Consistency, Replication and Fault Tolerance

● Introduction to replication and consistency

There are two primary reasons for replicating data: reliability and performance. First, data is replicated to increase the reliability of a system. If a file system has been replicated it may be possible to continue working after one replica crashes by simply switching to one of the other replicas. Also, by maintaining multiple copies, it becomes possible to provide better protection against corrupted data. For example, imagine there are three copies of a file and every read and write operation is performed on each copy. We can safeguard ourselves against a single, failing write operation, by considering the value that is returned by at least two copies as being the correct one.

The other reason for replicating data is performance. Replication for performance is important when the distributed system needs to scale in numbers and geographical area. Scaling in numbers. occurs, for example, when an increasing number of processes needs to access data that are managed by a single server. In that case, performance can be improved by replicating the server and subsequently dividing the work.

Scaling with respect to the size of a geographical area may also require replication. The basic idea is that by placing a copy of data in the proximity of the process using them, the time to access the data decreases. As a consequence, the performance as perceived by that process increases. This example also illustrates that the benefits of replication for performance may be hard to evaluate.

The problem with replication is that having multiple copies may lead to consistency problems. Whenever a copy is modified, that copy becomes different from the rest. Consequently, modifications have to be carried out on all copies to ensure consistency. Exactly when and how those modifications need to be carried out determines the price of replication.

Replication and caching for performance are widely applied as scaling techniques. Scalability issues generally appear in the form of performance problems.

Placing copies of data close to the processes using them can improve performance through reduction of access time and thus solve scalability problems. A more serious problem, however, is that keeping multiple copies consistent may itself be subject to serious scalability problems. Intuitively, a collection of copies is consistent when the copies are always the same.

This means that a read operation performed at any copy will always return the same result. Consequently, when an update operation is performed on one copy, the update should be propagated to all copies before a subsequent operation takes place, no matter at which copy that operation is initiated or performed.

This type of consistency is sometimes informally (and imprecisely) referred to as tight consistency as provided by what is also called synchronous replication. The key idea is that an update is performed at all copies as a single atomic operation, or transaction. Unfortunately, implementing atomicity involving a large number of replicas that may be widely dispersed across a large-scale network is inherently difficult when operations are also required to complete quickly.

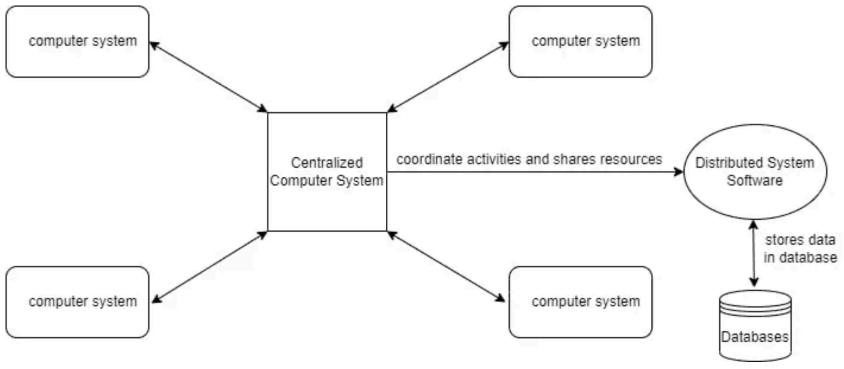
Difficulties come from the fact that we need to synchronize all replicas. In essence, this means that all replicas first need to reach agreement on when exactly an update is to be performed locally. In many cases, the only real solution is to loosen the consistency constraints.

In other words, if we can relax the requirement that updates need to be executed as atomic operations, we may be able to avoid (instantaneous) global synchronizations, and may thus gain performance.

● Data-Centric Consistency Models

The Data Centric Consistency Model is a set of principles and protocols used in distributed systems to define how data is accessed and updated, ensuring that all nodes in the system agree on the state of shared data.

It primarily focuses on the consistency guarantees provided to clients regarding the data they access and manipulate. In a distributed system, data is stored and managed across multiple nodes that communicate and collaborate to provide services and respond to client requests.


computer system
computer system
Centralized Computer System
coordinate activities and shares resources
Distributed System Software
stores data in database
computer system
computer system
Databases

However, due to factors like network delays, node failures, and concurrent data updates, achieving consistency in such an environment can be challenging.

Types of Data Centric Consistency Model:

Data-centric consistency models are a set of principles and protocols that define how data is accessed and updated in a distributed system, ensuring that all nodes agree on the state of shared data.

These models primarily focus on the consistency guarantees provided to clients regarding the data they access and manipulate. Let's explore the most common data-centric consistency models in detail:

Sequential Consistency

The time axis is always drawn horizontally, with time increasing from left to right. The symbols

$$W_i(x)a \text{ and } R_i(x)b$$

mean that a write by process P; to data item x with the value a and a read from that item by Pi returning b have been done, respectively. We assume that each data item is initially NIL. When there is no confusion concerning which process is accessing data, we omit the index from the symbols W and R.

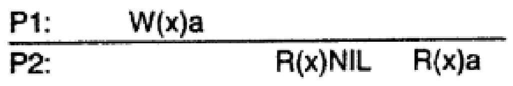


Figure 7-4. Behavior of two processes operating on the same data item. The horizontal axis is time.

As an example, in Fig. 7-4 PI does a write to a data item x, modifying its value to a. Note that, in principle, this operation WI (x)a is first performed on a copy of the data store that is local to PI, and is then subsequently propagated to the other local copies. In our example, P2 later reads the value NIL, and some time after that (from its local copy of the store). What we are seeing here is that it took some time to propagate the update of x to P2, which is perfectly acceptable.

In general, a data store is said to be sequentially consistent when it satisfies the following condition:

The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of-each individual process appear in this sequence in the order specified by its program.

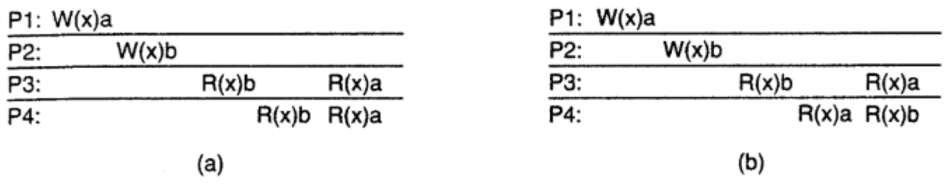


Figure 7-5. (a) A sequentially consistent data store. (b) A data store that is not sequentially consistent..

Consider four processes operating on the same data item x. In Fig. 7-5(a) process PI first performs W(x)a to x. Later (in absolute time), process P2 also performs a write operation,

by setting the value of x to b. However, both processes P3 and P4 first read value b, and later value a. In other words, the write operation of process P2 appears to have taken place before that of PI·

In contrast, Fig.7-5(b) violates sequential consistency because not all processes see the same interleaving of write operations. In particular, to process P3, it appears as if the data item has first been changed to b, and later to a. On the other hand, P4 will conclude that the final value is b.

Process P1	Process P2	Process P3
x ← 1;	y ← 1;	z ← 1;
print(y, z);	print(x, z);	print(x, y);

Figure 7-6. Three concurrently-executing processes.

To make the notion of sequential consistency more concrete, consider three concurrently-executing processes PI, P2, and P3, shown in Fig. 7-6 (Dubois et aI.,1988). The data items in this example are formed by the three integer variables x, y, and z, which are stored in a (possibly distributed) shared sequentially consistent data store. We assume that each variable is initialized to O. In this example, an assignment corresponds to a write operation, whereas a print statement corresponds to a simultaneous read operation of its two arguments. All statements are assumed to be indivisible.

Causal Consistency

The causal consistency model (Hutto and Ahamad, 1990) represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally related and those that are not. We already came across causality when discussing vector timestamps in the previous chapter. If event b is caused or influenced by an earlier event a, causality requires that everyone else first see a, then see b.

Consider a simple interaction by means of a distributed shared database. Suppose that process P, writes a data item x. Then P2 reads x and writes y. Here the reading of x and the writing of y are potentially causally related because the computation of y may have depended on the value of x as read by Pz (i.e., the value written by PI)' On the other hand, if two processes spontaneously and simultaneously write two different data items, these are not causally related. Operations that are not causally related are said to be concurrent. For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

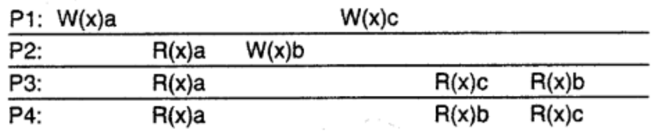


Figure 7-8. This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

As an example of causal consistency, consider Fig. 7-8. Here we have an event sequence that is allowed with a causally-consistent store, but which is forbidden with a sequentially-consistent store or a strictly consistent store. The thing to note is that the writes Wz(x)b and WI (x)c are concurrent, so it is not required that all processes see them in the same order.

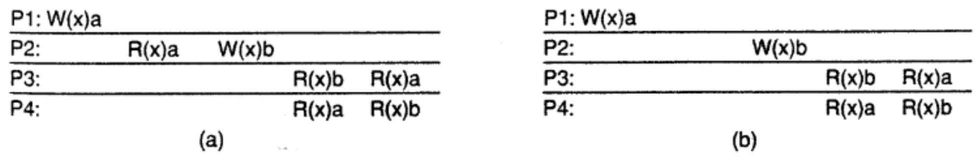


Figure 7-9. (a) A violation of a causally-consistent store. (b) A correct sequence of events in a causally-consistent store.

Now consider a second example. In Fig. 7-9(a) we have Wz(x)b potentially depending on WI (x)a because the b may be a result of a computation involving the value read by Rz(x)a. The two writes are causally related, so all processes must see them in the same order. Therefore, Fig. 7-9(a) is incorrect. On the other hand, in Fig. 7-9(b) the read has been removed, so WI (x)a and Wz(x)b are now concurrent writes. A causally-consistent store does not require concurrent writes to be globally ordered, so Fig.7-9(b) is correct. Note that Fig.7-9(b) reflects a situation that would not be acceptable for a sequentially consistent store.

Grouping Operations

Sequential and causal consistency are defined at the level of read and write operations. This level of granularity is for historical reasons: these models have initially been developed for shared-memory multiprocessor systems and were actually implemented at the hardware level.

The fine granularity of these consistency models in many cases did not match the granularity as provided by applications. What we see there is that concurrency between programs sharing data is generally kept under control through synchronization mechanisms for mutual exclusion and transactions. Effectively, what happens is that at the program level read and write operations are bracketed by the pair of operations ENTER_CS and LEAVE_CS where "CS" stands for critical section. As we explained in Chap. 6, the synchronization between processes takes place by means of these two operations. In terms of our distributed data store, this means that a process that has successfully executed ENTER_CS will ensure that the data in its local store is up to date. At that point, it can safely execute a series of read and write operations on that store, and subsequently wrap things up by calling LEAVE_CS.

In essence, what happens is that within a program the data that are operated on by a series of read and write operations are protected against concurrent accesses that would lead to seeing something else than the result of executing the series as a whole. Put differently, the bracketing turns the series of read and write operations into an atomically executed unit, thus raising the level of granularity.

In order to reach this point, we do need to have precise semantics concerning the operations ENTER_CS and LEAVE_CS. These semantics can be formulated in terms of shared synchronization variables. There are different ways to use these variables. We take the general approach in which each variable has some associated data, which could amount to the complete set of shared data. We adopt the convention that when a process enters its critical section it should acquire the relevant synchronization variables, and likewise when it leaves the critical section, it releases these variables. Note that the data in a process's critical section may be associated with different synchronization variables.

