts to send will not be delivered to the group members. That process will have to rejoin the group (as a ‘reincarnation’ of itself, with a new identifier), or abort its operations.

A false suspicion of a process and the consequent exclusion of the process from the group may reduce the group's effectiveness. The group has to manage without the extra reliability or performance that the withdrawn process could potentially have provided. The design challenge, apart from designing failure detectors to be as accurate as possible, is to ensure that a system based on group communication does not behave *incorrectly* if a process is falsely suspected.

An important consideration is how a group management service treats network partitions. Disconnection or the failure of components such as a router in a network may split a group of processes into two or more subgroups, with communication between the subgroups impossible. Group management services differ in whether they are *primary-partition* or *partitionable*. In the first case, the management service allows at most one subgroup (a majority) to survive a partition; the remaining processes are informed that they should suspend operations. This arrangement is appropriate for cases where the processes manage important data and the costs of inconsistencies between two or more subgroups outweigh any advantage of disconnected working.

On the other hand, in some circumstances it is acceptable for two or more subgroups to continue to operate – a partitionable group membership service allows this. For example, in an application in which users hold an audio or video conference to discuss some issues, it may be acceptable for two or more subgroups of users to continue their discussions independently despite a partition. They can merge their results when the partition heals and the subgroups are connected again.

View delivery • Consider the task of a programmer writing an application that runs in each process in a group and that must cope with new and lost members. The programmer needs to know that the system treats each member in a consistent way when the membership changes. It would be awkward if the programmer had to query the state of all the other members and reach a global decision whenever a membership change occurred, rather than being able to make a local decision on how to respond to the change. The programmer's life is made harder or easier according to the guarantees that apply when the system delivers views to the group members.

For each group g the group management service delivers to any member process $p \in g$ a series of views $v_0(g)$, $v_1(g)$, $v_2(g)$, etc. For example, a series of views could be $v_0(g) = (p)$, $v_1(g) = (p, p')$ and $v_2(g) = (p)$ – p joins an empty group, then p' joins the group, then p' leaves it. Although several membership changes may occur concurrently, such as when one process joins the group just as another leaves, the system imposes an order on the sequence of views given to each process.

We speak of a member *delivering a view* when a membership change occurs and the application is notified of the new membership, just as we speak of a process *delivering* a multicast message. As with multicast delivery, delivering a view is distinct from receiving a view. Group membership protocols keep proposed views on a hold-back queue until all extant members can agree to their delivery.

We also speak of an event as occurring *in a view* $v(g)$ at process p if, at the time of the event's occurrence, p has delivered $v(g)$ but has not yet delivered the next view, $v'(g)$.

Some basic requirements for view delivery are as follows:

Order: If a process p delivers view $v(g)$ and then view $v'(g)$, then no other process $q \neq p$ delivers $v'(g)$ before $v(g)$.

Integrity: If process p delivers view $v(g)$, then $p \in v(g)$.

Non-triviality: If process q joins a group and is or becomes indefinitely reachable from process $p \neq q$, then eventually q is always in the views that p delivers. Similarly, if the group partitions and remains partitioned, then eventually the views delivered in any one partition will exclude any processes in another partition.

The first of these requirements goes some way to giving the programmer a consistency guarantee by ensuring that view changes always occur in the same order at different processes. The second requirement is a ‘sanity check’. The third guards against trivial solutions. For example, a membership service that tells every process, regardless of its connectivity, that it is in a group all by itself is not of great interest. The non-triviality condition states that if two processes that have each joined the same group can eventually communicate indefinitely, then they should each be deemed members of that same group. Similarly, it requires that, when a partition occurs, the membership service should eventually reflect the partition. The condition does not state how the group membership service should behave in the problematic case of intermittent connectivity.

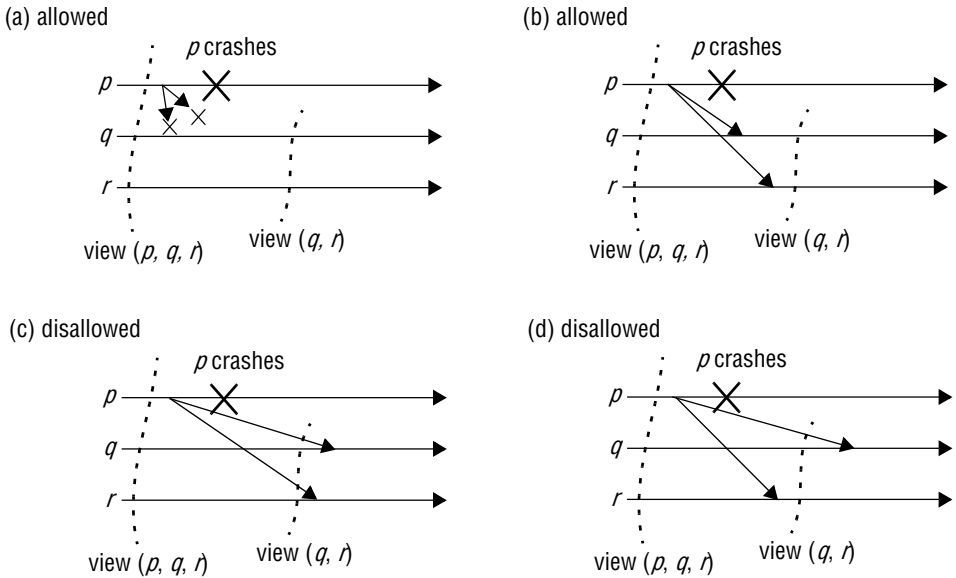
View-synchronous group communication • A *view-synchronous* group communication system makes guarantees additional to those above about the delivery ordering of view notifications with respect to the delivery of multicast messages. View-synchronous communication extends the reliable multicast semantics that we described in Chapter 15 to take account of changing group views. For the sake of simplicity, we restrict our discussion to the case where partitions may not occur. The guarantees provided by view-synchronous group communication are as follows:

Agreement: Correct processes deliver the same sequence of views (starting from the view in which they join the group) and the same set of messages in any given view. That is, if a correct process delivers message m in view $v(g)$, then all other correct processes that deliver m also do so in the view $v(g)$.

Integrity: If a correct process p delivers message m , then it will not deliver m again. Furthermore, $p \in \text{group}(m)$ and the process that sent m is in the view in which p delivers m .

Validity (closed groups): Correct processes always deliver the messages that they send. If the system fails to deliver a message to any process q , then it notifies the surviving processes by delivering a new view with q excluded, immediately after the view in which any of them delivered the message. That is, let p be any correct process that delivers message m in view $v(g)$. If some process $q \in v(g)$ does not deliver m in view $v(g)$, then the next view $v'(g)$ that p delivers has $q \notin v'(g)$.

Consider a group with three processes, p , q and r (see Figure 18.2). Suppose that p sends a message m while in view (p, q, r) but that p crashes soon after sending m , while q and r are correct. One possibility is that p crashes before m has reached any other process. In this case, q and r each deliver the new view (q, r) , but neither ever delivers m (Figure 18.2a). The other possibility is that m has reached at least one of the two surviving processes when p crashes. Then q and r both deliver first m and then the view (q, r) (Figure 18.2b). It is not allowed for q and r to deliver first the view (q, r) and then m (Figure 18.2c), since then they would deliver a message from a process that they have

Figure 18.2 View-synchronous group communication

been informed has failed; nor can the two deliver the message and the new view in opposite orders (Figure 18.2d).

In a view-synchronous system, the delivery of a new view draws a conceptual line across the system and every message that is delivered at all is consistently delivered on one side or the other of that line. This enables the programmer to draw useful conclusions about the set of messages that other correct processes have delivered when it delivers a new view, based only on the local ordering of message delivery and view delivery events.

An illustration of the usefulness of view-synchronous communication is how it can be used to achieve *state transfer* – the transfer of the working state from a current member of a process group to a new member of the group. For example, if the processes are replica managers that each hold the state of a diary, then a replica manager that joins the group for that diary needs to acquire the diary's current state when it joins. But the diary may be updated concurrently while the state is being captured. It is important that the replica manager does not miss any update messages that are not reflected in the state it acquires and that it does not reapply update messages that are already reflected in the state (unless those updates are idempotent).

To achieve this state transfer, we can use view-synchronous communication in a simple scheme such as the following. Upon delivery of the first view containing the new process, some distinct process from amongst the pre-existing members – say, the oldest – captures its state, sends it one-to-one to the new member and suspends its execution. All other pre-existing processes suspend their execution. Note that precisely the set of updates reflected in this state has, by definition, been applied at all other members. Upon

receipt of the state, the new process integrates it and multicasts a ‘commence!’ message to the group, at which point all proceed once more.

Discussion • The notion of view-synchronous group communication that we have presented is a formulation of the ‘virtually synchronous’ communication paradigm originally developed in the ISIS system [Birman 1993, Birman *et al.* 1991, Birman and Joseph 1987b]. Schiper and Sandoz [1993] describe a protocol for achieving view-synchronous (or as they call it, *view-atomic*) communication. Note that a group membership service achieves consensus, but it does so without flouting the impossibility result of Fischer *et al.* [1985]. As we discussed in Section 15.5.4, a system can circumvent that result by using an appropriate failure detector.

Schiper and Sandoz also provide a uniform version of view-synchronous communication in which the agreement condition covers the case of processes that crash. This is similar to uniform agreement for multicast communication, which we described in Section 15.4.2. In the uniform version of view-synchronous communication, even if a process crashes after it delivers a message, all correct processes are forced to deliver the message in the same view. This stronger guarantee is sometimes needed in fault-tolerant applications, since a process that has delivered a message may have had an effect on the outside world before crashing. For the same reason, Hadzilacos and Toueg [1994] consider uniform versions of the reliable and ordered multicast protocols described in Chapter 15.

The V system [Cheriton and Zwaenepoel 1985] was the first system to include support for process groups. After ISIS, process groups with some type of group membership service were developed in several other systems, including Horus [van Renesse *et al.* 1996], Totem [Moser *et al.* 1996] and Transis [Dolev and Malki 1996].

Variations on view synchrony have been proposed for partitionable group membership services, including support for partition-aware applications [Babaoglu *et al.* 1998] and extended virtual synchrony [Moser *et al.* 1994].

Finally, Cristian [1991] discusses a group membership service for *synchronous* distributed systems.

18.3 Fault-tolerant services

In this section, we examine how to provide a service that is correct despite up to f process failures, by replicating data and functionality at replica managers. For the sake of simplicity, we assume that communication remains reliable and that no partitions occur.

Each replica manager is assumed to behave according to a specification of the semantics of the objects it manages, when they have not crashed. For example, a specification of bank accounts would include an assurance that funds transferred between bank accounts can never disappear, and that only deposits and withdrawals affect the balance of any particular account.

Intuitively, a service based on replication is correct if it keeps responding despite failures and if clients cannot tell the difference between the service they obtain from an implementation with replicated data and one provided by a single correct replica manager. Care is needed in meeting these criteria. If precautions are not taken, then

anomalies can arise when there are several replica managers – even bearing in mind that we are considering the effects of individual operations, not transactions.

Consider a naive replication system, in which a pair of replica managers at computers A and B each maintain replicas of two bank accounts, x and y . Clients read and update the accounts at their local replica manager but try another replica manager if the local one fails. Replica managers propagate updates to one another in the background after responding to the clients. Both accounts initially have a balance of \$0.

Client 1 updates the balance of x at its local replica manager B to be \$1 and then attempts to update y 's balance to be \$2, but discovers that B has failed. Client 1 therefore applies the update at A instead. Now client 2 reads the balances at its local replica manager A . It finds first that y has \$2 and then that x has \$0 – the update to bank account x from B has not arrived, since B failed. The situation is shown below, where the operations are labelled by the computer at which they first took place and lower operations happen later:

Client 1:	Client 2:
$setBalance_B(x, 1)$	
$setBalance_A(y, 2)$	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$

This execution does not match a common-sense specification for the behaviour of bank accounts: client 2 should have read a balance of \$1 for x , given that it read the balance of \$2 for y , since y 's balance was updated after that of x . The anomalous behaviour in the replicated case could not have occurred if the bank accounts had been implemented by a single server. We can construct systems that manage replicated objects without the anomalous behaviour produced by the naive protocol in our example. First, we need to understand what counts as correct behaviour for a replicated system.

Linearizability and sequential consistency • There are various correctness criteria for replicated objects. The most strictly correct systems are *linearizable*, and this property is called *linearizability*. In order to understand linearizability, consider a replicated service implementation with two clients. Let the sequence of read and update operations that client i performs in some execution be $o_{i0}, o_{i1}, o_{i2}, \dots$. Each operation o_{ij} in these sequences is specified by the operation type and the arguments and return values as they occurred at runtime. We assume that every operation is synchronous. That is, clients wait for one operation to complete before requesting the next.

A single server managing a single copy of the objects would serialize the operations of the clients. In the case of an execution with only client 1 and client 2, this interleaving of the operations could be $o_{20}, o_{21}, o_{10}, o_{22}, o_{11}, o_{12}, \dots$, say. We define our correctness criteria for replicated objects by referring to a *virtual* interleaving of the clients' operations, which does not necessarily physically occur at any particular replica manager but that establishes the correctness of the execution.

A replicated shared object service is said to be linearizable if *for any execution* there is some interleaving of the series of operations issued by all the clients that satisfies the following two criteria:

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of operations in the interleaving is consistent with the real times at which the operations occurred in the actual execution.

This definition captures the idea that for any set of client operations there is a virtual canonical execution – the interleaved operations that the definition refers to – against a virtual single image of the shared objects. And each client sees a view of the shared objects that is consistent with that single image: that is, the results of the client's operations make sense as they occur within the interleaving.

