Principles of Computer System Design

An Introduction

Chapter 10
Consistency

Jerome H. Saltzer

M. Frans Kaashoek

Massachusetts Institute of Technology

Version 5.0

Suggestions, Comments, Corrections, and Requests to waive license restrictions: Please send correspondence by electronic mail to:

Saltzer@mit.edu

and

kaashoek@mit.edu

CHAPTER

Consistency

10

CHAPTER CONTENTS

Overview

The previous chapter developed *all-or-nothing atomicity* and *before-or-after atomicity,* two properties that define a transaction. This chapter introduces or revisits several applications that can make use of transactions. Section 10.1 introduces constraints and discusses how transactions can be used to maintain invariants and implement memory models that provide interface consistency. Sections 10.2 and 10.3 develop techniques used in two different application areas, *caching* and *geographically distributed replication*, to achieve higher performance and greater durability, respectively. Section 10.4 discusses *reconciliation*, which is a way of restoring the constraint that replicas be identical if their contents should drift apart. Finally, Section 10.5 considers some perspectives relating to Chapters 9[on-line] and 10.

10.1 Constraints and Interface Consistency

One common use for transactions is to maintain *constraints*. A constraint is an application-defined requirement that every update to a collection of data preserve some specified invariant. Different applications can have quite different constraints. Here are some typical constraints that a designer might encounter:

- Table management: The variable that tells the number of entries should equal the number of entries actually in the table.
- Double-linked list management: The forward pointer in a list cell, `A`, should refer a list cell whose back pointer refers to `A`.
- Disk storage management: Every disk sector should be assigned either to the free list or to exactly one file.
- Display management: The pixels on the screen should match the description in the display list.
- Replica management: A majority (or perhaps all) of the replicas of the data should be identical.
- Banking: The sum of the balances of all credit accounts should equal the sum of the balances of all debit accounts.
- Process control: At least one of the valves on the boiler should always be open.

As was seen in Chapter 9[on-line], maintaining a constraint over data within a single file can be relatively straightforward, for example by creating a shadow copy. Maintaining constraints across data that is stored in several files is harder, and that is one of the primary uses of transactions. Finally, two-phase commit allows maintaining a constraint that involves geographically separated files despite the hazards of communication.

A constraint usually involves more than one variable data item, in which case an update action by nature must be composite—it requires several steps. In the midst of those steps, the data will temporarily be inconsistent. In other words, there will be times when the data violates the invariant. During those times, there is a question about what to do if someone—another thread or another client—asks to read the data. This question is one of interface, rather than of internal operation, and it reopens the discussion of memory coherence and data consistency models introduced in Section 2.1.1.1. Different designers have developed several data consistency models to deal with this inevitable temporary inconsistency. In this chapter we consider two of those models: *strict consistency* and *eventual consistency*.

The first model, *strict consistency*, hides the constraint violation behind modular boundaries. Strict consistency means that actions outside the transaction performing the update will never see data that is inconsistent with the invariant. Since strict consistency is an interface concept, it depends on actions honoring abstractions, for example by using only the intended reading and writing operations. Thus, for a cache, read/write coherence is a strict consistency specification: "The result of a READ of a named object is always the value that was provided by the most recent WRITE to that object". This specification does not demand that the replica in the cache always be identical to the replica in the backing store, it requires only that the cache deliver data at its interface that meets the specification.

Applications can maintain strict consistency by using transactions. If an action is all-or-nothing, the application can maintain the outward appearance of consistency despite failures, and if an action is before-or-after, the application can maintain the outward appearance of consistency despite the existence of other actions concurrently reading or updating the same data. Designers generally strive for strict consistency in any situation where inconsistent results can cause confusion, such as in a multiprocessor system, and in situations where mistakes can have serious negative consequences, for example in banking and safety-critical systems. Section 9.1.6 mentioned two other consistency models, sequential consistency and external time consistency. Both are examples of strict consistency.

The second, more lightweight, way of dealing with temporary inconsistency is called *eventual consistency*. Eventual consistency means that after a data update the constraint may not hold until some unspecified time in the future. An observer may, using the standard interfaces, discover that the invariant is violated, and different observers may even see different results. But the system is designed so that once updates stop occurring, it will make a best effort drive toward the invariant.

Eventual consistency is employed in situations where performance or availability is a high priority and temporary inconsistency is tolerable and can be easily ignored. For example, suppose a Web browser is to display a page from a distant service. The page has both a few paragraphs of text and several associated images. The browser obtains the text immediately, but it will take some time to download the images. The invariant is that the appearance on the screen should match the Web page specification. If the browser renders the text paragraphs first and fills in the images as they arrive, the human reader finds that behavior not only acceptable, but perhaps preferable to staring at the previous screen until the new one is completely ready. When a person can say, "Oh, I see what is happening," eventual consistency is usually acceptable, and in cases such as the Web browser it can even improve human engineering. For a second example, if a librarian catalogs a new book and places it on the shelf, but the public version of the library catalog doesn't include the new book until the next day, there is an observable inconsistency, but most library patrons would find it tolerable and not particularly surprising.

Eventual consistency is sometimes used in replica management because it allows for relatively loose coupling among the replicas, thus taking advantage of independent failure. In some applications, continuous service is a higher priority than always-consistent answers. If a replica server crashes in the middle of an update, the other replicas may be able to continue to provide service, even though some may have been updated and some may have not. In contrast, a strict consistency algorithm may have to refuse to provide service until a crashed replica site recovers, rather than taking a risk of exposing an inconsistency.

The remaining sections of this chapter explore several examples of strict and eventual consistency in action. A cache can be designed to provide either strict or eventual consistency; Section 10.2 provides the details. The Internet Domain Name System, described in Section 4.4 and revisited in Section 10.2.2, relies on eventual consistency in updating its caches, with the result that it can on occasion give inconsistent answers. Similarly, for the geographically replicated durable storage of Section 10.3 a designer can choose either a strict or an eventual consistency model. When replicas are maintained on devices that are only occasionally connected, eventual consistency may be the only choice, in which case reconciliation, the topic of Section 10.4, drives occasionally connected replicas toward eventual consistency. The reader should be aware that these examples do not provide a comprehensive overview of consistency; instead they are intended primarily to create awareness of the issues involved by illustrating a few of the many possible designs.

10.2 Cache Coherence

10.2.1 Coherence, Replication, and Consistency in a Cache

Chapter 6 described the cache as an example of a multilevel memory system. A cache can also be thought of as a replication system whose primary goal is performance, rather than reliability. An invariant for a cache is that the replica of every data item in the primary store (that is, the cache) should be identical to the corresponding replica in the secondary memory. Since the primary and secondary stores usually have different latencies, when an action updates a data value, the replica in the primary store will temporarily be inconsistent with the one in the secondary memory. How well the multilevel memory system hides that inconsistency is the question.

A cache can be designed to provide either strict or eventual consistency. Since a cache, together with its backing store, is a memory system, a typical interface specification is that it provide read/write coherence, as defined in Section 2.1.1.1, for the entire name space of the cache:

- **The result of a read of a named object is always the value of the most recent write to that object.**

Read/write coherence is thus a specification that the cache provide strict consistency.

A write-through cache provides strict consistency for its clients in a straightforward way: it does not acknowledge that a write is complete until it finishes updating both the primary and secondary memory replicas. Unfortunately, the delay involved in waiting for the write-through to finish can be a performance bottleneck, so write-through caches are not popular.

A non-write-through cache acknowledges that a write is complete as soon as the cache manager updates the primary replica, in the cache. The thread that performed the write can go about its business expecting that the cache manager will eventually update the secondary memory replica and the invariant will once again hold. Meanwhile, if that same thread reads the same data object by sending a READ request to the cache, it will receive the updated value from the cache, even if the cache manager has not yet restored the invariant. Thus, because the cache manager masks the inconsistency, a non-write-through cache can still provide strict consistency.

On the other hand, if there is more than one cache, or other threads can read directly from the secondary storage device, the designer must take additional measures to ensure that other threads cannot discover the violated constraint. If a concurrent thread reads a modified data object via the same cache, the cache will deliver the modified version, and thus maintain strict consistency. But if a concurrent thread reads the modified data object directly from secondary memory, the result will depend on whether or not the cache manager has done the secondary memory update. If the second thread has its own cache, even a write-through design may not maintain consistency because updating the secondary memory does not affect a potential replica hiding in the second thread's cache. Nevertheless, all is not lost. There are at least three ways to regain consistency, two of which provide strict consistency, when there are multiple caches.