Each synchronization variable has a current owner, namely, the process that last acquired it. The owner may enter and exit critical sections repeatedly without having to send any messages on the network. A process not currently owning a synchronization variable but wanting to acquire it has to send a message to the current owner asking for ownership and the current values of the data associated with that synchronization variable. It is also possible for several processes to simultaneously own a synchronization variable in nonexclusive mode, meaning that they can read, but not write, the associated data.

We now demand that the following criteria are met:

1. An acquired access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.

2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.

3. After an exclusive mode access to a synchronization variable has been performed, any other process' next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

The first condition says that when a process does an acquire, the acquire may not complete (i.e., return control to the next statement) until all the guarded shared data have been brought up to date. In other words, at an acquire, all remote changes to the guarded data must be made visible.

The second condition says that before updating a shared data item, a process must enter a critical section in exclusive mode to make sure that no other process is trying to update the shared data at the same time.

The third condition says that if a process wants to enter a critical region in nonexclusive mode, it must first check with the owner of the synchronization variable guarding the critical region to fetch the most recent copies of the guarded shared data.

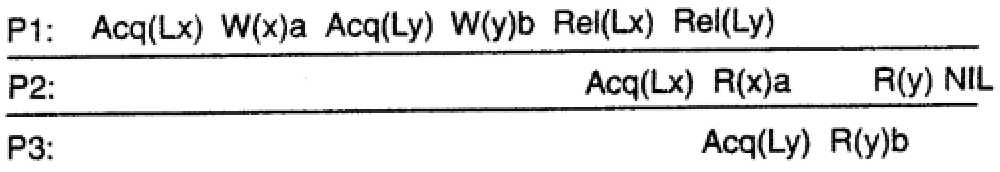

P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)
P2: Acq(Lx) R(x)a R(y) NIL
P3: Acq(Ly) R(y)b

Figure 7·10. A valid event sequence for entry consistency.

Fig. 7-10 shows an example of what is known as entry consistency. Instead of operating on the entire shared data, in this example we associate locks with each data item. In this case, P I does an acquire for x, changes x once, after which it also does an acquire for y. Process P2 does an acquire for x but not for Y'.so that it will read value a for x, but may read NIL for y. Because process P3 first does an acquire for y, it will read the value b when y is released by Pl'.

● Client-Centric Consistency Models

- ➢ Client –centric Consistency model-many Inconsistencies can be hidden in a cheap way.
- ➢ The client-centric consistency models arm at providing a consistent view on a database.
- ➢ An important assumption is that concurrent processes may simultaneously update and provide consistency in the face of such concurrency.
- ➢ client-centric consistency models are described using the following notations.
- ➢ Let xi[t] denote the version of data item x at Li at time t.
- ➢ version xi[t] is the result of a series of write operations of Li that took place since initialization; we denote this set as WS(xi[t]).
- ➢ If operations in WS(xi[t1]) have also been performed at local Lj at later time t2, we write WS(xi[t1];xj[t2]).

The Client-Centric consistency models are:

1) Monotonic Reads

2) Monotonic Writes

3) Read your Writes

4) Writes follow Reads

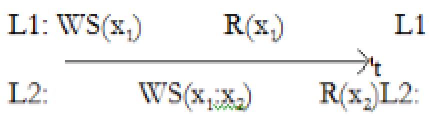
5) Eventual Consistency

1) Monotonic reads:

A data store is said to provide monotonic read consistency if the following conditions helds.

If a process reads the value of an algorithm x by what process will always return that same value or more recent value.

In other words, monotonic read consistency guarantees that if a process has seen a value of x at a later time t, it will never see an older version of x at a later time.



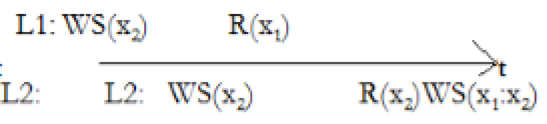
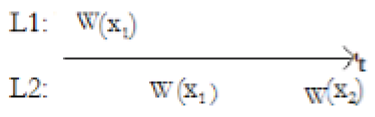


Fig.Monotonic read consistent data store

Fig. data store that does not provide monotonic reads.

2) Monotonic writes: in a monotonic write consistent store, the following conditions holds:

A write operation by a process on a data item x is completed before any successive write operation on x by the same process.



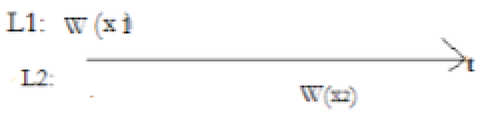


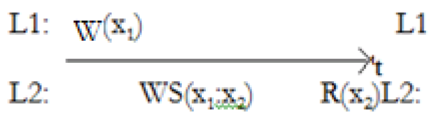
Fig.Monotonic write consistent data store

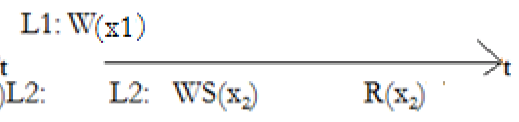
Fig. data store that does not provide monotonic write consistency

3) Read your writes:

A database is said to provide read your write consistency, if the following condition holds:

The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.





4) Write your reads:

A data store is said to provide writes follows reads consistency, if the following conditions holds:

A write operation by a process on a data item x following a previous read operation on x by the same process is generated to take place on the same or a more recent value of x that was read.

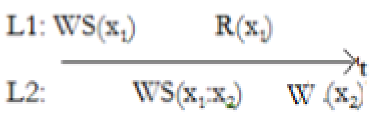


Fig. A write follows reads consistent data store

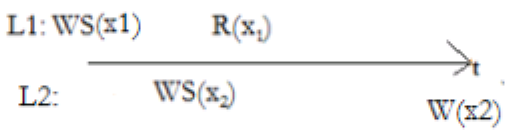


Fig. A datastore that does not provide write follows read consistency.

5) Eventual Consistency:

Data stores there are eventually consistent and have proper that in absence of updates, all replicas coverage toward identical copies of each other.

Eventual consistency essentially requires only that updates are guaranteed to propagate to all replicas.

● Replica Management

A key issue for any distributed system that supports replication is to decide where, when, and by whom replicas should be placed, and subsequently which mechanisms to use for keeping the replicas consistent. The placement problem itself should be split into two subproblems: that of placing replica servers, and that of placing content. The difference is a subtle but important one and the two issues are often not clearly separated. Replica-server placement is concerned with finding the best locations to place a server that can host (part of) a data store. Content placement deals with finding the best servers for placing content. Note that this often means that we are looking for the optimal placement of only a single data item. Obviously, before content placement can take place, replica servers will have to be placed first.

Replica-Server Placement

The placement of replica servers is not an intensively studied problem for the simple reason that it is often more of a management and commercial issue than an optimization problem. Nonetheless, analysis of client and network properties are useful to come to informed decisions.

There are various ways to compute the.best placement of replica servers, but all boil down to an optimization problem in which the best K out of N locations need to be selected (K < N). These problems are known to be computationally complex and can be solved only through heuristics.

Take the distance between clients and locations as their starting point. Distance can be measured in terms of latency or bandwidth. Their solution selects one server at a time such that the average distance between that server and its clients is minimal given that already k servers have been placed (meaning that there are N - k locations left).

Nodes are assumed to be positioned in an m-dimensional geometric space, as we discussed in the previous chapter. The basic idea is to identify the K largest clusters and assign a node from each cluster to host replicated content. To identify these clusters, the entire space is partitioned into cells. The K most dense cells are then chosen for placing a replica server. A cell is nothing but an m-dimensional hypercube. For a two-dimensional space, this corresponds to a rectangle.

Obviously, the cell size is important, as shown in Fig. 7-16.

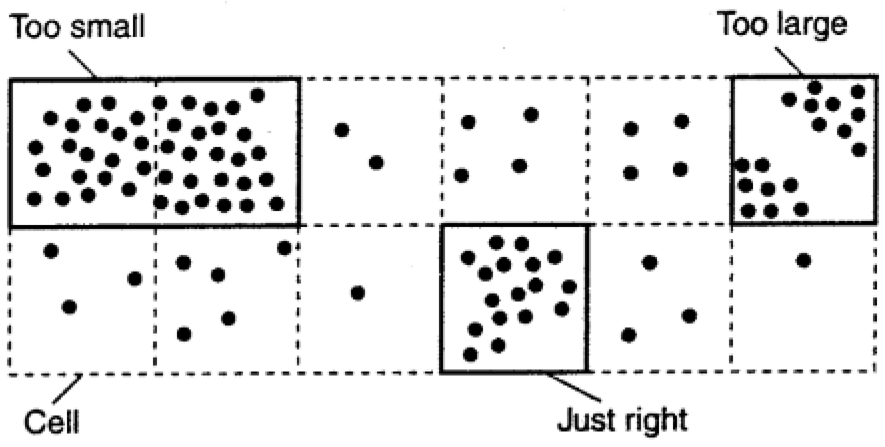


Figure 7-16. Choosing a proper cell *size* for server placement.

If cells are chosen too large, then multiple clusters of nodes may be contained in the same cell. In that case, too few replica servers for those clusters would be chosen. On the other hand, choosing small cells may lead to the situation that a single cluster is spread across a number of cells, leading to choosing too many replica servers. As it turns out, an appropriate cell size can be computed as a simple function of the average distance between two nodes and the number of required replicas.

Content Replication and Placement

When it comes to content replication and placement, three different types of replicas can be distinguished logically organized as shown in Fig. 7- 17.

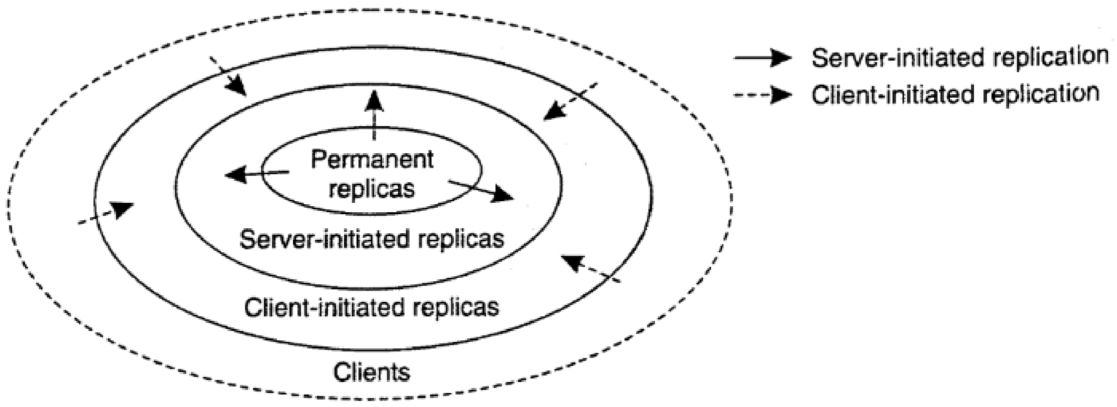

Server-initiated replication
Client-initiated replication
Permanent replicas
Server-initiated replicas
Client-initiated replicas
Clients

Figure 7-17. The logical organization of different kinds of copies of a data store into three concentric rings.

Permanent Replicas

Permanent replicas can be considered as the initial set of replicas that constitute a distributed data store. In many cases, the number of permanent replicas is small. Consider, for example, a Website. Distribution of a Web site generally comes in one of two forms. The first kind of distribution is one in which the files that constitute a site are replicated across a limited number of servers at a single location. Whenever a request comes in, it is forwarded to one of the servers, for instance, using a round-robin strategy. .

The second form of distributed Web sites is what is called mirroring. In this case, a Website is copied to a limited number of servers, called mirror sites. which are geographically spread across the Internet. In most cases, clients simply choose one of the various mirror sites from a list offered to them. Mirrored websites have in common with cluster-based Web sites that there are only a few number of replicas, which are more or less statically configured.

