Chapter 7

Consistency and replication

An important issue in distributed systems is the replication of data. Data are generally replicated to enhance reliability or improve performance. One of the major problems is keeping replicas consistent. Informally, this means that when one copy is updated we need to ensure that the other copies are updated as well; otherwise the replicas will no longer be the same. In this chapter, we take a detailed look at what consistency of replicated data actually means and the various ways that consistency can be achieved.

We start with a general introduction discussing why replication is useful and how it relates to scalability. We then continue by focusing on what consistency actually means. An important class of what are known as consistency models assumes that multiple processes simultaneously access shared data. Consistency for these situations can be formulated with respect to what processes can expect when reading and updating the shared data, knowing that others are accessing that data as well.

Consistency models for shared data are often hard to implement efficiently in large-scale distributed systems. Moreover, in many cases simpler models can be used, which are also often easier to implement. One specific class is formed by client-centric consistency models, which concentrate on consistency from the perspective of a single (possibly mobile) client. Client-centric consistency models are discussed in a separate section.

Consistency is only half of the story. We also need to consider how consistency is actually implemented. There are essentially two, more or less independent, issues we need to consider. First of all, we start with concentrating on managing replicas, which takes into account not only the placement of replica servers, but also how content is distributed to these servers.

The second issue is how replicas are kept consistent. In most cases, applications require a strong form of consistency. Informally, this means that updates are to be propagated more or less immediately between replicas. There are various alternatives for implementing strong consistency, which are

discussed in a separate section. Also, attention is paid to caching protocols, which form a special case of consistency protocols.

Being arguably the largest distributed system, we pay separate attention to caching and replication in Web-based systems, notably looking at content delivery networks as well as edge-server caching techniques.

7.1 Introduction

In this section, we start with discussing the important reasons for wanting to replicate data in the first place. We concentrate on replication as a technique for achieving scalability, and motivate why reasoning about consistency is so important.

Reasons for replication

There are two primary reasons for replicating data. First, data are 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 a distributed system needs to scale in terms of size or in terms of the geographical area it covers. Scaling with respect to size 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 workload among the processes accessing the data.

Scaling with respect to a geographical area may also require replication. The basic idea is that by placing a copy of data in 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. Although a client process may perceive better performance, it may also be the case that more network bandwidth is now consumed keeping all replicas up to date.

If replication helps to improve reliability and performance, who could be against it? Unfortunately, there is a price to be paid when data are replicated. 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.

To understand the problem, consider improving access times to Web pages. If no special measures are taken, fetching a page from a remote Web server may sometimes even take seconds to complete. To improve performance, Web browsers often locally store a copy of a previously fetched Web page (i.e., they *cache* a Web page). If a user requires that page again, the browser automatically returns the local copy. The access time as perceived by the user is excellent. However, if the user always wants to have the latest version of a page, he may be in for bad luck. The problem is that if the page has been modified in the meantime, modifications will not have been propagated to cached copies, making those copies out-of-date.

One solution to the problem of returning a stale copy to the user is to forbid the browser to keep local copies in the first place, effectively letting the server be fully in charge of replication. However, this solution may still lead to poor access times if no replica is placed near the user. Another solution is to let the Web server invalidate or update each cached copy, but this requires that the server keeps track of all caches and sending them messages. This, in turn, may degrade the overall performance of the server. We return to performance versus scalability issues below.

In the following we will mainly concentrate on replication for performance. Replication for reliability is discussed in Chapter 8.

Replication as scaling technique

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 possible trade-off that needs to be made is that keeping copies up to date may require more network bandwidth. Consider a process P that accesses a local replica N times per second, whereas the replica itself is updated M times per second. Assume that an update completely refreshes the previous version of the local replica. If $N \ll M$, that is, the access-to-update ratio is very low, we have the situation where many updated versions of the local replica will never be accessed by P, rendering the network communication for those versions useless. In this case, it may have been better not to install a local replica close to P, or to apply a different strategy for updating the replica.

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. (In Section 7.2, we will provide precise definitions of consistency and introduce a range of consistency models.) 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. For example, replicas may need to decide on a global ordering of operations using Lamport timestamps, or let a coordinator assign such an order. Global synchronization simply takes a lot of communication time, especially when replicas are spread across a wide-area network.

We are now faced with a dilemma. On the one hand, scalability problems can be alleviated by applying replication and caching, leading to improved performance. On the other hand, to keep all copies consistent generally requires global synchronization, which is inherently costly in terms of performance. The cure may be worse than the disease.

In many cases, the only real solution is to relax 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. The price paid is that copies may not always be the same everywhere. As it turns out, to what extent consistency can be relaxed depends highly on the access and update patterns of the replicated data, as well as on the purpose for which those data are used.

There are a range of consistency models and many different ways to implement models through what are called distribution and consistency protocols. Approaches to classifying consistency and replication can be found in [Gray et al., 1996; Wiesmann et al., 2000] and [Aguilera and Terry, 2016].

7.2 Data-centric consistency models

Traditionally, consistency has been discussed in the context of read and write operations on shared data, available by means of (distributed) shared memory, a (distributed) shared database, or a (distributed) file system. Here, we use the broader term **data store**. A data store may be physically distributed across

multiple machines. In particular, each process that can access data from the store is assumed to have a local (or nearby) copy available of the entire store. Write operations are propagated to the other copies, as shown in Figure 7.1. A data operation is classified as a write operation when it changes the data, and is otherwise classified as a read operation.

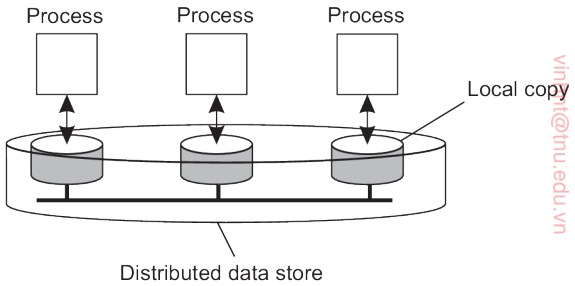

Figure 7.1: The general organization of a logical data store, physically distributed and replicated across multiple processes.

A **consistency model** is essentially a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly. Normally, a process that performs a read operation on a data item, expects the operation to return a value that shows the results of the last write operation on that data.

In the absence of a global clock, it is difficult to define precisely which write operation is the last one. As an alternative, we need to provide other definitions, leading to a range of consistency models. Each model effectively restricts the values that a read operation on a data item can return. As is to be expected, the ones with major restrictions are easy to use, for example when developing applications, whereas those with minor restrictions are generally considered to be difficult to use in practice. The trade-off is, of course, that the easy-to-use models do not perform nearly as well as the difficult ones. Such is life.

Continuous consistency

There is no such thing as a best solution to replicating data. Replicating data poses consistency problems that cannot be solved efficiently in a general way. Only if we loosen consistency can there be hope for attaining efficient solutions. Unfortunately, there are also no general rules for loosening consistency: exactly what can be tolerated is highly dependent on applications.

There are different ways for applications to specify what inconsistencies they can tolerate. Yu and Vahdat [2002] take a general approach by distinguishing three independent axes for defining inconsistencies: deviation in numerical values between replicas, deviation in staleness between replicas,

and deviation with respect to the ordering of update operations. They refer to these deviations as forming **continuous consistency** ranges.

Measuring inconsistency in terms of numerical deviations can be used by applications for which the data have numerical semantics. One obvious example is the replication of records containing stock market prices. In this case, an application may specify that two copies should not deviate more than $0.02, which would be an *absolute numerical deviation*. Alternatively, a *relative numerical deviation* could be specified, stating that two copies should differ by no more than, for example, 0.5%. In both cases, we would see that if a stock goes up (and one of the replicas is immediately updated) without violating the specified numerical deviations, replicas would still be considered to be mutually consistent.