The service that gave rise to the execution of the bank account clients in the preceding example is not linearizable. Even ignoring the real time at which the operations took place, there is no interleaving of the two clients' operations that would satisfy any correct bank account specification: for auditing purposes, if one account update occurred after another, then the first update should be observed if the second has been observed.

Note that linearizability concerns only the interleaving of individual operations and is not intended to be transactional. A linearizable execution may break application-specific notions of consistency if concurrency control is not applied.

The real-time requirement in linearizability is desirable in an ideal world, because it captures our notion that clients should receive up-to-date information. But, equally, the presence of real time in the definition raises the issue of linearizability's practicality, because we cannot always synchronize clocks to the required degree of accuracy. A weaker correctness condition is *sequential consistency*, which captures an essential requirement concerning the order in which requests are processed without appealing to real time. The definition keeps the first criterion from the definition for linearizability but modifies the second.

A replicated shared object service is said to be sequentially consistent if *for any execution* there is some interleaving of the series of operations issued by all the clients that satisfies the following two criteria:

- The interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
- The order of operations in the interleaving is consistent with the program order in which each individual client executed them.

Note that absolute time does not appear in this definition. Nor does any other *total* order on all operations. The only notion of ordering that is relevant is the order of events at each separate client – the program order. The interleaving of operations can shuffle the sequence of operations from a set of clients in any order, as long as each client's order is not violated and the result of each operation is consistent, in terms of the objects' specification, with the operations that preceded it. This is similar to shuffling together several packs of cards so that they are intermingled in such a way as to preserve the original order of each pack.

Every linearizable service is also sequentially consistent, since real-time order reflects each client's program order. The converse does not hold. An example execution

for a service that is sequentially consistent but not linearizable follows:

Client 1:	Client 2:
$setBalance_B(x, 1)$	$getBalance_A(y) \rightarrow 0$
	$getBalance_A(x) \rightarrow 0$
$setBalance_A(y, 2)$	

This execution is possible under a naive replication strategy even if neither of the computers A or B fails but if the update of x that client 1 made at B has not reached A when client 2 reads it. The real-time criterion for linearizability is not satisfied, since $getBalance_A(x) \rightarrow 0$ occurs later than $setBalance_B(x, 1)$; but the following interleaving satisfies both criteria for sequential consistency: $getBalance_A(y) \rightarrow 0$, $getBalance_A(x) \rightarrow 0$, $setBalance_B(x, 1)$, $setBalance_A(y, 2)$.

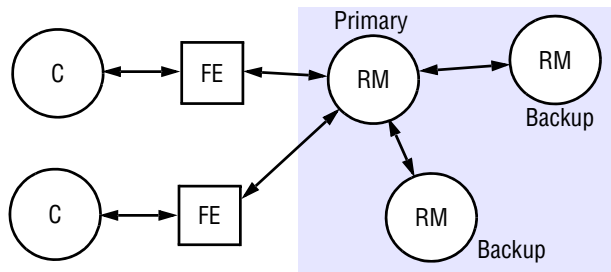
Lamport conceived of both sequential consistency [1979] and linearizability [1986] in relation to shared memory registers (although he used the term ‘atomicity’ instead of ‘linearizability’). Herlihy and Wing [1990] generalized the idea to cover arbitrary shared objects. Distributed shared memory systems also feature weaker consistency models, as discussed on the companion website for the book [www.cdk5.net/dsm].

18.3.1 Passive (primary-backup) replication

In the *passive* or *primary-backup* model of replication for fault tolerance (Figure 18.3), there is at any one time a single primary replica manager and one or more secondary replica managers – ‘backups’ or ‘slaves’. In the pure form of the model, front ends communicate only with the primary replica manager to obtain the service. The primary replica manager executes the operations and sends copies of the updated data to the backups. If the primary fails, one of the backups is promoted to act as the primary.

The sequence of events when a client requests an operation to be performed is as follows:

1. *Request*: The front end issues the request, containing a unique identifier, to the primary replica manager.
2. *Coordination*: The primary takes each request atomically, in the order in which it receives it. It checks the unique identifier, in case it has already executed the request, and if so it simply resends the response.
3. *Execution*: The primary executes the request and stores the response.
4. *Agreement*: If the request is an update, then the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowledgement.
5. *Response*: The primary responds to the front end, which hands the response back to the client.

Figure 18.3 The passive (primary-backup) model for fault tolerance

This system obviously implements linearizability if the primary is correct, since the primary sequences all the operations upon the shared objects. If the primary fails, then the system retains linearizability if a single backup becomes the new primary and if the new system configuration takes over exactly where the last left off. That is if:

- The primary is replaced by a unique backup (if two clients began using two backups, then the system could perform incorrectly).
- The replica managers that survive agree on which operations had been performed at the point when the replacement primary takes over.

Both of these requirements are met if the replica managers (primary and backups) are organized as a group and if the primary uses view-synchronous group communication to send the updates to the backups. The first of the above two requirements is then easily satisfied. When the primary crashes, the communication system eventually delivers a new view to the surviving backups, one that excludes the old primary. The backup that replaces the primary can be chosen by any function of that view. For example, the backups can choose the first member in that view as the replacement. That backup can register itself as the primary with a name service that the clients consult when they suspect that the primary has failed (or when they require the service in the first place).

The second requirement is also satisfied, by the ordering property of view-synchrony and the use of stored identifiers to detect repeated requests. The view-synchronous semantics guarantee that either all the backups or none of them will deliver any given update before delivering the new view. Thus the new primary and the surviving backups all agree on whether any particular client's update has or has not been processed.

Consider a front end that has not received a response. The front end retransmits the request to whichever backup takes over as the primary. The primary may have crashed at any point during the operation. If it crashed before the agreement stage (4), then the surviving replica managers cannot have processed the request. If it crashed during the agreement stage, then they may have processed the request. If it crashed after that stage, then they have definitely processed it. But the new primary does not have to know what stage the old primary was in when it crashed. When it receives a request, it proceeds from stage 2 above. By view-synchrony, no consultation with the backups is necessary, because they have all processed the same set of messages.

Discussion of passive replication • The primary-backup model may be used even where the primary replica manager behaves in a non-deterministic way, for example due to multi-threaded operation. Since the primary communicates the updated state from the operations rather than a specification of the operations themselves, the backups slavishly record the state determined by the primary's actions alone.

To survive up to f process crashes, a passive replication system requires $f+1$ replica managers (such a system cannot tolerate Byzantine failures). The front end requires little functionality to achieve fault tolerance. It just needs to be able to look up the new primary when the current primary does not respond.

Passive replication has the disadvantage of providing relatively large overheads. View-synchronous communication requires several rounds of communication per multicast, and if the primary fails then yet more latency is incurred while the group communication system agrees upon and delivers the new view.

In a variation of the model presented here, clients may be able to submit read requests to the backups, thus offloading work from the primary. The guarantee of linearizability is thereby lost, but the clients receive a sequentially consistent service.

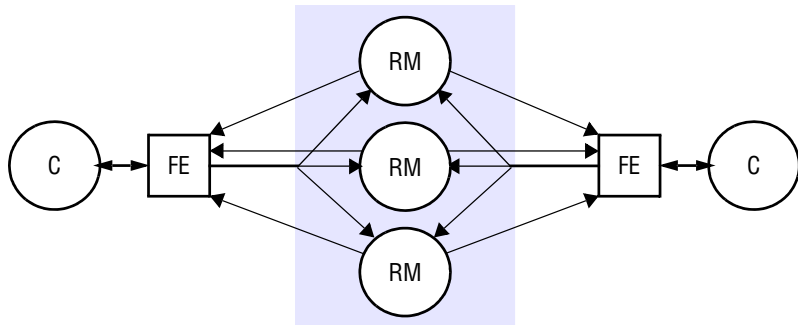
Passive replication is used in the Harp replicated file system [Liskov *et al.* 1991]. The Sun Network Information Service (NIS, formerly Yellow Pages) uses passive replication to achieve high availability and good performance, although with weaker guarantees than sequential consistency. The weaker consistency guarantees are still satisfactory for many purposes, such as storing certain types of system administration records. The replicated data is updated at a master server and propagated from there to slave servers using one-to-one (rather than group) communication. Clients may communicate with either a master or a slave server to retrieve information. In NIS, however, clients may not request updates: updates are made to the master's files.

18.3.2 Active replication

In the *active* model of replication for fault tolerance (see Figure 18.4), the replica managers are state machines that play equivalent roles and are organized as a group. Front ends multicast their requests to the group of replica managers and all the replica managers process the request independently but identically and reply. If any replica manager crashes, this need have no impact upon the performance of the service, since the remaining replica managers continue to respond in the normal way. We shall see that active replication can tolerate Byzantine failures, because the front end can collect and compare the replies it receives.

Under active replication, the sequence of events when a client requests an operation to be performed is as follows:

1. *Request*: The front end attaches a unique identifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive. The front end is assumed to fail by crashing at worst. It does not issue the next request until it has received a response.
2. *Coordination*: The group communication system delivers the request to every correct replica manager in the same (total) order.

Figure 18.4 Active replication

3. *Execution*: Every replica manager executes the request. Since they are state machines and since requests are delivered in the same total order, correct replica managers all process the request identically. The response contains the client's unique request identifier.

4. *Agreement*: No agreement phase is needed, because of the multicast delivery semantics.

5. *Response*: Each replica manager sends its response to the front end. The number of replies that the front end collects depends upon the failure assumptions and the multicast algorithm. If, for example, the goal is to tolerate only crash failures and the multicast satisfies uniform agreement and ordering properties, then the front end passes the first response to arrive back to the client and discards the rest (it can distinguish these from responses to other requests by examining the identifier in the response).

This system achieves sequential consistency. All correct replica managers process the same sequence of requests. The reliability of the multicast ensures that every correct replica manager processes the same set of requests and the total order ensures that they process them in the same order. Since they are state machines, they all end up with the same state as one another after each request. Each front end's requests are served in FIFO order (because the front end awaits a response before making the next request), which is the same as 'program order'. This ensures sequential consistency.

If clients do not communicate with other clients while waiting for responses to their requests, then their requests are processed in *happened-before* order. If clients are multi-threaded and can communicate with one another while awaiting responses from the service, then to guarantee request processing in *happened-before* order we would have to replace the multicast with one that is both causally and totally ordered.

The active replication system does not achieve linearizability. This is because the total order in which the replica managers process requests is not necessarily the same as the real-time order in which the clients made their requests. Schneider [1990] describes how, in a synchronous system with approximately synchronized clocks, the total order in which the replica managers process requests can be based on the order of physical

timestamps that the front ends supply with their requests. This does not guarantee linearizability, because the timestamps are not perfectly accurate; but it approximates it.

Discussion of active replication • We have assumed a solution to totally ordered and reliable multicast. As Chapter 15 pointed out, solving reliable and totally ordered multicast is equivalent to solving consensus. Solving consensus in turn requires either that the system is synchronous or that a technique such as employing failure detectors is used in an asynchronous system, to work around the impossibility result of Fischer *et al.* [1985].

Some solutions to consensus, such as that of Canetti and Rabin [1993], work even with the assumption of Byzantine failures. Given such a solution, and therefore a solution to totally ordered and reliable multicast, the active replication system can mask up to f Byzantine failures, as long as the service incorporates at least $2f + 1$ replica managers. Each front end waits until it has collected $f + 1$ identical responses and passes that response back to the client. It discards other responses to the same request. To be strictly sure of which response is really associated with which request (given Byzantine behaviour), we require that the replica managers digitally sign their responses.

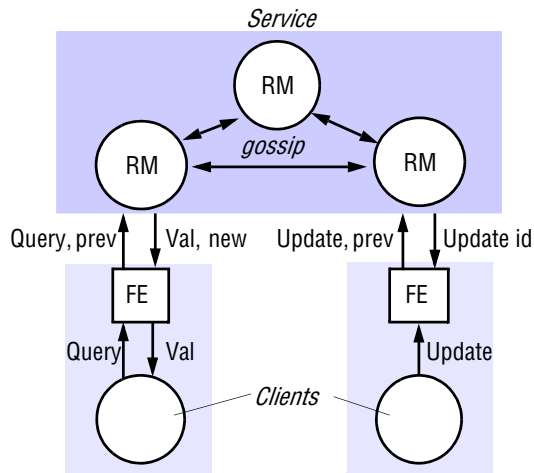
It may be possible to relax the system that we have described. First, we have assumed that all updates to the shared replicated objects must occur in the same order. However, in practice some operations may commute: that is, the effect of two operations performed in the order $o_1; o_2$ may be the same as if they were performed in the reverse order, $o_2; o_1$. For example, any two read-only operations (from different clients) commute, and any two operations that do not perform reads but update distinct objects commute. An active replication system may be able to exploit knowledge of commutativity in order to avoid the expense of ordering all the requests. We pointed out in Chapter 15 that some have proposed application-specific multicast ordering semantics [Cheriton and Skeen 1993, Pedone and Schiper 1999].

Finally, front ends may send read-only requests only to individual replica managers. In doing so, they lose the fault tolerance that comes with multicasting requests, but the service remains sequentially consistent. Moreover, the front end can easily mask the failure of a replica manager in this case, simply by submitting the read-only request to another replica manager.

