6.4 DISTRIBUTION PROTOCOLS

So far, we have concentrated only on different consistency models, but have hardly touched upon implementation issues. In this section, we discuss different ways of propagating, that is, distributing updates to replicas, independent of the consistency model that is to be supported. Specific consistency protocols are discussed in the next section.

6.4.1 Replica Placement

A major design issue for distributed data stores, is deciding where, when, and by whom copies of the data store are to be placed (see also Kermarrec et al., 1998). Three different types of copies can be distinguished, logically organized as shown in Fig. 6-19.

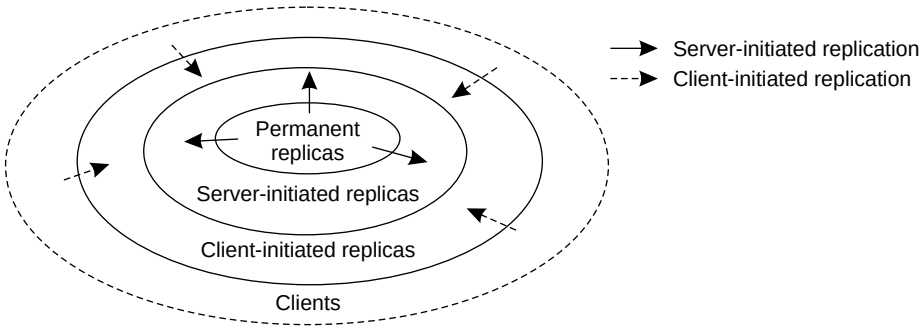


Figure 6-19. The logical organization of different kinds of copies of a data store into three concentric rings.

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 Web site. 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 on a single local-area network. Whenever a request comes in, it is forwarded to one of the servers, for instance, using a round-robin strategy (see also Chawathe and Brewer, 1998).

The second form of distributed Web sites is that of what is called **mirroring**. In this case, a Web site 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 mirror sites from a list offered to them. Mirrored Web

sites have in common with cluster-based Web sites that there are only a few number of replicas, which are more or less statically configured.

Similar static organizations also appear with distributed databases (Oszu and Valduriez, 1999). Again, the database can be distributed and replicated across a number of servers that together form a cluster of workstations (COW), often referred to as a **shared-nothing architecture**, emphasizing that neither disks nor main memory are shared by processors. Alternatively, a database is distributed and possibly replicated across a number of geographically dispersed number of sites. This architecture is generally deployed in federated databases (Sheth and Larson, 1990).

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. (Such bursts have happened on a number of occasions in the recent history of the Web.) In that case, it may be worthwhile to install a number of temporary replicas in regions where requests are coming from. Such replicas are also known as **push caches** (Gwertzman and Seltzer, 1995).

More recently, the problem of dynamically placing replicas is being addressed in Web hosting services. These services essentially 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.

One of the problems with server-initiated replicas is deciding exactly where and when replicas should be created or deleted. 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 files 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 C_1 and client C_2 share the same "closest" server P, all access requests for file F at server Q from C_1 and C_2 are jointly registered at Q as a single access count $cnt_Q(P,F)$. This situation is shown in Fig. 6-20.

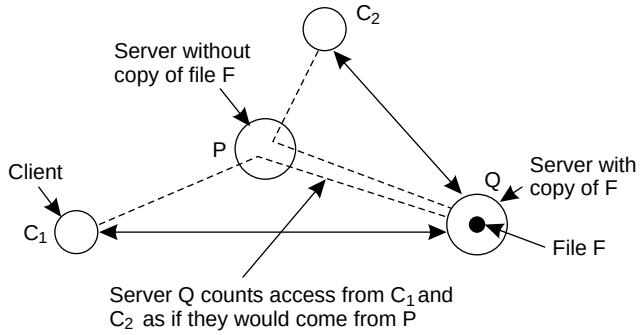


Figure 6-20. Counting access requests from different clients.

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 loads at other servers. Special measures are taken to ensure that at least one copy of each file continues to exist.

A replication threshold $rep(S,F)$, 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, $cnt_Q(P,F)$ 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, $cnt_Q(R,F)$ exceeds a certain fraction of all requests for F at Q, an attempt is made to replicate F to R.

Server-initiated replication is gradually increasing in popularity, 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 created at the initiative of clients. 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 have not been modified in the meantime.

Data are 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 hit** 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 C_1 may also be useful for a request from another nearby client C_2.