Server-Initiated Replicas

In contrast to permanent replicas, server-initiated replicas are copies of a data store that exist to enhance performance and which are created at the initiative of the (owner of the) data store. Consider, for example, a Web server placed in New York. Normally, this server can handle incoming requests quite easily, but it may happen that over a couple of days a sudden burst of requests come in from an unexpected location far from the server. In that case, it may be worthwhile to install a number of temporary replicas in regions where requests are coming from.

The problem of dynamically placing replicas is also being addressed in Web hosting services. These services offer a (relatively static) collection of servers spread across the Internet that can maintain and provide access to Web files belonging to third parties. To provide optimal facilities such hosting services can dynamically replicate files to servers where those files are needed to enhance performance, that is, close to demanding (groups of) clients.

Given that the replica servers are already in place, deciding where to place content is easier than in the case of server placement. An approach to dynamic replication of files in the case of a Web hosting service is described in Rabinovich et al. (1999). The algorithm is designed to support Web pages for which reason it assumes that updates are relatively rare compared to read requests. Using tiles as the unit of data, the algorithm works as follows.

The algorithm for dynamic replication takes two issues into account. First, replication can take place to reduce the load on a server. Second, specific files on a server can be migrated or replicated to servers placed in the proximity of clients that issue many requests for those files. In the following pages, we concentrate only on this second issue. We also leave out a number of details, which can be found in Rabinovich et al. (1999).

Each server keeps track of access counts per file, and where access requests come from. In particular, it is assumed that, given a client C, each server can determine which of the servers in the Web hosting service is closest to C. (Such information can be obtained, for example, from routing databases.) If client C1 and client C2 share the same "closest" server P, all access requests for file Fat server Q from eland C2 are jointly registered at Q as a single access count cntQ(P,F). This situation is shown in Fig. 7-18.

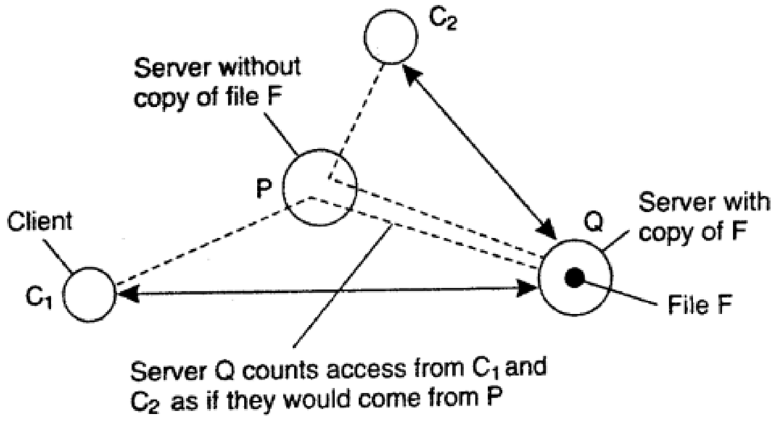

C_2
Server without copy of file F
P
Client
C_1
Q
Server with copy of F
File F
Server Q counts access from C_1 and C_2 as if they would come from P

Figure 7-18. Counting access requests from different elients.

When the number of requests for a specific file F at server S drops below a deletion threshold del (S,F), that file can be removed from S. As a consequence, the number of replicas of that file is reduced, possibly leading to higher work of each file continues to exist.

A replication threshold rep (5, F), which is always chosen higher than the deletion threshold, indicates that the number of requests for a specific file is so high that it may be worthwhile replicating it on another server. If the number of requests lie somewhere between the deletion and replication threshold, the file is allowed only to be migrated. In other words, in that case it is important to at least keep the number of replicas for that file the same.

When a server Q decides to reevaluate the placement of the files it stores, it checks the access count for each file. If the total number of access requests for F at Q drops below the deletion threshold del (Q,F), it will delete F unless it is the last copy. Furthermore, if for some server P, cntQ(p,F) exceeds more than half of the total requests for F at Q, server P is requested to take over the copy of F. In other words, server Q will attempt to migrate F to P.

Migration of file F to server P may not always succeed, for example, because P is already heavily loaded or is out of disk space. In that case, Q will attempt to replicate F on other servers. Of course, replication can take place only if the total number of access requests for F at Q exceeds the replication threshold rep (Q,F).

Server Q checks all other servers in the Web hosting service, starting with the one farthest away. If, for some server R, cntQ(R,F) exceeds a certain fraction of all re- quests for F at Q, an attempt is made to replicate F to R. Server-initiated replication continues to increase in popularity in time, especially in the context of Web hosting services such as the one just described. Note that as long as guarantees can be given that each data item is hosted by at least one server, it may suffice to use only server-initiated replication and not have any permanent replicas. Nevertheless, permanent replicas are still often useful as a back-up facility, or to be used as the only replicas that are allowed to be changed to guarantee consistency. Server-initiated replicas are then used for placing read-only copies close to clients.

Client-Initiated Replicas

An important kind of replica is the one initiated by a client. Client-initiated replicas are more commonly known as (client) caches. In essence, a cache is a local storage facility that is used by a client to temporarily store a copy of the data it has just requested. In principle, managing the cache is left entirely to the client.

The data store from where the data had been fetched has nothing to do with keeping cached data consistent. However, as we shall see, there are many occasions in which the client can rely on participation from the.data store to inform it when cached data has become stale.

Client caches are used only to improve access times to data. Normally, when a client wants access to some data, it connects to the nearest copy of the data store from where it fetches the data it wants to read, or to where it stores the data it had just modified. When most operations involve only reading data, performance can be improved by letting the client store requested data in a nearby cache. Such a cache could be located on the client's machine, or on a separate machine in the same local-area network as the client. The next time that same data needs to be read, the client can simply fetch it from this local cache. This scheme works fine as long as the fetched data has not been modified in the meantime.

Data is generally kept in a cache for a limited amount of time, for example, to prevent extremely stale data from being used, or simply to make room for other data. Whenever requested data can be fetched from the local cache, a cache bit is said to have occurred. To improve the number of cache hits, caches can be shared between clients. The underlying assumption is that a data request from client C1 may also be useful for a request from another nearby client C2·

Whether this assumption is correct depends very much on the type of data store. For example, in traditional file systems, data files are rarely shared at all (see, e.g., Muntz and Honeyman, 1992; and Blaze, 1993) rendering a shared cache useless. Likewise, it turns out that using Web caches to share data is also losing some ground, partly also because of the improvement in network and server performance. Instead, server-initiated replication schemes are becoming more effective.

Placement of client caches is relatively simple: a cache is normally placed on the same machine as its client, or otherwise on a machine shared by clients on the same local-area network. However, in some cases, extra levels of caching are introduced by system administrators by placing a shared cache between a number of departments or organizations, or even placing a shared cache for an entire region such as a province or country.

Yet another approach is to place (cache) servers at specific points in a wide area network and let a client locate the nearest server. When the server is located, it can be requested to hold copies of the data the client was previously fetching from somewhere else, as described in Noble et al. (1999). We will return to caching later in this chapter when discussing consistency protocols.

Content Distribution

Replica management also deals with propagation of (updated) content to the relevant replica servers. There are various trade-offs to make.

State versus Operations

An important design issue concerns what is actually to be propagated. Basically, there are three possibilities:

1. Propagate only a notification of an update.

2. Transfer data from one copy to another.

3. Propagate the update operation to other copies.

Propagating a notification is what invalidation protocols do. In an invalidation protocol, other copies are informed that an update has taken place and that the data they contain are no longer valid. The invalidation may specify which part of the data store has been updated, so that only part of a copy is actually invalidated.

The important issue is that no more than a notification is propagated. Whenever an operation on an invalidated copy is requested, that copy generally needs to be updated first, depending on the specific consistency model that is to be supported.

The main advantage of invalidation protocols is that they use little network bandwidth. The only information that needs to be transferred is a specification of which data are no longer valid. Such protocols generally work best when there are many update operations compared to read operations, that is, the read-to-write ratio is relatively small.

Consider, for example, a data store in which updates are propagated by sending the modified data to all replicas. If the size of the modified data is large, and updates occur frequently compared to read operations, we may have the situation that two updates occur after one another without any read operation being performed between them. Consequently, propagation of the first update to all replicas is effectively useless, as it will be overwritten by the second update. Instead, sending a notification that the data has been modified would have been more efficient.

Transferring the modified data among replicas is the second alternative, and is useful when the read-to-write ratio is relatively high. In that case, the probability that an update will be effective in the sense that the modified data will be read before the next update takes place is high. Instead of propagating modified data, it is also possible to log the changes and transfer only those logs to save bandwidth. In addition, transfers are often aggregated in the sense that multiple modifications are packed into a single message, thus saving communication overhead.

The third approach is not to transfer any data modifications at all, but to tell each replica which update operation it should perform (and send only the parameter values that those operations need). This approach, also referred to as active replication, assumes that each replica is represented by a process capable of "actively" keeping its associated data up to date by performing operations (Schneider, 1990). The main benefit of active replication is that updates can often be propagated at minimal bandwidth costs, provided the size of the parameters associated with an operation are relatively small. Moreover, the operations can be of arbitrary complexity, which may allow further improvements in keeping replicas consistent. On the other hand, more processing power may be required by each replica, especially in those cases when operations are relatively complex.

Pull versus Push Protocols

Another design issue is whether updates are pulled or pushed. In a push based approach, also referred to as server-based protocols, updates are propagated to other replicas without those replicas even asking for the updates.

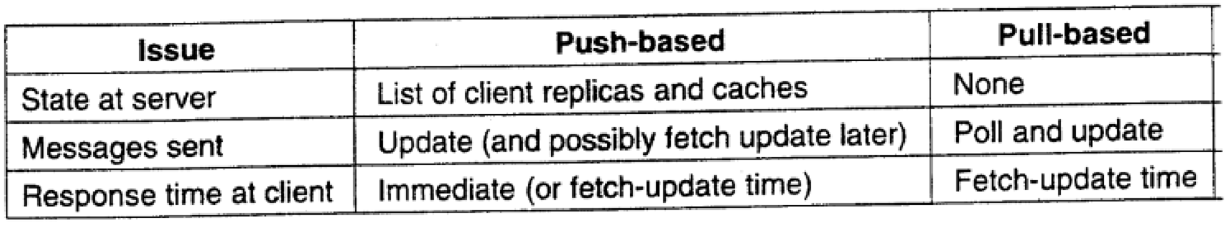
Push-based approaches are often used between permanent and server-initiated replicas, but can also be used to push updates to client caches. Server-based protocols are applied when replicas generally need to maintain a relatively high degree of consistency. In other words, replicas need to be kept identical.

This need for a high degree of consistency is related to the fact that permanent and server-initiated replicas, as well as large shared caches, are often shared by many clients, which, in tum, mainly perform read operations. Consequently, the read-to-update ratio at each replica is relatively high. In these cases, push-based protocols are efficient in the sense that every pushed update can be expected to be of use for one or more readers. In addition, push-based protocols make consistent data immediately available when asked for.

In contrast, in a pull-based approach, a server or client requests another server to send it any updates it has at that moment. Pull-based protocols, also called client-based protocols, are often used by client caches. For example, a common strategy applied to Web caches is first to check whether cached data items are still up to date. When a cache receives a request for items that are still locally available, the cache checks with the original Web server whether those data items have been modified since they were cached. In the case of a modification, the modified data are first transferred to the cache, and then returned to the requesting client. If no modifications took place, the cached data is returned. In other words, the client polls the server to see whether an update is needed.

A pull-based approach is efficient when the read-to-update ratio is relatively low. This is often the case with (nonshared) client caches, which have only one client. However, even when a cache is shared by many clients, a pull-based approach may also prove to be efficient when the cached data items are rarely shared. The main drawback of a pull-based strategy in comparison to a push-based approach is that the response time increases in the case of a cache miss.