18.4 Case studies of highly available services: The gossip architecture, Bayou and Coda

In this section, we consider how to apply replication techniques to make services highly available. Our emphasis now is on giving clients access to the service – with reasonable response times – for as much of the time as possible, even if some results do not conform to sequential consistency. For example, the user on the train described at the beginning of this chapter may be willing to cope with temporary inconsistencies between copies of data such as diaries if they can continue to work while disconnected and fix any problems later.

In Section 18.3, we saw that fault-tolerant systems transmit updates to the replica managers in an ‘eager’ fashion: all correct replica managers receive the updates as soon

Figure 18.5 Query and update operations in a gossip service

as possible and they reach collective agreement before passing control back to the client. This behaviour is undesirable for highly available operation. Instead, the system should provide an acceptable level of service using a minimal set of replica managers connected to the client. And it should minimize how long the client is tied up while replica managers coordinate their activities. Weaker degrees of consistency generally require less agreement and so allow shared data to be more available.

We now examine the design of three systems that provide highly available services: the gossip architecture, Bayou and Coda.

18.4.1 The gossip architecture

Ladin *et al.* [1992] developed what we call the *gossip architecture* as a framework for implementing highly available services by replicating data close to the points where groups of clients need it. The name reflects the fact that the replica managers exchange ‘gossip’ messages periodically in order to convey the updates they have each received from clients (see Figure 18.5). The architecture is based upon earlier work on databases by Fischer and Michael [1982] and Wu and Bernstein [1984]. It may be used, for example, to create a highly available electronic bulletin board or diary service.

A gossip service provides two basic types of operation: *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). A key feature is that front ends send queries and updates to any replica manager they choose, provided it is available and can provide reasonable response times. The system makes two guarantees, even though replica managers may be temporarily unable to communicate with one another:

Each client obtains a consistent service over time: In answer to a query, replica managers only ever provide a client with data that reflects at least the updates that the

client has observed so far. This is even though clients may communicate with different replica managers at different times, and therefore could in principle communicate with a replica manager that is ‘less advanced’ than one they used before.

Relaxed consistency between replicas: All replica managers eventually receive all updates and they apply updates with ordering guarantees that make the replicas sufficiently similar to suit the needs of the application. It is important to realize that while the gossip architecture can be used to achieve sequential consistency, it is primarily intended to deliver weaker consistency guarantees. Two clients may observe different replicas even though the replicas include the same set of updates, and a client may observe stale data.

To support relaxed consistency, the gossip architecture supports *causal* update ordering, as we defined it in Section 15.2.1. It also supports stronger ordering guarantees in the form of *forced* (total and causal) and *immediate* ordering. Immediate-ordered updates are applied in a consistent order relative to *any* other update at all replica managers, whether the other update ordering is specified as causal, forced or immediate. Immediate ordering is provided in addition to forced ordering, because a forced-order update and a causal-order update that are not related by the *happened-before* relation may be applied in different orders at different replica managers.

The choice of which ordering 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. Note that queries, which can be satisfied by any single replica manager, are always executed in causal order with respect to other operations.

Consider an electronic bulletin board application, in which a client program (which incorporates the front end) executes on the user’s computer and communicates with a local replica manager. The client sends the user’s postings to the local replica manager and the replica manager sends new postings in gossip messages to other replica managers. Readers of bulletin boards experience slightly out-of-date lists of posted items, but this does not usually matter if the delay is on the order of minutes or hours rather than days. Causal ordering could be used for posting items. This would mean that in general postings could appear in different orders at different replica managers but that, for example, a posting whose subject is ‘Re: oranges’ will always be posted after the message about ‘oranges’ to which it refers. Forced ordering could be used for adding a new subscriber to a bulletin board, so that there is an unambiguous record of the order in which users joined. Immediate ordering could be used for removing a user from a bulletin board’s subscription list, so that messages could not be retrieved by that user via some tardy replica manager once the deletion operation had returned.

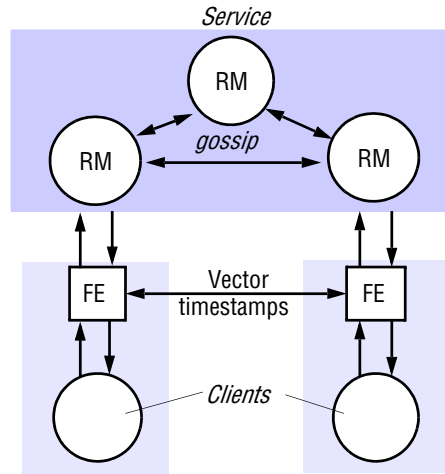
The front end for a gossip service handles operations that the client makes using an application-specific API and turns them into gossip operations. In general, client operations can either read the replicated state, modify it or both. Since in gossip updates purely modify the state, the front end converts an operation that both reads and modifies the state into a separate query and update.

In terms of our basic replication model, an outline of how a gossip service processes queries and update operations is as follows:

1. *Request*: The front end normally sends requests to only a single replica manager at a time. However, 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. Front ends, and thus clients, may be blocked on query operations. 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 $f+1$ replica managers, ensuring that it will be delivered everywhere despite up to f failures.
2. *Update response*: If the request is an update, then the replica manager replies as soon as it has received the update.
3. *Coordination*: The replica manager that receives a request does not process it until it can apply the request according to the required ordering constraints. This may involve receiving updates from other replica managers, in gossip messages. No other coordination between replica managers is involved.
4. *Execution*: The replica manager executes the request.
5. *Query response*: If the request is a query, then the replica manager replies at this point.
6. *Agreement*: The replica managers update one another by exchanging *gossip messages*, which contain the most recent updates they have received. They are said to update one another in a *lazy* 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 that it needs to process a request.

We now describe the gossip system in more detail. We begin by considering the timestamps and data structures that front ends and replica managers maintain in order to maintain update ordering guarantees. Then, in terms of these, we explain how replica managers process queries and updates. Much of the processing of vector timestamps needed to maintain causal updates is similar to the causal multicast algorithm of Section 15.4.3.

The front end's version timestamp • In order to control the ordering of operation processing, each front end keeps a vector timestamp that reflects the version of the latest data values accessed by the front end (and therefore accessed by the client). This timestamp, denoted *prev* in Figure 18.5, contains an entry for every replica manager. The front end sends it in every request message to a replica manager, together with a description of the query or update operation itself. When a replica manager returns a value as a result of a query operation, it supplies a new vector timestamp (*new* in Figure 18.5), since the replicas may have been updated since the last operation. Similarly, an update operation returns a vector timestamp (*Update ID* in Figure 18.5) that is unique to the update. Each returned timestamp is merged with the front end's previous timestamp

Figure 18.6 Front ends propagate their timestamps whenever clients communicate directly

to record the version of the replicated data that has been observed by the client. (See Section 14.4 for a definition of vector timestamp merging.)

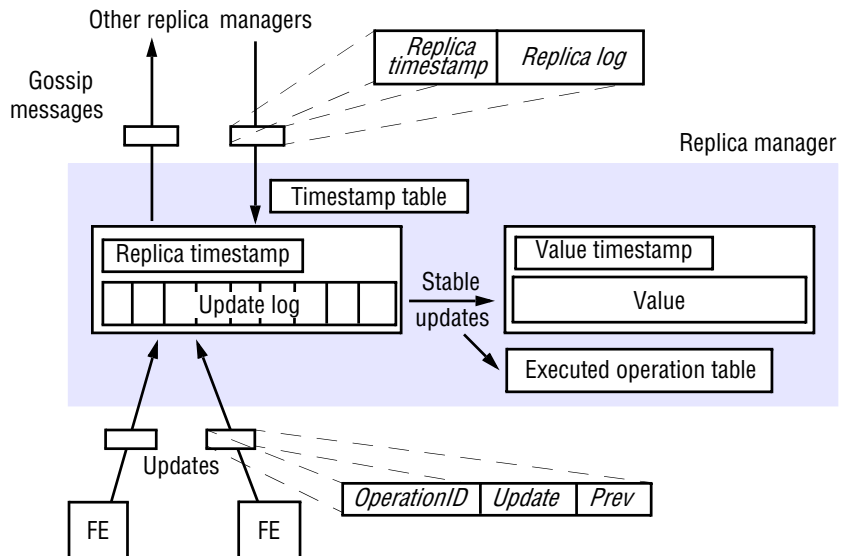
Clients exchange data by accessing the same gossip service and by communicating directly with one another. Since client-to-client communication can also lead to causal relationships between operations applied to the service, it also occurs via the clients' front ends. That way, the front ends can piggyback their vector timestamps on messages to other clients. The recipients merge them with their own timestamps in order that causal relationships can be inferred correctly. The situation is shown in Figure 18.6.

Replica manager state • Regardless of the application, a replica manager contains the following main state components (Figure 18.7):

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 and is thereafter solely the result of applying update operations to that state.

Value timestamp: This is the vector timestamp that represents the updates that are reflected in the value. It contains one entry for every replica manager. It is updated whenever an update operation is applied to the value.

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*. A stable update is one that may be applied consistently with its ordering guarantees (causal, forced or immediate). An update that is not yet stable must be held back and not yet processed. The second reason for keeping an update in the log is that, even though the update has become stable and has been applied to the value, the replica

Figure 18.7 A gossip replica manager, showing its main state components

manager has not received confirmation that this update has been received at all other replica managers. In the meantime, it propagates the update in gossip messages.

Replica timestamp: This vector 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.

Executed operation table: The same update may arrive at a given replica manager from a front end and in gossip messages from other replica managers. To prevent an update being applied twice, the ‘executed operation’ table is kept, containing the unique front-end-supplied identifiers of updates that have been applied to the value. The replica managers check this table before adding an update to the log.

Timestamp table: This table contains a vector timestamp for each other replica manager, filled with timestamps that arrive from them in gossip messages. Replica managers use the table to establish when an update has been applied at all replica managers.

The replica managers are numbered 0, 1, 2, ... : 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 0 would represent the fact that the value there reflects the first two updates accepted from front ends at manager 0, the first four at manager 1 and the first five at manager 2. The following sections look in more detail at how the timestamps are used to enforce the ordering.

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 $q.prev$ 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 $valueTS$ is the replica's value timestamp, then q can be applied to the replica's value if:

$$q.prev \leq valueTS$$

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 wait for the missing updates, which should eventually arrive in gossip messages, or request the updates from the replica managers concerned. For example, if $valueTS$ is (2,5,5) and $q.prev$ is (2,4,6), it can be seen that just one update is missing – from replica manager 2. (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 $valueTS$ to the front end as the timestamp new shown in Figure 18.5. The front end then merges this with its timestamp: $frontEndTS := merge(frontEndTS, new)$. The update at replica manager 1 that the front end has not seen before the query in the example just given ($q.prev$ has 4 where the replica manager has 5) will be reflected in the update to $frontEndTS$ (and potentially in the value returned, depending on the query).

Processing update operations in causal order • A front end submits an update request to one or more replica managers. Each update request u contains a specification of the update (its type and parameters) $u.op$, the front end's timestamp $u.prev$, and a unique identifier that the front end generates, $u.id$. If the front end sends the same request u to several replica managers, it uses the same identifier in u each time – so that it will not be processed as several different but identical requests.

When replica manager i receives an update request from a front end it checks that it has not already processed this request by looking up its operation identifier in the executed operation table and in the records in its log. The replica manager discards the update if it has already seen it; otherwise, it increments the i th element in its replica timestamp by one, to keep count of the number of updates it has received directly from front ends. Then the replica manager assigns to the update request u a unique vector timestamp whose derivation is given shortly, and a record for the update is placed in the replica manager's log. If ts is the unique timestamp that the replica manager assigns to the update, then the update record is constructed and stored in the log as the following tuple:

$$logRecord := \langle i, ts, u.op, u.prev, u.id \rangle$$

Replica manager i derives the timestamp ts from $u.prev$ by replacing $u.prev$'s i th element with the i th element of its replica timestamp (which it has just incremented). This action makes ts unique, thus ensuring that all system components will correctly record whether or not they have observed the update. The remaining elements in ts are copied from $u.prev$, since it is these values sent by the front end that must be used to determine when the update is stable. The replica manager then immediately passes ts

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 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:

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

This condition states that all the updates on which this update depends – that is, all the updates that have been observed by the front end that issued the update – 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 stability condition has been met for an update record r , the replica manager applies the update to the value and updates the value timestamp and the executed operation table, *executed*:

$$\begin{aligned} \text{value} &:= \text{apply}(\text{value}, r.u.op) \\ \text{valueTS} &:= \text{merge}(\text{valueTS}, r.ts) \\ \text{executed} &:= \text{executed} \cup \{r.u.id\} \end{aligned}$$

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 operation identifier is added to the set of identifiers of operations that have been executed – which is used to check for repeated operation requests.

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 Chapter 15 explained, a general method for generating sequence numbers is to use a single sequencer process. But reliance upon a single process is inadequate in the context of a highly available service. The solution is to designate a so-called *primary replica manager* as the sequencer and 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 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. Further details are provided in Ladin *et al.* [1992].

Gossip messages • Replica managers send gossip messages containing information concerning one or more updates so that other replica managers can bring their state up-to-date. A replica manager uses the entries in its timestamp table to estimate which updates any other replica manager has not yet received (it is an estimate because that replica manager may have received more updates by now).

A gossip message m consists of two items sent by the source replica manager: its log, $m.log$, and its replica timestamp, $m.ts$ (see Figure 18.7). The replica manager that receives a gossip message 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 executed operation table when it is known that the updates have been applied everywhere and for which there is no danger of repeats. Clearing redundant entries from the log and from the executed operation table is an important task, since they would otherwise grow without limit.