10.2.2 Eventual Consistency with Timer Expiration

The Internet Domain Name System, whose basic operation was described in Section 4.4, provides an example of an eventual consistency cache that does *not* meet the read/write coherence specification. When a client calls on a DNS server to do a recursive name lookup, if the DNS server is successful in resolving the name it caches a copy of the answer as well as any intermediate answers that it received. Suppose that a client asks some local name server to resolve the name `ginger.pedantic.edu`. In the course of doing so, the local name server might accumulate the following name records in its cache:

```
names.edu              198.41.0.4    name server for .edu
ns.pedantic.edu        128.32.25.19  name server for .pedantic.edu
ginger.pedantic.edu    128.32.247.24 target host name
```

If the client then asks for thyme.pedantic.edu the local name server will be able to use the cached record for ns.pedantic.edu to directly ask that name server, without having to go back up to the root to find names.edu and thence to names.edu to find ns.pedantic.edu.

Now, suppose that a network manager at Pedantic University changes the Internet address of ginger.pedantic.edu to 128.32.201.15. At some point the manager updates the authoritative record stored in the name server ns.pedantic.edu. The problem is that local DNS caches anywhere in the Internet may still contain the old record of the address of ginger.pedantic.edu. DNS deals with this inconsistency by limiting the lifetime of a cached name record. Recall that every name server record comes with an expiration time, known as the *time-to-live* (TTL) that can range from seconds to months. A typical time-to-live is one hour; it is measured from the moment that the local name server receives the record. So, until the expiration time, the local cache will be inconsistent with the authoritative version at Pedantic University. The system will eventually reconcile this inconsistency. When the time-to-live of that record expires, the local name server will handle any further requests for the name ginger.pedantic.edu by asking ns.pedantic.edu for a new name record. That new name record will contain the new, updated address. So this system provides eventual consistency.

There are two different actions that the network manager at Pedantic University might take to make sure that the inconsistency is not an inconvenience. First, the network manager may temporarily reconfigure the network layer of ginger.pedantic.edu to advertise both the old and the new Internet addresses, and then modify the authoritative DNS record to show the new address. After an hour has passed, all cached DNS records of the old address will have expired, and ginger.pedantic.edu can be reconfigured again, this time to stop advertising the old address. Alternatively, the network manager may have realized this change is coming, so a few hours in advance he or she modifies just the time-to-live of the authoritative DNS record, say to five minutes, without changing the Internet address. After an hour passes, all cached DNS records of this address will have expired, and any currently cached record will expire in five minutes or less. The manager now changes both the Internet address of the machine and also the authoritative DNS record of that address, and within a few minutes everyone in the Internet will be able to find the new address. Anyone who tries to use an old, cached, address will receive no response. But a retry a few minutes later will succeed, so from the point of view of a network client the outcome is similar to the case in which ginger.pedantic.edu crashes and restarts—for a few minutes the server is non-responsive.

There is a good reason for designing DNS to provide eventual, rather than strict, consistency, and for not requiring read/write coherence. Replicas of individual name records may potentially be cached in any name server anywhere in the Internet—there are thousands, perhaps even millions of such caches. Alerting every name server that might have cached the record that the Internet address of ginger.pedantic.edu changed would be a huge effort, yet most of those caches probably don't actually have a copy of this particular record. Furthermore, it turns out not to be that important because, as described in the previous paragraph, a network manager can easily mask any temporary inconsistency by configuring address advertisement or adjusting the time-to-live. Eventual consistency with expiration is an efficient strategy for this job.

10.2.3 Obtaining Strict Consistency with a Fluorescent Marking Pen

In certain special situations, it is possible to regain strict consistency, and thus read/write coherence, despite the existence of multiple, private caches: If only a few variables are actually both shared and writable, mark just those variables with a fluorescent marking pen. The meaning of the mark is "don't cache me". When someone reads a marked variable, the cache manager retrieves it from secondary memory and delivers it to the client, but does not place a replica in the cache. Similarly, when a client writes a marked variable, the cache manager notices the mark in secondary memory and does not keep a copy in the cache. This scheme erodes the performance-enhancing value of the cache, so it would not work well if most variables have don't-cache-me marks.

The World Wide Web uses this scheme for Web pages that may be different each time they are read. When a client asks a Web server for a page that the server has marked "don't cache me", the server adds to the header of that page a flag that instructs the browser and any intermediaries not to cache that page.

The Java language includes a slightly different, though closely related, concept, intended to provide read/write coherence despite the presence of caches, variables in registers, and reordering of instructions, all of which can compromise strict consistency when there is concurrency. The Java memory model allows the programmer to declare a variable to be **volatile**. This declaration tells the compiler to take whatever actions (such as writing registers back to memory, flushing caches, and blocking any instruction reordering features of the processor) might be needed to ensure read/write coherence for the **volatile** variable within the actual memory model of the underlying system. Where the fluorescent marking pen marks a variable for special treatment by the memory system, the **volatile** declaration marks a variable for special treatment by the interpreter.

10.2.4 Obtaining Strict Consistency with the Snoopy Cache

The basic idea of most cache coherence schemes is to somehow *invalidate* cache entries whenever they become inconsistent with the authoritative replica. One situation where a designer can use this idea is when several processors share the same secondary memory. If the processors could also share the cache, there would be no problem. But a shared cache tends to reduce performance, in two ways. First, to minimize latency the designer would prefer to integrate the cache with the processor, but a shared cache eliminates that option. Second, there must be some mechanism that arbitrates access to the shared cache by concurrent processors. That arbitration mechanism must enforce waits that increase access latency even more. Since the main point of a processor cache is to reduce latency, each processor usually has at least a small private cache.

Making the private cache write-through would ensure that the replica in secondary memory tracks the replica in the private cache. But write-through does not update any replicas that may be in the private caches of other processors, so by itself it doesn't provide read/write coherence. We need to add some way of telling those processors to invalidate any replicas their caches hold.

A naive approach would be to run a wire from each processor to the others and specify that whenever a processor writes to memory, it should send a signal on this wire. The other processors should, when they see the signal, assume that something in their cache has changed and, not knowing exactly what, invalidate everything their cache currently holds. Once all caches have been invalidated, the first processor can then confirm completion of its own write. This scheme would work, but it would have a disastrous effect on the cache hit rate. If 20% of processor data references are write operations, each processor will receive signals to invalidate the cache roughly every fifth data reference by each other processor. There would not be much point in having a big cache, since it would rarely have a chance to hold more than half a dozen valid entries.

To avoid invalidating the entire cache, a better idea would be to somehow communicate to the other caches the specific address that is being updated. To rapidly transmit an entire memory address in hardware could require adding a lot of wires. The trick is to realize that there is already a set of wires in place that can do this job: the memory bus. One designs each private cache to actively monitor the memory bus. If the cache notices that anyone else is doing a write operation via the memory bus, it grabs the memory address from the bus and invalidates any copy of data it has that corresponds to that address. A slightly more clever design will also grab the data value from the bus as it goes by and update, rather than invalidate, its copy of that data. These are two variations on what is called the *snoopy cache* [Suggestions for Further Reading 10.1.1]—each cache is snooping on bus activity. Figure 10.1 illustrates the snoopy cache.

The registers of the various processors constitute a separate concern because they may also contain copies of variables that were in a cache at the time a variable in the cache was invalidated or updated. When a program loads a shared variable into a register, it should be aware that it is shared, and provide coordination, for example through the use of locks, to ensure that no other processor can change (and thus invalidate) a variable that this processor is holding in a register. Locks themselves generally are implemented using write-through, to ensure that cached copies do not compromise the single-acquire protocol.

A small cottage industry has grown up around optimizations of cache coherence protocols for multiprocessor systems both with and without buses, and different designers have invented many quite clever speed-up tricks, especially with respect to locks. Before undertaking a multiprocessor cache design, a prospective processor architect should review the extensive literature of the area. A good place to start is with Chapter 8 of *Computer Architecture: A Quantitative Approach,* by Hennessy and Patterson [Suggestions for Further Reading 1.1.1].