Numerical deviation can also be understood in terms of the number of updates that have been applied to a given replica, but have not yet been seen by others. For example, a Web cache may not have seen a batch of operations carried out by a Web server. In this case, the associated deviation in the *value* is also referred to as its *weight*.

Staleness deviations relate to the last time a replica was updated. For some applications, it can be tolerated that a replica provides old data as long as it is not *too* old. For example, weather reports typically stay reasonably accurate over some time, say a few hours. In such cases, a main server may receive timely updates, but may decide to propagate updates to the replicas only once in a while.

Finally, there are classes of applications in which the ordering of updates are allowed to be different at the various replicas, as long as the differences remain bounded. One way of looking at these updates is that they are applied tentatively to a local copy, awaiting global agreement from all replicas. As a consequence, some updates may need to be rolled back and applied in a different order before becoming permanent. Intuitively, ordering deviations are much harder to grasp than the other two consistency metrics.

The notion of a conit

To define inconsistencies, Yu and Vahdat introduce a **consistency unit**, abbreviated to **conit**. A conit specifies the unit over which consistency is to be measured. For example, in our stock-exchange example, a conit could be defined as a record representing a single stock. Another example is an individual weather report.

To give an example of a conit, and at the same time illustrate numerical and ordering deviations, consider the situation of keeping track of a fleet of cars. In particular, the fleet owner is interested in knowing how much he pays on average for gas. To this end, whenever a driver tanks gasoline, he reports the amount of gasoline that has been tanked (recorded as `g`), the price paid (recorded as `p`), and the total distance since the last time he tanked (recorded

by the variable d). Technically, the three variables g, p, and d form a conit. This conit is replicated across two servers, as shown in Figure 7.2, and a driver regularly reports his gas usage to one of the servers by separately updating each variable (without further considering the car in question).

The task of the servers is to keep the conit "consistently" replicated. To this end, each replica server maintains a two-dimensional vector clock. We use the notation $\langle T, \mathsf{R} \rangle$ to express an operation that was carried out by replica R at (its) logical time T.

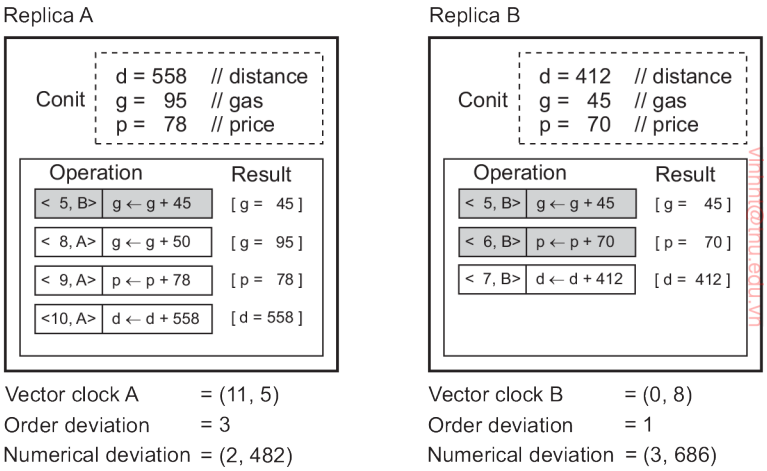

Figure 7.2: An example of keeping track of consistency deviations.

In this example we see two replicas that operate on a conit containing the data items g, p, and d from our example. All variables are assumed to have been initialized to 0. Replica A received the operation

$$\langle 5, \mathsf{B} \rangle : \mathsf{g} \leftarrow \mathsf{g} + 45$$

from replica B. We have shaded this operation gray to indicate that A has *committed* this operation to its local store. In other words, it has been made permanent and cannot be rolled back. Replica A also has three *tentative* update operations listed: $\langle 8, \mathsf{A} \rangle$, $\langle 9, \mathsf{A} \rangle$, and $\langle 10, \mathsf{A} \rangle$, respectively. In terms of continuous consistency, the fact that A has three tentative operations pending to be committed is referred to as an **order deviation** of, in this case, value 3. Analogously, with in total three operations of which two have been committed, B has an order deviation of 1.

From this example, we see that A's logical clock value is now 11. Because the last operation from B that A had received had timestamp 5, the vector clock at A will be $(11, 5)$, where we assume the first component of the vector is used for A and the second for B. Along the same lines, the logical clock at B is $(0, 8)$.

The **numerical deviation** at a replica R consists of two components: the number of operations at all *other* replicas that have not yet been seen by R, along with the sum of corresponding missed values (more sophisticated schemes are, of course, also possible). In our example, A has not yet seen operations $\langle 6, B\rangle$ and $\langle 7, B\rangle$ with a total value of 70 + 412 units, leading to a numerical deviation of $(2, 482)$. Likewise, B is still missing the three tentative operations at A, with a total summed value of 686, bringing B's numerical deviation to $(3, 686)$.

Using these notions, it becomes possible to specify specific consistency schemes. For example, we may restrict order deviation by specifying an acceptable maximal value. Likewise, we may want two replicas to never numerically deviate by more than 1000 units. Having such consistency schemes does require that a replica knows how much it is deviating from other replicas, implying that we need separate communication to keep replicas informed. The underlying assumption is that such communication is much less expensive than communication to keep replicas synchronized. Admittedly, it is questionable if this assumption also holds for our example.

Note 7.1 (Advanced: On the granularity of conits)
There is a trade-off between maintaining fine-grained and coarse-grained conits. If a conit represents a lot of data, such as a complete database, then updates are aggregated for all the data in the conit. As a consequence, this may bring replicas sooner in an inconsistent state. For example, assume that in Figure 7.3 two replicas may differ in no more than one outstanding update. In that case, when the data items in Figure 7.3 have each been updated once at the first replica, the second one will need to be updated as well. This is not the case when choosing a smaller conit, as shown in Figure 7.3 There, the replicas are still considered to be up to date. This problem is particularly important when the data items contained in a conit are used completely independently, in which case they are said to **falsely share** the conit.

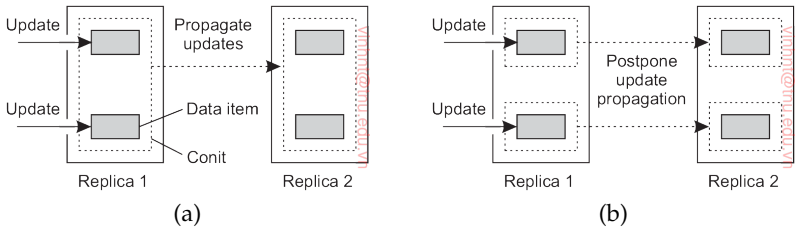

Figure 7.3: Choosing the appropriate granularity for a conit. (a) Two updates lead to update propagation. (b) No update propagation is needed.

Unfortunately, making conits very small is not a good idea, for the simple reason that the total number of conits that need to be managed grows as well. In other words, there is an overhead related to managing the conits that needs

to be taken into account. This overhead, in turn, may adversely affect overall performance, which has to be taken into account.

Although from a conceptual point of view conits form an attractive means for capturing consistency requirements, there are two important issues that need to be dealt with before they can be put to practical use. First, in order to enforce consistency we need to have protocols. Protocols for continuous consistency are discussed later in this chapter.

A second issue is that program developers must specify the consistency requirements for their applications. Practice indicates that obtaining such requirements may be extremely difficult. Programmers are generally not used to handling replication, let alone understanding what it means to provide detailed information on consistency. Therefore, it is mandatory that there are simple and easy-to-understand programming interfaces.