Merging the log contained in an arrived gossip message with the receiver's log is straightforward. Let $replicaTS$ denote the recipient's replica timestamp. A record r in $m.log$ is added to the receiver's log unless $r.ts \leq replicaTS$ – in which case it is already in the log or it has been applied to the value and then discarded.

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:

$$replicaTS := merge(replicaTS, m.ts)$$

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. The replica manager sorts the updates in the set according to the partial order ' \leq ' between vector timestamps. It then applies the updates in this order, smallest first. That is, each $r \in S$ is applied only when there is no $s \in S$ such that $s.prev < r.prev$.

The replica manager then looks for records in the log that can be discarded. If the gossip message was sent by replica manager j and if $tableTS$ is the table of replica timestamps of the replica managers, then the replica manager sets

$$tableTS[j] := m.ts$$

The replica manager can now discard any record r in the log for an update that has been received everywhere. That is, if c is the replica manager that created the record, then we require for all replica managers i :

$$tableTS[i][c] \geq r.ts[c]$$

The gossip architecture also defines how replica managers can discard entries in the executed operation table. It is important not to discard these entries too early; otherwise, a much-delayed operation could mistakenly be applied twice. Ladin *et al.* [1992] provide details of the scheme. In essence, front ends issue acknowledgements to the

replies to their updates, so replica managers know when a front end will stop sending the update. They assume a maximum update propagation delay from that point.

Update propagation • The gossip architecture does not specify when replica managers exchange gossip messages, or how a replica manager decides where to send its gossip messages. A robust update-propagation strategy is needed if all replica managers are to receive all updates in an acceptable time.

The time it takes for all replica managers to receive a given update depends upon three factors:

- the frequency and duration of network partitions;
- the frequency with which replica managers send gossip messages;
- the policy for choosing a partner with which to exchange gossip.

The first factor is beyond the system's control, although users can to some extent determine how often they work disconnectedly.

The desired gossip-exchange frequency may be tuned to the application. Consider a bulletin board system shared between several sites. It seems unnecessary for every item to be dispatched immediately to all sites. But what if gossip is only exchanged after long periods, say once a day? If only causal updates are used, then it is quite possible for clients at each site to have their own consistent debates over the same bulletin board, oblivious to the discussions at the other sites. Then at, say, midnight, all the debates will be merged; but debates on the same topic are likely to be incongruous, when it would have been preferable for them to take account of one another. A gossip-exchange period of minutes or hours seems more appropriate in this case.

There are several types of partner-selection policy. Golding and Long [1993] consider *random*, *deterministic* and *topological* policies for their 'timestamped anti-entropy protocol', which uses a gossip-style update propagation scheme.

Random policies choose a partner randomly but with weighted probabilities so as to favour some partners over others – for example, near partners over far partners. Golding and Long found that such a policy works surprisingly well under simulations. Deterministic policies utilize a simple function of the replica manager's state to make the choice of partner. For example, a replica manager could examine its timestamp table and choose the replica manager that appears to be the furthest behind in the updates it has received.

Topological policies arrange the replica managers into a fixed graph. One possibility is a mesh: replica managers send gossip messages to the four replica managers to which it is connected. Another is to arrange the replica managers in a circle, with each passing on gossip only to its neighbour (in the clockwise direction, say), so that updates from any replica manager eventually traverse the circle. There are many other possible topologies, including trees.

Different partner-selection policies such as these trade off the amount of communication against higher transmission latencies and the possibility that a single failure will affect other replica managers. The choice depends in practice on the relative importance of these factors. For example, the circle topology produces relatively little communication but is subject to high transmission latencies since gossip generally has to traverse several replica managers. Moreover, if one replica manager fails then the circle cannot function and needs to be reconfigured. By contrast, the random selection

policy is not susceptible to failures but may produce more variable update propagation times.

Discussion of the gossip architecture • The gossip architecture is aimed at achieving high availability for services. In its favour, this architecture ensures that clients can continue to obtain a service even when they are partitioned from the rest of the network, as long as at least one replica manager continues to function in their partition. But this type of availability is achieved at the expense of enforcing only relaxed consistency guarantees. For objects such as bank accounts, where sequential consistency is required, a gossip architecture can do no better than the fault-tolerant systems studied in Section 18.3 and supply the service only in a majority partition.

Its lazy approach to update propagation makes a gossip-based system inappropriate for updating replicas in near-real time, such as when users take part in a ‘real-time’ conference and update a shared document. A multicast-based system would be more appropriate for that case.

The scalability of a gossip system is another issue. As the number of replica managers grows, so does the number of gossip messages that have to be transmitted and the size of the timestamps used. If a client makes a query, then this normally takes two messages (between front end and replica manager). If a client makes a causal update operation and if each of the R replica managers normally collects G updates into a gossip message, then the number of messages exchanged is $2 + (R - 1)/G$. The first term represents communication between the front end and replica manager and the second is the update’s share of a gossip message sent to the other replica managers. Increasing G improves the number of messages but worsens the delivery latencies, because the replica manager waits for more updates to arrive before propagating them.

One approach to making gossip-based services scalable is to make most of the replicas read-only. In other words, these replicas are updated by gossip messages but do not receive updates directly from front ends. This arrangement is potentially useful where the *update/query* ratio is small. Read-only replicas can be situated close to client groups and updates can be serviced by relatively few central replica managers. Gossip traffic is reduced since read-only replicas have no gossip to propagate, and vector timestamps need only contain entries for the updateable replicas.

18.4.2 Bayou and the operational transformation approach

The Bayou system [Terry *et al.* 1995, Petersen *et al.* 1997] provides data replication for high availability with weaker guarantees than sequential consistency, like the gossip architecture and the timestamped anti-entropy protocol. As in those systems, Bayou replica managers cope with variable connectivity by exchanging updates in pairs, in what the designers also call an *anti-entropy protocol*. But Bayou adopts a markedly different approach in that it enables domain-specific conflict detection and conflict resolution to take place.

Consider the user who needs to update a diary while working disconnectedly. If strict consistency is required, in the gossip architecture updates would have to be performed using a forced (totally ordered) operation. But then only users in a majority partition could update the diary. The users’ access to the diary might thus be limited, regardless of whether they in fact are making updates that would break the diary’s

integrity. Users who want to fill in a non-conflicting appointment are treated the same as users who might have unwittingly double-booked a time slot.

In Bayou, by contrast, the users on the train and at work may make any updates they like. All the updates are applied and recorded at whatever replica manager they reach. When updates received at any two replica managers are merged during an anti-entropy exchange, however, the replica managers detect and resolve conflicts. Any domain-specific criterion for resolving conflicts between operations may be applied. For example, if an executive working offsite, and her secretary have added appointments in the same time slot, then a Bayou system detects this after the executive has reconnected their laptop. Moreover, it resolves the conflict according to a domain-specific policy. In this case, it could, for example, confirm the executive's appointment and remove the secretary's booking in the slot. Such an effect, in which one or more of a set of conflicting operations are undone or altered in order to resolve them, is called an *operational transformation*.

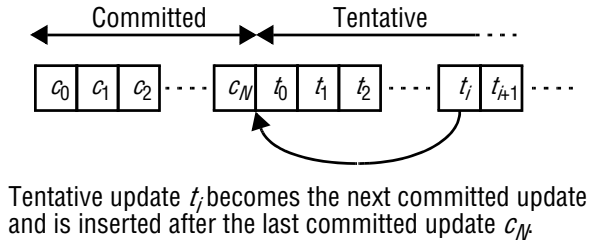
The state that Bayou replicates is held in the form of a database, supporting queries and updates (that may insert, modify or delete items in the database). Although we shall not concentrate on this aspect here, a Bayou update is a special case of a transaction. It consists of a single operation, an invocation of a 'stored procedure', that affects several objects within each replica manager but is carried out with the ACID guarantees. Bayou may undo and redo updates to the database as execution proceeds.

The Bayou guarantee is that, eventually, every replica manager receives the same set of updates and it eventually applies those updates in such a way that the replica managers' databases are identical. In practice, there may be a continuous stream of updates and the databases may never become identical; but they would become identical if the updates ceased.

Committed and tentative updates • Updates are marked as *tentative* when they are first applied to a database. Bayou arranges that tentative updates are eventually placed in a canonical order and marked as *committed*. While updates are tentative, the system may undo and reapply them as necessary to produce a consistent state. Once committed, they remain applied in their allotted order. In practice, the committed order can be achieved by designating some replica manager as the *primary* replica manager. In the usual way, this decides the committed order as that in which it receives the tentative updates and it propagates that ordering information to other replica managers. For the primary, users can choose, for example, a fast machine that is usually available, or the replica manager on an executive's laptop, if that user's updates take priority.

At any one time, the state of a database replica derives from a (possibly empty) sequence of committed updates followed by a (possibly empty) sequence of tentative updates. If the next committed update arrives, or if one of the tentative updates that has been applied becomes the next committed update, then a reordering of the updates must take place. In Figure 18.8, t_i has become committed. All tentative updates after c_N need to be undone; t_i is then applied after c_N and t_0 to t_{i-1} and t_{i+1} etc. reapplied after t_i .

Dependency checks and merge procedures • An update may conflict with some other operation that has already been applied. Because of this possibility, every Bayou update contains a *dependency check* and a *merge procedure* in addition to the operation's specification (the operation type and parameters). All these components of an update are domain-specific.

Figure 18.8 Committed and tentative updates in Bayou

A replica manager calls the dependency check procedure before applying the operation. It checks whether a conflict would occur if the update was applied and it may examine any part of the database to do that. For example, consider the case of booking an appointment in a diary. The dependency check could, most simply, test for a *write-write* conflict: that is, whether another client has filled the required slot. But the dependency check could also test for a *read-write* conflict. For example, it could test that the desired slot is empty and that the number of appointments on that day is fewer than six.

If the dependency check indicates a conflict, then Bayou invokes the operation's merge procedure. That procedure alters the operation that will be applied so that it achieves something similar to the intended effect but avoids a conflict. For example, in the case of the diary the merge procedure could choose another slot at a nearby time instead or, as we mentioned above, it could use a simple priority scheme to decide which appointment is more important and impose that one. The merge procedure may fail to find a suitable alteration of the operation, in which case the system indicates an error. The effect of a merge procedure is deterministic, however – Bayou replica managers are state machines.

Discussion • Bayou differs from the other replication schemes that we have considered in that it makes replication non-transparent to the application. It exploits knowledge of the application's semantics in order to increase the availability of data while maintaining a replicated state that is what we might call *eventually sequentially consistent*.

This approach has a few disadvantages. The first is the increased complexity for the application programmer, who must supply dependency checks and merge procedures. Both may be complex to produce, given the potentially large number of possible conflicts that need to be detected and resolved. The second disadvantage is the increased complexity for the user. Users are expected to deal not only with data that are read while they are still tentative but also with the fact that the operations they specify may turn out to have been altered later. For example, the user may book a slot in a diary, only to find later that the booking has 'jumped' to a nearby slot. It is very important that the user be given a clear indication of which data are tentative and which are committed.

The operational transformation approach used by Bayou appears particularly in systems to support computer-supported cooperative working (CSCW), where conflicting updates between geographically separated users may occur [Kindberg *et al.* 1996, Sun and Ellis 1998]. The approach is limited, in practice, to applications where conflicts are relatively rare, where the underlying data semantics are relatively simple; and where users can cope with tentative information.

18.4.3 The Coda file system

The Coda file system is a descendent of AFS (see Section 12.4) that aims to address several requirements that AFS does not meet – particularly the requirement to provide high availability despite disconnected operation. It was developed in a research project undertaken by Satyanarayanan and his coworkers at Carnegie-Mellon University [Satyanarayanan *et al.* 1990, Kistler and Satyanarayanan 1992]. The design requirements for Coda were derived from experience with AFS at CMU and elsewhere involving its use in large-scale distributed systems on both local and wide area communication networks.

While the performance and ease of administration of AFS were found to be satisfactory under the conditions of use at CMU, it was felt that the limited form of replication (restricted to read-only volumes) offered by AFS was bound to become a limiting factor at some scale, especially for accessing widely shared files such as electronic bulletin boards and other system-wide databases.

In addition, there was room for improvement in the availability of the service offered by AFS. The most common difficulties experienced by users of AFS arose from the failure (or scheduled interruption) of servers and network components. The scale of the system at CMU was such that a few service failures occurred every day and they could seriously inconvenience many users for periods ranging from a few minutes to many hours.

Finally, a mode of computer use was emerging that AFS did not cater for – the mobile use of portable computers. This led to a requirement to make all of the files needed for a user to continue their work available while disconnected from the network without resorting to manual methods for managing the locations of files.

Coda aims to meet all three of these requirements under the general heading of *constant data availability*. The aim was to provide users with the benefits of a shared file repository but to allow them to rely entirely on local resources when the repository is partially or totally inaccessible. In addition to these aims, Coda retains the original goals of AFS with regard to scalability and the emulation of UNIX file semantics.

In contrast to AFS, where read-write volumes are stored on just one server, the design of Coda relies on the replication of file volumes to achieve a higher throughput of file access operations and a greater degree of fault tolerance. In addition, Coda relies on an extension of the mechanism used in AFS for caching copies of files at client computers to enable those computers to operate when they are not connected to the network.

Coda is like Bayou in so far as it follows an optimistic strategy. That is, it allows clients to update data while the system is partitioned, on the basis that conflicts are relatively unlikely and that they can be fixed if they do occur. Like Bayou, it detects conflicts, but unlike Bayou it performs these checks without regard to the semantics of the data stored in files. Also unlike Bayou, it provides only limited system support for resolving conflicting replicas.