10.3 Durable Storage Revisited: Widely Separated Replicas

10.3.1 Durable Storage and the Durability Mantra

Chapter 8[on-line] demonstrated how to create durable storage using a technique called *mirroring*, and Section 9.7[on-line] showed how to give the mirrored replicas the all-or-nothing property when reading and writing. Mirroring is characterized by writing the replicas synchronously—that is, waiting for all or a majority of the replicas to be written before going on to the next action. The replicas themselves are called *mirrors*, and they are usually created on a physical unit basis. For example, one common RAID configuration uses multiple disks, on each of which the same data is written to the same numbered sector, and a write operation is not considered complete until enough mirror copies have been successfully written.

Mirroring helps protect against internal failures of individual disks, but it is not a magic bullet. If the application or operating system damages the data before writing it, all the replicas will suffer the same damage. Also, as shown in the fault tolerance analyses in the previous two chapters, certain classes of disk failure can obscure discovery that a replica was not written successfully. Finally, there is a concern for where the mirrors are physically located.

Placing replicas at the same physical location does not provide much protection against the threat of environmental faults, such as fire or earthquake. Having them all

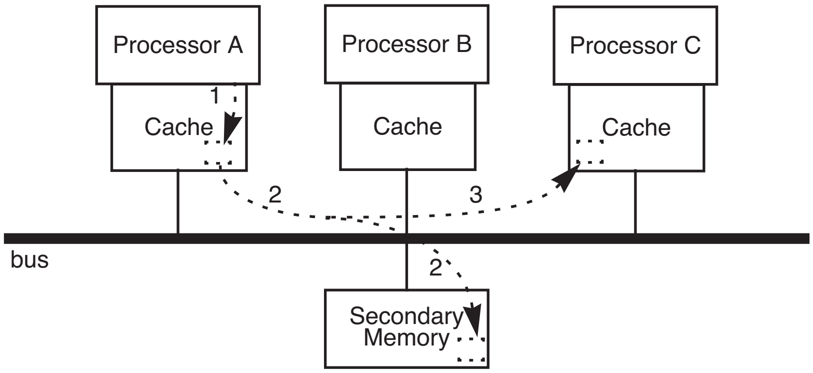

FIGURE 10.1

A configuration for which a snoopy cache can restore strict consistency and read/write coherence. When processor A writes to memory (arrow 1), its write-through cache immediately updates secondary memory using the next available bus cycle (arrow 2). The caches for processors B and C monitor (“snoop on”) the bus address lines, and if they notice a bus write cycle for an address they have cached, they update (or at least invalidate) their replica of the contents of that address (arrow 3).

under the same administrative control does not provide much protection against administrative bungling. To protect against these threats, the designer uses a powerful design principle:

The durability mantra

Multiple copies, widely separated and independently administered…
Multiple copies, widely separated and independently administered…

Sidebar 4.5 referred to Ross Anderson's Eternity Service, a system that makes use of this design principle. Another formulation of the durability mantra is "lots of copies keep stuff safe" [Suggestions for Further Reading 10.2.3]. The idea is not new: "…let us save what remains; not by vaults and locks which fence them from the public eye and use in consigning them to the waste of time, but by such a multiplication of copies, as shall place them beyond the reach of accident."*

The first step in applying this design principle is to separate the replicas geographically. The problem with separation is that communication with distant points has high latency and is also inherently unreliable. Both of those considerations make it problematic to write the replicas synchronously. When replicas are made asynchronously, one of the replicas (usually the first replica to be written) is identified as the *primary copy*, and the site that writes it is called the *master*. The remaining replicas are called *backup copies*, and the sites that write them are called *slaves*.

The constraint usually specified for replicas is that they should be identical. But when replicas are written at different times, there will be instants when they are not identical; that is, they violate the specified constraint. If a system failure occurs during one of those instants, violation of the constraint can complicate recovery because it may not be clear which replicas are authoritative. One way to regain some simplicity is to organize the writing of the replicas in a way understandable to the application, such as file-by-file or record-by-record, rather than in units of physical storage such as disk sector-by-sector. That way, if a failure does occur during replica writing, it is easier to characterize the state of the replica: some files (or records) of the replica are up to date, some are old, the one that was being written may be damaged, and the application can do any further recovery as needed. Writing replicas in a way understandable to the application is known as making *logical copies*, to contrast it with the *physical copies* usually associated with mirrors. Logical copying has the same attractions as logical locking, and also some of the performance disadvantages, because more software layers must be involved and it may require more disk seek arm movement.

In practice, replication schemes can be surprisingly complicated. The primary reason is that the purpose of replication is to suppress unintended changes to the data caused by random decay. But decay suppression also complicates intended changes, since one must

* Letter from Thomas Jefferson to the publisher and historian Ebenezer Hazard, February 18, 1791. Library of Congress, *The Thomas Jefferson Papers Series 1. General Correspondence. 1651-1827.*

now update more than one copy, while being prepared for the possibility of a failure in the midst of that update. In addition, if updates are frequent, the protocols to perform update must not only be correct and robust, they must also be efficient. Since multiple replicas can usually be read and written concurrently, it is possible to take advantage of that possibility to enhance overall system performance. But performance enhancement can then become a complicating requirement of its own, one that interacts strongly with a requirement for strict consistency.

10.3.2 Replicated State Machines

Data replicas require a management plan. If the data is written exactly once and never again changed, the management plan can be fairly straightforward: make several copies, put them in different places so they will not all be subject to the same environmental faults, and develop algorithms for reading the data that can cope with loss of, disconnection from, and decay of data elements at some sites.

Unfortunately, most real world data need to be updated, at least occasionally, and update greatly complicates management of the replicas. Fortunately, there exists an easily-described, systematic technique to ensure correct management. Unfortunately, it is surprisingly hard to meet all the conditions needed to make it work.

The systematic technique is a *sweeping simplification* known as the *replicated state machine*. The idea is to identify the data with the state of a finite state machine whose inputs are the updates to be made to the data, and whose operation is to make the appropriate changes to the data, as illustrated in Figure 10.2. To maintain identical data replicas, co-locate with each of those replicas a replica of the state machine, and send the same inputs to each state machine. Since the state of a finite state machine is at all times determined by its prior state and its inputs, the data of the various replicas will, in principle, perfectly match one another.

The concept is sound, but four real-world considerations conspire to make this method harder than it looks:

1. All of the state machine replicas must receive the same inputs, in the same order. Agreeing on the values and order of the inputs at separated sites is known as achieving *consensus*. Achieving consensus among sites that do not have a common clock, that can crash independently, and that are separated by a best-effort communication network is a project in itself. Consensus has received much attention from theorists, who begin by defining its core essence, known as *the consensus problem*: to achieve agreement on a single binary value. There are various algorithms and protocols designed to solve this problem under specified conditions, as well as proofs that with certain kinds of failures consensus is impossible to reach. When conditions permit solving the core consensus problem, a designer can then apply bootstrapping to come to agreement on the complete set of values and order of inputs to a set of replicated state machines.

2. All of the data replicas (in Figure 10.2, the "prior state") must be identical. The problem is that random decay events can cause the data replicas to drift apart, and updates that occur when they have drifted can cause them to drift further apart. So there needs to be a plan to check for this drift and correct it. The mechanism that identifies such differences and corrects them is known as *reconciliation*.
3. The replicated state machines must also be identical. This requirement is harder to achieve than it might at first appear. Even if all the sites run copies of the same program, the operating environment surrounding that program may affect its behavior, and there can be transient faults that affect the operation of individual state machines differently. Since the result is again that the data replicas drift apart, the same reconciliation mechanism that fights decay may be able to handle this problem.
4. To the extent that the replicated state machines really are identical, they will contain identical implementation faults. Updates that cause the faults to produce errors in the data will damage all the replicas identically, and reconciliation can neither detect nor correct the errors.