When comparing push-based and pull-based solutions, there are a number of trade-offs to be made, as shown in Fig. 7-19. For simplicity, consider a client-server system consisting of a single, non-distributed server, and a number of client processes, each having their own cache.



Issue	Push-based	Pull-based
State at server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

Figure 7-19. A comparison between push-based and pull-based protocols in the case of multiple-client, single-server systems.

An important issue is that in push-based protocols, the server needs to keep track of all client caches. Apart from the fact that stateful servers are often less fault tolerant, as we discussed in Chap. 3, keeping track of all client caches may introduce a considerable overhead at the server. For example, in a push-based approach, a Web server may easily need to keep track of tens of thousands of client caches. Each time a Web page is updated, the server will need to go through its list of client caches holding a copy of that page, and subsequently propagate the update. Worse yet, if a client purges a page due to lack of space, it has to inform the server, leading to even more communication.

The messages that need to be sent between a client and the server also differ.

In a push-based approach, the only communication is that the server sends updates to each client. When updates are actually only invalidations, additional communication is needed by a client to fetch the modified data. In a pull-based approach, a client will have to poll the server, and, if necessary. fetch the modified data.

Finally, the response time at the client is also different. When a server pushes modified data to the client caches, it is clear that the response time at the client side is zero. When invalidations are pushed, the response time is the same as in the pull-based approach, and is determined by the time it takes to fetch the modified data from the server.

● Fault Tolerance: Introduction

To understand the role of fault tolerance in distributed systems we first need to take a closer look at what it actually means for a distributed system to tolerate faults. Being fault tolerant is strongly related to what are called dependable systems. Dependability is a term that covers a number of useful requirements for distributed systems including the following (Kopetz and Verissimo, 1993):

1. Availability

2. Reliability

3. Safety

4. Maintainability

Avail ability is defined as the property that a system is ready to be used immediately. In general, it refers to the probability that the system is operating correctly. at any given moment and is available to perform its functions on behalf of its users. In other words, a highly available system is one that will most likely be working at a given instant in time.

Reliability refers to the property that a system can run continuously without failure. In contrast to availability, reliability is defined in terms of a time interval instead of an instant in time. A highly-reliable system is one that will most likely continue to work without interruption during a relatively long period of time. This is a subtle but important difference when compared to availability. If a system goes down for one millisecond every hour, it has an availability of over 99.9999 percent, but is still highly unreliable. Similarly, a system that never crashes but is shut down for two weeks every August has high reliability but only 96 percent availability. The two are not the same.

Safety refers to the situation that when a system temporarily fails to operate correctly, nothing catastrophic happens. For example, many process control systems, such as those used for controlling nuclear power plants or sending people into space, are required to provide a high degree of safety. If such control systems temporarily fail for only a very brief moment, the effects could be disastrous.

Many examples from the past (and probably many more yet to come) show how hard it is to build safe systems.

Finally, maintainability refers to how easy a failed system can be repaired. A highly maintainable system may also show a high degree of availability, especially if failures can be detected and repaired automatically. However, as we shall see later in this chapter, automatically recovering from failures is easier said than done.

A system is said to fail when it cannot meet its promises. In particular, if a distributed system is designed to provide its users with a number of services, the system has failed when one or more of those services cannot be (completely) provided. An error is a part of a system's state that may lead to a failure. For example, when transmitting packets across a network, it is to be expected that some packets have been damaged when they arrive at the receiver. Damaged in this context means that the receiver may incorrectly sense a bit value (e.g., reading a 1 instead of a 0), or may even be unable to detect that something has arrived.

The cause of an error is called a fault. Clearly, finding out what caused an error is important. For example, a wrong or bad transmission medium may easily cause packets to be damaged. In this case, it is relatively easy to remove the fault.

However, transmission errors may also be caused by bad weather conditions such as in wireless networks. Changing the weather to reduce or prevent errors is a bit trickier.

Building dependable systems closely relates to controlling faults. A distinction can be made between preventing, removing, and forecasting faults (Avizienis et aI., 2004). For our purposes, the most important issue is fault tolerance, meaning that a system can provide its services even in the presence of faults. In other words, the system can tolerate faults and continue to operate normally.

Faults are generally classified as transient, intermittent, or permanent. Transient faults occur once and then disappear. If the operation is repeated, the fault goes away. A bird flying through the beam of a microwave transmitter may cause lost bits on some network (not to mention a roasted bird). If the transmission times out and is retried, it will probably work the second time.

An intermittent fault occurs, then vanishes of its own accord, then reappears, and so on. A loose contact on a connector will often cause an intermittent fault. Intermittent faults cause a great deal of aggravation because they are difficult to diagnose. Typically, when the fault doctor shows up, the system works fine.

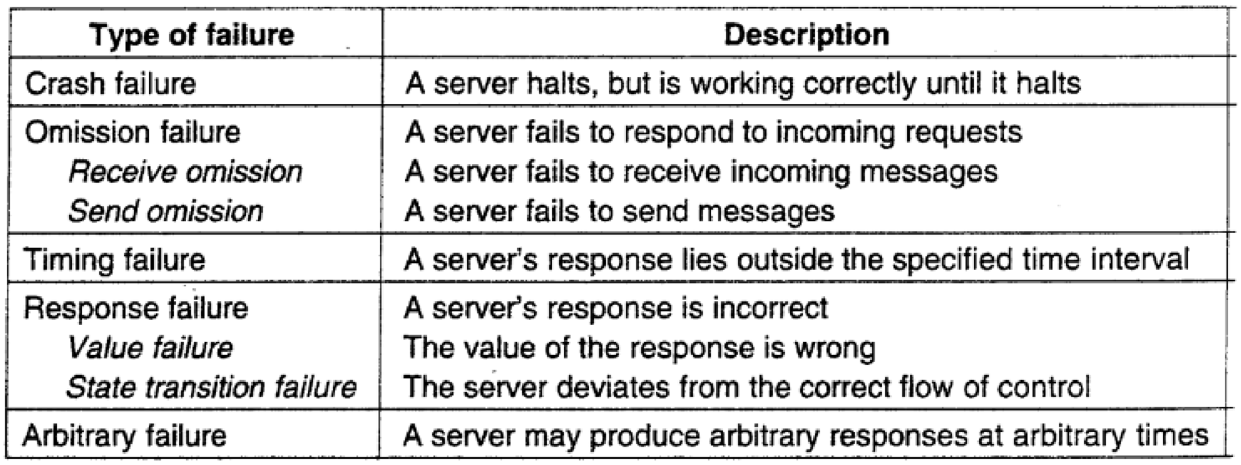
A permanent fault is one that continues to exist until the faulty component is replaced. Burnt-out chips, software bugs, and disk head crashes are examples of permanent faults.

Failure Models

A system that fails is not adequately providing the services it was designed for. If we consider a distributed system as a collection of servers that communicate with one another and with their clients, not adequately providing services means that servers, communication channels, or possibly both, are not doing what they are supposed to do. However, a malfunctioning server itself may not always be the fault we are looking for. If such a server depends on other servers to adequately provide its services, the cause of an error may need to be searched for somewhere else.

Such dependency relations appear in abundance in distributed systems. A failing disk may make life difficult for a file server that is designed to provide a highly available file system. If such a file server is part of a distributed database, the proper working of the entire database may be at stake, as only part of its data may be accessible.

To get a better grasp on how serious a failure actually is, several classification schemes have been developed. One such scheme is shown in Fig. 8-1, and is based on schemes described in Cristian (1991) and Hadzilacos and Toueg (1993).



Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure	A server's response is incorrect
Value failure	The value of the response is wrong
State transition failure	The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Figure 8-1. Different types of failures.

A crash failure occurs when a server prematurely halts, but was working correctly until it stopped. An important aspect of crash failures is that once the server has halted, nothing is heard from it anymore. A typical example of a crash failure is an operating system that comes to a grinding halt, and for which there is only one solution: reboot it. Many personal computer systems suffer from crash failures so often that people have come to expect them to be normal.

Consequently, moving the reset button from the back of a cabinet to the front was done for good reason. Perhaps one day it can be moved to the back again, or even removed altogether.

An omission failure occurs when a server fails to respond to a request. Several things might go wrong. In the case of a receive omission failure, possibly the server never got the request in the first place. Note that it may well be the case that the connection between a client and a server has been correctly established, but that there was no thread listening to incoming requests. Also, a receive omission failure will generally not affect the current state of the server, as the server is unaware of any message sent to it.

Likewise, a send omission failure happens when the server has done its work, but somehow fails in sending a response. Such a failure may happen, for example, when a send buffer overflows while the server was not prepared for such a situation. Note that, in contrast to a receive omission failure, the server may now be in a state reflecting that it has just completed a service for the client. As a consequence, if the sending of its response fails, the server has to be prepared for the client to reissue its previous request.

Other types of omission failures not related to communication may be caused by software errors such as infinite loops or improper memory management by which the server is said to "hang." Another class of failures is related to timing. Timing failures occur when the response lies outside a specified real-time interval. As we saw with isochronous data streams in Chap. 4, providing data too soon may easily cause trouble for a recipient if there is not enough buffer space to hold all the incoming data. More common, however, is that a server responds too late, in which case a performance failure is said to occur.

A serious type of failure is a response failure, by which the server's response is simply incorrect. Two kinds of response failures may happen. In the case of a value failure, a server simply provides the wrong reply to a request. For example, a search engine that systematically returns Web pages not related to any of the search terms used. has failed.

The other type of response failure is known as a state transition failure. This kind of failure happens when the server reacts unexpectedly to an incoming request. For example, if a server receives a message it cannot recognize, a state transition failure happens if no measures have been taken to handle such messages. In particular, a faulty server may incorrectly take default actions it should never have initiated.

The most serious are arbitrary failures, also known as Byzantine failures. In effect, when arbitrary failures occur, clients should be prepared for the worst. In particular, it may happen that a server is producing output it should never have produced, but which cannot be detected as being incorrect. Worse yet a faulty server may even be maliciously working together with other servers to produce intentionally wrong answers. This situation illustrates why security is also considered an important requirement when talking about dependable systems. The term "Byzantine" refers to the Byzantine Empire, a time (330-1453) and place (the Balkans and modern Turkey) in which endless conspiracies, intrigue, and untruthfulness were alleged to be common in ruling circles. Byzantine faults were first analyzed by Pease et al. (1980) and Lamport et al. (1982). We return to such failures below.

Arbitrary failures are closely related to crash failures. The definition of crash failures as presented above is the most benign way for a server to halt. They are also referred to as fail-stop failures. In effect, a fail-stop server will simply stop producing output in such a way that its halting can be detected by other processes.

In the best case, the server may have been so friendly to announce it is about to crash; otherwise it simply stops. Of course, in real life, servers halt by exhibiting omission or crash failures, and are not so friendly as to announce in advance that they are going to stop. It is up to the other processes to decide that a server has prematurely halted. However, in such fail-silent systems, the other process may incorrectly conclude that a server has halted. Instead, the server may just be unexpectedly slow, that is, it is exhibiting performance failures.

Finally, there are also occasions in which the server is producing random output, but this output can be recognized by other processes as plain junk. The server is then exhibiting arbitrary failures, but in a benign way. These faults are also referred to as being fail-safe.

If a system is to be fault tolerant, the best it can do is to try to hide the occurrence of failures from other processes. The key technique for masking faults is to use redundancy. Three kinds are possible: information redundancy, time redundancy, and physical redundancy [see also Johnson (1995)]. With information redundancy, extra bits are added to allow recovery from garbled bits. For example, a Hamming code can be added to transmitted data to recover from noise on the transmission line.

With time redundancy, an action is performed, and then. if need be, it is performed again. Transactions (see Chap. 1) use this approach. If a transaction aborts, it can be redone with no harm. Time redundancy is especially helpful when the faults are transient or intermittent.