Note 7.2 (Advanced: Programming conits)
Continuous consistency can be implemented as a toolkit which appears to programmers as just another library that they link with their applications. A conit is simply declared alongside an update of a data item. For example, the fragment of pseudocode

```
AffectsConit(ConitQ, 1, 1);
append message m to queue Q;
```

states that appending a message to queue `Q` belongs to a conit named `ConitQ`. Likewise, operations may now also be declared as being dependent on conits:

```
DependsOnConit(ConitQ, 4, 0, 60);
read message m from head of queue Q;
```

In this case, the call to `DependsOnConit()` specifies that the numerical deviation, ordering deviation, and staleness should be limited to the values 4, 0, and 60 (seconds), respectively. This can be interpreted as that there should be at most 4 unseen update operations at other replicas, there should be no tentative local updates, and the local copy of `Q` should have been checked for staleness no more than 60 seconds ago. If these requirements are not fulfilled, the underlying middleware will attempt to bring the local copy of `Q` to a state such that the read operation can be carried out.

The question, of course, is how does the system know that `Q` is associated with `ConitQ`? For practical reasons, we can avoid explicit declarations of conits and concentrate only on the grouping of operations. The data to be replicated is collectively considered to belong together. By subsequently associating a write operation with a named conit, and likewise for a read operation, we tell the middleware layer when to start synchronizing the *entire* replica. Indeed, there may be a considerable amount of false sharing in such a case. If false sharing needs to be avoided, we would have to introduce a separate programming construct to explicitly declare conits.

Consistent ordering of operations

There is a huge body of work on data-centric consistency models from the past decades. An important class of models comes from the field of parallel programming. Confronted with the fact that in parallel and distributed computing multiple processes will need to share resources and access these resources simultaneously, researchers have sought to express the semantics of concurrent accesses when shared resources are replicated. The models that we discuss here all deal with consistently ordering operations on shared, replicated data.

In principle, the models augment those of continuous consistency in the sense that when tentative updates at replicas need to be committed, replicas will need to reach agreement on a global, that is, consistent ordering of those updates.

Sequential consistency

In the following, we will use a special notation in which we draw the operations of a process along a time axis. The time axis is always drawn horizontally, with time increasing from left to right. We use the notation $W_i(x)a$ to denote that process P_i writes value a to data item x. Similarly, $R_i(x)b$ represents the fact that process P_i reads x and is returned the value b. We assume that each data item has initial value 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 Figure 7.4 P_1 does a write to a data item x, modifying its value to a. Note that, according to our system model the operation $W_1(x)a$ is first performed on a copy of the data store that is local to P_1, and only then is it propagated to the other local copies. In our example, P_2 later reads the value NIL, and some time after that a (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 P_2, which is perfectly acceptable.

Sequential consistency is an important data-centric consistency model, which was first defined by Lamport [1979] in the context of shared memory for multiprocessor systems. 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.

What this definition means is that when processes run concurrently on (possibly) different machines, any valid interleaving of read and write operations is acceptable behavior, but *all processes see the same interleaving of operations*. Note that nothing is said about time; that is, there is no reference to the "most recent" write operation on a data item. Also, a process "sees" the writes from all processes but only through its own reads.

That time does not play a role can be seen from Figure 7.5. Consider four processes operating on the same data item x. In Figure 7.5(a) process P_1 first performs $W_1(x)a$ on x. Later (in absolute time), process P_2 also performs a write operation $W_2(x)b$, by setting the value of x to b. However, both processes P_3 and P_4 *first* read value b, and *later* value a. In other words, the write operation $W_2(x)b$ of process P_2 appears to have taken place before $W_1(x)a$ of P_1.

Figure 7.5: (a) A sequentially consistent data store. (b) A data store that is not sequentially consistent.

In contrast, Figure 7.5(b) violates sequential consistency because not all processes see the same interleaving of write operations. In particular, to process P_3, it appears as if the data item has first been changed to b, and later to a. On the other hand, P_4 will conclude that the final value is b.

Process P_1	Process P_2	Process P_3
`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 P_1, P_2, and P_3, shown in Figure 7.6 (taken from [Dubois et al., 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 0. 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.

Various interleaved execution sequences are possible. With six independent statements, there are potentially 720 (6!) possible execution sequences, although some of these violate program order. Consider the 120 (5!) sequences that begin with x ← 1. Half of these have print(x,z) before y ← 1 and thus violate program order. Half also have print(x,y) before z ← 1 and also violate program order. Only 1/4 of the 120 sequences, or 30, are valid. Another 30 valid sequences are possible starting with y ← 1 and another 30 can begin with z ← 1, for a total of 90 valid execution sequences. Four of these are shown in Figure 7.7.

Execution 1	Execution 2	Execution 3	Execution 4
P_1: x ← 1;	P_1: x ← 1;	P_2: y ← 1;	P_2: y ← 1;
P_1: print(y,z);	P_2: y ← 1;	P_3: z ← 1;	P_1: x ← 1;
P_2: y ← 1;	P_2: print(x,z);	P_3: print(x,y);	P_3: z ← 1;
P_2: print(x,z);	P_1: print(y,z);	P_2: print(x,z);	P_2: print(x,z);
P_3: z ← 1;	P_3: z ← 1;	P_1: x ← 1;	P_1: print(y,z);
P_3: print(x,y);	P_3: print(x,y);	P_1: print(y,z);	P_3: print(x,y);
Prints: 001011	*Prints:* 101011	*Prints:* 010111	*Prints:* 111111
Signature: 00 10 11	*Signature:* 10 10 11	*Signature:* 11 01 01	*Signature:* 11 11 11
(a)	(b)	(c)	(d)

Figure 7.7: Four valid execution sequences for the processes of Figure 7.6. The vertical axis is time.

In Figure 7.7(a) the three processes are run in order, first P_1, then P_2, then P_3. The other three examples demonstrate different, but equally valid, interleavings of the statements in time. Each of the three processes prints two variables. Since the only values each variable can take on are the initial value (0), or the assigned value (1), each process produces a 2-bit string. The numbers after *Prints* are the actual outputs that appear on the output device.

If we concatenate the output of P_1, P_2, and P_3 in that order, we get a 6-bit string that characterizes a particular interleaving of statements. This is the string listed as the *Signature* in Figure 7.7. Below we will characterize each ordering by its signature rather than by its printout.

Not all 64 signature patterns are allowed. As a trivial example, 00 00 00 is not permitted, because that would imply that the print statements ran before the assignment statements, violating the requirement that statements are executed in program order. A more subtle example is 00 10 01. The first two bits, 00, mean that y and z were both 0 when P_1 did its printing. This situation occurs only when P_1 executes both statements before P_2 or P_3 starts. The next two bits, 10, mean that P_2 must run after P_1 has started but before

P_3 has started. The last two bits, 01, mean that P_3 must complete before P_1 starts, but we have already seen that P_1 must go first. Therefore, 00 10 01 is not allowed.

In short, the 90 different valid statement orderings produce a variety of different program results (less than 64, though) that are allowed under the assumption of sequential consistency. The contract between the processes and the distributed shared data store is that the processes must accept all of these as valid results. In other words, the processes must accept the four results shown in Figure 7.7 and all the other valid results as proper answers, and must work correctly if any of them occurs. A program that works for some of these results and not for others violates the contract with the data store and is incorrect.

Note 7.3 (Advanced: The importance and intricacies of sequential consistency) There is no doubt that sequential consistency is an important model. In essence, of all consistency models that exist and have been developed, it is the easiest one to understand when developing concurrent and parallel applications. This is due to the fact that the model matches best our expectations when we let several programs operate on shared data simultaneously. At the same time, implementing sequential consistency is far from trivial [Adve and Boehm, 2010]. To illustrate, consider the example involving two variables x and y, shown in Figure 7.8.

P1:	W(x)a		W(y)a	R(x)b
P2:		W(y)b	W(x)b	R(y)a

Figure 7.8: Both x and y are each handled in a sequentially consistent manner, but taken together, sequential consistency is violated.