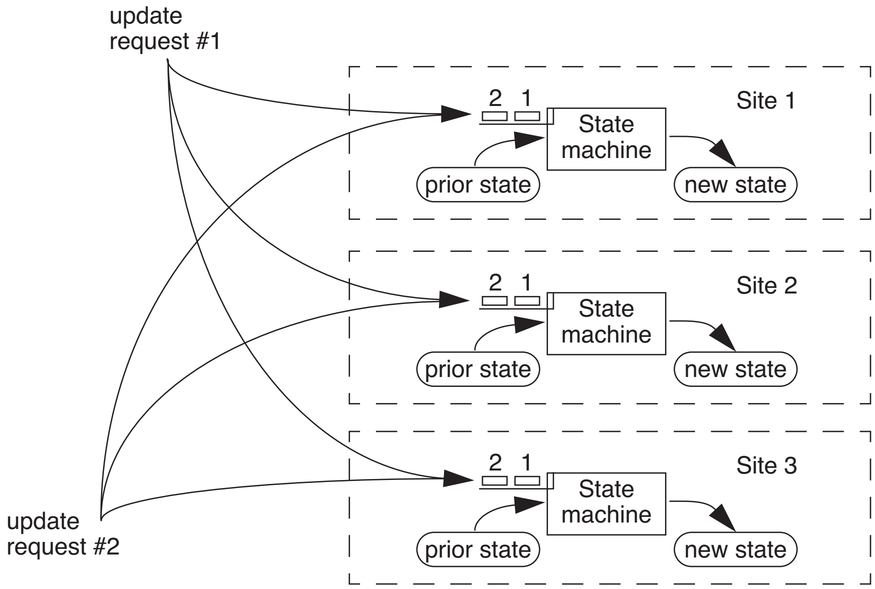

FIGURE 10.2

Replicated state machines. If *N* identical state machines that all have the same prior state receive and perform the same update requests in the same order, then all *N* of the machines will enter the same new state.

The good news is that the replicated state machine scheme not only is systematic, but it lends itself to modularization. One module can implement the consensus-achieving algorithm; a second set of modules, the state machines, can perform the actual updates; and a third module responsible for reconciliation can periodically review the data replicas to verify that they are identical and, if necessary, initiate repairs to keep them that way.

10.3.3 Shortcuts to Meet more Modest Requirements

The replicated state machine method is systematic, elegant, and modular, but its implementation requirements are severe. At the other end of the spectrum, some applications can get along with a much simpler method: implement just a *single state machine*. The idea is to carry out all updates at one replica site, generating a new version of the database at that site, and then somehow bring the other replicas into line. The simplest, brute force scheme is to send a copy of this new version of the data to each of the other replica sites, completely replacing their previous copies. This scheme is a particularly simple example of master/slave replication. One of the things that makes it simple is that there is no need for consultation among sites; the master decides what to do and the slaves just follow along.

The single state machine with brute force copies works well if:

- The data need to be updated only occasionally.
- The database is small enough that it is practical to retransmit it in its entirety.
- There is no urgency to make updates available, so the master can accumulate updates and perform them in batches.
- The application can get along with temporary inconsistency among the various replicas. Requiring clients to read from the master replica is one way to mask the temporary inconsistency. On the other hand if, for improved performance, clients are allowed to read from any available replica, then during an update a client reading data from a replica that has received the update may receive different answers from another client reading data from a different replica to which the update hasn't propagated yet.

This method is subject to data decay, just as is the replicated state machine, but the effects of decay are different. Undetected decay of the master replica can lead to a disaster in which the decay is propagated to the slave replicas. On the other hand, since update installs a complete new copy of the data at each slave site, it incidentally blows away any accumulated decay errors in slave replicas, so if update is frequent, it is usually not necessary to provide reconciliation. If updates are so infrequent that replica decay is a hazard, the master can simply do an occasional dummy update with unchanged data to reconcile the replicas.

The main defect of the single state machine is that even though data access can be fault tolerant—if one replica goes down, the others may still available for reading—data update is not: if the primary site fails, no updates are possible until that failure is detected and repaired. Worse, if the primary site fails while in the middle of sending out an update, the replicas may remain inconsistent until the primary site recovers. This whole approach doesn't work well for some applications, such as a large database with a requirement for strict consistency and a performance goal that can be met only by allowing concurrent reading of the replicas.

Despite these problems, the simplicity is attractive, and in practice many designers try to get away with some variant of the single state machine method, typically tuned up with one or more enhancements:

- The master site can distribute just those parts of the database that changed (the updates are known as "deltas" or "diffs") to the replicas. Each replica site must then run an engine that can correctly update the database using the information in the deltas. This scheme moves back across the spectrum in the direction of the replicated state machine. Though it may produce a substantial performance gain, such a design can end up with the disadvantages of both the single and the replicated state machines.
- Devise methods to reduce the size of the time window during which replicas may appear inconsistent to reading clients. For example, the master could hold the new version of the database in a shadow copy, and ask the slave sites to do the same, until all replicas of the new version have been successfully distributed. Then, short messages can tell the slave sites to make the shadow file the active database. (This model should be familiar: a similar idea was used in the design of the two-phase commit protocol described in Chapter 9[on-line].)
- If the database is large, partition it into small regions, each of which can be updated independently. Section 10.3.7, below, explores this idea in more depth. (The Internet Domain Name System is for the most part managed as a large number of small, replicated partitions.)
- Assign a different master to each partition, to distribute the updating work more evenly and increase availability of update.
- Add fault tolerance for data update when a master site fails by using a consensus algorithm to choose a new master site.
- If the application is one in which the data is insensitive to the order of updates, implement a replicated state machine without a consensus algorithm. This idea can be useful if the only kind of update is to add new records to the data and the records are identified by their contents, rather than by their order of arrival. Members of a workgroup collaborating by e-mail typically see messages from other group members this way. Different users may find that received messages appear in different orders, and may even occasionally see one member answer a question that another member apparently hasn't yet asked, but if the e-mail system is working correctly, eventually everyone sees every message.

- The master site can distribute just its update log to the replica sites. The replica sites can then run REDO on the log entries to bring their database copies up to date. Or, the replica site might just maintain a complete log replica rather than the database itself. In the case of a disaster at the master site, one of the log replicas can then be used to reconstruct the database.

This list just touches the surface. There seem to be an unlimited number of variations in application-dependent ways of doing replication.

10.3.4 Maintaining Data Integrity

In updating a replica, many things can go wrong: data records can be damaged or even completely lost track of in memory buffers of the sending or receiving systems, transmission can introduce errors, and operators or administrators can make blunders, to name just some of the added threats to data integrity. The durability mantra suggests imposing physical and administrative separation of replicas to make threats to their integrity more independent, but the threats still exist.

The obvious way to counter these threats to data integrity is to apply the method suggested on page 9–94 to counter spontaneous data decay: plan to periodically compare replicas, doing so often enough that it is unlikely that all of the replicas have deteriorated. However, when replicas are not physically adjacent this obvious method has the drawback that bit-by-bit comparison requires transmission of a complete copy of the data from one replica site to another, an activity that can be time-consuming and possibly expensive.

An alternative and less costly method that can be equally effective is to calculate a *witness* of the contents of a replica and transmit just that witness from one site to another. The usual form for a witness is a hash value that is calculated over the content of the replica, thus attesting to that content. By choosing a good hash algorithm (for example, a cryptographic quality hash such as described in Sidebar 11.7) and making the witness sufficiently long, the probability that a damaged replica will have a hash value that matches the witness can be made arbitrarily small. A witness can thus stand in for a replica for purposes of confirming data integrity or detecting its loss.

The idea of using witnesses to confirm or detect loss of data integrity can be applied in many ways. We have already seen checksums used in communications, both for end-to-end integrity verification (page 7–31) and in the link layer (page 7–40); checksums can be thought of as weak witnesses. For another example of the use of witnesses, a file system might calculate a separate witness for each newly written file, and store a copy of the witness in the directory entry for the file. When later reading the file, the system can recalculate the hash and compare the result with the previously stored witness to verify the integrity of the data in the file. Two sites that are supposed to be maintaining replicas of the file system can verify that they are identical by exchanging and comparing lists of witnesses. In Chapter 11[on-line] we will see that by separately protecting a witness one can also counter threats to data integrity that are posed by an adversary.

10.3.5 Replica Reading and Majorities

So far, we have explored various methods of creating replicas, but not how to use them. The simplest plan, with a master/slave system, is to direct all client read and write requests to the primary copy located at the master site, and treat the slave replicas exclusively as backups whose only use is to restore the integrity of a damaged master copy. What makes this plan simple is that the master site is in a good position to keep track of the ordering of read and write requests, and thus enforce a strict consistency specification such as the usual one for memory coherence: that a read should return the result of the most recent write.