With physical redundancy, extra equipment or processes are added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components. Physical redundancy can thus be done either in hardware or in software. For example, extra processes can be added to the system so that if a small number of them crash, the system can still function correctly. In other words, by replicating processes, a high degree of fault tolerance may be achieved. We return to this type of software redundancy below.

Physical redundancy is a well-known technique for providing fault tolerance. It is used in biology (mammals have two eyes, two ears, two lungs, etc.), aircraft (747s have four engines but can fly on three), and sports (multiple referees in case one misses an event). It has also been used for fault tolerance in electronic circuits for years; it is illustrative to see how it has been applied there. Consider, for example, the circuit of Fig. 8-2(a). Here signals pass through devices A, B, and C, in sequence. If one of them is faulty, the final result will probably be incorrect.

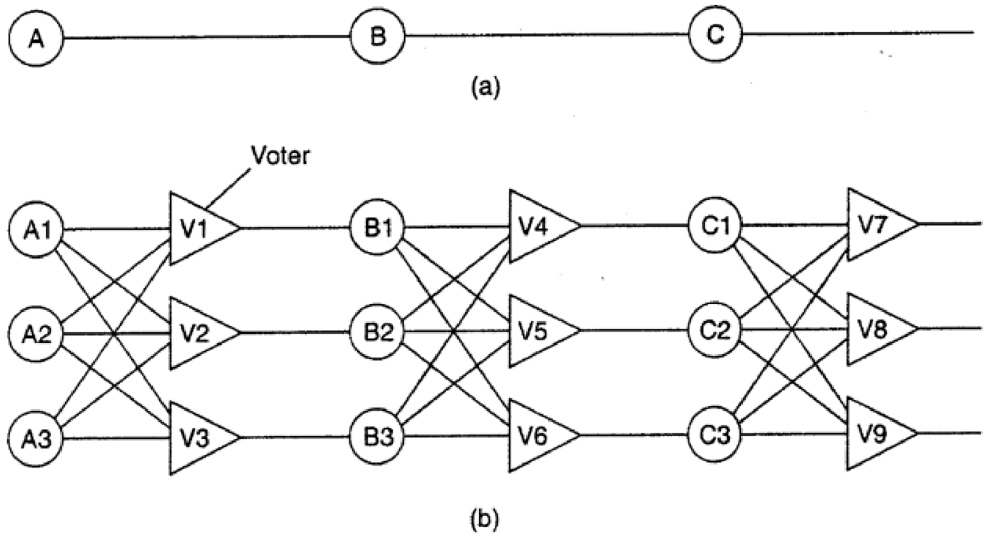


Figure 8-2. Triple modular redundancy.

In Fig. 8-2(b), each device is replicated three times. Following each stage in the circuit is a triplicated voter. Each voter is a circuit that has three inputs and one output. If two or three of the inputs are the same, the output is equal to that input. If all three inputs are different, the output is undefined. This kind of design is known as TMR (Triple Modular Redundancy).

Suppose that element Az fails. Each of the voters, Vb Vz, and V3 gets two good (identical) inputs and one rogue input, and each of them outputs the correct value to the second stage. In essence, the effect of Az failing is completely masked, so that the inputs to B I, Bz, and B3 are exactly the same as they would have been had no fault occurred.

Now consider what happens if B3 and C1 are also faulty, in addition to Az. These effects are also masked, so the three final outputs are still correct. At first it may not be obvious why three voters are needed at each stage. After all, one voter could also detect and pass though the majority view.

However, a voter is also a component and can also be faulty. Suppose, for example, that voter V I malfunctions. The input to B I will then be wrong, but as long as everything else works, Bz and B3 will produce the same output and V4, Vs, and V6 will all produce the correct result into stage three. A fault in VI is effectively no different than a fault in B I.In both cases B I produces incorrect output, but in both cases it is voted down later and the final result is still correct.

● Process resilience

The key approach to tolerating a faulty process is to organize several identical processes into a group. The key property that all groups have is that when a message is sent to the group itself, all members of the group receive it. In this way, if one process in a group fails, hopefully some other process can take over for it. Process groups may be dynamic. New groups can be created and old groups can be destroyed. A process can join a group or leave one during system operation. A process can be a member of several groups at the same time. Consequently, mechanisms are needed for managing groups and group membership.

Groups are roughly analogous to social organizations. Alice might be a member of a book club, a tennis club, and an environmental organization. On a particular day, she might receive mailings (messages) announcing a new birthday cake cookbook from the book club, the annual Mother's Day tennis tournament from the tennis club, and the start of a campaign to save the Southern groundhog from the environmental organization. At any moment, she is free to leave any or all of these groups, and possibly join other groups.

The purpose of introducing groups is to allow processes to deal with collections of processes as a single abstraction. Thus a process can send a message to a group of servers without having to know who they are or how many there are or where they are, which may change from one call to the next.

Flat Groups versus Hierarchical Groups

An important distinction between different groups has to do with their internal structure. In some groups, all the processes are equal. No one is the boss and all decisions are made collectively. In other groups, some kind of hierarchy exists. For example, one process is the coordinator and all the others are workers. In this model, when a request for work is generated, either by an external client or by one of the workers, it is sent to the coordinator. The coordinator then decides which worker is best suited to carry it out, and forwards it there. More complex hierarchies are also possible, of course. These communication patterns are illustrated in Fig. 8-3.

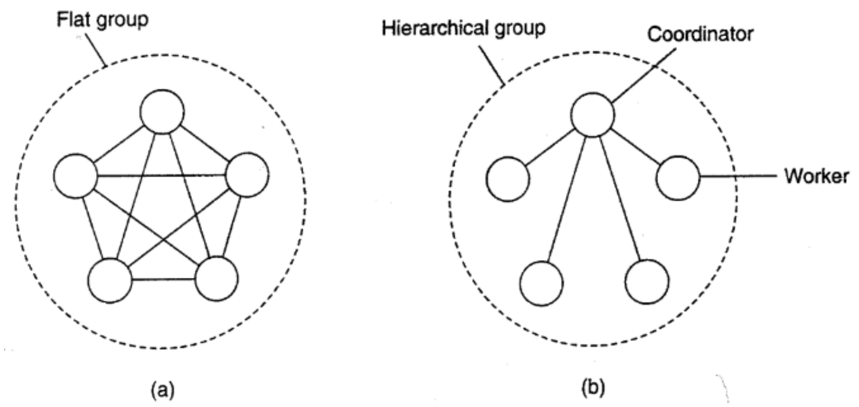


Figure 8-3. (a) Communication in a flat group. (b) Communication in a simple hierarchical group.

Each of these organizations has its own advantages and disadvantages. The flat group is symmetrical and has no single point of failure. If one of the processes crashes, the group simply becomes smaller, but can otherwise continue. A disadvantage is that decision making is more complicated. For example, to decide anything, a vote often has to be taken, incurring some delay and overhead.

The hierarchical group has the opposite properties. Loss of the coordinator brings the entire group to a grinding halt, but as long as it is running, it can make decisions without bothering everyone else.

Group Membership

When group communication is present, some method is needed for creating and deleting groups, as well as for allowing processes to join and leave groups. One possible approach is to have a group server to which all these requests can be sent. The group server can then maintain a complete database of all the groups and their exact membership. This method is straightforward, efficient, and fairly easy to implement. Unfortunately, it shares a major disadvantage with all centralized techniques: a single point of failure. If the group server crashes, group management ceases to exist. Probably most or all groups will have to be reconstructed from scratch, possibly terminating whatever work was going on.

The opposite approach is to manage group membership in a distributed way.

For example, if (reliable) multicasting is available, an outsider can send a message to all group members announcing its wish to join the group.

Ideally, to leave a group, a member just sends a goodbye message to everyone. In the context of fault tolerance, assuming fail-stop semantics is generally not appropriate. The trouble is, there is no polite announcement that a process crashes as there is when a process leaves voluntarily. The other members have to discover this experimentally by noticing that the crashed member no longer responds to anything. Once it is certain that the crashed member is really down (and not just slow), it can be removed from the group.

Another knotty issue is that leaving and joining have to be synchronous with data messages being sent. In other words, starting at the instant that a process has joined a group, it must receive all messages sent to that group. Similarly, as soon as a process has left a group, it must not receive any more messages from the group, and the other members must not receive any more messages from it. One way of making sure that a join or leave is integrated into the message stream at the right place is to convert this operation into a sequence of messages sent to the whole group.

One final issue relating to group membership is what to do if so many machines go down that the group can no longer function-at all. Some protocol is needed to rebuild the group. Invariably, some process will have to take the initiative to start the ball rolling, but what happens if two or three try at the same time?

The protocol must be able to withstand this.

Failure Masking and Replication

Process groups are part of the solution for building fault-tolerant systems. In particular, having a group of identical processes allows us to mask one or more faulty processes in that group. In other words, we can replicate processes and organize them into a group to replace a single (vulnerable) process with a (fault tolerant) group. As discussed in the previous chapter, there are two ways to approach such replication: by means of primary-based protocols, or through replicated-write protocols.

Primary-based replication in the case of fault tolerance generally appears in the form of a primary-backup protocol. In this case, a group of processes is organized in a hierarchical fashion in which a primary coordinates all write operations.

In practice, the primary is fixed, although its role can be taken over by one of the backups. if need be. In effect, when the primary crashes, the backups execute some election algorithm to choose a new primary.

An important issue with using process groups to tolerate faults is how much replication is needed. To simplify our discussion, let us consider only replicated write systems. A system is said to be k fault tolerant if it can survive faults in k components and still meet its specifications. If the components, say processes, fail silently, then having k + 1 of them is enough to provide k fault tolerance. If k of them simply stop, then the answer from the other one can be used.

On the other hand, if processes exhibit Byzantine failures, continuing to run when sick and sending out erroneous or random replies, a minimum of 2k + 1 processors are needed to achieve k fault tolerance. In the worst case, the k failing processes could accidentally (or even intentionally) generate the same reply.

However, the remaining k + 1 will also produce the same answer, so the client or voter can just believe the majority. Of course, in theory it is fine to say that a system is k fault tolerant and just let the k + I identical replies outvote the k identical replies, but in practice it is hard to imagine circumstances in which one can say with certainty that k processes can fail but k + 1 processes cannot fail. Thus even in a fault-tolerant system some kind of statistical analysis may be needed.

Failure Detection

It may have become clear from our discussions so far that in order to properly mask failures, we generally need to detect them as well. Failure detection is one of the cornerstones of fault tolerance in distributed systems. What it all boils down to is that for a group of processes, non faulty members should be able to decide who is still a member, and who is not. In other words, we need to be able to detect when a member has failed.

When it comes to detecting process failures, there are essentially only two mechanisms. Either processes actively send "are you alive?" messages to each other (for which they obviously expect an answer), or passively wait until messages come in from different processes. The latter approach makes sense only when it can be guaranteed that there is enough communication between processes.

In practice, actively pinging processes is usually followed.

There has been a huge body of theoretical work on failure detectors. What it all boils down to is that a timeout mechanism is used to check whether a process has failed. In real settings, there are two major problems with this approach. First, due to unreliable networks, simply stating that a process has failed because it does not return an answer to a ping message may be wrong. In other words, it is quite easy to generate false positives. If a false positive has the effect that a perfectly a healthy process is removed from a membership list, then clearly we are doing something wrong.

Another serious problem is that timeouts are just plain crude. As noticed by Birman (2005), there is hardly any work on building proper failure detection subsystems that take more into account than only the lack of a reply to a single message. This statement is even more evident when looking at industry-deployed distributed systems.

There are various issues that need to be taken into account when designing a failure detection subsystem [see also Zhuang et al. (2005)]. For example, failure detection can take place through gossiping in which each node regularly announces to its neighbors that it is still up and running. As we mentioned, an alternative is to let nodes actively probe each other.

