```
            a.SendMessage("Re[8]:(A,B)","B");
            a.SendMessage("Re[9]:(A,B)","B");
            a.SendMessage("Re[10]:(A,B)","B");
            a.SendMessage("Re[11]:(A,B)","B");
            a.SendMessage("Re[12]:(A,B)","B");
            a.SendMessage("Re[13]:(A,B)","B");
            a.SendMessage("Re[14]:(A,B)","B");
            for(int i = 0 ; i < 14 ; i++){
                    System.out.println("Message to B -
                        " + b.ReceiveMessageAsIs());
            }
            for(int i = 0 ; i < 14 ; i++){
                    System.out.println("Message to C -
                        " + c.ReceiveMessageAsIs());
            }
        }
}
```

16 Replicated Data Management

16.1 INTRODUCTION

Data replication is an age-old technique for tolerating faults, increasing system availability, and reducing latency in data access. Consider the telephone directory that you use at your home, or at work, or while traveling. You may not want to carry it with you because of its bulky size; so you may decide to make multiple copies of it, keeping one for home and one in your car. Even if the copy in your car is stolen the loss is not catastrophic since you can always drive home to get the other copy. However, having an extra copy in the car saves the time to drive home, and therefore reduces the time to access the directory. Recall that the implementation of stable storage (Chapter 14) used data replication on independent disks. Replication is widely used in cache memory, distributed shared memory, distributed file systems, and bulletin boards.

In addition to fault-tolerance, replication reduces access latency. DNS servers at the upper levels are highly replicated — without this, IP address lookup will be unacceptably slow. In large systems (like peer-to-peer systems and grids), how many replicas will be required to bring down the latency to an acceptable level and where to place them are interesting questions. Another major problem in replication management is that of replica update. The problem does not exist if the data is read-only, which is true for program codes or immutable data. When a replica is updated, every other copy of that data has to be eventually updated to maintain consistency. However, due to the finite computation speed of processes and the latencies involved in updating geographically dispersed copies, it is possible that even after one copy has been updated, users of the other copies still access the old version. What inconsistencies are permissible and how consistency can be restored are the central themes of replicated data management.

16.1.1 Reliability vs. Availability

Two primary motivations behind data replication are *reliability* and *availability*. Consider a distributed file system, where a recently accessed file has been cached into the hard disk of your desktop. Even if your Internet connection is disrupted for some time, your work may not be disrupted. On the World Wide Web, proxy servers provide service when the main server becomes overloaded. These show how replication can improve data or service availability. There may be cases in which the service provided by a proxy server is not as efficient as the service provided by the main server — but certainly it is better that having no service at all. This is an example of *graceful degradation*. When all servers are up and running and client accesses are uniform, the quality of service goes up since the server loads are balanced. In mobile terminals and handheld devices, disconnected modes of operation are important, and replication of data or service minimizes the disruption of service. RAID (Redundant Array of Inexpensive Disks) is another example of providing reliability and improving availability through replication.

Reliability and availability address two orthogonal issues. A server that is reliable but rarely available serves no purpose. Similarly, a server that is available but frequently malfunctions or supplies incorrect or stale data causes headache for all.

Consider two users Alice and Bob updating a shared file **F**. Each user's life is as follows:

```
do true →
        read F;
        modify F;
        write F
od
```

Depending on how the file is maintained, the write operation (i) may update a central copy of **F** that is shared by both, (ii) or may update the local copies of **F** separately maintained by Alice and Bob. In either case, ideally, Alice's updates must reach Bob before he initiates the read operation, and vice versa. However, with independent local copies this may not be possible due to latencies associated with the physical channels; so Bob may either read a stale copy, or postpone reading until the update arrives. What is an acceptable semantics of sharing **F**? This depends on *data consistency* models.

16.2 ARCHITECTURE OF REPLICATED DATA MANAGEMENT

Ideally replicated data management must be transparent. Transparency implies that users have the illusion of using a single copy of the data or the object — even if multiple copies of a shared data exist or replicated servers provide a specific service, the clients should not have any knowledge of which replica is supplying with the data or which server is providing the service. It is conceptually reassuring for the clients to believe in the existence of a single server that is highly available and trustworthy, although in real life different replicas may take turns to create the illusion of a single copy of data or server. Ideal replication transparency can rarely be achieved, approximating replication transparency is one of the architectural goals of replicated data management.

16.2.1 Passive vs. Active Replication

For maintaining replication transparency, two different architectural models are widely used: *passive replication* and *active replication*. In passive replication, every client communicates with a single replica called the *primary*. In addition to the primary, one or more replicas are used as backup copies. Figure 16.1 illustrates this. A client requesting service always communicates with the primary copy. If the primary is up and running, then it provides the desired service. If the client requests do not modify the state of the server, then no further action is necessary. If however the service modifies the server state, then to keep the states of the backup servers consistent, the primary performs an atomic multicast of the updates to the backup servers before sending the response to the client. If the primary crashes, then one of the backup servers is elected as the new primary.