If we just consider the write and read operations on x, the fact that P_1 reads the value a is perfectly consistent. The same holds for the operation $R_2(y)b$ by process P_2. However, when taken together, there is no way that we can order the write operations on x and y such that we can have $R_1(x)a$ and $R_2(y)b$ (note that we need to keep the ordering as executed by each process):

Ordering of operations	Result	
$W_1(x)a; W_1(y)a; W_2(y)b; W_2(x)b$	$R_1(x)b$	$R_2(y)b$
$W_1(x)a; W_2(y)b; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a; W_2(y)b; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_2(x)b; W_1(x)a; W_1(y)a$	$R_1(x)a$	$R_2(y)a$

In terms of transactions, the operations carried out by P_1 and P_2 are not **serializable**. Our example shows that sequential consistency is not **compositional**: when having data items that are each kept sequentially consistent, their composition as a set need not be so [Herlihy and Shavit, 2008]. The problem of noncompositional consistency can be solved by assuming **linearizability**. This is best explained

by making a distinction between the start and completion of an operation, and assuming that it may take some time. Linearizability states that:

> *Each operation should appear to take effect instantaneously at some moment between its start and completion.*

Returning to our example, Figure 7.9 shows the same set of write operations, but we have now also indicated when they take place: the shaded area designates the time the operation is being executed. Linearizability states that the effect of an operation should take place somewhere during the interval indicated by the shaded area. In principle, this means that at the time of completion of a write operation, the results should be propagated to the other data stores.

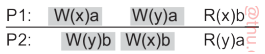

Figure 7.9: An example of taking linearizable sequential consistency into account, with only one possible outcome for x and y.

With that in mind, the possibilities for properly ordering become limited:

Ordering of operations	Result	
$W_1(x)a; W_2(y)b; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_1(x)a; W_2(y)b; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_1(y)a; W_2(x)b$	$R_1(x)b$	$R_2(y)a$
$W_2(y)b; W_1(x)a; W_2(x)b; W_1(y)a$	$R_1(x)b$	$R_2(y)a$

In particular, $W_2(y)b$ is completed before $W_1(y)a$ starts, so that y will have the value a. Likewise, $W_1(x)a$ completes before $W_2(x)b$ starts, so that x will have value b. It should not come as a surprise that implementing linearizability on a many-core architecture may impose serious performance problems. Yet at the same time, it eases programmability considerably, so a trade-off needs to be made.

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_1 writes a data item x. Then P_2 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 P_2 (i.e., the value written by P_1).

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.*

As an example of causal consistency, consider Figure 7.10. 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 $W_2(x)b$ and $W_1(x)c$ are concurrent, so it is not required that all processes see them in the same order.

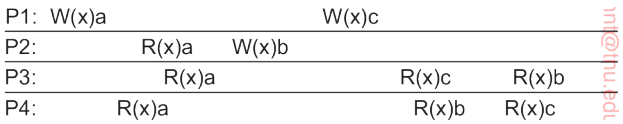

Figure 7.10: This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

Now consider a second example. In Figure 7.11(a) we have $W_2(x)b$ potentially depending on $W_1(x)a$ because writing the value b into x may be a result of a computation involving the previously read value by $R_2(x)a$. The two writes are causally related, so all processes must see them in the same order. Therefore, Figure 7.11(a) is incorrect. On the other hand, in Figure 7.11(b) the read has been removed, so $W_1(x)a$ and $W_2(x)b$ are now concurrent writes. A causally consistent store does not require concurrent writes to be globally ordered, so Figure 7.11(b) is correct. Note that Figure 7.11(b) reflects a situation that would not be acceptable for a sequentially consistent store.

Implementing causal consistency requires keeping track of which processes have seen which writes. There are many subtle issues to take into account. To illustrate, assume we replace $W_2(x)b$ in Figure 7.11(a) with $W_2(y)b$, and likewise $R_3(x)b$ with $R_3(y)b$, respectively. This situation is shown in Figure 7.12.

Let us first look at operation $R_3(x)$. Process P_3 executes this operation after $R_3(y)b$. We know at this point for sure that $W(x)a$ *happened before* $W(y)b$. In particular, $W(x)a \rightarrow R(x)a \rightarrow W(y)b$, meaning that if we are to preserve causality, reading x after reading b from y can return only a. If the system would return NIL to P_3 it would violate the preservation of causal relationships.

What about $R_4(y)$? Could it return the initial value of y, namely NIL? The answer is affirmative: although we have the formal *happened-before* relationship $W(x)a \rightarrow W(y)b$, without having read b from y, process P_4 can still justifiably observe that $W(x)a$ took place independently from the initialization of y.

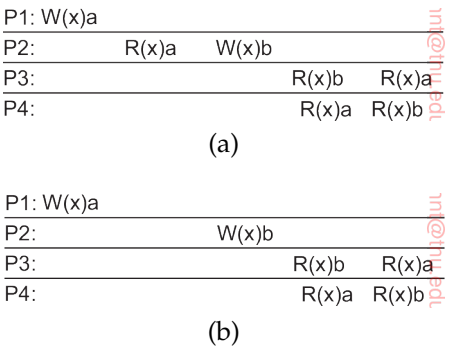

Figure 7.11: (a) A violation of a causally-consistent store. (b) A correct sequence of events in a causally-consistent store.

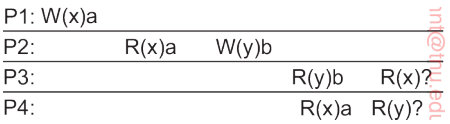

Figure 7.12: A slight modification of Figure 7.11(a). What should $R_3(x)$ or $R_4(y)$ return?

Implementationwise, preserving causality introduces some interesting questions. Consider, for example, the middleware underlying process P_3 from Figure 7.12. At the point that this middleware returns the value b from reading y, it must know about the relationship $W(x)a \rightarrow W(y)b$. In other words, when the most recent value of y was propagated to P_3's middleware, at the very least metadata on y's dependency should have been propagated as well. Alternatively, the propagation may have also been done together with updating x at P_3's node. By-and-large, the bottom line is that we need a dependency graph of which operation is dependent on which other operations. Such a graph may be pruned at the moment that dependent data is also locally stored.

Grouping operations

Many consistency models are defined at the level of elementary 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 does 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. A process that has successfully executed ENTER_CS will be ensured that all 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. Data and instructions between ENTER_CS and LEAVE_CS is denoted as a **critical section**.

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**, or simply **locks**. A lock has shared data items associated with it, and each shared data item is associated with at most one lock. In the case of course-grained synchronization, all shared data items would be associated to just a single lock. Fine-grained synchronization is achieved when each shared data item has its own unique lock. Of course, these are just two extremes of associating shared data to a lock. When a process enters a critical section it should *acquire* the relevant locks, and likewise when it leaves the critical section, it *releases* these locks.

Each lock has a current owner, namely, the process that last acquired it. A process not currently owning a lock 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 lock. While having *exclusive access* to a lock, a process is allowed to perform read and write operations. It is also possible for several processes to simultaneously have *nonexclusive access* to a lock, meaning that they can read, but not write, the associated data. Of course, nonexclusive access can be granted if and only if there is no other process having exclusive access.

We now demand that the following criteria are met [Bershad et al., 1993]:

- Acquiring a lock can succeed only when all updates to its associated shared data have completed.
- Exclusive access to a lock can succeed only if no other process has exclusive or nonexclusive access to that lock.
- Nonexclusive access to a lock is allowed only if any previous exclusive access has been completed, including updates on the lock's associated data.

Note that we are effectively demanding that the usage of locks is linearized,

adhering to sequential consistency. Figure 7.13 shows an example of what is known as **entry consistency**. We associate a lock with each data item separately. We use the notation L(x) as an abbreviation for acquiring the lock for x, that is, *locking* x. Likewise, U(x) stands for releasing the lock on x, or *unlocking* it. In this case, P_1 locks x, changes x once, after which it locks y. Process P_2 also acquires the lock for x but not for y, so that it will read value a for x, but may read NIL for y. However, because process P_3 first acquires the lock for y, it will read the value b when y was unlocked by P_1. It is important to note here that each process has a *copy* of a variable, but that this copy need not be instantly or automatically updated. When locking or unlocking a variable, a process is explicitly telling the underlying distributed system that the copies of that variable need to be synchronized. A simple read operation without locking may thus result in reading a local value that is effectively stale.