Failure detection can also be done as a side-effect of regularly exchanging information with neighbors, as is the case with gossip-based information dissemination (which we discussed in Chap. 4). This approach is essentially also adopted in Obduro (Vogels, 2003): processes periodically gossip about their service availability.

This information is gradually disseminated through the network by gossiping. Eventually, every process will know about every other process, but more importantly, will have enough information locally available to decide whether a process has failed or not. A member for which the availability information is old, will presumably have failed.

Another important issue is that a failure detection subsystem should ideally be able to distinguish network failures from node failures. One way of dealing with this problem is not to let a single node decide whether one of its neighbors has crashed. Instead, when noticing a timeout on a ping message, a node requests other neighbors to see whether they can reach the presumed failing node. Of course, positive information can also be shared: if a node is still alive, that information can be forwarded to other interested parties (who may be detecting a link failure to the suspected node).

This brings us to another key issue: when a member failure is detected, how should other non faulty processes be informed? One simple, and somewhat radical approach is the one followed in FUSE (Dunagan et al., 2004). In FUSE, processes can be joined in a group that spans a wide-area network. The group members create a spanning tree that is used for monitoring member failures. Members send ping messages to their neighbors. When a neighbor does not respond, the pinging node immediately switches to a state in which it will also no longer respond to pings from other nodes. By recursion, it is seen that a single node failure is rapidly promoted to a group failure notification. FUSE does not suffer a lot from link failures for the simple reason that it relies on point-to-point TCP connections between group members.

● Reliable client-server communication

In many cases, fault tolerance in distributed systems concentrates on faulty processes. However, we also need to consider communication failures. Most of the failure models discussed previously apply equally well to communication channels. In particular, a communication channel may exhibit crash, omission, timing, and arbitrary failures. In practice, when building reliable communication channels, the focus is on masking crashes and omission failures. Arbitrary failures may occur in the form of duplicate messages, resulting from the fact that in a computer network messages may be buffered for a relatively long time, and are reinjected into the network after the original sender has already issued a retransmission.

Point-to-Point Communication

In many distributed systems, reliable point-to-point communication is established by making use of a reliable transport protocol, such as TCP. TCP masks omission failures, which occur in the form of lost messages, by using acknowledgments and retransmissions. Such failures are completely hidden from a TCP client.

However, crash failures of connections are not masked. A crash failure may occur when (for whatever reason) a TCP connection is abruptly broken so that no more messages can be transmitted through the channel. In most cases, the client is informed that the channel has crashed by raising an exception. The only way to mask such failures is to let the distributed system attempt to automatically set up a new connection, by simply resending a connection request. The underlying assumptions that the other side is still, or again, responsive to such requests.

RPC Semantics in the Presence of Failures

Let us now take a closer look at client-server communication when using high-level communication facilities such as Remote Procedure Calls (RPCs). The goal of RPC is to hide communication by making remote procedure calls look just like local ones. With a few exceptions, so far we have come fairly close. Indeed, as long as both client and server are functioning perfectly, RPC does its job well.

The problem comes about when errors occur. It is then that the differences between local and remote calls are not always easy to mask.

To structure our discussion, let us distinguish between five different classes of failures that can occur in RPC systems, as follows:

1. The client is unable to locate the server.

2. The request message from the client to the server is lost.

3. The server crashes after receiving a request.

4. The reply message from the server to the client is lost.

5. The client crashes after sending a request.

Each of these categories poses different problems and requires different solutions.

Client Cannot Locate the Server

To start with, it can happen that the client cannot locate a suitable server. All servers might be down, for example. Alternatively, suppose that the client is compiled using a particular version of the client stub, and the binary is not used for a considerable period of time. In the meantime, the server evolves and a new version of the interface is installed; new stubs are generated and put into use. When the client is eventually run, the binder will be unable to match it up with a server and will report failure. While this mechanism is used to protect the client from accidentally trying to talk to a server that may not agree with it in terms of what parameters are required or what it is supposed to do, the problem remains of how this failure should be dealt with.

One possible solution is to have the error raise an exception. In some languages, (e.g., Java), programmers can write special procedures that are invoked upon specific errors, such as division by zero. In C, signal handlers can be used for this purpose. In other words, we could define a new signal type SIGNO SERVER, and allow it to be handled in the same way as other signals.

This approach, too, has drawbacks. To start with, not every language has exceptions or signals. Another point is that having to write an exception or signal handler destroys the transparency we have been trying to achieve. Suppose that you are a programmer and your boss tells you to write the sum procedure. You smile and tell her it will be written, tested, and documented in five minutes. Then she mentions that you also have to write an exception handler as well, just in case the procedure is not there today. At this point it is pretty hard to maintain the illusion that remote procedures are no different from local ones, since writing an exception handler for "Cannot locate server" would be a rather unusual request in a single-processor system. So much for transparency.

Lost Request Messages

The second item on the list is dealing with lost request messages. This is the easiest one to deal with: just have the operating system or client stub start a timer when sending the request. If the timer expires before a reply or acknowledgment comes back, the message is sent again. If the message was truly lost, the server will not be able to tell the difference between the retransmission and the original, and everything will work fine. Unless, of course, so many request messages are lost that the client gives up and falsely concludes that the server is down, in which case we are back to "Cannot locate server." If the request was not lost, the only thing we need to do is let the server be able to detect it is dealing with a retransmission. Unfortunately, doing so is not so simple, as we explain when discussing lost replies.

Server Crashes

The next failure on the list is a server crash. The normal sequence of events at a server is shown in Fig. 8-7(a). A request arrives, is carried out, and a reply is sent. Now consider Fig. 8-7(b). A request arrives and is carried out, just as before, but the server crashes before it can send the reply. Finally, look at Fig. 8-7(c). Again a request arrives, but this time the server crashes before it can even be carried out. And, of course, no reply is sent back.

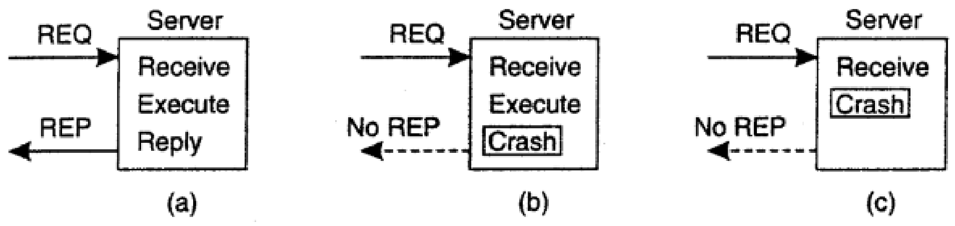


Figure 8-7. A server in client-server communication. (a) The normal case. (b) Crash after execution. (c) Crash before execution.

The annoying part of Fig. 8-7 is that the correct treatment differs for (b) and (c). In (b) the system has to report failure back to the client (e.g., raise, an exception), whereas in (c) it can just retransmit the request. The problem is that the client's operating system cannot tell which is which. All it knows is that its timer has expired.

Assume that the server crashes and subsequently recovers. It announces to all clients that it has just crashed but is now up and running again. The problem is that the client does not know whether its request to print some text will actually be carried out.

There are four strategies the client can follow. First, the client can decide to never reissue a request, at the risk that the text will not be printed. Second, it can decide to always reissue a request, but this may lead to its text being printed twice. Third, it can decide to reissue a request only if it did not yet receive an acknowledgment that its print request had been delivered to the server. In that case, the client is counting on the fact that the server crashed before the print request could be delivered. The fourth and last strategy is to reissue a request only if it has received an acknowledgment for the print request.

With two strategies for the server, and four for the client, there are a total of eight combinations to consider. Unfortunately, no combination is satisfactory. To explain, note that there are three events that can happen at the server: send the completion message (M), print the text (P), and crash (C). These events can occur in six different orderings:

1. M ~P ~C: A crash occurs after sending the completion message and printing the text.

2. M ~C (~P): A crash happens after sending the completion message, but before the text could be printed.

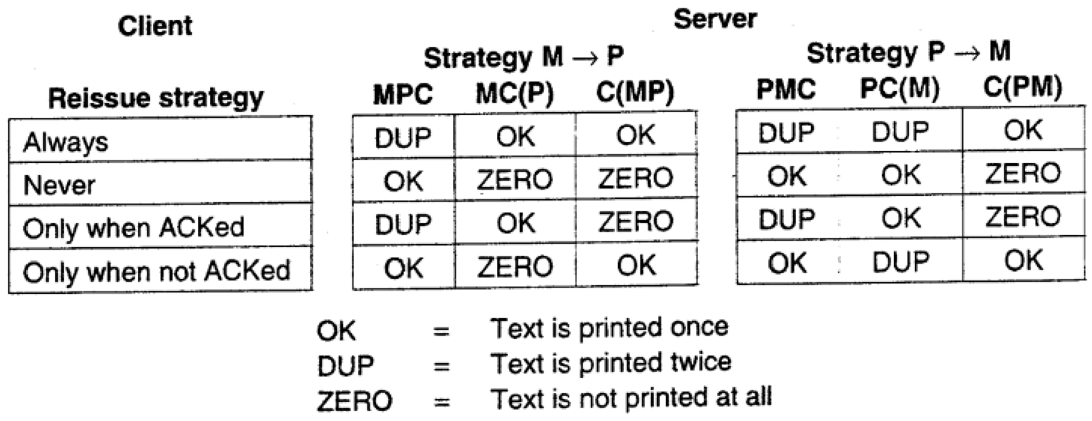
3. p ~M ~C: A crash occurs after sending the completion message and printing the text.

4. P~C(~M): The text printed, after which a crash occurs before the completion message could be sent.

5. C (~P ~M): A crash happens before the server could do anything.

6. C(~M ~P): A crash happens before the server could do anything.

The parentheses indicate an event that can no longer happen because the server already crashed. Fig. 8-8 shows all possible combinations. As can be readily verified, there is no combination of client strategy and server strategy that will work correctly under all possible event sequences. The bottom line is that the client can never know whether the server crashed just before or after having the text printed.



Client	Server					
	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK	=	Text is printed once
DUP	=	Text is printed twice
ZERO	=	Text is not printed at all

Figure 8-8. Different combinations of client and server strategies in the presence of server crashes.

In short, the possibility of server crashes radically changes the nature of RPC and clearly distinguishes single-processor systems from distributed systems. In the former case, a server crash also implies a client crash, so recovery is neither possible nor necessary. In the latter it is both possible and necessary to take action.

Lost Reply Messages

Lost replies can also be difficult to deal with. The obvious solution is just to rely on a timer again that has been set by the client's operating system. If no reply is forthcoming within a reasonable period, just send the request once more. The trouble with this solution is that the client is not really sure why there was no answer. Did the request or reply get lost, or is the server merely slow? It may make a difference.

In particular, some operations can safely be repeated as often as necessary with no damage being done. A request such as asking for the first 1024 bytes of a file has no side effects and can be executed as often as necessary without any harm being done. A request that has this property is said to be idempotent.

Now consider a request to a banking server asking to transfer a million dollars from one account to another. If the request arrives and is carried out, but the reply is lost, the client will not know this and will retransmit the message. The bank server will interpret this request as a new one, and will carry it out too. Two million dollars will be transferred. Heaven forbid that the reply is lost 10 times. Transferring money is not idempotent.

One way of solving this problem is to try to structure all the requests in an idempotent way. In practice, however, many requests (e.g., transferring money) are inherently non idempotent, so something else is needed. Another method is to have the client assign each request a sequence number. By having the server keep track of the most recently received sequence number from each client that is using it, the server can tell the difference between an original request and a retransmission and can refuse to carry out any request a second time. However, the server will still have to send a response to the client. Note that this approach does require that the server maintains administration on each client. Furthermore, it is not clear how long to maintain this administration. An additional safeguard is to have a bit in the message header that is used to distinguish initial requests from retransmissions.