A common enhancement to a replica system, intended to increase availability for read requests, is to allow reads to be directed to any replica, so that the data continues to be available even when the master site is down. In addition to improving availability, this enhancement may also have a performance advantage, since the several replicas can probably provide service to different clients at the same time. Unfortunately, the enhancement has the complication that there will be instants during update when the several replicas are not identical, so different readers may obtain different results, a violation of the strict consistency specification. To restore strict consistency, some mechanism that ensures before-or-after atomicity between reads and updates would be needed, and that before-or-after atomicity mechanism will probably erode some of the increased availability and performance.

Both the simple and the enhanced schemes consult only one replica site, so loss of data integrity, for example from decay, must be detected using just information local to that site, perhaps with the help of a witness stored at the replica site. Neither scheme takes advantage of the data content of the other replicas to verify integrity. A more expensive, but more reliable, way to verify integrity is for the client to also obtain a second copy (or a witness) from a different replica site. If the copy (or witness) from another site matches the data (or a just-calculated hash of the data) of the first site, confidence in the integrity of the data can be quite high. This idea can be carried further to obtain copies or witnesses from several of the replicas, and compare them. Even when there are disagreements, if a majority of the replicas or witnesses agree, the client can still accept the data with confidence, and might in addition report a need for reconciliation.

Some systems push the majority idea further by introducing the concept of a *quorum*. Rather than simply “more than half the replicas”, one can define separate read and write quorums, Q_r and Q_w, that have the property that $Q_r + Q_w > N_{replicas}$. This scheme declares a write to be confirmed after writing to at least a write quorum, Q_w, of replicas (while the system continues to try to propagate the write to the remaining replicas), and a read to be successful if at least a read quorum, Q_r, agree on the data or witness value. By varying Q_r and Q_w, one can configure such a system to bias availability in favor of either reads or writes in the face of multiple replica outages. In these terms, the enhanced availability scheme described above is one for which $Q_w = N_{replicas}$ and $Q_r = 1$.

Alternatively, one might run an $N_{replicas} = 5$ system with a rule that requires that all updates be made to at least $Q_w = 4$ of the replicas and that reads locate at least $Q_r = 2$

replicas that agree. This choice biases availability modestly in favor of reading: a successful write requires that at least 4 of the 5 replicas be available, while a read will succeed if only 2 of the replicas are available and agree, and agreement of 2 is ensured if any 3 are available. Or, one might set $Q_w = 2$ and $Q_r = 4$. That configuration would allow someone doing an update to receive confirmation that the update has been accomplished if any two replicas are available for update, but reading would then have to wait at least until the update gets propagated to two more replicas. With this configuration, write availability should be high but read availability might be quite low.

In practice, quorums can actually be quite a bit more complicated. The algorithm as described enhances durability and allows adjusting read versus write availability, but it does not provide either before-or-after or all-or-nothing atomicity, both of which are likely to be required to maintain strict consistency if there is either write concurrency or a significant risk of system crashes. Consider, for example, the system for which $N_{replicas} = 5$, $Q_w = 4$, and $Q_r = 2$. If an updater is at work and has successfully updated two of the replicas, one reader could read the two replicas already written by the updater while another reader might read two of the replicas that the updater hasn't gotten to yet. Both readers would believe they had found a consistent set of replicas, but the read/write coherence specification has not been met. Similarly, with the same system parameters, if an updater crashes after updating two of replicas, a second updater might come along and begin updating a different two of the replicas and then crash. That scenario would leave a muddled set of replicas in which one reader could read the replicas written by the first updater while another reader might read the replicas written by the second updater.

Thus a practical quorum scheme requires some additional before-or-after atomicity mechanism that serializes writes and ensures that no write begins until the previous write has sufficiently propagated to ensure coherence. The complexity of the mechanism depends on the exact system configuration. If all reading and updating originates at a single site, a simple sequencer at that site can provide the needed atomicity. If read requests can come from many different sources but all updates originate at a single site, the updating site can associate a version number with each data update and reading sites can check the version numbers to ensure that they have read the newest consistent set. If updates can originate from many sites, a protocol that provides a distributed sequencer implementation might be used for atomicity. Performance maximization usually is another complicating consideration. The interested reader should consult the professional literature, which describes many (sometimes quite complex) schemes for providing serialization of quorum replica systems. All of these mechanisms are specialized solutions to the generic problem of achieving atomicity across multiple sites, which was discussed at the end of Chapter 9[on-line].

10.3.6 Backup

Probably the most widely used replication technique for durable storage that is based on a single state machine is to periodically make backup copies of a complete file system on an independent, removable medium such as magnetic tape, writable video disk (DVD), or removable hard disk. Since the medium is removable, one can make the copy locally and introduce geographic separation later. If a disk fails and must be replaced, its contents can be restored from the most recent removable medium replica. Removable media are relatively cheap, so it is not necessary to recycle previous backup copies immediately. Older backup copies can serve an additional purpose, as protection against human error by acting as archives of the data at various earlier times, allowing retrieval of old data values.

The major downside of this technique is that it may take quite a bit of time to make a complete backup copy of a large storage system. For this reason, refinements such as *incremental backup* (copy only files changed since the last backup) and partial backup (don't copy files that can be easily reconstructed from other files) are often implemented. These techniques reduce the time spent making copies, but they introduce operational complexity, especially at restoration time.

A second problem is that if updates to the data are going on at the same time as backup copying, the backup copy may not be a snapshot at any single instant—it may show some results of a multi-file update but not others. If internal consistency is important, either updates must be deferred during backup or some other scheme, such as logging updates, must be devised. Since complexity also tends to reduce reliability, the designer must use caution when going in this direction.

It is worth repeating that the success of data replication depends on the independence of failures of the copies, and it can be difficult to assess correctly the amount of independence between replicas. To the extent that they are designed by the same designer and are modified by the same software, replicas may be subject to the same design or implementation faults. It is folk wisdom among system designers that the biggest hazard for a replicated system is replicated failures. For example, a programming error in a replicated state machine may cause all of the data replicas to become identically corrupted. In addition, there is more to achieving durable storage than just replication. Because a thread can fail at a time when some invariant on the data is not satisfied, additional techniques are needed to recover the data.

Complexity can also interfere with success of a backup system. Another piece of folk wisdom is that the more elaborate the backup system, the less likely that it actually works. Most experienced computer users can tell tales of the day that the disk crashed, and for some reason the backup copy did not include the most important files. (But the tale usually ends with a story that the owner of those files didn't trust the backup system, and was able to restore those important files from an *ad hoc* copy he or she made independently.)

10.3.7 Partitioning Data

A quite different approach to tolerating failures of storage media is to simply partition the data, thereby making the system somewhat fail-soft. In a typical design, one would divide a large collection of data into several parts, each of about the same size, and place each part on a different physical device. Failure of any one of the devices then compromises availability of only one part of the entire set of data. For some applications this approach can be useful, easy to arrange and manage, easy to explain to users, and inexpensive. Another reason that partition is appealing is that access to storage is often a bottleneck. Partition can allow concurrent access to different parts of the data, an important consideration in high-performance applications such as popular Web servers.

Replication can be combined with partition. Each partition of the data might itself be replicated, with the replicas placed on different storage devices, and each storage device can contain replicas of several of the different partitions. This strategy ensures continued availability if any single storage device fails, and at the same time an appropriate choice of configuration can preserve the performance-enhancing feature of partition.

10.4 Reconciliation

A typical constraint for replicas is that a majority of them be identical. Unfortunately, various events can cause them to become different: data of a replica can decay, a replicated state machine may experience an error, an update algorithm that has a goal of eventual consistency may be interrupted before it reaches its goal, an administrator of a replica site may modify a file in a way that fails to respect the replication protocol, or a user may want to make an update at a time when some replicas are disconnected from the network. In all of these cases, a need arises for an after-the-fact procedure to discover the differences in the data and to recover consistency. This procedure, called *reconciliation,* makes the replicas identical again.

Although reconciliation is a straightforward concept in principle, in practice three things conspire to make it more complicated than one might hope:

1. For large bodies of data, the most straightforward methods (e.g., compare all the bits) are expensive, so performance enhancements dominate, and complicate, the algorithms.
2. A system crash during a reconciliation can leave a body of data in worse shape than if no reconciliation had taken place. The reconciliation procedure itself must be resilient against failures and system crashes.
3. During reconciliation, one may discover *conflicts*, which are cases where different replicas have been modified in inconsistent ways. And in addition to files decaying, decay may also strike records kept by the reconciliation system itself.