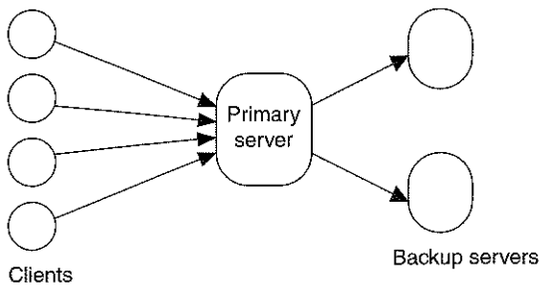

FIGURE 16.1 Passive replication of servers.

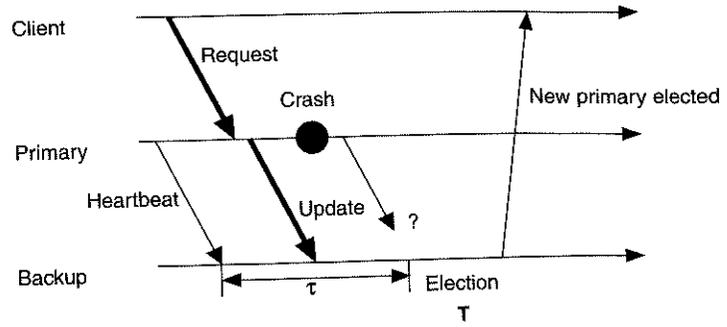

FIGURE 16.2 An illustration of the primary-backup protocol: The thin lines represent the heartbeat messages.

The *primary-backup* protocol has the following specifications:

1. At most one replica can be the primary server at any time.
2. Each client maintains a variable **L** (leader) that specifies the replica to which it will send requests. Requests are queued at the primary server.
3. Backup servers ignore client requests.

There may be periods of time when there is no primary server — this happens during a changeover, and the period is called the failover time. When repairs are ignored, the primary-backup approach implements a service that can tolerate a bounded number of faults over the lifetime of the service. Here, unless specified otherwise, a fault implies server crash. Since the primary server returns a response to the client after completing the atomic multicast, when a client receives a response it is assured that each nonfaulty replica has received the update. The primary server is also required to periodically broadcast heartbeat messages. If a backup server fails to receive this message within a specific window of time, then it concludes that the primary has crashed and initiates an election. The new leader takes over as the new primary and notifies the clients. Figure 16. 2 illustrates the steps of the primary-backup protocol.

To maintain replication transparency, the above switchover must be atomic. But in real life this may not be true. Consider the following description of the life of a client:

```
do    request for service → receive response from the server
□     timeout             → retransmit the request to the server
od
```

If the response is not received due to a server crash, then the request for service is retransmitted. Multiple retransmissions may be necessary until a backup server becomes the new primary server. This happens during the failover time. An important question is: given that at most **m** servers can fail over a given time, what are the smallest possible values of the degree of replication and the failover time? We will briefly examine these issues in presence of crash failures only.

Considering only crash failures, at least **(m+1)** replicas are sufficient to tolerate the crash of **m** servers, since to provide service it is sufficient to have only one server up and running. Also, from Figure 16.2 the smallest failover time is $\tau + 2\delta + T$ where τ is the interval between the reception of two consecutive heartbeat messages, **T** is the election time, and δ is the maximum message propagation delay from the primary to a backup server. This corresponds to the case when the primary crashes immediately after sending a heartbeat message and an update message to the backup servers.

An alternative to passive replication is active replication. Here, each of the **n** clients transparently communicates with a group of **k** servers (also called replica managers). Unlike the primary-backup model, these servers do not have any master–slave relationship among them. Figure 16.3 shows a

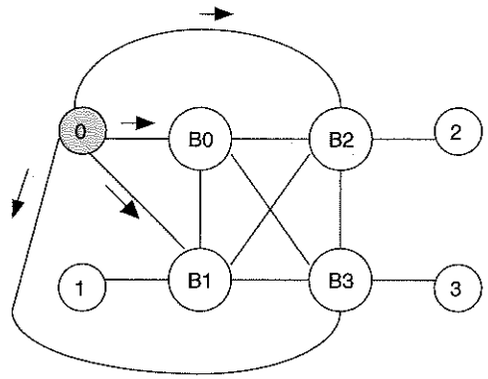

FIGURE 16.3 An example of active replication with four clients and four servers: The client's machine multicasts the updates to every other server.

bulletin board shared by a group of members. Each member **j** uses her local copy of the bulletin board. Whenever a member posts an update, her local copy is updated and her machine propagates the update to each of the **k** servers using total order multicast (so that eventually all copies of the bulletin board become identical). To read a message, the client sends a request to her machine, which eventually sends a response. Depending on the consistency model, a read request may be forwarded to the other servers before the response is generated.