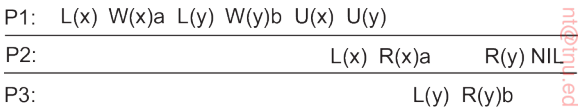

Figure 7.13: A valid event sequence for entry consistency.

One of the programming problems with entry consistency is properly associating data with locks. One straightforward approach is to explicitly tell the middleware which data are going to be accessed, as is generally done by declaring which database tables will be affected by a transaction. In an object-based approach, we could associate a unique lock with each declared object, effectively serializing all invocations to such objects.

Consistency versus coherence

At this point, it is useful to clarify the difference between two closely related concepts. The models we have discussed so far all deal with the fact that a number of processes execute read and write operations on a set of data items. A **consistency model** describes what can be expected with respect to that set when multiple processes concurrently operate on that data. The set is then said to be consistent if it adheres to the rules described by the model.

Where data consistency is concerned with a set of data items, **coherence models** describe what can be expected to hold for only a single data item [Cantin et al., 2005]. In this case, we assume that a data item is replicated; it is said to be coherent when the various copies abide to the rules as defined by its associated consistency model. A popular model is that of sequential consistency, but now applied to only a single data item. In effect, it means that in the case of concurrent writes, all processes will eventually see the same order of updates taking place.

Eventual consistency

To what extent processes actually operate in a concurrent fashion, and to what extent consistency needs to be guaranteed, may vary. There are many examples in which concurrency appears only in a restricted form. For example, in many database systems, most processes hardly ever perform update operations; they mostly read data from the database. Only one, or very few processes perform update operations. The question then is how fast updates should be made available to only-reading processes. In the advent of globally operating content delivery networks, developers often choose to propagate updates slowly, implicitly assuming that most clients are always redirected to the same replica and will therefore never experience inconsistencies.

Another example is the Web. In virtually all cases, Web pages are updated by a single authority, such as a webmaster or the actual owner of the page. There are normally no write-write conflicts to resolve. On the other hand, to improve efficiency, browsers and Web proxies are often configured to keep a fetched page in a local cache and to return that page upon the next request. An important aspect of both types of Web caches is that they may return out-of-date Web pages. In other words, the cached page that is returned to the requesting client is an older version compared to the one available at the actual Web server. As it turns out, many users find this inconsistency acceptable (to a certain degree), as long as they have access only to the same cache. In effect, they remain unaware of the fact that an update had taken place, just as in the previous case of content delivery networks.

Yet another example, is a worldwide naming system such as DNS. The DNS name space is partitioned into domains, where each domain is assigned to a naming authority, which acts as owner of that domain. Only that authority is allowed to update its part of the name space. Consequently, conflicts resulting from two operations that both want to perform an update on the same data (i.e., **write-write conflicts**), never occur. The only situation that needs to be handled are **read-write conflicts**, in which one process wants to update a data item while another is concurrently attempting to read that item. As it turns out, also in this case is it often acceptable to propagate an update in a lazy fashion, meaning that a reading process will see an update only after some time has passed since the update took place.

These examples can be viewed as cases of (large scale) distributed and replicated databases that tolerate a relatively high degree of inconsistency. They have in common that if no updates take place for a long time, all replicas will gradually become consistent, that is, have exactly the same data stored. This form of consistency is called **eventual consistency** [Vogels, 2009].

Data stores that are eventually consistent thus have the property that in the absence of write-write conflicts, all replicas will converge toward identical copies of each other. Eventual consistency essentially requires only that updates are guaranteed to propagate to all replicas. Write-write conflicts

are often relatively easy to solve when assuming that only a small group of processes can perform updates. In practice, we often also see that in the case of conflicts, one specific write operation is (globally) declared as "winner," overwriting the effects of any other conflicting write operation. Eventual consistency is therefore often cheap to implement.

Note 7.4 (Advanced: Making eventual consistency stronger)
Eventual consistency is a relatively easy model to understand, but equally important is the fact that it is also relatively easy to implement. Nevertheless, it is a weak-consistency model with its own peculiarities. Consider a calendar shared between Alice, Bob, and Chuck. A meeting M has two attributes: a proposed starting time and a set of people who have confirmed their attendance. When Alice proposes to start meeting M at time T, and assuming no one else has confirmed attendance, she executes the operation $W_A(M)[T, \{A\}]$. When Bob confirms his attendance, he will have read the tuple $[T, \{A\}]$ and update M accordingly: $W_B(M)[T, \{A, B\}]$. In our example two meetings M_1 and M_2 need to be planned.

Assume the sequence of events

$$W_A(M_1)[T_1, \{A\}] \rightarrow R_B(M_1)[T_1, \{A\}] \rightarrow$$

$$W_B(M_1)[T_1, \{A, B\}] \rightarrow W_B(M_2)[T_2, \{B\}].$$

In other words, Bob confirms his attendance at M_1 and then immediately proposes to schedule M_2 at T_2. Unfortunately, Chuck *concurrently* proposes to schedule M_1 at T_3 when Bob confirms he can attend M_1 at T_1. Formally, using the symbol "$\|$" to denote concurrent operations, we have,

$$W_B(M_1)[T_1, \{A, B\}] \parallel W_C(M_1)[T_3, \{C\}]$$

Using our usual notation, these operations can be illustrated as shown in Figure 7.14.

Figure 7.14: The situation of updating two meetings M_1 and M_2.

Eventual consistency may lead to very different scenarios. There is a number of write-write conflicts, but in any case, eventually $[T_2, \{B\}]$ will be stored for meeting M_2, as the result of the associated write operation by Bob. For the value of meeting M_1 there are different options. In principle, we have *three* possible outcomes: $[T_1, \{A\}]$, $[T_1, \{A, B\}]$, and $[T_3, \{C\}]$. Assuming we can maintain some notion of a global clock, it is not very likely that $W_A(M_1)[T_1, \{A\}]$ will prevail. However, the two write operations $W_B(M_1)[T_1, \{A, B\}]$ and $W_C(M_1)[T_1, \{C\}]$ are truly in conflict. In practice, one of them will win, presumably through a decision by a central coordinator.

Researchers have been seeking to combine eventual consistency with stricter guarantees on ordering. Bailis et al. [2013] propose to use a separate layer

that operates on top of an eventually consistent, distributed store. This layer implements causal consistency, of which it has been formerly proven that it is the best attainable consistency in the presence of network partitioning [Mahajan et al., 2011]. In our example, we have only one chain of dependencies:

$$W_A(M_1)[T_1, \{A\}] \rightarrow R_B(M_1)[T_1, \{A\}] \rightarrow$$

$$W_B(M_1)[T_1, \{A, B\}] \rightarrow W_B(M_2)[T_2, \{B\}].$$

An important observation is that with causal consistency in place, once a process reads $[T_2, \{B\}]$ for meeting M_2, obtaining the value for M_1 returns either $[T_1, \{A, B\}]$ or $[T_3, \{C\}]$, but certainly not $[T_1, \{A\}]$. The reason is that $W(M_1)[T_1, \{A, B\}]$ immediately precedes $W(M_2)[T_2, \{B\}]$, and at worse may have been overwritten by $W(M_1)[T_3, \{C\}]$. Causal consistency rules out that the system could return $[T_1, \{A\}]$.