Client Crashes

The final item on the list of failures is the client crash. What happens if a client sends a request to a server to do some work and crashes before the server replies? At this point a computation is active and no parent is waiting for the result. Such an unwanted computation is called an orphan.

Orphans can cause a variety of problems that can interfere with normal operation of the system. As a bare minimum, they waste CPU cycles. They can also lock files or otherwise tie up valuable resources. Finally, if the client reboots and does the RPC again, but the reply from the orphan comes back immediately afterward, confusion can result.

What can be done about orphans? Nelson (1981) proposed four solutions. In solution 1, before a client stub sends an RPC message, it makes a log entry telling what it is about to do. The log is kept on disk or some other medium that survives crashes. After a reboot, the log is checked and the orphan is explicitly killed off.

This solution is called orphan extermination.

The disadvantage of this scheme is the horrendous expense of writing a disk record for every RPC. Furthermore, it may not even work, since orphans themselves may do RPCs, thus creating grand orphans or further descendants that are difficult or impossible to locate. Finally, the network may be partitioned, due to a failed gateway, making it impossible to kill them, even if they can be located. All in all, this is not a promising approach.

In solution 2. called reincarnation, all these problems can be solved without the need to write disk records. The way it works is to divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message to all machines declaring the start of a new epoch. When such a broadcast comes in, all remote computations on behalf of that client are killed. Of course, if the network is partitioned, some orphans may survive. Fortunately, however, when they report back, their replies will contain an obsolete epoch number, making them easy to detect.

Solution 3 is a variant on this idea, but somewhat less draconian. It is called gentle reincarnation. When an epoch broadcast comes in, each machine checks to see if it has any remote computations running locally, and if so, tries its best to locate their owners. Only if the owners cannot be located anywhere is the computation killed.

Finally, we have solution 4, expiration, in which each RPC is given a standard amount of time, T, to do the job. If it cannot finish, it must explicitly ask for another quantum, which is a nuisance. On the other hand, if after a crash the client waits a time T before rebooting, all orphans are sure to be gone. The problem to be solved here is choosing a reasonable value of Tin the face of RPCs with wildly differing requirements.

● Reliable group communication

Basic Reliable-Multicasting Schemes

Although most transport layers offer reliable point-to-point channels, they rarely offer reliable communication to a collection of processes. The best they can offer is to let each process set up a point-to-point connection to each other process it wants to communicate with. Obviously, such an organization is not very efficient as it may waste network bandwidth. Nevertheless, if the number of processes is small, achieving reliability through multiple reliable point-to-point channels is a simple and often straightforward solution.

To go beyond this simple case, we need to define precisely what reliable multicasting is. Intuitively, it means that a message that is sent to a process group should be delivered to each member of that group. However, what happens if during communication a process joins the group? Should that process also receive the message? Likewise, we should also determine what happens if a (sending) process crashes during communication.

To cover such situations, a distinction should be made between reliable communication in the presence of faulty processes, and reliable communication when processes are assumed to operate correctly. In the first case, multicasting is considered to be reliable when it can be guaranteed that all non faulty group members receive the message. The tricky part is that agreement should be reached on what the group actually looks like before a message can be delivered, in addition to various ordering constraints. We return to these matters when we discuss atomic multicasts below.

The situation becomes simpler if we assume agreement exists on who is a member of the group and who is not. In particular, if we assume that processes do not fail, and processes do not join or leave the group while communication is going on, reliable multicasting simply means that every message should be delivered to each current group member. In the simplest case, there is no requirement that all group members receive messages in the same order, but sometimes this feature is needed.

This weaker form of reliable multicasting is relatively easy to implement again subject to the condition that the number of receivers is limited. Consider the case that a single sender wants to multicast a message to multiple receivers.

Assume that the underlying communication system offers only unreliable multicasting, meaning that a multicast message may be lost part way and delivered to some, but not all, of the intended receivers.

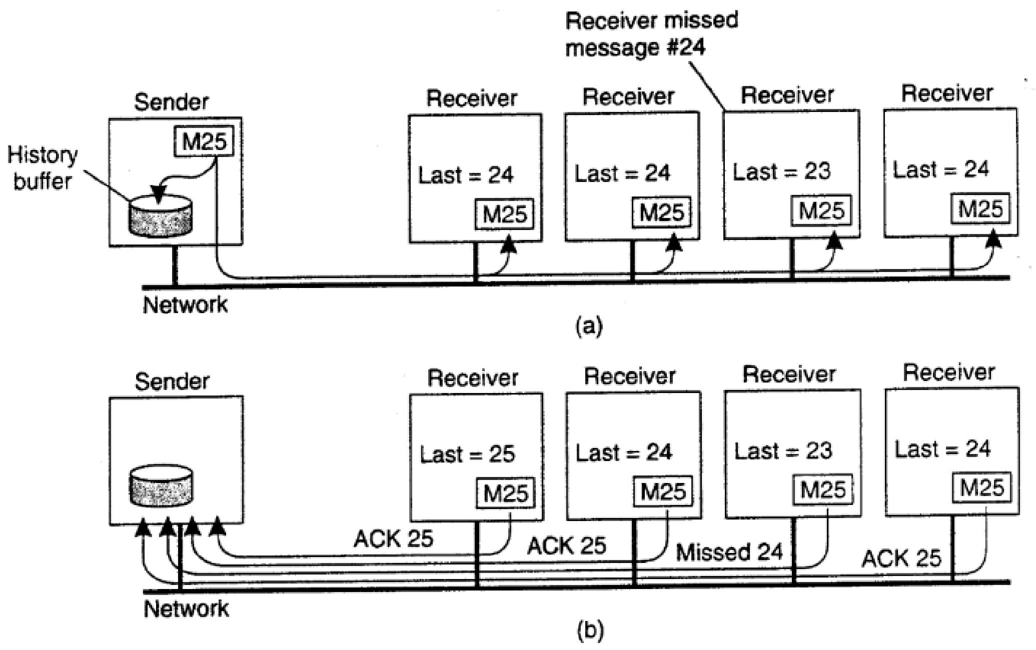


Figure 8-9. A simple solution to reliable multicasting when all receivers are known and are assumed not to fail. (a) Message transmission. (b) Reporting feedback..

A simple solution is shown in Fig. 8-9. The sending process assigns a sequence number to each message it multicasts. We assume that messages are received in the order they are sent. In this way, it is easy for a receiver to detect it is missing a message. Each multicast message is stored locally in a history buffer at the sender. Assuming the receivers are known to the sender, the sender simply keeps the message in its history buffer until each receiver has returned an acknowledgment. If a receiver detects it is missing a message, it may return a negative acknowledgement, requesting the sender for a retransmission. Alternatively, the sender may automatically retransmit the message when it has not received all acknowledgments within a certain time.

There are various design trade-offs to be made. For example, to reduce the number of messages returned to the sender, acknowledgments could possibly be piggybacked with other messages. Also, retransmitting a message can be done using point-to-point communication to each requesting process, or using a single multicast message sent to all processes.

Scalability in Reliable Multicasting

The main problem with the reliable multicast scheme just described is that it cannot support large numbers of receivers. If there are N receivers, the sender must be prepared to accept at least N acknowledgments. With many receivers, the sender may be swamped with such feedback messages, which is also referred to as a feedback implosion. In addition, we may also need to take into account that the receivers are spread across a wide-area network.

One solution to this problem is not to have receivers acknowledge the receipt of a message. Instead, a receiver returns a feedback message only to inform the sender it is missing a message. Returning only such negative acknowledgments can be shown to generally scale better [see, for example, Towsley et al. (1997)]~ but no hard guarantees can be given that feedback implosions will never happen.

Another problem with returning only negative acknowledgments is that the sender will, in theory, be forced to keep a message in its history buffer forever. Because the sender can never know if a message has been correctly delivered to all receivers, it should always be prepared for a receiver requesting the retransmission of an old message. In practice, the sender will remove a message from its history buffer after some time has elapsed to prevent the buffer from overflowing.

However, removing a message is done at the risk of a request for a retransmission not being honored. Several proposals for scalable reliable multicasting exist. A comparison between different schemes can be found in Levine and Garcia-Luna-Aceves (1998).

We now briefly discuss two very different approaches that are representative of many existing solutions.

1. Nonhierarchical Feedback Control
2. Hierarchical Feedback Control

Nonhierarchical Feedback Control

The key issue to scalable solutions for reliable multicasting is to reduce the number of feedback messages that are returned to the sender. A popular model that has been applied to several wide-area applications is feedback suppression. This scheme underlies the Scalable Reliable Multicasting (SRM) protocol developed by Floyd et al. (1997) and works as follows.

First, in SRM, receivers never acknowledge the successful delivery of a multicast message, but instead, report only when they are missing a message. How message loss is detected is left to the application. Only negative acknowledgments are returned as feedback. Whenever a receiver notices that it missed a message, it multicasts its feedback to the rest of the group.

Multicasting feedback allows another group member to suppress its own feedback. Suppose several receivers missed message m. Each of them will need to return a negative acknowledgment to the sender, S, so that m can be retransmitted.

However, if we assume that retransmissions are always multicast to the entire group, it is sufficient that only a single request for retransmission reaches S.

For this reason, a receiver R that did not receive message 111 schedules a feedback message with some random delay. That is, the request for retransmission is not sent until some random time has elapsed. If, in the meantime, another request for retransmission for m reaches R, R will suppress its own feedback, knowing that m will be retransmitted shortly. In this way, ideally, only a single feedback message will reach S, which in turn subsequently retransmits m. This scheme is shown in Fig. 8-10.

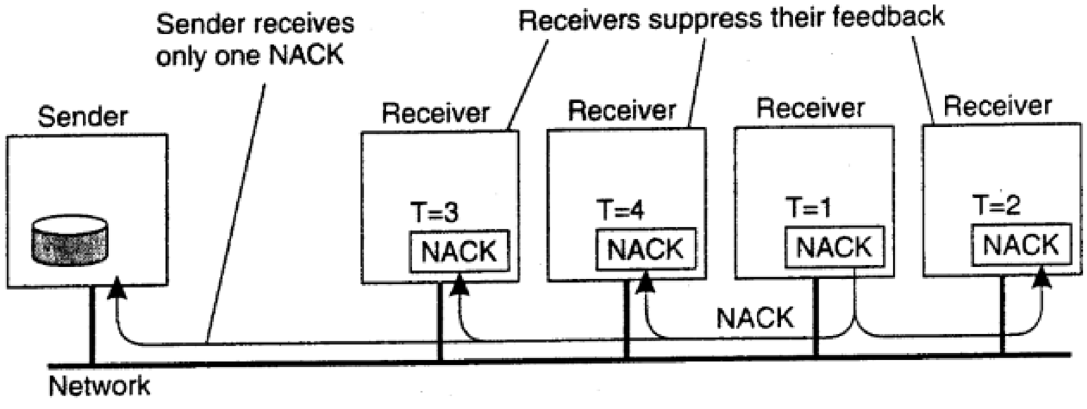


Figure 8·10. Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others.

Feedback suppression has shown to scale reasonably well, and has been used as the underlying mechanism for a number of collaborative Internet applications, such as a shared whiteboard. However, the approach also introduces a number of serious problems. First, ensuring that only one request for retransmission is returned to the sender requires a reasonably accurate scheduling of feedback messages at each receiver. Otherwise, many receivers will still return their feedback at the same time. Setting timers accordingly in a group of processes that is dispersed across a wide-area network is not that easy. Another problem is that multicasting feedback also interrupts those processes to which the message has been successfully delivered. In other words, other receivers are forced to receive and process messages that are useless to them. The only solution to this problem is to let receivers that have not received message 111 join a separate multicast group for m, as explained in Kasera et al. (1997). Unfortunately, this solution requires that groups can be managed in a highly efficient manner, which is hard to accomplish in a wide-area system. A better approach is therefore to let receivers that tend to miss the same messages team up and share the same multicast channel for feedback messages and retransmissions. Details on this approach are found in Liu et al. (1998).