16.2.2 Fault-Tolerant State Machines

The value of **k** will depend on the failure model. If there is no failure, then **k = 1** will suffice. If one or more of the replicas fail, **k** must increase. The value of **k** will depend on the nature and the extent of failure. Schneider [Sch90] presented a theory of fault-tolerant state machines, and it captures the essence of maintaining a fault-tolerant service via active replication. Recall how reliable atomic multicast is done (Chapter 15). When an update is multicast as a sequence of unicasts, the client's machine can crash at the middle of it. To preserve atomicity, servers have to communicate among themselves.

Each server is a state machine whose state is modified by a client request. If their initial states are identical and all updates are delivered in the same total order (in spite of failures), then all correct replicas will always be in the same state. The replica coordination problem can be reduced to the consensus problem, since all state machines agree to the choice of the next request to be used to update its state. This has two components:

Agreement. Every correct replica receives all the requests.
Order. Every correct replica receives the requests in the same order.

In purely asynchronous systems, the order part cannot be solved using deterministic protocols if processes crash, so replica coordination is impossible. Total order reliable multicast cannot be implemented on asynchronous distributed systems (since it will disprove the FLP impossibility result; see Theorem 15.1); so a synchronous model must be used. By appropriately combining the outputs of the replicas one can design a fault-tolerant state machine.

To model failures, there are two options (1) the client machines multicast updates to all the replica servers — both client machines and replicas may fail, or (2) client machines never fail, only the replicas exhibit faulty behavior. For simplicity, we consider the second version. Let **m** be the maximum number of faulty replicas. If the failures are only fail-stop, then **(m+1)** replicas are adequate since the faulty processes produce correct outputs until they fail, and the failure can be detected. To read a value, a client's machine queries the servers, and the first response that it receives

is the desired value. If the failures are byzantine, then to compensate for the erroneous behavior of faulty processes, at least **(2m+1)** replicas will be required, so that the faulty processes become a minority and they can be voted out.

For fail-stop failures, the agreement part can be solved using the reliable atomic multicast protocol of Section 15.2. For byzantine failures, one of the protocols from Section 13.3 can be used. The signed message algorithm offers much better resilience. The oral message algorithm will require more than **3m** replicas to tolerate m failures.

The order part can be satisfied by modifying the total order multicast protocols of Section 15.5.1 so that it deals with faulty replicas. Assume that timestamps determine the desired order in which requests will be delivered to the replicas. A client request is stable, if no request with a lower timestamp is expected to arrive after it. Only stable requests will be delivered to the state machine. A receiver can detect a fail-stop failure of the sender only after it has received the last message from it. If nonfaulty replicas communicate infinitely often and the channels are FIFO, then each state machine replica will receive the requests in the ascending order of timestamps. The dilemma here is that every nonfaulty client may not have a valid request to send for a long time. This affects the stability test. To overcome this and make progress, each nonfaulty client will be required to periodically send out *null* requests[1] when it has nothing to send. This will handle stability in the presence of fail-stop failures.

If the failure is byzantine, then a faulty client (or a faulty replica connected to a client) may refuse to send the null message. However, since the upper bound of the message propagation delay is known, the absence of a message can be detected.

16.3 DATA-CENTRIC CONSISTENCY MODELS

Replica consistency requires all copies of data to be eventually identical. However, due to the inherent latency in message propagation, it is sometimes possible for the clients to receive anomalous responses. Here is an example: In a particular flight, all the seats were sold out, and two persons **A** and **B** in two different cities are trying to make reservations. At 9:37:00 there was a cancellation by a passenger **C**. **A** tried to reserve a seat at 9:37:25 but failed to reserve the seat. Surprisingly, **B** tried to make the reservation at 9:38:00, and could grab it! While this appears awkward at a first glance, its acceptability hinges on the type of consistency model that is being used. Consistency determines what responses are acceptable following an update of a replica. It is a contract between the clients and the replica management system.

The implementation of *distributed shared memory* (DSM) that creates the illusion of a shared memory on top of a message passing system supports many data consistency models. Each machine maintains a local copy of the shared data (Figure 16.4). Users read and write their local copies, and the implementation of DSM conforms to the semantics of sharing. Below we present a few well-known consistency models — each model has a different implementation.

1. Strict consistency
2. Linearizability
3. Sequential consistency
4. Causal consistency

16.3.1 Strict Consistency

For a shared variable **x,** the trace of a computation is a sequence of read (**R**) and write (**W**) operations on **x**. One or more processes can execute these operations.

[1] An alternative is to use the known upper bound of the delay and deduce that the sender did not have anything to send up to a certain time.