The Coda architecture • Coda runs what it calls ‘Venus’ processes at the client computers and ‘Vice’ processes at file server computers, adopting the AFS terminology. The Vice processes are what we have called replica managers. The Venus processes are a hybrid of front ends and replica managers. They play the front end’s role of hiding the service implementation from local client processes, but since they manage a local cache

of files they are also replica managers (although of a different type to the Vice processes).

The set of servers holding replicas of a file volume is known as the *volume storage group* (VSG). At any instant, a client wishing to open a file in such a volume can access some subset of the VSG, known as the *available volume storage group* (AVSG). The membership of the AVSG varies as servers become accessible or are made inaccessible by network or server failures.

Normally, Coda file access proceeds in a similar manner to AFS, with cached copies of files being supplied to the client computers by any one of the servers in the current AVSG. As in AFS, clients are notified of changes via a *callback promise* mechanism, but this now depends on an additional mechanism for the distribution of updates to each replica. On *close*, copies of modified files are broadcast in parallel to all of the servers in the AVSG.

In Coda, disconnected operation is said to occur when the AVSG is empty. This may be due to network or server failures, or it may be a consequence of the deliberate disconnection of the client computer, as in the case of a laptop. Effective disconnected operation relies on the presence in the client computer's cache of *all* of the files that are required for the user's work to proceed. To achieve this, the user must cooperate with Coda to generate a list of files that should be cached. A tool is provided that records a historical list of file usage while connected, and this serves as a basis for predicting usage while disconnected.

It is a principle of the design of Coda that the copies of files residing on servers are more reliable than those residing in the caches of client computers. Although it might be possible logically to construct a file system that relies entirely on cached copies of files in client computers, it is unlikely that a satisfactory quality of service would be achieved. The Coda servers exist to provide the necessary quality of service. The copies of files residing in client computer caches are regarded as useful only as long as their currency can be periodically revalidated against the copies residing in servers. In the case of disconnected operation, revalidation occurs when disconnected operation ceases and the cached files are reintegrated with those in the servers. In the worst case, this may require some manual intervention to resolve inconsistencies or conflicts.

The replication strategy • Coda's replication strategy is optimistic – it allows modification of files to proceed when the network is partitioned or during disconnected operation. It relies on the attachment to each version of a file of a *Coda version vector* (CVV). A CVV is a vector timestamp with one element for each server in the relevant VSG. Each element of the CVV is an estimate of the number of modifications performed on the version of the file that is held at the corresponding server. The purpose of the CVVs is to provide sufficient information about the update history of each file replica to enable potential conflicts to be detected and submitted for manual intervention and for stale replicas to be updated automatically.

If the CVV at one of the sites is greater than or equal to all the corresponding CVVs at the other sites (Section 14.4 defines the meaning of $v_1 \geq v_2$ for vector timestamps v_1 and v_2), then there is no conflict. Older replicas (with strictly smaller timestamps) include all the updates in a newer replica and they can automatically be brought up-to-date with it.

When this is not the case – that is, when neither $v_1 \geq v_2$ nor $v_2 \geq v_1$ holds for two CVVs – then there is a conflict: each replica reflects at least one update that the other does not reflect. Coda does not, in general, resolve conflicts automatically. The file is marked as ‘inoperable’ and the owner of the file is informed of the conflict.

When a modified file is closed, each site in the current AVSG is sent an update message by the Venus process at the client, containing the current CVV and the new contents for the file. The Vice process at each site checks the CVV and, if it is greater than the one currently held, stores the new contents for the file and returns a positive acknowledgement. The Venus process then computes a new CVV with modification counts increased for the servers that responded positively to the update message and distributes the new CVV to the members of the AVSG.

Since the message is sent only to the members of the AVSG and not the VSG, servers that are not in the current AVSG do not receive the new CVV. Any CVV will therefore always contain an accurate modification count for the local server, but the counts for non-local servers will in general be lower bounds, since they will be updated only when the server receives an update message.

The box below contains an example illustrating the use of CVVs to manage the updating of a file replicated at three sites. Further details on the use of CVVs for the

Example: Consider a sequence of modifications to a file F in a volume that is replicated at three servers, S_1 , S_2 and S_3 . The VSG for F is $\{S_1, S_2, S_3\}$. F is modified at about the same time by two clients, C_1 and C_2 . Because of a network fault, C_1 can access only S_1 and S_2 (C_1 ’s AVSG is $\{S_1, S_2\}$) and C_2 can access only S_3 (C_2 ’s AVSG is $\{S_3\}$).

1. Initially, the CVVs for F at all three servers are the same – say, $[1,1,1]$.
2. C_1 runs a process that opens F , modifies it and then closes it. The Venus process at C_2 broadcasts an update message to its AVSG, $\{S_1, S_2\}$, finally resulting in new versions of F and a CVV $[2,2,1]$ at S_1 and S_2 but no change at S_3 .
3. Meanwhile, C_2 runs two processes, each of which opens F , modifies it and then closes it. The Venus process at C_2 broadcasts an update message to its AVSG, $\{S_3\}$, after each modification, finally resulting in a new version of F and a CVV $[1,1,3]$ at S_3 .
4. At some later time, the network fault is repaired, and C_2 makes a routine check to see whether the inaccessible members of the VSG have become accessible (the process by which such checks are made is described later) and discovers that S_1 and S_2 are now accessible. It modifies its AVSG to $\{S_1, S_2, S_3\}$ for the volume containing F and requests the CVVs for F from all members of the new AVSG. When they arrive, C_2 discovers that S_1 and S_2 each have CVVs $[2,2,1]$ whereas S_3 has $[1,1,3]$. This represents a *conflict* requiring manual intervention to bring F up-to-date in a manner that minimizes the loss of update information.

On the other hand, consider a similar but simpler scenario that follows the same sequence of events as the one above, but omitting item (3), so that F is not modified by C_2 . The CVV at S_3 therefore remains unchanged as $[1,1,1]$, and when the network fault is repaired, C_2 discovers that the CVVs at S_1 and S_2 ($[2,2,1]$) *dominate* that at S_3 . The version of the file at S_1 or S_2 should replace that at S_3 .

management of updates can be found in Satyanarayanan *et al.* [1990]. CVVs are based on the replication techniques used in the Locus system [Popek and Walker 1985].

In normal operation, the behaviour of Coda appears similar to that of AFS. A cache miss is transparent to users and only imposes a performance penalty. The advantages deriving from the replication of some or all file volumes on multiple servers are:

- The files in a replicated volume remain accessible to any client that can access at least one of the replicas.
- The performance of the system can be improved by sharing some of the load of servicing client requests on a replicated volume between all of the servers that hold replicas.

In disconnected operation (when none of the servers for a volume can be accessed by the client) a cache miss prevents further progress and the computation is suspended until the connection is resumed or the user aborts the process. It is therefore important to load the cache before disconnected operation commences so that cache misses can be avoided.

In summary, compared with AFS, Coda enhances availability both by the replication of files across servers and by the ability of clients to operate entirely out of their caches. Both methods depend upon the use of an optimistic strategy for the detection of update conflicts in the presence of network partitions. The mechanisms are complementary and independent of each other. For example, a user can exploit the benefits of disconnected operation even though the required file volumes are stored on a single server.

Update semantics • The currency guarantees offered by Coda when a client opens a file are weaker than for AFS, reflecting the optimistic update strategy. The single server S referred to in the currency guarantees for AFS is replaced by a set of servers \bar{S} (the file's VSG) and the client C can access a subset of servers \bar{s} (the AVSG for the file seen by C).

Informally, the guarantee offered by a successful *open* in Coda is that it provides the most recent copy of F from the current AVSG, and if no server is accessible, a locally cached copy of F is used if one is available. A successful *close* guarantees that the file has been propagated to the currently accessible set of servers, or, if no server is available, that the file has been marked for propagation at the earliest opportunity.

A more precise definition of these guarantees, taking into account the effect of lost callbacks, can be made using an extension of the notation used for AFS. In each definition except the last there are two cases: the first, beginning $\bar{s} \neq \emptyset$, refers to all situations in which the AVSG is not empty, and the second deals with disconnected operation:

- after a successful *open*: $(\bar{s} \neq \emptyset \text{ and } (\text{latest}(F, \bar{s}, 0)$
 $\text{or } (\text{latest}(F, \bar{s}, T) \text{ and } \text{lostCallback}(\bar{s}, T) \text{ and } \text{inCache}(F))))$
 $\text{or } (\bar{s} = \emptyset \text{ and } \text{inCache}(F))$
- after a failed *open*: $(\bar{s} \neq \emptyset \text{ and } \text{conflict}(F, \bar{s}))$
 $\text{or } (\bar{s} = \emptyset \text{ and } \neg \text{inCache}(F))$

after a successful *close*: $(\bar{s} \neq \emptyset \text{ and } \text{updated}(F, \bar{s}))$
 or $(\bar{s} = \emptyset)$

after a failed *close*: $\bar{s} \neq \emptyset \text{ and } \text{conflict}(F, \bar{s})$

This model assumes a synchronous system: T is the longest time for which a client can remain unaware of an update elsewhere to a file that is in its cache; $\text{latest}(F, \bar{s}, T)$ denotes the fact that the current value of F at C was the latest across all the servers in \bar{s} at some instant in the last T seconds and that there were no conflicts among the copies of F at that instant; $\text{lostCallback}(\bar{s}, T)$ means that a callback was sent by some member of \bar{s} in the last T seconds and was not received at C ; and $\text{conflict}(F, \bar{s})$ means that the values of F at some servers in \bar{s} are currently in conflict.

Accessing replicas • The strategy used on *open* and *close* to access the replicas of a file is a variant of the *read-one/write-all* approach described in Section 18.5. On *open*, if a copy of the file is not present in the local cache the client identifies a preferred server from the AVSG for the file. The preferred server may be chosen at random, or on the basis of performance criteria such as physical proximity or server load. The client requests a copy of the file attributes and contents from the preferred server, and on receiving it, it checks with all the other members of the AVSG to verify that the copy is the latest available version. If not, a member of the AVSG with the latest version is made the preferred site, the file contents are refetched and the members of the AVSG are notified that some members have stale replicas. When the fetch has been completed, a callback promise is established at the preferred server.

When a file is closed at a client after modification, its contents and attributes are transmitted in parallel to all the members of the AVSG using a multicast remote procedure calling protocol. This maximizes the probability that every replication site for a file has the current version at all times. It doesn't guarantee it, because the AVSG does not necessarily include all the members of the VSG. It minimizes the server load by giving clients the responsibility for propagating changes to the replication sites in the normal case (servers are involved only when a stale replica is discovered on *open*).

Since maintaining callback state in all the members of an AVSG would be expensive, the callback promise is maintained only at the preferred server. But this introduces a new problem: the preferred server for one client need not be in the AVSG of another client. If this is the case, an update by the second client will not cause a callback to the first client. The solution adopted to this problem is discussed in the next subsection.

Cache coherence • The Coda currency guarantees stated above mean that the Venus process at each client must detect the following events within T seconds of their occurrence:

- enlargement of an AVSG (due to the accessibility of a previously inaccessible server);
- shrinking of an AVSG (due to a server becoming inaccessible);
- a lost callback event.

To achieve this, Venus sends a probe message to all the servers in VSGs of the files that it has in its cache every T seconds. Responses will be received only from accessible

servers. If Venus receives a response from a previously inaccessible server it enlarges the corresponding AVSG and drops the callback promises on any files that it holds from the relevant volume. This is done because the cached copy may no longer be the latest version available in the new AVSG.

If it fails to receive a response from a previously accessible server Venus shrinks the corresponding AVSG. No callback changes are required unless the shrinkage is caused by the loss of a preferred server, in which case all callback promises from that server must be dropped. If a response indicates that a callback message was sent but not received, the callback promise on the corresponding file is dropped.

We are now left with the problem, mentioned above, of updates that are missed by a server because it is not in the AVSG of a different client that performs an update. To deal with this case, Venus is sent a *volume version vector* (*volume CVV*) in response to each probe message. The volume CVV contains a summary of the CVVs for all of the files in the volume. If Venus detects any mismatch between the volume CVVs then some members of the AVSG must have some file versions that are not up-to-date. Although the outdated files may not be the ones that are in its local cache, Venus makes a pessimistic assumption and drops the callback promises on all of the files that it holds from the relevant volume.

Note that Venus only probes servers in the VSGs of files for which it holds cached copies and that a single probe message serves to update the AVSGs and check the callbacks for all of the files in a volume. This, combined with a relatively large value for T (on the order of 10 minutes in the experimental implementation), means that the probes are not an obstacle to the scalability of Coda to large numbers of servers and wide area networks.

Disconnected operation • During brief disconnections, such as those that may occur because of unexpected service interruptions, the least recently used cache replacement policy normally adopted by Venus may be sufficient to avoid cache misses on the disconnected volumes. But it is unlikely that a client could operate in disconnected mode for extended periods without generating references to files or directories that are not in the cache unless a different policy is adopted.

Coda therefore allows users to specify a prioritized list of files and directories that Venus should strive to retain in the cache. Objects at the highest level are identified as *sticky* and these must be retained in the cache at all times. If the local disk is large enough to accommodate all of them, the user is assured that they will remain accessible. Since it is often difficult to know exactly what file accesses are generated by any sequence of user actions, a tool is provided that enables the user to bracket a sequence of actions; Venus notes the file references generated by the sequence and flags them with a given priority.

When disconnected operation ends, a process of *reintegration* begins. For each cached file or directory that has been modified, created or deleted during disconnected operation, Venus executes a sequence of update operations to make the AVSG replicas identical to the cached copy. Reintegration proceeds top-down from the root of each cached volume.

