- 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.

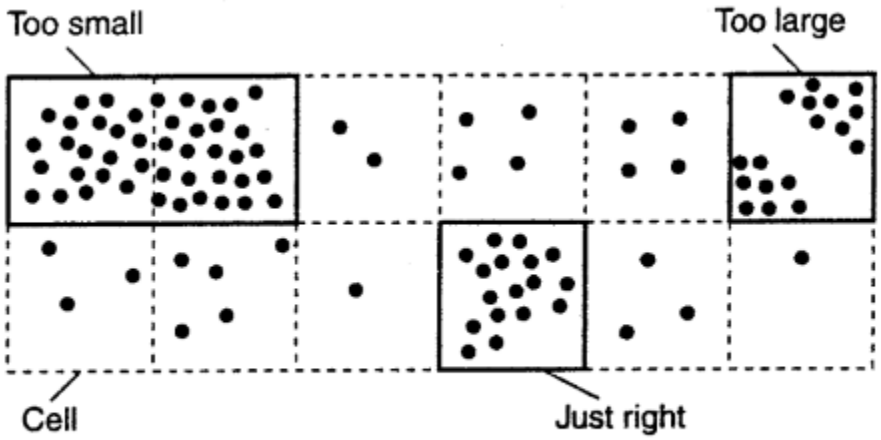

Figure 7-16. Choosing a proper cell *size* for server placement.

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.

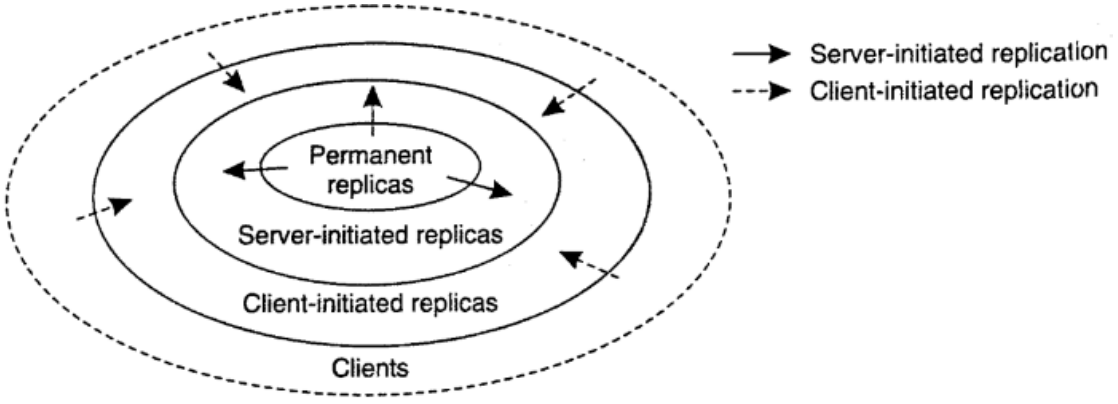

Figure 7-17. The logical organization of different kinds of copies of a data store into three concentric rings.

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.

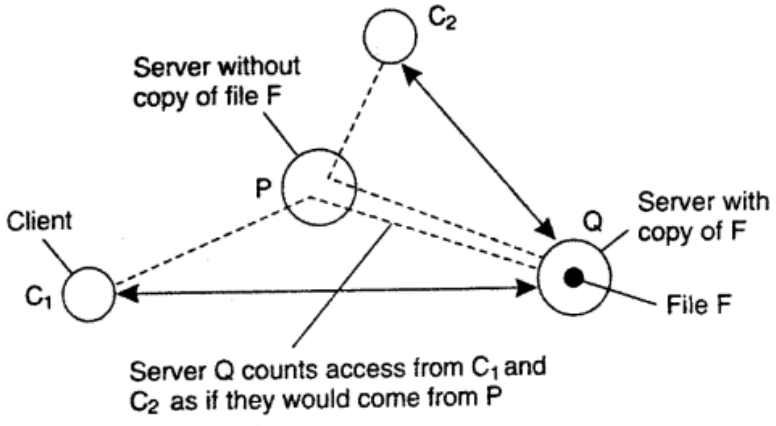

Figure 7-18. Counting access requests from different elients.

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.