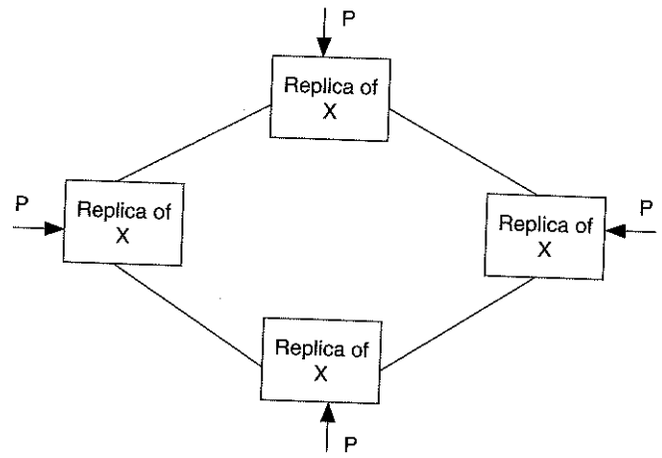

FIGURE 16.4 A distributed shared memory with four processes sharing a read–write object X.

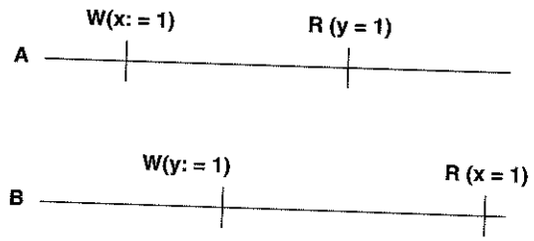

FIGURE 16.5 A linearizable shared memory. Initially **x=y=0**.

Strict consistency corresponds to true replication transparency. If one of the processes executes **x := 5** at real time **t** and this is the latest write operation, then at a real time $t' > t$, every process trying to read **x** will receive the value **5**. Strict consistency criterion requires that regardless of the number of replicas of **x**, every process receive a response that is consistent with the real time.

The scenario in the previous example of two passengers **A** and **B** trying to reserve airline seats does not satisfy the strict consistency criteria, since even if a seat has been released at 9:37:00 due to a cancellation, the availability of the seat was not visible to **A** at 9:37:25. Due to the nonzero latency of the messages, strict consistency is difficult to implement.

16.3.2 Linearizability

Strict consistency is too strong, and requires all replicas to be updated instantaneously. A slightly weaker version of consistency is linearizability.

Each trace interleaves of the individual reads and writes into a single total order that respects the local ordering of the reads and writes of every process. A trace is consistent, when every read returns the latest value written into the shared variable preceding that read operation. A trace is linearizable, when (1) it is consistent, and (2) if t_1, t_2 are the times at which two distinct processes perform operations and $t_1 < t_2$, then the consistent trace must satisfy the condition $t_1 < t_2$.
A shared memory is linearizable, when it has a linearizable trace. Figure 16.5 shows an example with two shared variables **x** and **y**. Initially **x = y = 0**.

The read and write operations by the two processes **A** and **B** can be reduced to a trace **W(x := 1) W(y := 1) R(y = 1) R(x = 1)**. This satisfies both criteria listed above. So linearizability holds.

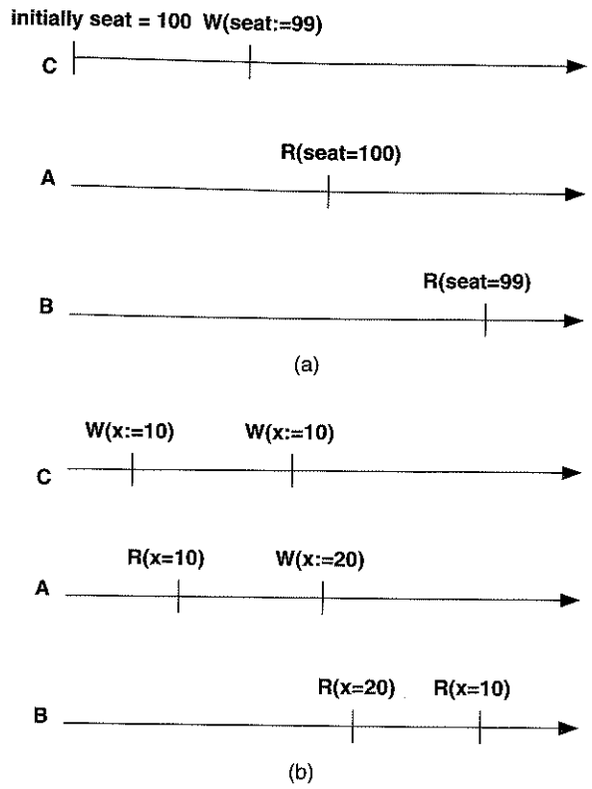

FIGURE 16.6 (a) Sequential consistency is satisfied. (b) Sequential consistency is violated.

16.3.3 Sequential Consistency

A slightly weaker (and much more widely used) form of consistency is sequential consistency. Sequential consistency requires only the first criterion of linearizability and is not concerned with real time.