One way to simplify thinking about reconciliation is to decompose it into two distinct modular components:

1. Detecting differences among the replicas.
2. Resolving the differences so that all the replicas become identical.

At the outset, every difference represents a potential conflict. Depending on how much the reconciliation algorithm knows about the semantics of the replicas, it may be able to algorithmically resolve many of the differences, leaving a smaller set of harder-to-handle conflicts. The remaining conflicts generally require more understanding of the semantics of the data, and ultimately may require a decision to be made on the part of a person. To illustrate this decomposition, the next section examines one widely-implemented reconciliation application, known as *occasionally connected* operation, in some detail.

10.4.1 Occasionally Connected Operation

A common application for reconciliation arises when a person has both a desktop computer and a laptop computer, and needs to work with the same files on both computers. The desktop computer is at home or in an office, while the laptop travels from place to place, and because the laptop is often not network-connected, changes made to a file on one of the two computers can not be automatically reflected in the replica of that file on the other. This scenario is called *occasionally connected* operation. Moreover, while the laptop is disconnected files may change on either the desktop or the laptop (for example, the desktop computer may pick up new incoming mail or do an automatic system update while the owner is traveling with the laptop and editing a report). We are thus dealing with a problem of concurrent update to multiple replicas.

Recall from the discussion on page 9–63 that there are both pessimistic and optimistic concurrency control methods. Either method can be applied to occasionally connected replicas:

- Pessimistic: Before disconnecting, identify all of the files that might be needed in work on the laptop computer and mark them as “checked out” on the desktop computer. The file system on the desktop computer then blocks any attempts to modify checked-out files. A pessimistic scheme makes sense if the traveler can predict exactly which files the laptop should check out and it is likely that someone will also attempt to modify them at the desktop.
- Optimistic: Allow either computer to update any file and, the next time that the laptop is connected, detect and resolve any conflicting updates. An optimistic scheme makes sense if the traveler cannot predict which files will be needed while traveling and there is little chance of conflict anyway.

Either way, when the two computers can again communicate, reconciliation of their replicas must take place. The same need for reconciliation applies to the handheld computers known as “personal digital assistants” which may have replicas of calendars, address books, to-do lists, or databases filled with business cards. The popular term for this kind of reconciliation is “file synchronization”. We avoid using that term because “synchronization” has too many other meanings.

The general outline of how to reconcile the replicas seems fairly simple: If a particular file changed on one computer but not on the other, the reconciliation procedure can resolve the difference by simply copying the newer file to the other computer. In the pessimistic case that is all there is to it. If the optimistic scheme is being used, the same file may have changed on both computers. If so, that difference is a conflict and reconciliation requires more guidance to figure out how the resolve it. For the file application, both the detection step and the resolution step can be fairly simple.

The most straightforward and accurate way to detect differences would be to read both copies of the file and compare their contents, bit by bit, with a record copy that was made at the time of the last reconciliation. If either file does not match the record copy, there is a difference; if both files fail to match the record copy, there is a conflict. But this approach would require maintaining a record copy of the entire file system as well as transmitting all of the data of at least one of the file systems to the place that holds the record copy. Thus there is an incentive to look for shortcuts.

One shortcut is to use a witness in place of the record copy. The reconciliation algorithm can then detect both differences and conflicts by calculating the current hash of a file and comparing it with a witness that was stored at the time of the previous reconciliation. Since a witness is likely to be much smaller than the original file, it does not take much space to store and it is easy to transmit across a network for comparison. The same set of stored witnesses can also support a decay detector that runs in a low-priority thread, continually reading files, recalculating their hash values, and comparing them with the stored witnesses to see if anything has changed.

Since witnesses require a lot of file reading and hash computation, a different shortcut is to just examine the time of last modification of every file on both computers, and compare that with the time of last reconciliation. If either file has a newer modification timestamp, there is a difference, and if both have newer modification timestamps, there is a conflict. This shortcut is popular because most file systems maintain modification timestamps as part of the metadata associated with a file. One requirement of this shortcut is that the timestamp have a resolution fine enough to ensure that every time a file is modified its timestamp increases. Unfortunately, modification timestamps are an approximation to witnesses that have several defects. First, the technique does not discover decay because decay events change file contents without updating modification times. Second, if someone modifies a file, then undoes the changes, perhaps because a transaction was aborted, the file will have a new timestamp and the reconciliation algorithm will consider the file changed, even though it really hasn't. Finally, the system clocks of disconnected computers may drift apart or users may reset system clocks to match their wristwatches (and some file systems allow the user to "adjust" the modification timestamp on a file), so algorithms based on comparing timestamps may come to wrong conclusions as to which of two file versions is "newer". The second defect affects performance rather than correctness, and the impact may be inconsequential, but the first and third defects can create serious correctness problems.

A file system can provide a different kind of shortcut by maintaining a systemwide sequence number, known as a *generation number*. At some point when the replicas are known to be identical, both file systems record as part of the metadata of every file a starting generation number, say zero, and they both set their current systemwide generation numbers to one. Then, whenever a user modifies a file, the file system records in the metadata of that file the current generation number. When the reconciliation program next runs, by examining the generation numbers on each file it can easily determine whether either or both copies of a file were modified since the last reconciliation: if either copy of the file has the current generation number, there is a difference; if both copies of the file have the current generation number, there is a conflict. When the reconciliation is complete and the two replicas are again identical, the file systems both increase their current generation numbers by one in preparation for the next reconciliation. Generation numbers share two of the defects of modification timestamps. First, they do not allow discovery of decay, since decay events change file contents without updating generation numbers. Second, an aborted transaction can leave one or more files with a new generation number even though the file contents haven't really changed. An additional problem that generation numbers do not share with modification timestamps is that implementation of generation numbers is likely to require modifying the file system.

The resolution step usually starts with algorithmic handling of as many detected differences as possible, leaving (one hopes) a short list of conflicts for the user to resolve manually.

10.4.2 A Reconciliation Procedure

To illustrate some of the issues involved in reconciliation, Figure 10.3 shows a file reconciliation procedure named RECONCILE, which uses timestamps. To simplify the example, files have path names, but there are no directories. The procedure reconciles two sets of files, named *left* and *right*, which were previously reconciled at *last_reconcile_time*, which acts as a kind of generation number. The procedure assumes that the two sets of files were identical at that time, and its goal is to make the two sets identical again, by examining the modification timestamps recorded by the storage systems that hold the files. The function MODIFICATION_TIME(*file*) returns the time of the last modification to *file*. The **copy** operation, in addition to copying a file from one set to another, also copies the time of last modification, if necessary creating a file with the appropriate file name.

RECONCILE operates as a transaction. To achieve all-or-nothing atomicity, RECONCILE is constructed to be idempotent; in addition, the **copy** operation must be atomic. To achieve before-or-after atomicity, RECONCILE must run by itself, without anyone else making more changes to files while its executes, so it begins by quiescing all file activity, perhaps by setting a lock that prevents new files from being opened by anyone other than itself, and then waiting until all files opened by other threads have been closed. For durability, reconcile depends on the underlying file system. Its constraint is that when it exits, the two sets *left* and *right* are identical.