However, eventual consistency may overwrite previously stored data items. In doing so, dependencies may be lost. To make this point clear, it is important to realize that in practice an operation at best keeps track of the immediate preceding operation it depends on. As soon as $W_c(M_1)[T_3, \{C\}]$ *overwrites* $W_B(M_1)[T_1, \{A, B\}]$ (and propagates to all replicas), we also break the chain of dependencies

$$W_A(M_1)[T_1, \{A\}] \rightarrow R_B(M_1)[T_1, \{A\}] \rightarrow \cdots \rightarrow W_B(M_2)[T_2, \{B\}]$$

which would normally prevent $W_A(M_1)[T_1, \{A\}]$ ever overtaking $W_B(M_1)[T_1, \{A, B\}]$ and any operation depending on it. As a consequence, maintaining causal consistency requires that we do maintain a history of dependencies, instead of just keeping track of immediately preceding operations.

7.3 Client-centric consistency models

Data-centric consistency models aim at providing a systemwide consistent view on a data store. An important assumption is that concurrent processes may be simultaneously updating the data store, and that it is necessary to provide consistency in the face of such concurrency. For example, in the case of object-based entry consistency, the data store guarantees that when an object is called, the calling process is provided with a copy of the object that reflects all changes to the object that have been made so far, possibly by other processes. During the call, it is also guaranteed that no other process can interfere, that is, mutual exclusive access is provided to the calling process.

Being able to handle concurrent operations on shared data while maintaining strong consistency is fundamental to distributed systems. For performance reasons, strong consistency may possibly be guaranteed only when processes use mechanisms such as transactions or synchronization variables. Along the same lines, it may be impossible to guarantee strong consistency, and weaker

forms need to be accepted, such as causal consistency in combination with eventual consistency.

In this section, we take a look at a special class of distributed data stores. The data stores we consider are characterized by the lack of simultaneous updates, or when such updates happen, it is assumed that they can be relatively easily resolved. Most operations involve reading data. These data stores offer a weak consistency model, such as eventual consistency. By introducing special client-centric consistency models, it turns out that many inconsistencies can be hidden in a relatively cheap way.

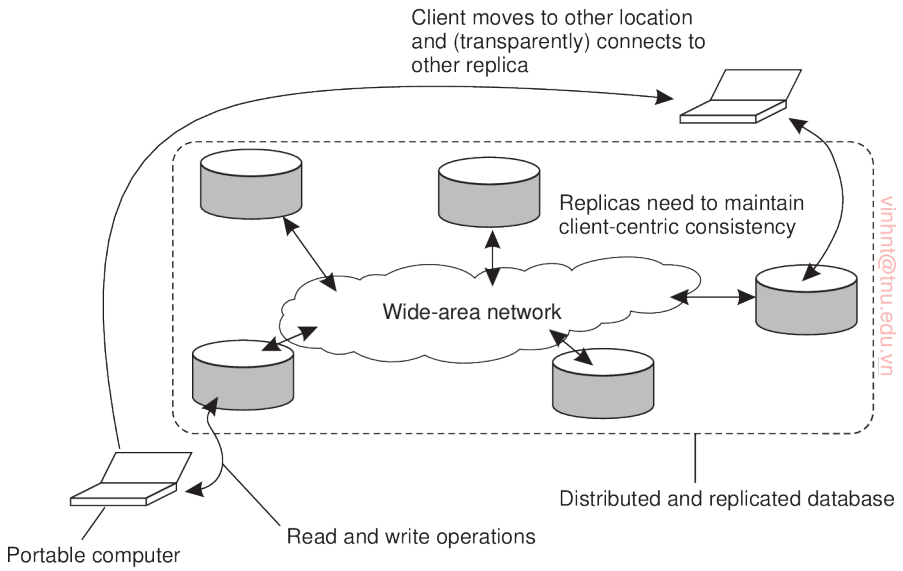

Figure 7.15: The principle of a mobile user accessing different replicas of a distributed database.

Eventually consistent data stores generally work fine as long as clients always access the same replica. However, problems arise when different replicas are accessed over a short period of time. This is best illustrated by considering a mobile user accessing a distributed database, as shown in Figure 7.15.

The mobile user, say, Alice, accesses the database by connecting to one of the replicas in a transparent way. In other words, the application running on Alice's mobile device is unaware on which replica it is actually operating. Assume Alice performs several update operations and then disconnects again. Later, she accesses the database again, possibly after moving to a different location or by using a different access device. At that point, she may be connected to a different replica than before, as shown in Figure 7.15. However, if the updates performed previously have not yet been propagated, Alice will notice inconsistent behavior. In particular, she would expect to see all

previously made changes, but instead, it appears as if nothing at all has happened.

This example is typical for eventually consistent data stores and is caused by the fact that users may sometimes operate on different replicas while updates have not been fully propagated. The problem can be alleviated by introducing **client-centric consistency**. In essence, client-centric consistency provides guarantees *for a single client* concerning the consistency of accesses to a data store by that client. No guarantees are given concerning concurrent accesses by different clients. If Bob modifies data that is shared with Alice but which is stored at a different location, we may easily create write-write conflicts. Moreover, if neither Alice nor Bob access the same location for some time, such conflicts may take a long time before they are discovered.

Client-centric consistency models originate from the work on Bayou and, more general, from mobile-data systems (see, for example, Terry et al. [1994], Terry et al. [1998], or Terry [2008]). Bayou is a database system developed for mobile computing, where it is assumed that network connectivity is unreliable and subject to various performance problems. Wireless networks and networks that span large areas, such as the Internet, fall into this category.

Bayou essentially distinguishes four different consistency models. To explain these models, we again consider a data store that is physically distributed across multiple machines. When a process accesses the data store, it generally connects to the locally (or nearest) available copy, although, in principle, any copy will do just fine. All read and write operations are performed on that local copy. Updates are eventually propagated to the other copies.

Client-centric consistency models are described using the following notations. Let x_i denote the *version* of data item x. Version x_i is the result of a series of write operations that took place since initialization, its **write set** $\mathsf{WS}(x_i)$. By appending write operations to that series we obtain another version x_j and say that x_j *follows from* x_i. We use the notation $\mathsf{WS}(x_i;x_j)$ to indicate that x_j follows from x_i. If we do not know if x_j follows from x_i, we use the notation $\mathsf{WS}(x_i|x_j)$.

Monotonic reads

The first client-centric consistency model is that of monotonic reads. A (distributed) data store is said to provide **monotonic-read consistency** if the following condition holds:

> *If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value.*

In other words, monotonic-read consistency guarantees that once a process has seen a value of x, it will never see an older version of x.

As an example where monotonic reads are useful, consider a distributed e-mail database. In such a database, each user's mailbox may be distributed

and replicated across multiple machines. Mail can be inserted in a mailbox at any location. However, updates are propagated in a lazy (i.e., on demand) fashion. Only when a copy needs certain data for consistency are those data propagated to that copy. Suppose a user reads his mail in San Francisco. Assume that only reading mail does not affect the mailbox, that is, messages are not removed, stored in subdirectories, or even tagged as having already been read, and so on. When the user later flies to New York and opens his mailbox again, monotonic-read consistency guarantees that the messages that were in the mailbox in San Francisco will also be in the mailbox when it is opened in New York.

Using a notation similar to that for data-centric consistency models, monotonic-read consistency can be graphically represented as shown in Figure 7.16. Rather than showing *processes* along the vertical axis, we now show *local data stores*, in our example L_1 and L_2. A write or read operation is indexed by the process that executed the operation, that is, $W_1(x)a$ denotes that process P_1 wrote value a to x. As we are not interested in specific values of shared data items, but rather their versions, we use the notation $W_1(x_2)$ to indicate that process P_1 produces version x_2 without knowing anything about other versions. $W_2(x_1;x_2)$ indicates that process P_2 is responsible for producing version x_2 that follows from x_1. Likewise, $W_2(x_1|x_2)$ denotes that process P_2 producing version x_2 *concurrently* to version x_1 (and thus potentially introducing a write-write conflict). $R_1(x_2)$ simply means that P_1 reads version x_2.