To enhance the scalability of SRM, it is useful to let receivers assist in local recovery. In particular, if a receiver to which message m has been successfully delivered, receives a request for retransmission, it can decide to multicast m even before the retransmission request reaches the original sender. Further details can be found in Floyd et al. (1997) and Liu et aI. (1998).

Hierarchical Feedback Control

Feedback suppression as just described is basically a nonhierarchical solution. However, achieving scalability for very large groups of receivers requires that hierarchical approaches are adopted. In essence, a hierarchical solution to reliable multicasting works as shown in Fig. 8-11.

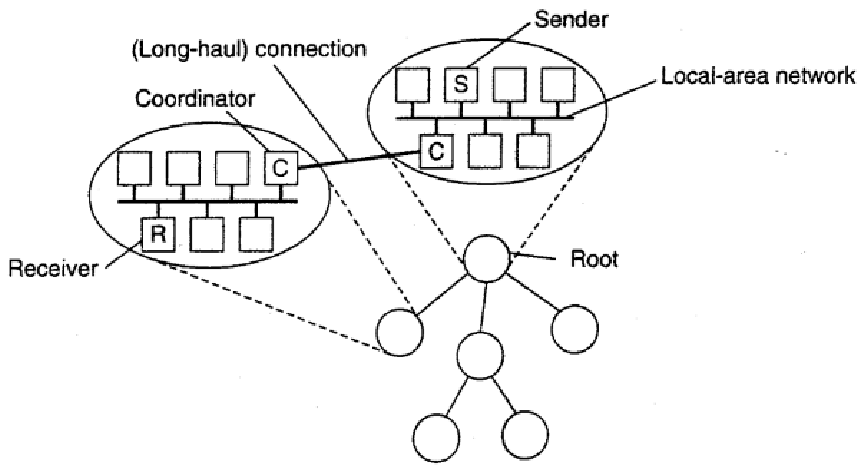


Figure 8-11. The essence of hierarchical reliable multicasting. Each local coordinator forwards the message to its children and later handles retransmission requests.

To simplify matters, assume there is only a single sender that needs to multicast messages to a very large group of receivers. The group of receivers is partitioned into a number of subgroups, which are subsequently organized into a tree. The subgroup containing the sender forms the root of the tree. Within each subgroup, any reliable multicasting scheme that works for small groups can be used.

Each subgroup appoints a local coordinator, which is responsible for handling retransmission requests of receivers contained in its subgroup. The local coordinator will thus have its own history buffer. If the coordinator itself has missed a message m, it asks the coordinator of the parent subgroup to retransmit m. In a scheme based on acknowledgments, a local coordinator sends an acknowledgment to its parent if it has received the message. If a coordinator has received acknowledgments for message m from all members in its subgroup, as well as from its children, it can remove m from its history buffer.

The main problem with hierarchical solutions is the construction of the tree. In many cases, a tree needs to be constructed dynamically. One approach is to make use of the multicast tree in the underlying network, if there is one. In principle, the approach is then to enhance each multicast router in the network layer in such a way that it can act as a local coordinator in the way just described. Unfortunately, as a practical matter, such adaptations to existing computer networks are not easy to do.In conclusion, building reliable multicast schemes that can scale to a large number of receivers spread across a wide-area network, is a difficult problem. No single best solution exists, and each solution introduces new problems.

● Recovery

Fundamental to fault tolerance is the recovery from an error. Recall that an error is that part of a system that may lead to a failure. The whole idea of error recovery is to replace an erroneous state with an error-free state. There are essentially two forms of error recovery.

In backward recovery, the main issue is to bring the system from its present erroneous state back into a previously correct state. To do so, it will be necessary to record the system's state from time to time, and to restore such a recorded state when things go wrong. Each time (part of) the system's present state is recorded, a checkpoint is said to be made.

Another form of error recovery is forward recovery. In this case, when the system has entered an erroneous state, instead of moving back to a previous, checkpointed state, an attempt is made to bring the system in a correct new state from which it can continue to execute. The main problem with forward error recovery mechanisms is that it has to be known in advance which errors may occur.

Only in that case is it possible to correct those errors and move to a new state. The distinction between backward and forward error recovery is easily explained when considering the implementation of reliable communication. The common approach to recover from a lost packet is to let the sender retransmit that packet. In effect, packet retransmission establishes that we attempt to go back to a previous, correct state, namely the one in which the packet that was lost is being sent. Reliable communication through packet retransmission is therefore an example of applying backward error recovery techniques.

An alternative approach is to use a method known as erasure correction. In this approach. a missing packet is constructed from other, successfully delivered packets. For example, in an (n,k) block erasure code, a set of k source packets is encoded into a set of n encoded packets, such that any set of k encoded packets is enough to reconstruct the original k source packets. Typical values are k =16' or k=32, and k<11~2k [see, for example, Rizzo (1997)]. If not enough packets have yet been delivered, the sender will have to continue transmitting packets until a previously lost packet can be constructed. Erasure correction is a typical example of a forward error recovery approach.

By and large, backward error recovery techniques are widely applied as a general mechanism for recovering from failures in distributed systems. The major benefit of backward error recovery is that it is a generally applicable method independent of any specific system or process. In other words, it can be integrated into (the middleware layer) of a distributed system as a general-purpose service.

However, backward error recovery also introduces some problems (Singhal and Shivaratri, 1994). First, restoring a system or process to a previous state is generally a relatively costly operation in terms of performance. As will be discussed in succeeding sections, much work generally needs to be done to recover from, for example, a process crash or site failure. A potential way out of this problem is to devise very cheap mechanisms by which components are simply rebooted. We will return to this approach below.

Second, because backward error recovery mechanisms are independent of the distributed application for which they are actually used, no guarantees can be given that once recovery has taken place, the same or similar failure will not happen again. If such guarantees are needed, handling errors often requires that the application gets into the loop of recovery. In other words, full-fledged failure transparency can generally not be provided by backward error recovery mechanisms.

Finally, although backward error recovery requires checkpointing, some states can simply never be rolled back to. For example, once a (possibly malicious) person has taken the $1.000 that suddenly came rolling out of the incorrectly functioning automated teller machine, there is only a small chance that money will be stuffed back in the machine. Likewise, recovering to a previous state in most UNIX systems after having enthusiastically typed rrn -fr * but from the wrong working directory, may turn a few people pale. Some things are simply irreversible.

Checkpointing allows the recovery to a previous correct state. However, taking a checkpoint is often a costly operation and may have a severe performance penalty. As a consequence, many fault-tolerant distributed systems combine checkpointing with message logging. In this case, after a checkpoint has been taken, a process logs its messages before sending them off (called sender-based logging). An alternative solution is to let the receiving process first log an incoming message before delivering it to the application it is executing. This scheme is also referred to as receiver-based logging. When a receiving process crashes, it is necessary to restore the most recently checked state, and from there on replay the messages that have been sent. Consequently, combining checkpoints with message logging makes it possible to restore a state that lies beyond the most recent checkpoint without the cost of checkpointing.

Another important distinction between checkpointing and schemes that additionally use logs follows. In a system where only checkpointing is used, processes will be restored to a checkpointed state. From there on, their behavior may be different than it was before the failure occurred. For example, because communication times are not deterministic, messages may now be delivered in a different order, in turn leading to different reactions by the receivers. However, if message logging takes place, an actual replay of the events that happened since the last checkpoint takes place. Such a replay makes it easier to interact with the outside world, For example, consider the case that a failure occurred because a user provided erroneous input. If only checkpointing is used, the system would have to take a checkpoint before accepting the user's input in order to recover to exactly the same state. With message logging, an older checkpoint can be used, after which a replay of events can take place up to the point that the user should provide input.

In practice, the combination of having fewer checkpoints and message logging is more efficient than having to take many checkpoints.

Stable Storage

To be able to recover to a previous state, it is necessary that information needed to enable recovery is safely stored. Safely in this context means that recovery information survives process crashes and site failures, but possibly also various storage media failures. Stable storage plays an important role when it comes to recovery in distributed systems. We discuss it briefly here.

Storage comes in three categories. First there is ordinary RAM memory, which is wiped out when the power fails or a machine crashes. Next there is disk storage, which survives CPU failures but which can be lost in disk head crashes.

Finally, there is also stable storage, which is designed to survive anything except major calamities such as floods and earthquakes. Stable storage can be implemented with a pair of ordinary disks, as shown in Fig. 8-23(a). Each block on drive 2 is an exact copy of the corresponding block on drive 1. When a block is updated, first the block on drive 1 is updated and verified. then the same block on drive 2 is done.

Suppose that the system crashes after drive 1 is updated but before the update on drive 2, as shown in Fig. 8-23(b). Upon recovery, the disk can be compared block for block.

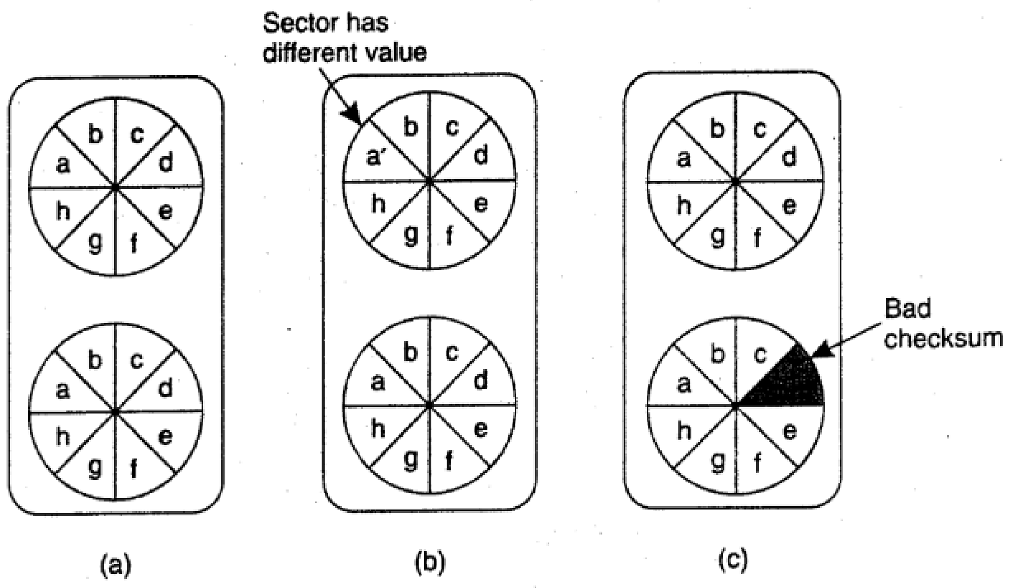


Figure 8-23. (a) Stable storage. (b) Crash after drive 1 is updated. (c) Bad spot.

Whenever two corresponding blocks differ, it can be assumed that drive 1 is the correct one (because drive 1 is always updated before drive 2), so the new block is copied from drive 1 to drive 2. When the recovery process is complete, both drives will again be identical.

Another potential problem is the spontaneous decay of a block. Dust particles or general wear and tear can give a previously valid block a sudden checksum error, without cause or warning, as shown in Fig. 8-23(c). When such an error is detected, the bad block can be regenerated from the corresponding block on the other drive.

As a consequence of its implementation, stable storage is well suited to applications that require a high degree of fault tolerance, such as atomic transactions.

When data is written to stable storage and then read back to check that it has been written correctly, the chance of it subsequently being lost is extremely small.