RECONCILE prepares for reconciliation by reading from a dedicated disk sector the timestamp of the previous reconciliation and enumerating the names of the files on both sides. From the two enumerations, program lines *7* through *9* create three lists:

- names of files that appear on both sides (*common_list*),

```
procedure RECONCILE (reference left, reference right,
                                      reference last_reconcile_time)
    quiesce all activity on left and right    // Shut down all file-using applications
    ALL_OR_NOTHING_GET (last_reconcile_time, reconcile_time_sector)
    left_list ← enumerate(left)
    right_list ← enumerate(right)
    common_list ← intersect(left_list, right_list)
    left_only_list ← remove members of common_list from left_list
    right_only_list ← remove members of common_list from right_list
    conflict_list ← NIL

    for each named_file in common_list do   // Reconcile files found both sides
        left_new ← (MODIFICATION_TIME (left.named_file) > last_reconcile_time)
        right_new ← (MODIFICATION_TIME (right.named_file) > last_reconcile_time)
        if left_new and right_new then
            add named_file to conflict_list
        else if left_new then
            copy named_file from left to right
        else if right_new then
            copy named_file from right to left
        else if MODIFICATION_TIME (left.named_file) ≠
                    (MODIFICATION_TIME (right.named_file)
            then TERMINATE ("Something awful has happened.")

    for each named_file in left_only_list do   // Reconcile files found one side
        if MODIFICATION_TIME (left.named_file) > last_reconcile_time then
            copy named_file from left to right
        else
            delete left.named_file
    for each named_file in right_only_list do
        if MODIFICATION_TIME (right.named_file) > last_reconcile_time then
            copy named_file from right to left
        else
            delete right.named_file

    for each named_file in conflict_list do           // Handle conflicts
        MANUALLY_RESOLVE (right.named_file, left.named_file)
    last_reconcile_time ← NOW ()
    ALL_OR_NOTHING_PUT (last_reconcile_time, reconcile_time_sector)
    Allow activity to resume on left and right
```

FIGURE 10.3

A simple reconciliation algorithm.

- names of files that appear only on the left (*left_only_list*), and
- names of files that appear only on the right (*right_only_list*).

These three lists drive the rest of the reconciliation. Line 10 creates an empty list named *conflict_list*, which will accumulate names of any files that it cannot algorithmically reconcile.

Next, RECONCILE reviews every file in *common_list*. It starts, on lines 12 and 13, by checking timestamps to see whether either side has modified the file. If both sides have timestamps that are newer than the timestamp of the previous run of the reconciliation program, that indicates that both sides have modified the file, so it adds that file name to the list of conflicts. If only one side has a newer timestamp, it takes the modified version to be the authoritative one and copies it to the other side. (Thus, this program does some difference resolution at the same time that it is doing difference detection. Completely modularizing these two steps would require two passes through the lists of files, and thereby reduce performance.) If both file timestamps are older than the timestamp of the previous run, it checks to make sure that the timestamps on both sides are identical. If they are not, that suggests that the two file systems were different at the end of the previous reconciliation, perhaps because something went wrong during that attempt to reconcile, so the program terminates with an error message rather than blundering forward and taking a chance on irreparably messing up both file systems.

Having handled the list of names of files found on both sides, RECONCILE then considers those files whose names it found on only one side. This situation can arise in three ways:

1. one side deletes an old file,
2. the other side creates a new file, or
3. one side modifies a file that the other side deletes.

The first case is easily identified by noticing that the side that still has the file has not modified it since the previous run of the reconciliation program. For this case RECONCILE deletes the remaining copy. The other two cases cannot, without keeping additional state, be distinguished from one another, so RECONCILE simply copies the file from one side to the other. A consequence of this choice is that a deleted file will silently reappear if the other side modified it after the previous invocation of RECONCILE. An alternative implementation would be to declare a conflict, and ask the user to decide whether to delete or copy the file. With that choice, every newly created file requires manual intervention at the next run of RECONCILE. Both implementations create some user annoyance. Eliminating the annoyance is possible but requires an algorithm that remembers additional, per-file state between runs of RECONCILE.

Having reconciled all the differences that could be resolved algorithmically, RECONCILE asks the user to resolve any remaining conflicts by manual intervention. When the user finishes, RECONCILE is ready to commit the transaction, which it does by recording the current time in the dedicated disk sector, in line 36. It then allows file creation activity to resume, and it exits. The two sets of files are again identical.

10.4.3 Improvements

There are several improvements that we could make to this simple reconciliation algorithm to make it more user-friendly or comprehensive. As usual, each improvement adds complexity. Here are some examples:

1. Rather than demanding that the user resolve all remaining conflicts on the spot, it would be possible to simply notify the user that there is a non-empty conflict list and let the user resolve those conflicts at leisure. The main complication this improvement adds is that the user is likely to be modifying files (and changing file modification timestamps) at the same time that other file activity is going on, including activity that may be generating new inconsistencies among the replicas. Changes that the user makes to resolve the conflicts may thus look like new conflicts next time the reconciliation program runs. A second complication is that there is no assurance that the user actually reconciles the conflicts; the conflict list may still be non-empty the next time that the reconciliation program runs, and it must take that possibility into account. A simple response could be for the program to start by checking the previous conflict list to see if it is empty, and if it is not asking the user to take care of it before proceeding.

2. Some of the remaining conflicts may actually be algorithmically resolvable, with the help of an application program that understands the semantics and format of a particular file. Consider, for example, an appointment calendar application that stores the entire appointment book in a single file. If the user adds a 1 p.m. meeting to the desktop replica and a 4 p.m. meeting to the laptop replica, both files would have modification timestamps later than the previous reconciliation, so the reconciliation program would flag these files as a conflict. On the other hand, the calendar application program might be able to resolve the conflict by copying both meeting records to both files. What is needed is for the calendar application to perform the same kind of detection/resolution reconciliation we have already seen, but applied to individual appointment records rather than to the whole file. Any application that maintains suitable metadata (e.g. a record copy, witnesses, a generation number, or a timestamp showing when each entry in its database was last modified) can do such a record-by-record reconciliation. Of course, if the calendar application encounters two conflicting changes to the same appointment record, it probably would refer that conflict to the user for advice. The result of the application-specific reconciliation should be identical files on both replicas with identical modification timestamps.

   Application-specific reconciliation procedures have been designed for many different specialized databases such as address books, to-do lists, and mailboxes; all that is required is that the program designer develop an appropriate reconciliation algorithm. For convenience, it is helpful to integrate these application-specific procedures with the main reconciliation procedure. The usual method is for such applications to register their reconciliation procedures, along with a list of files or file types that each reconciliation procedure can handle, with the main reconciliation program. The main reconciliation program then adds a step of reviewing its conflict list to see if there is an application-specific program available for each file. If there is, it invokes that program, rather than asking the user to resolve the conflict.

3. As it stands, the reconciliation procedure enumerates only files. If it were to be applied to a file system that has directories, links, and file metadata other than file names and modification times, it might do some unexpected things. For example, the program would handle links badly, by creating a second copy of the linked file, rather than creating a link. Most reconciliation programs have substantial chunks of code devoted to detecting and resolving differences in directories and metadata. Because the semantics of the directory management operations are usually known to the writer of the reconciliation program, many differences between directories can be resolved algorithmically. However, there can still be a residue of conflicts that require user guidance to resolve, such as when a file named A has been created in a directory on one side and a different file named A has been created in the same directory on the other side.

10.4.4 Clock Coordination

This RECONCILE program is relatively fragile. It depends, for example, on the timestamps being accurate. If the two sets of files are managed by different computer systems with independent clocks, and someone sets the clock incorrectly on one side, the timestamps on that side will also be incorrect, with the result that RECONCILE may not notice a conflict, it may overwrite a new version of a file with an old version, it may delete a file that should not be deleted, or it may incorrectly revive a deleted file. For the same reason, RECONCILE must carefully preserve the variable *last_reconcile_time* from one run to the next.

Some reconciliation programs try to minimize the possibility of accidental damage by reading the current clock value from both systems, noting the difference, and taking that difference into account. If the difference has not changed since the previous reconciliation, reconcile can simply add (or subtract, as appropriate) the time difference and proceed as usual. If the difference has changed, the amount of the change can be considered a delta of uncertainty; any file whose fate depends on that uncertainty is added to the list of conflicts for the user to resolve manually.

10.5 Perspectives

In [on-line] Chapters 9 and 10 we have gone into considerable depth on various aspects of atomicity and systematic approaches to providing it. At this point it is appropriate to stand back from the technical details and try to develop some perspective on how all these ideas relate to the real world. The observations of this section are wide-ranging: history, trade-offs, and unexplored topics. Individually these observations appear somewhat disconnected, but in concert they may provide the reader with some preparation for the way that atomicity fits into the practical world of computer system design.