Conflicts may be detected during reintegration due to updates to AVSG replicas by other clients. When this occurs, the cached copy is stored in a temporary location on the server, and the user that initiated the reintegration is informed. This approach is

based on the design philosophy adopted in Coda, which assigns priority to server-based replicas over cached copies. The temporary copies are stored in a *covolume*, which is associated with each volume on a server. Covolumes resemble the *lost+found* directories found in conventional UNIX systems. They mirror just those parts of the file directory structure needed to hold the temporary data. Little additional storage is required, because the covolumes are almost empty.

Performance • Satyanarayanan *et al.* [1990] compared the performance of Coda with AFS under benchmark loads designed to simulate user populations ranging from 5 to 50 typical AFS users.

With no replication, there is no significant difference between the performance of AFS and that of Coda. With threefold replication, the time taken for Coda to perform a benchmark load equivalent to 5 typical users exceeds that of AFS without replication by only 5%. However, with threefold replication and a load equivalent to 50 users, the time required to complete the benchmark is increased by 70%, whereas that for AFS without replication is increased by only 16%. This difference is attributed only in part to the overheads associated with replication – differences in the tuning of the implementation are said to account for part of the difference in performance.

Discussion • We pointed out above that Coda is similar to Bayou in that it also employs an optimistic approach to achieving high availability (although they differ in several other ways, not least because one manages files and the other databases). We also described how Coda uses CVVs to check for conflicts, without regard to the semantics of the data stored in files. The approach can detect potential write-write conflicts but not read-write conflicts. These are ‘potential’ write-write conflicts because at the level of the application semantics there may be no actual conflict: clients may have updated different objects in the file compatibly such that a simple automatic merge would be possible.

Coda’s overall approach of semantics-free conflict detection and manual resolution is sensible in many cases, especially in applications that require human judgement or in systems with no knowledge of the data’s semantics.

Directories are a special case in Coda. Automatically maintaining the integrity of these key objects through conflict resolution is sometimes possible, since their semantics are relatively simple: the only changes that can be made to directories are the insertion or deletion of directory entries. Coda incorporates its own method for resolving directories. It has the same effect as Bayou’s approach of operational transformation, but Coda merges the state of conflicting directories directly, since it has no record of the operations that clients performed.

Replication in Dynamo • Section 16.7 introduced Dynamo, the storage service that Amazon uses for applications such as shopping carts that require only key/value access. In Dynamo [DeCandia *et al.* 2007], data is partitioned and replicated; all updates reach all replicas eventually.

Like Bayou and Coda, Dynamo uses optimistic replication techniques; changes are allowed to propagate to replicas in the background and concurrent, disconnected work is tolerated. This approach can lead to conflicting changes that must be detected and resolved.

In Dynamo, writes are always accepted and written as immutable versions, so that customers can always add and remove items to and from their shopping carts.

Vector timestamps are used to determine causal ordering between different versions of the same object. The timestamps are compared as described in Section 14.4. When the vector timestamp of one version is less than that of another, the earlier version is discarded. Otherwise, the two versions conflict and must be resolved. Both versions of the data are stored and then given to a client as the results of a read operation.

This client is responsible for resolving the conflict. Dynamo provides both the application-level approach of Bayou and the system-level approach of Coda. The former approach is used for shopping carts, where all the *add item* operations in conflicting versions are merged and sometimes a deleted item can reappear. When application semantics cannot be used, Dynamo uses simple timestamp-based resolution – the object with the largest physical timestamp value is chosen as the correct version.

18.5 Transactions with replicated data

So far in this chapter we have considered systems in which clients request single operations at a time on replicated sets of objects. Chapters 16 and 17 explained that transactions are *sequences* of one or more operations, applied in such a way as to enforce the ACID properties. As with the systems in Section 18.4, objects in transactional systems may be replicated to increase both availability and performance.

From a client's viewpoint, a transaction on replicated objects should appear the same as one with non-replicated objects. In a non-replicated system, transactions appear to be performed one at a time in some order. This is achieved by ensuring a serially equivalent interleaving of clients' transactions. The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at a time on a single set of objects. This property is called *one-copy serializability*. It is similar to, but not to be confused with, sequential consistency. Sequential consistency considers valid executions without any notion of aggregating the client operations into transactions.

Each replica manager provides concurrency control and recovery of its own objects. In this section, we assume that two-phase locking is used for concurrency control.

Recovery is complicated by the fact that a failed replica manager is a member of a collection and that the other members continue to provide a service during the time that it is unavailable. When a replica manager recovers from a failure, it uses information obtained from the other replica managers to restore its objects to their current values, taking into account all the changes that have occurred during the time it was unavailable.

This section first introduces the architecture for transactions with replicated data. Architectural questions are whether a client request can be addressed to any of the replica managers; how many replica managers are required for the successful completion of an operation; whether the replica manager contacted by a client can defer the forwarding of requests until a transaction is committed; and how to carry out a two-phase commit protocol.

The implementation of one-copy serializability is illustrated by *read-one/write-all* – a simple replication scheme in which *read* operations are performed by a single replica manager and *write* operations are performed by all of them.

The section then discusses the problems of implementing replication schemes in the presence of server crashes and recovery. It introduces available copies replication, a variant of the read-one/write-all replication scheme in which *read* operations are performed by any single replica manager and *write* operations are performed by all of those that are available.

Finally, the section presents three replication schemes that work correctly when the collection of replica managers is divided into subgroups by a network partition:

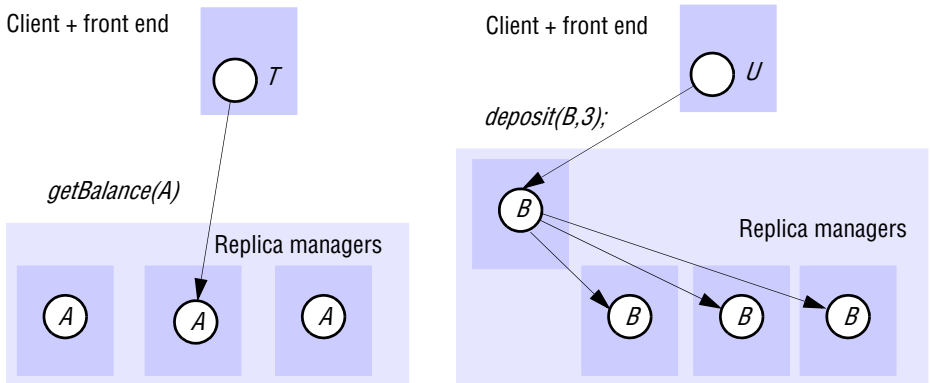
- *Available copies with validation*: Available copies replication is applied in each partition. When a partition is repaired, a validation procedure is applied and any inconsistencies are dealt with.
- *Quorum consensus*: A subgroup must have a quorum (meaning that it has sufficient members) in order to be allowed to continue providing a service in the presence of a partition. When a partition is repaired (and when a replica manager restarts after a failure), replica managers get their objects up-to-date by means of recovery procedures.
- *Virtual partition*: A combination of quorum consensus and available copies. If a virtual partition has a quorum, it can use available copies replication.

18.5.1 Architectures for replicated transactions

As with the range of systems we have already considered in previous sections, a front end may either multicast client requests to groups of replica managers or send each request to a single replica manager, which is then responsible for processing the request and responding to the client. Wiesmann *et al.* [2000] and Schiper and Raynal [1996] consider the case of multicast requests, and we shall not deal with it here. Henceforth, we assume that a front end sends client requests to one of the group of replica managers of a logical object. In the *primary copy* approach, all front ends communicate with a distinguished ‘primary’ replica manager to perform an operation, and that replica manager keeps the backups up-to-date. Alternatively, front ends may communicate with any replica manager to perform an operation – but coordination between the replica managers is consequently more complex.

The replica manager that receives a request to perform an operation on a particular object is responsible for getting the cooperation of the other replica managers in the group that have copies of that object. Different replication schemes have different rules as to how many of the replica managers in a group are required for the successful completion of an operation. For example, in the read-one/write-all scheme, a *read* request can be performed by a single replica manager, whereas a *write* request must be performed by all the replica managers in the group, as shown in Figure 18.9 (there can be different numbers of replicas of the various objects). Quorum consensus schemes are designed to reduce the number of replica managers that must perform update operations, but at the expense of increasing the number of replica managers required to perform *read-only* operations.

Another issue is whether the replica manager contacted by a front end should defer the forwarding of update requests to other replica managers in the group until after a transaction commits – the so-called *lazy* approach to update propagation – or,

Figure 18.9 Transactions on replicated data

conversely, whether replica managers should forward each update request to all the necessary replica managers within the transaction and before it commits – the *eager* approach. The lazy approach is an attractive alternative because it reduces the amount of communication between the replica managers that takes place before responding to the updating client. However, concurrency control must also be considered. The lazy approach is sometimes used in primary copy replication (see below), where a single primary replica manager serializes the transactions. But if several different transactions may attempt to access the same objects at different replica managers in a group, to ensure that the transactions are correctly serialized at all the replica managers in the group, each replica manager needs to know about the requests performed by the others. The eager approach is the only one available in that case.

The two-phase commit protocol • The two-phase commit protocol becomes a two-level nested two-phase commit protocol. As before, the coordinator of a transaction communicates with the workers. But if either the coordinator or a worker is a replica manager, it will communicate with the other replica managers to which it passed requests during the transaction.

That is, in the first phase, the coordinator sends *canCommit?* requests to the workers, which pass them on to the other replica managers and collect their replies before replying to the coordinator. In the second phase, the coordinator sends the *doCommit* or *doAbort* request, which is passed on to the members of the groups of replica managers.

Primary copy replication • Primary copy replication may be used in the context of transactions. In this scheme, all client requests (whether or not they are read-only) are directed to a single primary replica manager (see Figure 18.3). For primary copy replication, concurrency control is applied at the primary. To commit a transaction, the primary communicates with the backup replica managers and then, in the eager approach, replies to the client. This form of replication allows a backup replica manager to take over consistently if the primary fails. In the lazy alternative, the primary responds to front ends before it has updated its backups. In that case, a backup that replaces a failed front end will not necessarily have the latest state of the database.

Read-one/write-all • We use this simple replication scheme to illustrate how two-phase locking at each replica manager can be used to achieve one-copy serializability, where front ends may communicate with any replica manager. Every *write* operation must be performed at all of the replica managers, each of which sets a write lock on the object affected by the operation. Each *read* operation is performed by a single replica manager, which sets a read lock on the object affected by the operation.

Consider pairs of operations of different transactions on the same object: any pair of *write* operations will require conflicting locks at all of the replica managers, while a *read* operation and a *write* operation will require conflicting locks at a single replica manager. Thus one-copy serializability is achieved.

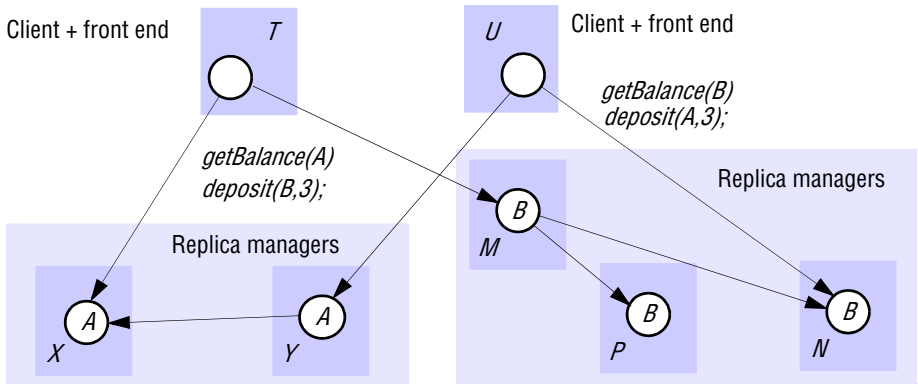
18.5.2 Available copies replication

Simple read-one/write-all replication is not a realistic scheme, because it cannot be carried out if some of the replica managers are unavailable, either because they have crashed or because of a communication failure. The available copies scheme is designed to allow for some replica managers being temporarily unavailable. The strategy is that a client's *read* request on a logical object may be performed by any available replica manager but that a client's update request must be performed by all available replica managers in the group with copies of the object. The idea of the 'available members of a group of replica managers' is similar to Coda's available volume storage group, described in Section 18.4.3.

In the normal case, client requests are received and performed by a functioning replica manager. *read* requests can be performed by the replica manager that receives them. *write* requests are performed by the receiving replica manager and all the other available replica managers in the group. For example, in Figure 18.10, the *getBalance* operation of transaction *T* is performed by *X*, whereas its *deposit* operation is performed by *M*, *N* and *P*. Concurrency control at each replica manager affects the operations performed locally. For example, at *X*, transaction *T* has read *A* and therefore transaction *U* is not allowed to update *A* with the *deposit* operation until transaction *T* has completed. So long as the set of available replica managers does not change, local concurrency control achieves one-copy serializability in the same way as in read-one/write-all replication. Unfortunately, this is not the case if a replica manager fails or recovers during the progress of the conflicting transactions.

Replica manager failure • We assume that replica managers fail benignly by crashing. However, a crashed replica manager is replaced by a new process, which recovers the committed state of the objects from a recovery file. Front ends use timeouts to decide that a replica manager is not currently available. When a client makes a request to a replica manager that has crashed, the front end times out and retries the request at another replica manager in the group. If the request is received by a replica manager at which the object is out of date because the replica manager has not completely recovered from failure, the replica manager rejects the request and the front end retries the request at another replica manager in the group.