Sequential consistency requires that some interleaving that preserves the local temporal order of the read and write operations, be a consistent trace. Thus, if a write **x := u** precedes another write **x := v** in a trace, then no process reading **x** will read **(x = v)** before **(x = u)**. In Figure 16.5, suppose that process **A** reads the value of y as 0 instead of 1. Will linearizability be satisfied? To determine this, note that the only possible trace that satisfies consistency is **W(x := 1) R(y = 0) W(y := 1) R(x := 1)**. However assuming that the time scales are consistent in the diagram it violates the real time requirement since **R(y = 0)** occurred after **W(y := 1)**. Therefore it is not linearizable. However, it satisfies sequential consistency.

Consider the example of airline reservation again (Figure 16.6a), and assume that the total number of available seats in the flight is 100. This behavior is sequentially consistent, since all processes could agree to the consistent interleaving shown below:

$$W(\text{seat} := 100 \text{ by C}) \prec R(\text{seat} = 100 \text{ by A}) \prec W(\text{seat} := 99 \text{ by C}) \prec R(\text{seat} = 99 \text{ by B})$$

Real time is not taken into consideration in sequential consistency. If a client is unhappy with this anomalous behavior of the reservation system, then she is perhaps asking for linearizability.

However, the scenario in Figure 16.6b does not satisfy sequential consistency. From the behaviors of **C** and **A** it follows that **W(x := 10) ≺ R(x = 10) ≺ (x := 20)**. However, process **B** reads **x** as

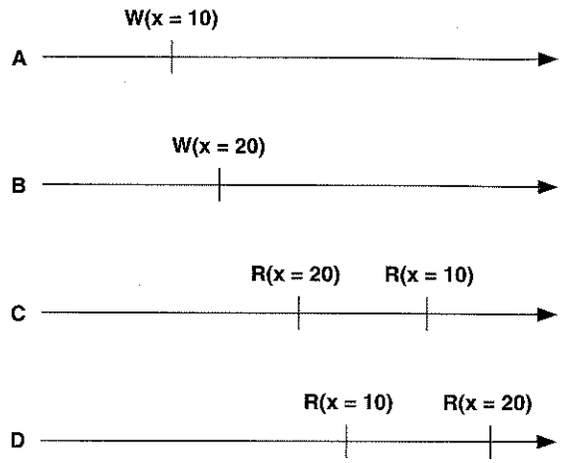

FIGURE 16.7 A behavior that is causally consistent, but not sequentially consistent.

20 first, and then reads **x** as **10**, so the two consecutive reads violate the program order of the two writes, and it is not possible to build a consistent total order that conforms to the local orders.

16.3.4 Causal Consistency

In the causal consistency model, all writes that are causally related must be seen by every process in the same order. The writes may be executed by the same process, or by different processes. The order of values returned by read operations must be consistent with this causal order. Writes that are not causally related to one another can however be seen in any order. Consequently, these do not impose any constraint on the order of values read by a process. Figure 16.7 illustrates a causally consistent behavior. Note that there is no causal order between **W(x:=10** by **A)** and **W(x:=20** by **B).** Therefore, processes **C** and **D** are free to read these values in any order, building their own perception of the history of the write operations.

Why are there so many different consistency criteria? Although true replication transparency is the ultimate target, the implementation of a stronger consistency criteria is expensive. Weaker consistency models have fewer restrictions and are cheaper to implement.

Consistency issues have been investigated in many different contexts. In addition to Distributed Shared Memory (DSM) on multicomputers, consistency models has also been extensively studied for cache coherence on shared-memory multiprocessors, web-caching, distributed file systems, distributed databases, and various highly available services.

16.4 CLIENT-CENTRIC CONSISTENCY PROTOCOLS

The various data consistency models have one thing in common: When multiple clients concurrently write a shared data, a write–write conflict results, and has to be appropriately resolved. In some cases there are no write–write conflicts to resolve. Consider the example of a webpage in the World Wide Web. The page is updated only by its owner. Furthermore, updates are infrequent, and the number of reads far outnumber the number of updates. Since browsers and web-proxies keep a copy of the fetched page in their local cache and return it to the viewer following the next request, quite often, stale versions of the webpage may be returned to the viewer. To maintain freshness, the cached copies are periodically discarded and updated pages are pulled from the server. As a result, updates propagate to the clients in a lazy manner. Protocols that deal with consistency issues in such cases are known as client-centric protocols. Below we present a few models of client-centric consistency.

16.4.1 Eventual Consistency

If the updates are infrequent, then eventually all readers will receive the correct view of the webpage, and all replicas become identical. This is the notion of eventual consistency. Similar models apply to many large databases like the DNS (Domain Name Server), where updates in a particular domain are performed by designated naming authorities and propagated in a lazy manner. The implementation is cheap.

Eventual consistency is acceptable as long as the clients access the same replica of the shared data. Mobile clients can potentially access different version of the replica and eventually consistency may be unsatisfactory. Consider a busy executive trying to update a database. She performs a part of the update, then leaves for a meeting in a different city, and in the evening decides to finish the remaining updates. Since she will not be working on the same replica that she was working on in the morning, she may notice that many of the updates that she made in the morning are lost.