Whether this assumption is correct depends very much on the type of data store. For example, in traditional file systems, files are rarely shared at all (see, e.g., Muntz and Honeyman, 1992; and Blaze, 1993) rendering a shared cache useless. In contrast, shared caches in the Web have proven to be useful, although their impact with respect to performance is gradually declining for the reason that sharing between different clients occurs less often as the total number of Web pages grows (Barford et al., 1999).

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.

6.4.2 Update Propagation

Update operations on a distributed and replicated data store are generally initiated at a client and subsequently forwarded to one of the copies. From there, the update should be propagated to the other copies, while at the same time ensuring consistency. There are different design issues to consider with respect to propagating updates, which we discuss next.

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 have 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. 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. On the other hand, more processing power may be required by each replica, especially 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 turn, 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 are 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. 6-21. For simplicity, consider a client-server system consisting of a single, nondistributed 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 6-21. 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.

These trade-offs have lead to a hybrid form of update propagation based on

leases. A **lease** is a promise by the server that it will push updates to the client for a specified time. When a lease expires, the client is forced to poll the server for updates and pull in the modified data if necessary. An alternative is that a client requests a new lease for pushing updates when the previous lease expires.

Leases were originally introduced by Gray and Cheriton (1989). They provide a convenient mechanism for dynamically switching between a push-based and pull-based strategy. Duvvuri et al. (2000) describe a flexible lease system that allows the expiration time to be dynamically adapted depending on different lease criteria. They distinguish the following three types of leases. (Note that in all cases, updates are pushed by the server as long as the lease has not expired.)

First, age-based leases are given out on data items depending on the last time the item was modified. The underlying assumption is that data that have not been modified for a long time can be expected to remain unmodified for some time yet to come. This assumption has shown to be reasonable in the case of Web-based data. By granting long-lasting leases to data items that are expected to remain unmodified, the number of update messages can be strongly reduced compared to the case where all leases have the same expiration time.

Another lease criterion is how often a specific client requests its cached copy to be updated. With renewal-frequency based leases, a server will hand out a long-lasting lease to a client whose cache often needs to be refreshed. On the other hand, a client that asks only occasionally for a specific data item will be handed a short-term lease for that item. The effect of this strategy is that the server essentially keeps track only of those clients where its data are popular; moreover, those clients are offered a high degree of consistency.

The last criterion is that of state-space overhead at the server. When the server realizes that it is gradually becoming overloaded, it lowers the expiration time of new leases it hands out to clients. The effect of this strategy is that the server needs to keep track of fewer clients as leases expire more quickly. In other words, the server dynamically switches to a more stateless mode of operation, thereby offloading itself so that it can handle requests more efficiently.

Unicasting versus Multicasting

Related to pushing or pulling updates is deciding whether unicasting or multicasting should be used. In unicast communication, when a server that is part of the data store sends its update to N other servers, it does so by sending N separate messages, one to each server. With multicasting, the underlying network takes care of sending a message efficiently to multiple receivers.

In many cases, it is cheaper to use available multicasting facilities. An extreme situation is when all replicas are located in the same local-area network and that hardware broadcasting is available. In that case, broadcasting or multicasting a message is no more expensive than a single point-to-point message. Unicasting updates would then be less efficient.

Multicasting can often be efficiently combined with a push-based approach to propagating updates. In that case, a server that decides to push its updates to a number of other servers simply uses a single multicast group to send its updates. In contrast, with a pull-based approach, it is generally only a single client or server that requests its copy to be updated. In that case, unicasting may be the most efficient solution.

6.4.3 Epidemic Protocols

We already mentioned that for many data stores it is sufficient to provide only eventual consistency. In other words, in the absence of updates, it need only be guaranteed that all replicas are eventually identical. Update propagation in eventual-consistent data stores is often implemented by a class of algorithms known as **epidemic protocols**. Epidemic algorithms do not solve any update conflicts for which separate solutions are used. Instead, their only concern is propagating updates to all replicas in as few messages as possible. For this purpose, updates are often aggregated into a single message, and subsequently exchanged between two servers. Epidemic algorithms form the basis for the Bayou system described previously; its various update propagation schemes are described in (Petersen et al., 1997).

To explain the general principles of these algorithms, we assume that all updates for a specific data item are initiated at a single server. In this way, we simply avoid write-write conflicts. The following presentation is based on the classical paper by Demers et al. (1987) on epidemic algorithms.

Update Propagation Models

As the name suggests, epidemic algorithms are based on the theory of epidemics, which studies the spreading of infectious diseases. In the case of replicated data stores, instead of spreading diseases, they spread updates. Research on epidemics for data stores also aims at a completely different goal: whereas health organizations will do their utmost best to prevent infectious diseases from spreading across large groups of people, designers of epidemic algorithms for distributed data stores will try to ''infect'' all replicas with new updates as fast as possible.