Figure 7.16: The read operations performed by a single process P at two different local copies of the same data store. (a) A monotonic-read consistent data store. (b) A data store that does not provide monotonic reads.

In Figure 7.16(a) process P_1 first performs a write operation on x at L_1, producing version x_1 and later reads this version. At L_2 process P_2 first produces version x_2, following from x_1. When process P_1 moves to L_2 and reads x again, it finds a more recent value, but one that at least took its previous write into account.

Figure 7.16(b) shows a situation in which monotonic-read consistency is violated. After process P_1 has read x_1 at L_1, it later performs the operation $R_1(x_2)$ at L_2. However, the preceding write operation $W_2(x_1|x_2)$ by process P_2 at L_2 is known to produce a version that does not follow from x_1. As a consequence, P_1's read operation at L_2 is known not to include the effect of the write operations when it performed $R_1(x_1)$ at location L_1.

Monotonic writes

In many situations, it is important that write operations are propagated in the correct order to all copies of the data store. This property is expressed in monotonic-write consistency. In a **monotonic-write consistent** store, the following condition 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.*

More formally, if we have two successive operations $W_k(x_i)$ and $W_k(x_j)$ by process P_k, then, regardless where $W_k(x_j)$ takes place, we also have $WS(x_i;x_j)$. Thus, completing a write operation means that the copy on which a successive operation is performed reflects the effect of a previous write operation by the same process, no matter where that operation was initiated. In other words, a write operation on a copy of item x is performed only if that copy has been brought up to date by means of any preceding write operation by that same process, which may have taken place on other copies of x. If need be, the new write must wait for old ones to finish.

Note that monotonic-write consistency resembles data-centric FIFO consistency. The essence of FIFO consistency is that write operations by the same process are performed in the correct order everywhere. This ordering constraint also applies to monotonic writes, except that we are now considering consistency only for a single process instead of for a collection of concurrent processes.

Bringing a copy of x up to date need not be necessary when each write operation completely overwrites the present value of x. However, write operations are often performed on only part of the state of a data item. Consider, for example, a software library. In many cases, updating such a library is done by replacing one or more functions, leading to a next version. With monotonic-write consistency, guarantees are given that if an update is performed on a copy of the library, all preceding updates will be performed first. The resulting library will then indeed become the most recent version and will include all updates that have led to previous versions of the library.

Monotonic-write consistency is shown in Figure 7.17. In Figure 7.17(a) process P_1 performs a write operation on x at L_1, presented as the operation $W_1(x_1)$. Later, P_1 performs another write operation on x, but this time at L_2, shown as $W_1(x_2;x_3)$. The version produced by P_1 at L_2 follows from an update by process P_2, in turn based on version x_1. The latter is expressed by the operation $W_2(x_1;x_2)$. To ensure monotonic-write consistency, it is necessary that the previous write operation at L_1 has already been propagated to L_2, and possibly updated.

In contrast, Figure 7.17(b) shows a situation in which monotonic-write consistency is not guaranteed. Compared to Figure 7.17(a) what is missing is the propagation of x_1 to L_2 before another version of x is produced, expressed

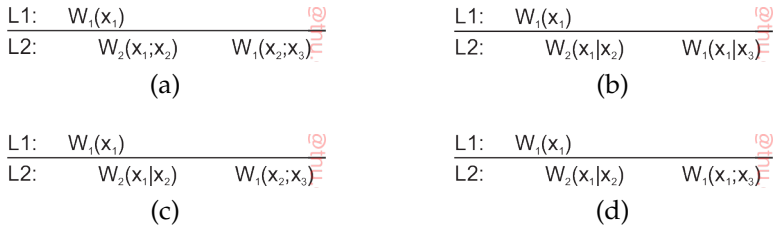

Figure 7.17: The write operations performed at two different local copies of the same data store. (a) A monotonic-write consistent data store. (b) A data store that does not provide monotonic-write consistency. (c) Again, no consistency as $WS(x_1|x2)$ and thus also $WS(x_1|x_3)$. (d) Consistent as $WS(x_1;x_3)$ although x_1 has apparently overwritten x_2.

by the operation $W_2(x_1|x_2)$. In this case, process P_2 produced a concurrent version to x_1, after which process P_1 simply produces version x_3, but again concurrently to x_1. Only slightly more subtle, but still violating monotonic-write consistency, is the situation sketched in Figure 7.17(c). Process P_1 now produces version x_3 which follows from x_2. However, because x_2 does not incorporate the write operations that led to x_1, that is, $WS(x_1|x_2)$, we also have $WS(x_1|x_3)$.

An interesting case is shown in Figure 7.17(d). The operation $W_2(x_1|x_2)$ produces version x_2 concurrently to x_1. However, later process P_1 produces version x_3, but apparently based on the fact that version x_1 had become available at L_2. How and when x_1 was transferred to L_2 is left unspecified, but in any case a write-write conflict was created with version x_2 and resolved in favor of x_1. A consequence is that the situation shown in Figure 7.17(d) follows the rules for monotonic-write consistency. Note, however, that any subsequent write by process P_2 at L_2 (without having read version x_1) will immediately violate consistency again. How such a violation can be prevented is left as an exercise to the reader.

Note that, by the definition of monotonic-write consistency, write operations by the same process are performed in the same order as they are initiated. A somewhat weaker form of monotonic writes is one in which the effects of a write operation are seen only if all preceding writes have been carried out as well, but perhaps not in the order in which they have been originally initiated. This consistency is applicable in those cases in which write operations are commutative, so that ordering is really not necessary. Details are found in [Terry et al., 1994].

Read your writes

A data store is said to provide **read-your-writes 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.*

In other words, a write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place.

The absence of read-your-writes consistency is sometimes experienced when updating Web documents and subsequently viewing the effects. Update operations frequently take place by means of a standard editor or word processor, perhaps embedded as part of a content management system, which then saves the new version on a file system that is shared by the Web server. The user's Web browser accesses that same file, possibly after requesting it from the local Web server. However, once the file has been fetched, either the server or the browser often caches a local copy for subsequent accesses. Consequently, when the Web page is updated, the user will not see the effects if the browser or the server returns the cached copy instead of the original file. Read-your-writes consistency can guarantee that if the editor and browser are integrated into a single program, the cache is invalidated when the page is updated, so that the updated file is fetched and displayed.

Similar effects occur when updating passwords. For example, to enter a digital library on the Web, it is often necessary to have an account with an accompanying password. However, changing a password may take some time to come into effect, with the result that the library may be inaccessible to the user for a few minutes. The delay can be caused because a separate server is used to manage passwords and it may take some time to subsequently propagate (encrypted) passwords to the various servers that constitute the library.

Figure 7.18(a) shows a data store that provides read-your-writes consistency. Note that Figure 7.18(a) is very similar to Figure 7.16(a), except that consistency is now determined by the last write operation by process P_1, instead of its last read.

Figure 7.18: (a) A data store that provides read-your-writes consistency. (b) A data store that does not.

In Figure 7.18(a) process P_1 performed a write operation $W_1(x_1)$ and later a read operation at a different local copy. Read-your-writes consistency guarantees that the effects of the write operation can be seen by the succeeding read operation. This is expressed by $W_2(x_1;x_2)$, which states that a process P_2 produced a new version of x, yet one based on x_1. In contrast, in Figure 7.18(b) process P_2 produces a version concurrently to x_1, expressed as $W_2(x_1|x_2)$.

This means that the effects of the previous write operation by process P_1 have not been propagated to L_2 at the time x_2 was produced. When P_1 reads x_2, it will not see the effects of its own write operation at L_1.

Writes follow reads

The last client-centric consistency model is one in which updates are propagated as the result of previous read operations. A data store is said to provide **writes-follow-reads** consistency, if the following holds.