16.4.2 Consistency Models for Mobile Clients

When replicas are geographically distributed, a mobile user will most likely use a replica closest to her. The following four consistency models are based on the sequence of operations that can be carried out by a mobile user:

1. Read-after-read
2. Write-after-write
3. Read-after-write
4. Write-after-read

Read-after-read consistency. When a read of a data set from one server **S** is followed by a read of its replica from another server **S***, each read from **S*** should return a value that is at least as old as, or more recent than the value previously read from **S**.

Relate this to the story of the busy executive. To make sense, if she reads a value **x1** at time **t1**, then at time **t2 > t1**, she must read a value that is either more recent or at least as old as the value **x1**. To implement this, all updates seen at time **t1** must be propagated to all replicas before time **t2**. Furthermore, structured data may often be partially updated: for example, each update operation on a salary database may update the salary of only one employee. To retain all such updates, replicas written after time **t1** must receive all updates done up to time **t1**, otherwise, a tuple **(x1, y1, z1)** may be updated to **(x2, y1, z1)** after the first update, and then to **(x1, y1, z2)** after a second update, although the correct value of the tuple should have been **(x2, y1, z2)** after the two update operations.
Write-after-write consistency. When a write on one replica in server **S** is followed by a write on another replica in a different server **S***, it is important that the earlier updates propagate to all replicas (including **S***) before the next write is performed.

All replicas should be updated in the same order. In the data-centric consistency model, this is equivalent to sequential consistency that expects some total order among all writes. If writes are propagated in the incorrect order, then a recent update may be overwritten by an older update. The order of updates can be relaxed when they are commutative: for example, when the write operations update disjoint variables.

To implement read-after-read consistency, when a client issues a read request for a data set **D** to a server, the server first determines if all prior writes to **D** have been locally updated. This requires logging the server id into the write set **D**, so that the updates can be fetched from those servers. Similarly, to implement write-after-write consistency, before a client carries out a write operation on a data set **D**, the server must guarantee that all prior writes to **D** have been locally performed.
Read-after-write consistency. Each client must be able to see the updates in a server **S** following every write operation by itself on another server **S***.

Consider updating a webpage. If the browser is not synchronized with the editor and the editor sends the updated HTML page to the server, the browser may return an old copy of the page to the viewer. To implement this consistency model, the editor must invalidate the cached copy, forcing the browser to fetch the recently uploaded version from the server.

Write-after-read consistency. Each write following a read should take effect on the previously read copy, or a more recent version of it. Thus if a read **R** performed at server **S**, precedes a write **W** is performed at server **S***, then all relevant updates found in **R** are also performed at **S***, and before **W**: To implement this consistency model, the server must guarantee that it has obtained all updates on the read set.

Thus, if a process reads a tuple as **(x1, y1, z1)** at **S**, and then wants to modify the tuple at **S***, then the write must take effect on **(x1, y1, z1)** or a later version of it and not on a stale version.

16.5 IMPLEMENTATION OF DATA-CENTRIC CONSISTENCY MODELS

Ordered multicasts (Chapter 15) play an important role in the implementation of consistency models. Linearizability requires the clocks of every process to be perfectly synchronized with the real time, and message propagation delays to be bounded. It can be implemented using total order multicast that forces every process to accept and handle all reads and writes in the same real time order in which they are issued. Here are the steps:

1. A process that wants to read (or write) a shared variable **x**, calls a procedure **read (x)** (or **write (x)**) at its local server. The local server sends out the read and write requests all other servers using a total order multicast.
2. Upon delivery of these requests, the replica servers update their copies of **x** in response to a write, but only send acknowledgments for the reads.
3. Upon return (that signals the completion of the total order multicast), the local copy is returned to the reader, or an acknowledgment is returned to the writer. This signals the completion of the **read (x)** or **write (x)** operation.

The correctness follows from the fact that every replica sees and applies all writes in the same real time order, so the value of **x** returned to the reader always reflects the latest write that has been completed in this order. The implementation is much easier for passive replication, where a master replica plays the role of a sequencer.

To implement sequential consistency, each read operation immediately returns the local copy. Only write operations trigger a total order multicast. In response to the multicast, other servers update their local copies and return an acknowledgment. As a result, every replica sees all writes in the same order, and each read returns a value consistent with some write in this total order. The physical clocks need not be synchronized.

To implement causal consistency, one approach is to use vector timestamps. As usual, every write operation, appended by the current value of the vector clock, is multicast to all replicas (recall the causal order multicast protocol in Chapter 15). Upon receiving and accepting these writes, each replica updates its local copy. This satisfies the causality requirement. Read operation immediately returns the local copy, which is consistent with some write in this causal order. The physical clocks need not be synchronized.