Using the terminology from epidemics, a server that is part of a distributed data store is called **infective** if it holds an update that it is willing to spread to other servers. A server that has not yet been updated, is called **susceptible**. Finally, an updated server that is not willing or able to spread its update, is said to be **removed**.

A popular propagation model is that of **anti-entropy**. In this model, a server *P* picks another server *Q* at random, and subsequently exchanges updates with *Q*. There are three approaches to exchanging updates:

1. *P* only pushes its own updates to *Q*
2. *P* only pulls in new updates from *Q*
3. *P* and *Q* send updates to each other (i.e., a push-pull approach)

When it comes to rapidly spreading updates, only pushing updates turns out to be a bad choice. Intuitively, this can be understood as follows. First, note that in a pure push-based approach, updates can be propagated only by infective servers. However, if many servers are infective, the probability of each one selecting a susceptible server is relatively small. Consequently, chances are that a particular server remains susceptible for a long period simply because it is not selected by an infective server.

In contrast, the pull-based approach works much better when many servers are infective. In that case, spreading updates is essentially triggered by susceptible servers. Chances are large that such a server will contact an infective one to subsequently pull in the updates and become infective as well.

It can be shown that if only a single server is infective, updates will eventually be spread across all servers using either form of anti-entropy. However, to ensure that updates are spread quickly, it makes sense to use additional mechanisms by which at least a number of servers immediately become more or less infective. Typically, directly pushing an update to a number of servers helps.

One specific variant of this approach is called **rumor spreading**, or simply **gossiping**. It works as follows. If server *P* has just been updated for data item *x*, it contacts an arbitrary other server *Q* and tries to push the update to *Q*. However, it is possible that *Q* was already updated by another server. In that case, *P* may lose interest in spreading the update any further, say with probability $1/k$. In other words, it then becomes removed.

Gossiping is completely analogous to real life. When Bob has some hot news to spread around, he may phone his friend Alice telling her all about it. Alice, like Bob, will be really excited to spread the gossip to her friends as well. However, she will become disappointed when phoning a friend, say Chuck, only to hear that the news has already reached him. Chances are that she will stop phoning other friends, for what good is it if they already know?

Gossiping turns out to be a really good way of rapidly spreading updates. However, it cannot guarantee that all servers will actually be updated (Demers et al., 1987). It can be shown that in the case the data store consists of a large number of servers, the fraction *s* of servers that will remain ignorant of an update, that is, remain susceptible, satisfies the equation:

$$s = e^{-(k+1)(1-s)}$$

For example, if $k = 3$, s is less than 0.02, meaning that less than 2% of the servers remain susceptible. Nevertheless, special measures are needed to guarantee that those servers will also be updated. Combining anti-entropy with gossiping

will do the trick.

One of the main advantages of epidemic algorithms is their scalability, due to the fact that the number of synchronizations between processes is relatively small compared to other propagation methods. For wide-area systems, Lin and Marzullo show that it makes sense to take the actual network topology into account to achieve better results (Lin and Marzullo, 1999). In their approach, servers that are connected to only a few other servers are contacted with a relatively high probability. The underlying assumption is that such servers form a bridge to other remote parts of the network; therefore, they should be contacted as soon as possible.

Removing Data

Epidemic algorithms are good for spreading updates in eventual-consistent data stores. However, they have a rather strange side-effect: spreading the *deletion* of a data item is hard. The essence of the problem lies in the fact that deletion of a data item destroys all information on that item. Consequently, when a data item is simply removed from a server, that server will eventually receive old copies of the data item and interpret those as updates on something it did not have before.

The trick is to record the deletion of a data item as just another update, and keep a record of that deletion. In this way, old copies will not be interpreted as something new, but merely treated as versions that have been updated by a delete operation. The recording of a deletion is done by spreading **death certificates**.

Of course, the problem with death certificates is that they should eventually be cleaned up, or otherwise each server will gradually build a huge local database of historical information on deleted data items that is otherwise not used. Demers et al. propose to use what they call dormant death certificates. Each death certificate is timestamped when it is created. If it can be assumed that updates propagate to all servers within a known finite time, then death certificates can be removed after this maximum propagation time has elapsed.

However, to provide hard guarantees that deletions are indeed spread to all servers, only a very few servers maintain dormant death certificates that are never thrown away. Assume server P has such a certificate for data item x. If by any chance an obsolete update for x reaches P, P will react by simply spreading the death certificate for x again.