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

In other words, any successive write operation by a process on a data item x will be performed on a copy of x that is up to date with the value most recently read by that process.

Writes-follow-reads consistency can be used to guarantee that users of a network newsgroup see a posting of a reaction to an article only after they have seen the original article [Terry et al., 1994]. To understand the problem, assume that a user first reads an article A. Then, she reacts by posting a response B. By requiring writes-follow-reads consistency, B will be written to any copy of the newsgroup only after A has been written as well. Note that users who only read articles need not require any specific client-centric consistency model. The writes-follows-reads consistency assures that reactions to articles are stored at a local copy only if the original is stored there as well.

Figure 7.19: (a) A writes-follow-reads consistent data store. (b) A data store that does not provide writes-follow-reads consistency.

This consistency model is shown in Figure 7.19. In Figure 7.19(a), process P_2 reads version x_1 at local copy L_1. This version of x was previously produced at L_1 by process P_1 through the operation $W_1(x_1)$. That version was subsequently propagated to L_2, and used by another process P_3 to produce a new version x_2, expressed as $W_3(x_1;x_2)$. When process P_2 later updates its version of x after moving to L_2, it is known that it will operate on a version that follows from x_1, expressed as $W_2(x_2;x_3)$. Because we also have $W_3(x_1;x_2)$, we known that $WS(x_1;x_3)$.

The situation shown in Figure 7.19(b) is different. Process P_3 produces a version x_2 concurrently to that of x_1. As a consequence, when P_2 updates x

after reading x_1, it will be updating a version it had not read before. Writes-follow-reads consistency is then violated.

7.4 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 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.

Finding the best server location

Where perhaps over a decade ago one could be concerned about where to place an individual server, matters have changed considerably with the advent of the many large-scale data centers located across the Internet. Likewise, connectivity continues to improve, making *precisely* locating servers less critical.

Note 7.5 (Advanced: 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. Qiu et al. [2001] 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).

As an alternative, Radoslavov et al. [2001] propose to ignore the position of clients and only take the topology of the Internet as formed by the autonomous systems. An **autonomous system** (**AS**) can best be viewed as a network in which the nodes all run the same routing protocol and which is managed by a single organization. As of 2015, there were some 30,000 ASes. Radoslavov et al. first consider the largest AS and place a server on the router with the largest number

of network interfaces (i.e., links). This algorithm is then repeated with the second largest AS, and so on.

As it turns out, client-unaware server placement achieves similar results as client-aware placement, under the assumption that clients are uniformly distributed across the Internet (relative to the existing topology). To what extent this assumption is true is unclear. It has not been well studied.

One problem with these algorithms is that they are computationally expensive. For example, both the previous algorithms have a complexity that is higher than $\mathcal{O}(N^2)$, where N is the number of locations to inspect. In practice, this means that for even a few thousand locations, a computation may need to run for tens of minutes. This may be unacceptable.

Szymaniak et al. [2006] have developed a method by which a region for placing replicas can be quickly identified. A region is identified to be a collection of nodes accessing the same content, but for which the internode latency is low. The goal of the algorithm is first to select the most demanding regions–that is, the one with the most nodes–and then to let one of the nodes in such a region act as replica server.

To this end, 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.

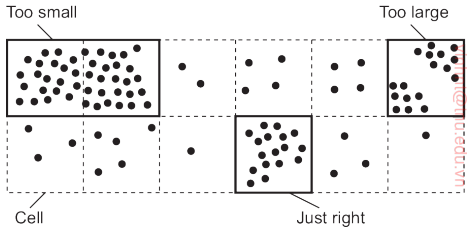

Figure 7.20: Choosing a proper cell size for server placement.

Obviously, the cell size is important, as shown in Figure 7.20. 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. With this cell size, it can be shown that the algorithm performs as well as the close-to-optimal one described by Qiu et al. [2001], but having a much lower complexity: $\mathcal{O}(N \times \max\{\log(N), K\})$. To give an impression what this result means: experiments show that computing the 20 best replica locations for a

collection of 64,000 nodes is approximately 50,000 times faster. As a consequence, replica-server placement can now be done in real time.

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 Figure 7.21.

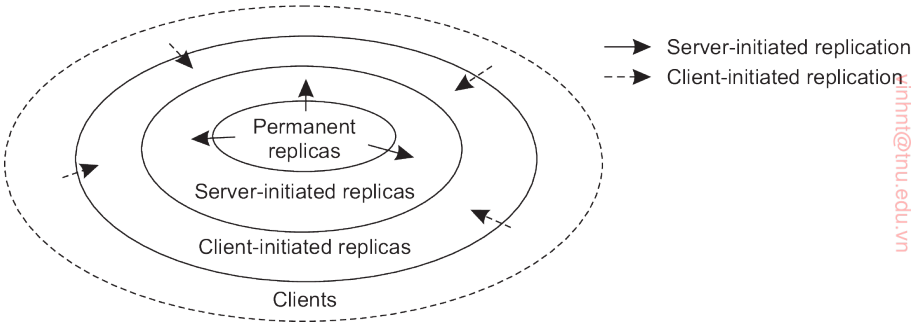

Figure 7.21: The logical organization of different kinds of copies of a data store into three concentric rings.

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 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 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 various 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 replicas, which are more or less statically configured.

Similar static organizations also appear with distributed databases [Kemme et al., 2010; Özsu and Valduriez, 2011]. Again, the database can be distributed and replicated across a number of servers that together form a cluster of servers, 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 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 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.

Note 7.6 (Advanced: An example of dynamic Web-content placement)
The problem of dynamically placing replicas has since long been addressed in Web hosting services. These services offer an often 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 not that difficult. An early case toward dynamic replication of files in the case of a Web hosting service is described by 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, 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, when a client C enters the service, it does so through a server close to it. 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 $\mathsf{cnt}_Q(P, F)$. This situation is shown in Figure 7.22.

When the number of requests for a specific file F at server S drops below a deletion threshold $\mathsf{del}(S, F)$, 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.

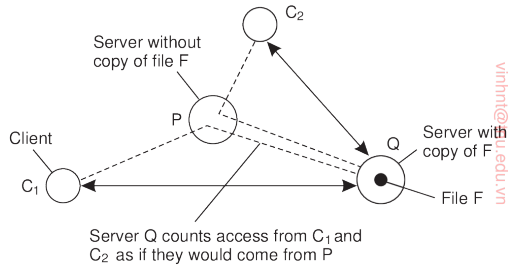

Figure 7.22: Counting access requests from different clients.

A replication threshold $\mathsf{rep}(S, F)$, 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 to be only 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 $\mathsf{del}(Q, F)$, it will delete F unless it is the last copy. Furthermore, if for some server P, $\mathsf{cnt}_Q(P, F)$ 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 $\mathsf{rep}(Q, F)$. Server Q checks all other servers in the Web hosting service, starting with the one farthest away. If, for some server R, $\mathsf{cnt}_Q(R, F)$ exceeds a certain fraction of all requests for F at Q, an attempt is made to replicate F to R.

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. However, permanent replicas are 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, 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, 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 has been losing 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 [Noble et al., 1999].

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:

- Propagate only a notification of an update.
- Transfer data from one copy to another.
- 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 (and sending 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 generally applied when strong consistency is required.

This need for strong 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 at least one, but perhaps 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 Figure 7.23. 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 7.23: 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, 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. In the case of replica management, 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, originally introduced by Gray and Cheriton [1989], provide a convenient mechanism for dynamically switching between a push-based and pull-based strategy. Consider the following lease system that allows the expiration time to be dynamically adapted depending on different lease criteria, described in [Duvvuri et al., 2003]. We 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, for example, Web-based data and regular files. 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 **state-based lease** 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 expecting to offload itself so that it can handle requests more efficiently. The obvious drawback is that it may need to do more work when the read-to-update ratio is high.

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. When the two are carefully integrated, 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.