16.5.1 Quorum-Based Protocols

A classic method of implementing consistency uses the idea of a quorum. The relevance can be understood when network outages partition the replicas into two or more subgroups. One approach

of dealing with partitions is to maintain internal consistency within each subgroup, although no consistency can be enforced across subgroups due to the lack of communication. Mutual consistency is restored only after the connectivity is restored. Another approach is to allow updates in only one group that contains the majority of replicas, and postpone the updates in the remaining replicas — so these contain out-of-date copies until the partition is repaired.

Quorum systems use the second approach. A quorum system engages a designated minimum number of the replicas for every read or write — this number is called the read quorum or write quorum. The scheme was originally proposed by Thomas [T79]. Here is a description of how it works: Let there be **N** servers. To complete a write, a client must successfully place the updated data item on more than **N/2** servers. The updated data item will be assigned a new version number that is obtained by incrementing its current version number. To read the data, the client must read out the copies from any subset of at least **N/2** servers. From these, it will find out the copy with the highest version number and accept that copy. For concurrency control, two-phase locking can be used — for initial query from a set replicas, a reader can use read locks and a writer can use write locks. These locks will be released after the operation is completed or aborted.

Since the write quorum engages a majority of the servers, two distinct write operations cannot succeed at the same time. Therefore, all write operations are serialized. Furthermore, the intersection of the read quorum and the write quorum is nonempty, so reads do not overlap with writes. As a result, every read operation returns the latest version that was written, and single-copy serializability is maintained.

The above scheme can be further generalized to allow relaxed choices of the read and write quorums. For example, if **W** is the size of the write quorum, and **R** is the size of the read quorum, then it is possible to design a quorum-based protocol if the following two conditions are satisfied:

1. $W + R > N$
2. $W > N/2$

In a system with 10 replica servers, an example of a possible choice is **W=9, R=2**. The extreme case of **W=N, R=1** (known as *read-one write all*) is useful when the writes are very infrequent compared to the reads. It allows faster read and better parallelism. This generalization is due to Gifford [G79].

16.6 REPLICA PLACEMENT

There are many different policies for the placement of replicas of shared data. Here is a summary of some of the well-known strategies:

Mirror sites. On the World Wide Web, a mirror site contains a replica of the original web site on a different server. There may be several mirror sites for a website that expect a large number of hits. These replica servers are geographically dispersed and improve both availability and reliability. A client can choose any one of them to cut down the response time or improve availability. Sometimes a client may also be automatically connected to the nearest site. Such mirror sites are permanent replicas.

Server-generated replicas. Here replication is used for load balancing. The primary server copies a fraction of its files to a replica site only when its own load increases. Such replica sites are hosted by web-hosting services that temporarily rent their spaces to third party on demand. The primary server monitors the access counts for each of its files. When the count exceeds a predetermined threshold **h1**, the file is copied into a host server that is geographically closer to the callers, and all calls are rerouted. The joint access counts are periodically monitored. When the access count falls below a second threshold **h2**($h1 > h2$), the file is removed from the remote server and rerouting is discontinued. The relation $h1 > h2$ helps overcome a possible oscillation in and out of the replication mode.

Client caches. The client maintains a replica in its own machine or another machine in the same LAN. The administration of the cache is entirely the client's responsibility and the server has no contractual obligation to keep it consistent. In some cases, the client may request the server for a notification in case another client modified the server's copy. Upon receiving such a notification, the client invalidates its local copy and prepares to pull a fresh copy from the server during a subsequent read. This restores consistency.

16.7 CASE STUDIES

In this section, we discuss two replication-based systems, and study how they manage the replicas and maintain consistency. The first one is the distributed file system Coda, and the second one is the highly available service Bayou that is based on the gossip architecture.

16.7.1 Replication Management in Coda

Coda (acronym for Constant Data Availability) is a distributed file system designed by Satyanarayanan and his group in Carnegie Mellon University. It descended from their earlier design of AFS (Andrew File System). In addition to supporting most features of AFS, Coda also supports data availability in the disconnected mode.

Disconnected modes of operation have gained importance in recent times. Mobile users frequently get disconnected from the network, but still want to continue their work without much disruption. Also, network problems can cause partitions, making some replicas unreachable. This affects the implementation of consistency protocols, since partitions prevent update propagation. As a result, clients in disconnected components may receive inconsistent values. To sustain operation and maintain availability, consistency properties will at best hold within each connected component, and with non-faulty replicas only. If applications consider this preferable to the non-availability of the service, then contracts have to be rewritten, leading to weaker consistency models. An additional requirement in disconnected operation is that after the repair is done or the connectivity is restored, all replicas must be brought up-to-date, and reintegrated into the system. To track the recentness of the updates and to identify conflicts, various schemes are used. Coda uses vector timestamps for this purpose.