One-copy serializability requires that crashes and recoveries be serialized with respect to transactions. According to whether it can access an object or not, a transaction observes that a failure occurs after it has finished or before it started. One-copy

Figure 18.10 Available copies

serializability is not achieved when different transactions make conflicting failure observations.

Consider the case in Figure 18.10 where the replica manager *X* fails just after *T* has performed *getBalance* and replica manager *N* fails just after *U* has performed *getBalance*. Assume that both of these replica managers fail before *T* and *U* have performed their *deposit* operations. This implies that *T*'s *deposit* will be performed at replica managers *M* and *P* and *U*'s *deposit* will be performed at replica manager *Y*. Unfortunately, the concurrency control on *A* at replica manager *X* does not prevent transaction *U* from updating *A* at replica manager *Y*. Neither does the concurrency control on *B* at replica manager *N* prevent transaction *T* updating *B* at replica managers *M* and *P*.

This is contrary to the requirement for one-copy serializability. If these operations were to be performed on single copies of the objects, they would be serialized either with transaction *T* before *U* or with transaction *U* before *T*. This ensures that one of the transactions will read the value set by the other. Local concurrency control on copies of objects is not sufficient to ensure one-copy serializability in the available copies replication scheme.

As *write* operations are directed to all available copies, local concurrency control does ensure that conflicting writes on an object are serialized. In contrast, a *read* by one transaction and a *write* by another do not necessarily affect the same copy of an object. Therefore, the scheme requires additional concurrency control to prevent the dependencies between a *read* operation of one transaction and a *write* operation of another transaction forming a cycle. Such dependencies cannot arise if the failures and recoveries of replicas of objects are serialized with respect to transactions.

Local validation • We refer to the additional concurrency control procedure as local validation. The local validation procedure is designed to ensure that any failure or recovery event does not appear to happen during the progress of a transaction. In our example, as *T* has read from an object at *X*, *X*'s failure must be after *T*. Similarly, as *T* observes the failure of *N* when it attempts to update the object, *N*'s failure must be before *T*. That is:

N fails $\rightarrow T$ reads object A at X ; T writes object B at M and $P \rightarrow T$ commits $\rightarrow X$ fails

It can also be argued for transaction U that:

X fails $\rightarrow U$ reads object B at N ; U writes object A at $Y \rightarrow U$ commits $\rightarrow N$ fails

The local validation procedure ensures that two such incompatible sequences cannot both occur. Before a transaction commits it checks for any failures (and recoveries) of replica managers of objects it has accessed. In the example, transaction T would check that N is still unavailable and X , M and P are still available. If this is the case, T can commit. This implies that X fails after T validated and before U validated. In other words, U 's validation is after T 's validation. U 's validation fails because N has already failed.

Whenever a transaction has observed a failure, the local validation procedure attempts to communicate with the failed replica managers to ensure that they have not yet recovered. The other part of the local validation, which is testing that replica managers have not failed since objects were accessed, can be combined with the two-phase commit protocol.

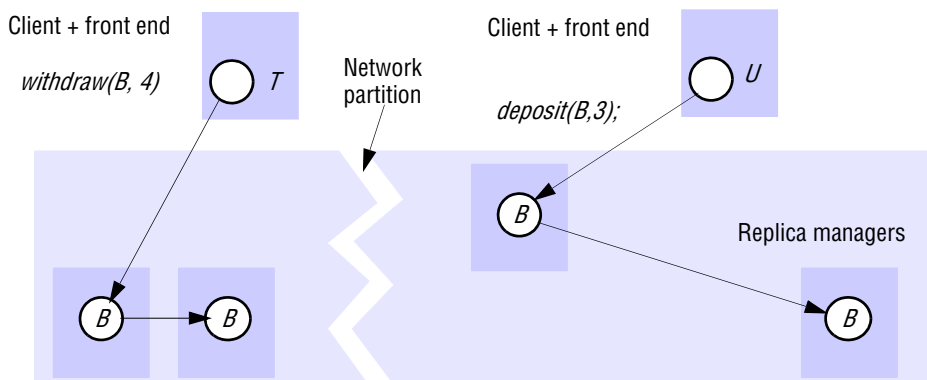
Available copies algorithms cannot be used in environments in which functioning replica managers are unable to communicate with one another.

18.5.3 Network partitions

Replication schemes need to take into account the possibility of network partitions. A network partition separates a group of replica managers into two or more subgroups in such a way that the members of one subgroup can communicate with one another but members of different subgroups cannot communicate with one another. For example, in Figure 18.11, the replica managers receiving the *deposit* request cannot send it to the replica managers receiving the *withdraw* request.

Replication schemes are designed with the assumption that partitions will eventually be repaired. Therefore, the replica managers within a single partition must ensure that any requests that they execute during a partition will not make the set of replicas inconsistent when the partition is repaired.

Figure 18.11 Network partition



Davidson *et al.* [1985] discuss many different approaches, which they categorize as being either optimistic or pessimistic with regard to whether inconsistencies are likely to occur. The optimistic schemes do not limit availability during a partition, whereas pessimistic schemes do.

The optimistic approaches allow updates in all partitions – this can lead to inconsistencies between partitions, which must be resolved when the partition is repaired. An example of this approach is a variant of the available copies algorithm in which updates are allowed in partitions and, after the partition has been repaired, the updates are validated – any updates that break the one-copy serializability criterion are aborted.

The pessimistic approach limits availability even when there are no partitions, but it prevents any inconsistencies occurring during partitions. When a partition is repaired, all that needs to be done is to update the copies of the objects. The quorum consensus approach is pessimistic. It allows updates in the partition that has the majority of replica managers and propagates the updates to the other replica managers when the partition is repaired.

18.5.4 Available copies with validation

The available copies algorithm is applied within each partition. This optimistic approach maintains the normal level of availability for *read* operations, even during partitions. When a partition is repaired, the possibly conflicting transactions that have taken place in the separate partitions are validated. If the validation fails, then some steps must be taken to overcome the inconsistencies. If there had been no partition, one of a pair of transactions with conflicting operations would have been delayed or aborted. Unfortunately, as there has been a partition, pairs of conflicting transactions have been allowed to commit in different partitions. The only choice after the event is to abort one of them. This requires making changes in the objects and in some cases, compensating effects in the real world, such as dealing with overdrawn bank accounts. The optimistic approach is only feasible with applications where such compensating actions can be taken.

Version vectors can be used to validate conflicts between pairs of *write* operations. These are used in the Coda file system and are described in Section 18.4.3. This approach cannot detect *read-write* conflicts but works well in file systems where transactions tend to access a single file and *read-write* conflicts are unimportant. It is not suitable for applications such as our banking example where *read-write* conflicts are important.

Davidson [1984] used *precedence graphs* to detect inconsistencies between partitions. Each partition maintains a log of the objects affected by the *read* and *write* operations of transactions. This log is used to construct a precedence graph whose nodes are transactions and whose edges represent conflicts between the *read* and *write* operations of transactions. Such a graph will not contain any cycles, since concurrency control has been applied within the partition. The validation procedure takes the precedence graphs from the partitions and adds edges, representing conflicts, between transactions in different partitions. If the resulting graph contains cycles, then the validation fails.

18.5.5 Quorum consensus methods

One way of preventing transactions in different partitions from producing inconsistent results is to make a rule that operations can be carried out within only one of the partitions. As the replica managers in different partitions cannot communicate with one another, the subgroup of replica managers within each partition must be able to decide independently whether they are allowed to carry out operations. A quorum is a subgroup of replica managers whose size gives it the right to carry out operations. For example, if having a majority is the criterion, a subgroup that has the majority of the members of a group would form a quorum because no other subgroup could have a majority.

In quorum consensus replication schemes an update operation on a logical object may be completed successfully by a subgroup of its group of replica managers. The other members of the group will therefore have out-of-date copies of the object. Version numbers or timestamps may be used to determine whether copies are up-to-date. If versions are used, the initial state of an object is the first version, and after each change we have a new version. Each copy of an object has a version number, but only the copies that are up-to-date have the current version number, whereas out-of-date copies have earlier version numbers. Operations should be applied only to copies with the current version number.

Gifford [1979] developed a file replication scheme in which a number of ‘votes’ are assigned to each physical copy at a replica manager of a single logical file. A vote can be regarded as a weighting related to the desirability of using a particular copy. Each *read* operation must first obtain a read quorum of R votes before it can proceed to read from any up-to-date copy, and each *write* operation must obtain a write quorum of W votes before it can proceed with an update operation. R and W are set for a group of replica managers such that

$$W > \text{half the total votes}$$

$$R + W > \text{total number of votes for the group}$$

This ensures that any pair consisting of a read quorum and a write quorum or two write quora must contain common copies. Therefore if there is a partition, it is not possible to perform conflicting operations on the same copy, but in different partitions.

To perform a *read* operation, a read quorum is collected by making sufficient version number enquiries to find a set of copies, the sum of whose votes is not less than R . Not all of these copies need be up-to-date. Since each read quorum overlaps with every write quorum, every read quorum is certain to include at least one current copy. The read operation may be applied to any up-to-date copy.

To perform a *write* operation, a write quorum is collected by making sufficient version number enquiries to find a set of replica managers with up-to-date copies, the sum of whose votes is not less than W . If there are insufficient up-to-date copies, then a non-current file is replaced with a copy of the current file, to enable the quorum to be established. The updates specified in the *write* operation are then applied by each replica manager in the write quorum, the version number is incremented and completion of the write is reported to the client.

The files at the remaining available replica managers are then updated by performing the *write* operation as a background task. Any replica manager whose copy of the file has an older version number than the one used by the write quorum updates it

by replacing the entire file with a copy obtained from a replica manager that is up-to-date.

Two-phase read-write locking may be used for concurrency control in Gifford’s replication scheme. The preliminary version number enquiry to obtain the read quorum, R , causes read locks to be set at each replica manager contacted. When a *write* operation is applied to the write quorum, W , a write lock is set at each replica manager involved. (Locks are applied with the same granularity as version numbers.) The locks ensure one-copy serializability, as any read quorum overlaps with any write quorum and any two write quora overlap.

Configurability of groups of replica managers • An important property of the weighted voting algorithm is that groups of replica managers can be configured to provide different performance or reliability characteristics. Once the general reliability and performance of a group of replica managers is established by its voting configuration, the reliability and performance of *write* operations may be increased by decreasing W and similarly for *reads* by decreasing R .

The algorithm can also allow for the use of copies of files on local disks at client computers as well as those at file servers. The copies of files in client computers are regarded as *weak representatives* and are always allocated zero votes. This ensures that they are not included in any quorum. A *read* operation may be performed at any up-to-date copy, once a read quorum has been obtained. Therefore a *read* operation may be carried out on the local copy of the file if it is up-to-date. Weak representatives can be used to speed up *read* operations.

An example from Gifford • Gifford gives three examples showing the range of properties that can be achieved by allocating weights to the various replica managers in a group and assigning R and W appropriately. We now reproduce Gifford’s examples, which are based on the table below. The blocking probabilities give an indication of the probability that a quorum cannot be obtained when a *read* or *write* request is made. They

		Example 1	Example 2	Example 3
Latency (milliseconds)	Replica 1	75	75	75
	Replica 2	65	100	750
	Replica 3	65	750	750
Voting configuration	Replica 1	1	2	1
	Replica 2	0	1	1
	Replica 3	0	1	1
Quorum sizes	R	1	2	1
	W	1	3	3
Derived performance of file suite:				
Read	Latency	65	75	75
	Blocking probability	0.01	0.0002	0.000001
Write	Latency	75	100	750
	Blocking probability	0.01	0.0101	0.03

are calculated assuming that there is a 0.01 probability that any single replica manager will be unavailable at the time of a request.

Example 1 is configured for a file with a high read-to-write ratio in an application with several weak representatives and a single replica manager. Replication is used to enhance the performance of the system, not the reliability. There is one replica manager on the local network that can be accessed in 75 milliseconds. Two clients have chosen to make weak representatives on their local disks, which they can access in 65 milliseconds, resulting in lower latency and less network traffic.

Example 2 is configured for a file with a moderate read-to-write ratio, which is accessed primarily from one local network. The replica manager on the local network is assigned two votes and the replica managers on the remote networks are assigned one vote apiece. Reads can be satisfied from the local replica manager, but writes must access the local replica manager and one remote replica manager. The file will remain available in read-only mode if the local replica manager fails. Clients could create local weak representatives for lower read latency.

Example 3 is configured for a file with a very high read-to-write ratio, such as a system directory in a three-replica-manager environment. Clients can read from any replica manager, and the probability that the file will be unavailable is small. Updates must be applied to all copies. Once again, clients could create weak representatives on their local machines for lower read latency.

The main disadvantage of quorum consensus is that the performance of *read* operations is degraded by the need to collect a read quorum from R replica managers.

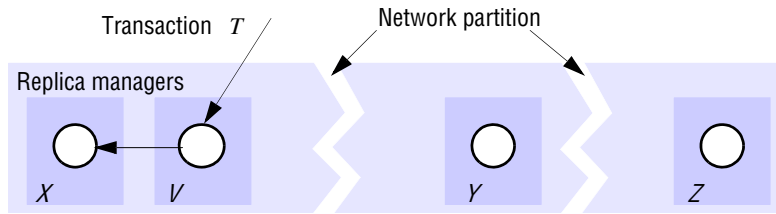
Herlihy [1986] proposed an extension of the quorum consensus method for abstract data types. This method allows the semantics of operations to be taken into account, to increase the availability of objects. Herlihy's method uses timestamps instead of version numbers. This has the advantage that there is no need to make version number enquiries in order to get a new version number before performing a write operation. The main advantage claimed by Herlihy is that the use of semantic knowledge can increase the number of choices for a quorum.

Quorum consensus in Dynamo • Dynamo uses a quorum-like approach for maintaining consistency amongst the replicas. As with Gifford's scheme, read and write operations must use R and W nodes respectively and $R + W > N$. In Dynamo, N is the number of nodes with replicas. The values of W and R affect availability, durability and consistency. DeCandia *et al.* [2007] state that a common configuration in Dynamo has $[N, R, W] = [3, 2, 2]$.

In the case of partition, Gifford's quorum can operate only in a 'majority' partition. But Dynamo uses a 'sloppy quorum' that involves N nodes, where replicas may be stored at substitute nodes that will pass on the values when the intended node recovers.

18.5.6 Virtual partition algorithm

This algorithm, which was proposed by El Abbadi *et al.* [1985], combines the quorum consensus approach with the available copies algorithm. Quorum consensus works correctly in the presence of partitions but the available copies algorithm is less expensive for *read* operations. A *virtual partition* is an abstraction of a real partition and contains a set of replica managers. Note that the term 'network partition' refers to the

Figure 18.12 Two network partitions

barrier that divides replica managers into several parts, whereas the term ‘virtual partition’ refers to the parts themselves. Although they are not connected with multicast communication, virtual partitions are similar to group views, which we introduced in Section 18.2.2. A transaction can operate in a virtual partition if it contains sufficient replica managers to have a read quorum and a write quorum for the objects accessed. In this case, the transaction uses the available copies algorithm. This has the advantage that *read* operations need only ever access a single copy of an object and may enhance performance by choosing the ‘nearest’ copy. If a replica manager fails and the virtual partition changes during a transaction, then the transaction is aborted. This ensures one-copy serializability of transactions because all transactions that survive see the failures and recoveries of replica managers in the same order.

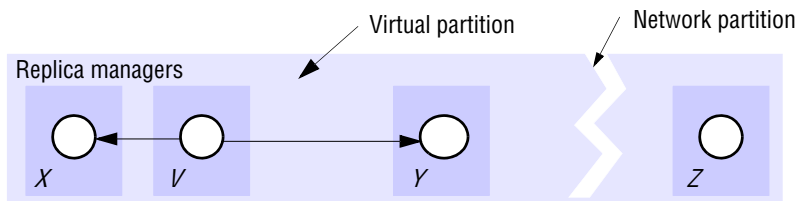
Whenever a member of a virtual partition detects that it cannot access one of the other members – for example, when a *write* operation is not acknowledged – it attempts to create a new virtual partition with a view to obtaining a virtual partition with read and write quora.

Suppose, for example, that we have four replica managers, *V*, *X*, *Y* and *Z*, each of which has one vote, and that the read and write quora are $R = 2$ and $W = 3$. Initially, all the managers can contact one another. So long as they remain in contact, they can use the available copies algorithm. For example, a transaction *T* consisting of a *read* followed by a *write* operation will perform the *read* at a single replica manager (for example, *V*) and the *write* operation at all four of them.

Suppose that transaction *T* starts by performing its *read* at *V* at a time when *V* is still in contact with *X*, *Y* and *Z*. Now suppose that a network partition occurs as in Figure 18.12, in which *V* and *X* are in one part and *Y* and *Z* are in different ones. Then when transaction *T* attempts to apply its *write*, *V* will notice that it cannot contact *Y* and *Z*.

When a replica manager cannot contact managers that it could previously contact, it keeps on trying until it can create a new virtual partition. For example, *V* will keep on trying to contact *Y* and *Z* until one or both of them replies – as, for example, in Figure 18.13, when *Y* can be accessed. The group of replica managers *V*, *X* and *Y* comprise a virtual partition because they are sufficient to form read and write quora.

When a new virtual partition is created during a transaction that has performed an operation at one of the replica managers (such as transaction *T*), the transaction must be aborted. In addition, the replicas within a new virtual partition must be brought up-to-date by copying them from other replicas. Version numbers can be used as in Gifford’s algorithm to determine which copies are up-to-date. It is essential that all replicas be up-to-date, because *read* operations are performed on any single replica.

Figure 18.13 Virtual partition

Implementation of virtual partitions • A virtual partition has a creation time, a set of potential members and a set of actual members. Creation times are logical timestamps. The actual members of a particular virtual partition have the same idea as to its creation time and membership (a shared *view* of the replica managers with which they can communicate). For example, in Figure 18.13 the potential members are V, X, Y, and Z and the actual members are V, X and Y.

The creation of a new virtual partition is achieved by a cooperative protocol carried out by those of the potential members that can be accessed by the replica managers that initiated it. Several replica managers may attempt to create a new virtual partition simultaneously. For example, suppose that the replica managers Y and Z shown in Figure 18.12 keep making attempts to contact the others, and after a while the network partition is partially repaired so that Y cannot communicate with Z but the two groups V, X, Y and V, X, Z can communicate among themselves. Then there is a danger that two overlapping virtual partitions, such as V_1 and V_2 shown in Figure 18.14, might both be created.

Consider the effect of executing different transactions in the two virtual partitions. The *read* operation of the transaction in V, X, Y might be applied at the replica manager Y, in which case its read lock will not conflict with write locks set by a *write* operation of a transaction in the other virtual partition. Overlapping virtual partitions are contrary to one-copy serializability.

The aim of the protocol is to create new virtual partitions consistently, even if real partitions occur during the protocol. The protocol for creating a new virtual partition has two phases, as shown in Figure 18.15.

A replica manager that replies *Yes* in phase 1 does not belong to a virtual partition until it receives the corresponding *Confirmation* message in phase 2.

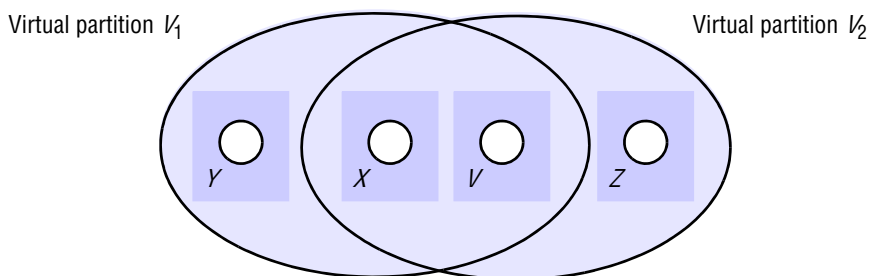
Figure 18.14 Two overlapping virtual partitions

Figure 18.15 Creating a virtual partition

Phase 1:

- The initiator sends a *Join* request to each potential member. The argument of *Join* is a proposed logical timestamp for the new virtual partition.
- When a replica manager receives a *Join* request, it compares the proposed logical timestamp with that of its current virtual partition.
 - If the proposed logical timestamp is greater it agrees to join and replies *Yes*;
 - If it is less, it refuses to join and replies *No*.

Phase 2:

- If the initiator has received sufficient *Yes* replies to have *read* and *write* quora, it may complete the creation of the new virtual partition by sending a *Confirmation* message to the sites that agreed to join. The creation timestamp and list of actual members are sent as arguments.
 - Replica managers receiving the *Confirmation* message join the new virtual partition and record its creation timestamp and list of actual members.
-

In our example above, the replica managers *Y* and *Z* shown in Figure 18.12 each attempt to create a virtual partition, and whichever one has the higher logical timestamp will be the one that is used in the end.

This is an effective method when partitions are not a common occurrence. Each transaction uses the available copies algorithm within a virtual partition.

18.6 Summary

Replicating objects is an important means of achieving services with good performance, high availability and fault tolerance in a distributed system. We described architectures for services in which replica managers hold replicas of objects, and in which front ends make this replication transparent. Clients, front ends and replica managers may be separate processes or exist in the same address space.

The chapter began by describing a system model in which each logical object is implemented by a set of physical replicas. Often, updates to these replicas can be made conveniently by group communication. We expanded our account of group communication to include group views and view-synchronous communication.

We defined linearizability and sequential consistency as correctness criteria for fault-tolerant services. These criteria express how the services must provide the equivalent of a single image of the set of logical objects, even though those objects are replicated. The most practically significant of the criteria is sequential consistency.

In passive (primary-backup) replication, fault tolerance is achieved by directing all requests through a distinguished replica manager and having a backup replica manager take over if this fails. In active replication, all replica managers process all requests independently. Both forms of replication can be conveniently implemented using group communication.

Next we considered highly available services. Gossip and Bayou both allow clients to make updates to local replicas while partitioned. In each system, replica managers exchange updates with one another when they become reconnected. Gossip provides its highest availability at the expense of relaxed, causal consistency. Bayou provides stronger eventual consistency guarantees, employing automatic conflict detection and the technique of operational transformation to resolve conflicts. Coda is a highly available file system that uses version vectors to detect potentially conflicting updates.

Finally, we considered the performance of transactions against replicated data. Both primary-backup architectures and architectures in which front ends may communicate with any replica manager exist for this case. We discussed how transactional systems allow for replica manager failures and network partitions. The techniques of available copies, quorum consensus and virtual partitions enable operations within transactions to make progress even in some circumstances where not all replicas are reachable.

EXERCISES

- 18.1 Three computers together provide a replicated service. The manufacturers claim that each computer has a mean time between failure of five days; a failure typically takes four hours to fix. What is the availability of the replicated service? page 766
- 18.2 Explain why a multi-threaded server might not qualify as a state machine. page 768
- 18.3 In a multi-user game, the players move figures around a common scene. The state of the game is replicated at the players' workstations and at a server, which contains services controlling the game as a whole, such as collision detection. Updates are multicast to all replicas. Consider the following conditions:
 - i) The figures may throw projectiles at one another, and a hit debilitates the unfortunate recipient for a limited time. What type of update ordering is required here? Hint: consider the 'throw', 'collide' and 'revive' events.
 - ii) The game incorporates magic devices that may be picked up by a player to assist them. What type of ordering should be applied to the 'pick-up-device' operation? page 770
- 18.4 A router separating process p from two others, q and r , fails immediately after p initiates the multicasting of message m . If the group communication system is view-synchronous, explain what happens to p next. page 773
- 18.5 You are given a group communication system with a totally ordered multicast operation and a failure detector. Is it possible to construct view-synchronous group communication from these components alone? page 773
- 18.6 A *sync-ordered* multicast operation is one whose delivery ordering semantics are the same as those for delivering views in a view-synchronous group communication system. In a *thingumajig* service, operations upon thingumajigs are causally ordered. The service supports lists of users able to perform operations on each particular thingumajig. Explain why removing a user from a list should be a sync-ordered operation. page 773
- 18.7 What is the consistency issue raised by state transfer? page 774

- 18.8 An operation X upon an object o causes o to invoke an operation upon another object o' . It is now proposed to replicate o but not o' . Explain the difficulty that this raises concerning invocations upon o' , and suggest a solution. page 773
- 18.9 Explain the difference between linearizability and sequential consistency, and why the latter is more practical to implement, in general. page 777
- 18.10 Explain why allowing backups to process *read* operations leads to sequentially consistent rather than linearizable executions in a passive replication system. page 780
- 18.11 Could the gossip architecture be used for a distributed computer game like the one described in Exercise 18.3? page 783
- 18.12 In the gossip architecture, why does a replica manager need to keep both a ‘replica’ timestamp and a ‘value’ timestamp? page 786
- 18.13 In a gossip system, a front end has the vector timestamp (3, 5, 7) representing the data it has received from members of a group of three replica managers. The three replica managers have vector timestamps (5, 2, 8), (4, 5, 6) and (4, 5, 8), respectively. Which replica manager(s) could immediately satisfy a query from the front end, and what would the resultant timestamp of the front end be? Which could incorporate an update from the front end immediately? page 788
- 18.14 Explain why making some replica managers read-only may improve the performance of a gossip system. page 792
- 18.15 Write pseudo-code for dependency checks and merge procedures (as used in Bayou) suitable for a simple room-booking application. page 793
- 18.16 In the Coda file system, why is it sometimes necessary for users to intervene manually in the process of updating the copies of a file at multiple servers? page 800
- 18.17 Devise a scheme for integrating two replicas of a file system directory that underwent separate updates during disconnected operation. Either use Bayou’s operational transformation approach, or supply a solution for Coda. page 801
- 18.18 Available copies replication is applied to data items A and B with replicas A_x, A_y and B_m, B_n . The transactions T and U are defined as:
 $T: \text{Read}(A); \text{Write}(B, 44)$. $U: \text{Read}(B); \text{Write}(A, 55)$.
 Show an interleaving of T and U , assuming that two-phase locks are applied to the replicas. Explain why locks alone cannot ensure one-copy serializability if one of the replicas fails during the progress of T and U . Explain with reference to this example how local validation ensures one-copy serializability. page 805
- 18.19 Gifford’s quorum consensus replication is in use at servers X, Y and Z , which all hold replicas of data items A and B . The initial values of all replicas of A and B are 100 and the votes for A and B are 1 at each of X, Y and Z . Also, $R = W = 2$ for both A and B . A client reads the value of A and then writes it to B .
- i) At the time the client performs these operations, a partition separates servers X and Y from server Z . Describe the quora obtained and the operations that take place if the client can access servers X and Y .
 - ii) Describe the quora obtained and the operations that take place if the client can access only server Z .
 - iii) The partition is repaired and then another partition occurs so that X and Z are separated from Y . Describe the quora obtained and the operations that take place if the client can access servers X and Z . page 810