To maintain high availability, the clients' files are replicated and stored in Volume Storage Groups (**VSG**). Depending on the current state of these replicas and the connectivity between the client and the servers, a client can access only a subset of these called **AVSG** (Accessible **VSG**). To open a file, the client downloads a copy of it from a preferred server in its **AVSG**, and caches it in his local machine. The preferred server is chosen depending on its physical proximity or its available bandwidth, but the client also makes sure that the preferred server indeed contains the latest copy of the file — otherwise, a server that contains the most recent updates is chosen as the preferred server. While closing the file, the client sends its updates to all the servers in its **AVSG**. This is called the read-one-write-all strategy. When the client becomes disconnected from the network, its **AVSG** becomes empty and the client relies on the locally cached copies to continue operation.

Files are replicated in the servers, as well as in the client cache. Coda considers the server copies to be more trustworthy (first-class objects) than the client copies (second-class objects), since clients have limited means and resources to ensure the quality of the object. There are two approaches to file sharing: pessimistic and optimistic. In a pessimistic sharing, file modifications are not permitted until the client has exclusive access to all the copies. This maintains strict consistency but has poor availability. The optimistic approach allows a client to make progress regardless of whatever copies are accessible and is the preferred design choice in Coda. Reintegration to restore consistency is postponed to a later stage.

The reintegration occurs as follows: Each server replica has a vector (called Coda Version Vector or **CVV**) attached to it — this reflects the update history of the replica. The **k**th component of the

CVV (call it **CVV[k]**), of the replica of a file **F** represents the number of updates made on the replica of that file at server **k**. As an example, consider four replicas of a file **F** stored in the servers **V0, V1, V2, V3** shared by two clients **0** and **1**. Of these four replicas, the copies **V0, V1** are accessible to client **0** only and copies **V2, V3** are accessible to client **1** only. At time **A**, client **0** updates its local copy and multicasts them to **V0** and **V1**, so **CVV** of these versions become **1, 1, 0, 0**. If client **1** also updates in overlapped time (time **B**) and multicasts its updates to the servers, the **CVV** of client **1**'s version in **V2, V3** becomes **0, 0, 1, 1**. When the connections are restored and the **AVSG** of both clients include all four servers, each server compares the two **CVVs** and finds them incomparable.[2] This leads to a conflict that cannot always be resolved by the system, and may have to be resolved manually. If however **CVV(F) = 3, 2, 1, 1** for client **0**, and **CVV(F) = 0, 0, 1, 1** for client **1**, then the first **CVV** is greater than the second one. In such a case, during reintegration, the **CVVs** of all four replicas are modified to **3, 2, 1, 1** and replicas **V2** and **V3** are updated using the copies in **V0** or **V1**. Coda observed that write–write conflicts are rare (which is true for typical UNIX environments). The traces of the replicas reflect the serialization of all the updates in some arbitrary order and preserves sequential consistency.

Coda guarantees that a change in the size of the available VSG is reported to the clients within a specific period of time. When a file is fetched from a server, the client receives a promise that the server will call its back when another client has made any change into the file. This is known as *callback* promise. When a shared file **F** is modified, a preferred server sends out callbacks to fulfill its promise. This invalidates the local copy of **F** in the client's cache. When the client opens file **F**, it has to load the updated copy from the server. If no callback is received, then the copy in the local cache is used. However, in Coda, since the disconnected mode of operation is supported, there is a possibility that the callback may be lost because the preferred server is down or out of the client's reach. In such a case, the callback promise is not fulfilled, that is, the designated server is freed from the obligation of sending the callback when another client modifies the file. To open a file, the client fetches a fresh copy from its available VSG.

16.7.2 Replication Management in Bayou

Bayou is a distributed data management service that emphasizes high availability. It is designed to operate under diverse network conditions, and supports an environment for computer supported cooperative work.

The system satisfies *eventual consistency*, which only guarantees that all replicas eventually receive all updates. Update propagation only relies on occasional pairwise communications between servers. The system does not provide replication transparency — the application explicitly participates in conflict detection and resolution. The updates received by two different replica managers are checked for conflicts and resolved during occasional *anti-entropy sessions* that incrementally steer the system towards a consistent configuration. Each replica applies (or discards) the updates in such a manner that eventually the replicas become identical to one another.

A given replica enqueues the pending updates in the increasing order of timestamps. Initially all updates are tentative. In case of a conflict, a replica may undo a tentative update or reapply the updates in a different order. However, at some point, an update must be committed or stabilized, so that the result takes permanent effect. An update becomes stable when (i) all previous updates have become stable, and (ii) no update operation with a smaller timestamp will ever arrive at that replica. Bayou designates one of the replicas as the primary — the commit timestamps assigned by the primary determine the total order of the updates. During the antientropy sessions, each server examines the timestamps of the replica on other servers — an update is stable at a server when it has a lower timestamp than the logical clocks of all other servers. The number of trial updates

[2] See Chapter 6 to find out how two vector timestamps can be compared.