10.5.1 History

Systematic application of atomicity to recovery and to coordination is relatively recent. Ad hoc programming of concurrent activities has been common since the late 1950s, when machines such as the IBM 7030 (STRETCH) computer and the experimental TX–0 at M.I.T. used interrupts to keep I/O device driver programs running concurrently with the main computation. The first time-sharing systems (in the early 1960s) demonstrated the need to be more systematic in interrupt management, and many different semantic constructs were developed over the next decade to get a better grasp on coordination problems: Edsger Dijkstra's semaphores, Per Brinch Hansen's message buffers, David Reed and Raj Kanodia's eventcounts, Nico Habermann's path expressions, and Anthony Hoare's monitors are examples. A substantial literature grew up around these constructs, but a characteristic of all of them was a focus on properly coordinating concurrent activities, each of which by itself was assumed to operate correctly. The possibility of failure and recovery of individual activities, and the consequences of such failure and recovery on coordination with other, concurrent activities, was not a focus of attention. Another characteristic of these constructs is that they resemble a machine language, providing low-level tools but little guidance in how to apply them.

Failure recovery was not simply ignored in those early systems, but it was handled quite independently of coordination, again using ad hoc techniques. The early time-sharing system implementers found that users required a kind of durable storage, in which files could be expected to survive intact in the face of system failures. To this end most time-sharing systems periodically made backup copies of on-line files, using magnetic tape as the backup medium. The more sophisticated systems developed incremental backup schemes, in which recently created or modified files were copied to tape on an hourly basis, producing an almost-up-to-date durability log. To reduce the possibility that a system crash might damage the on-line disk storage contents, salvager programs were developed to go through the disk contents and repair obvious and common kinds of damage. The user of a modern personal computer will recognize that some of these techniques are still in widespread use.

These *ad hoc* techniques, though adequate for some uses, were not enough for designers of serious database management systems. To meet their requirements, they developed the concept of a transaction, which initially was exactly an all-or-nothing action applied to a database. Recovery logging protocols thus developed in the database environment, and it was some time before it was recognized that recovery semantics had wider applicability.

Within the database world, coordination was accomplished almost entirely by locking techniques that became more and more systematic and automatic, with the realization that the definition of correctness for concurrent atomic actions involved getting the same result as if those atomic actions had actually run one at a time in some serial order. The database world also contributed the concept of maintaining constraints or invariants among different data objects, and the word *transaction* came to mean an action that is both all-or-nothing and before-or-after and that can be used to maintain constraints and provide durability. The database world also developed systematic replication schemes, primarily to enhance reliability and availability, but also to enhance performance.

The understanding of before-or-after atomicity, along with a requirement for hierarchical composition of programs, in turn led to the development of version history (also called *temporal database* or *time domain addressing*) systems. Version histories systematically provide both recovery and coordination with a single mechanism, and they simplify building big atomic actions out of several, independently developed, smaller ones.

This text has reversed this order of development because the relatively simple version history is pedagogically more straightforward, while the higher complexity of the logging/locking approach is easier to grasp after seeing why version histories work. Version histories are used in source code management systems and also in user interfaces that provide an UNDO button, but virtually all commercial database management systems use logs and locking in order to attain maximum performance.

10.5.2 Trade-Offs

An interesting set of trade-offs applies to techniques for coordinating concurrent activities. Figure 10.4 suggests that there is a spectrum of coordination possibilities, ranging from totally serialized actions on the left to complete absence of coordination on the right. Starting at the left, we can have great simplicity (for example by scheduling just one thread at a time) but admit no concurrency at all. Moving toward the right, the complexity required to maintain correctness increases but so does the possibility of improved performance, since more and more concurrency is admitted. For example, the mark-point and simple locking disciplines might lie more toward the left end of this spectrum while two-phase locking would be farther to the right. The solid curved line in the figure represents a boundary of increasing minimum complexity, below which that level of coordination complexity can no longer ensure correctness; outcomes that do not correspond to any serial schedule of the same actions become possible. (For purposes of illustration, the figure shows the boundary line as a smooth increasing curve, but that is a gross oversimplification. At the first hint of concurrency, the complexity leaps upward.)

Continuing to traverse the concurrency spectrum to the right, one passes a point, indicated by the dashed vertical line, beyond which correctness cannot be achieved no matter how clever or complex the coordination scheme. The closer one approaches this limit from the left, the higher the performance, but at the cost of higher complexity. All of the algorithms explored in [on-line] Chapters 9 and 10 are intended to operate to the left of the correctness limit, but we might inquire about the possibilities of working on the other side. Such a possibility is not as unthinkable as it might seem at first. If interference between concurrent activities is rare, and the cost of an error is small, one might actually be willing to permit concurrent actions that can lead to certifiably wrong answers. Section 9.5.4[on-line] suggested that designers sometimes employ locking protocols that operate in this region.

For example, in an inventory control system for a grocery store, if an occasional sale of a box of cornflakes goes unrecorded because two point-of-sale terminals tried to update the cornflakes inventory concurrently, the resulting slight overstatement of inventory may not be a serious problem. The grocery store must do occasional manual inventory anyway because other boxes of cornflakes are misplaced, damaged, and stolen, and employees sometimes enter wrong numbers when new boxes are delivered. This higher-layer data recovery mechanism will also correct any errors that creep in because of miscoordination in the inventory management system, so its designer might well decide to use a coordination technique that allows maximum concurrency, is simple, catches the most common miscoordination problems, but nevertheless operates below or to the right of the strict correctness line. A decision to operate a data management system

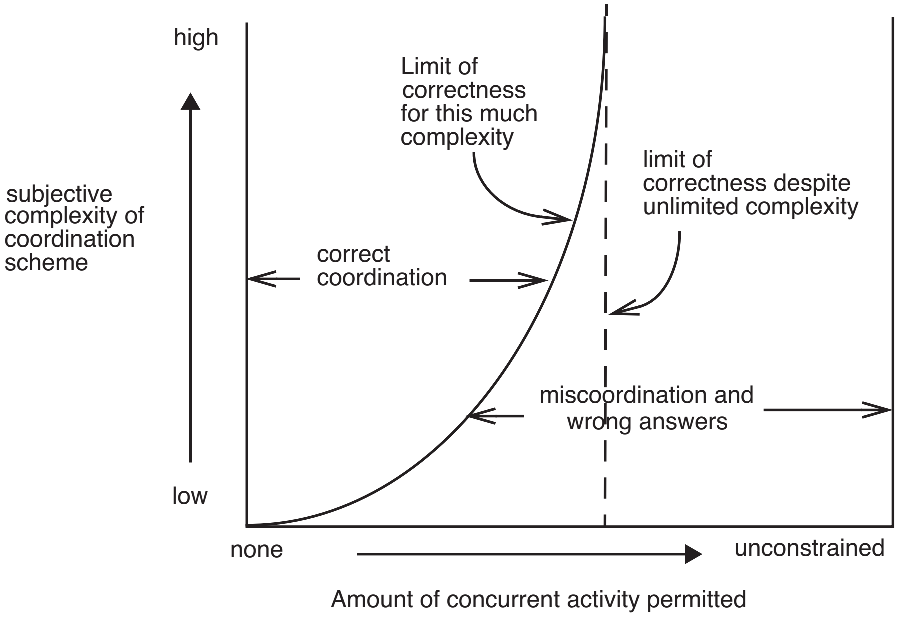

FIGURE 10.4

The trade-off among concurrency, complexity, and correctness. The choice of where in this chart to position a system design depends on the answers to two questions: 1) How frequently will concurrent activities actually interfere with one another? 2) How important are 100% correct results? If interference is rare, it is appropriate to design farther to the right. If correctness is not essential, it may be acceptable to design even to the right of the two correctness boundaries.

in a mode that allows such errors can be made on a rational basis. One would compare the rate at which the system loses track of inventory because of its own coordination errors with the rate at which it loses track because of outside, uncontrolled events. If the latter rate dominates, it is not necessary to press the computer system for better accuracy.

Another plausible example of acceptable operation outside the correctness boundary is the calculation, by the Federal Reserve Bank, of the United States money supply. Although in principle one could program a two-phase locking protocol that includes every bank account in every bank that contains U.S. funds, the practical difficulty of accomplishing that task with thousands of independent banks distributed over a continent is formidable. Instead, the data is gathered without locking, with only loose coordination and it is almost certain that some funds are counted twice and other funds are overlooked. However, great precision is not essential in the result, so lack of perfect coordination among the many individual bank systems operating concurrently is acceptable.

Although allowing incorrect coordination might appear usable only in obscure cases, it is actually applicable to a wider range of situations than one might guess. In almost all database management applications, the biggest cause of incorrect results is wrong input by human operators. Typically, stored data already has many defects before the transaction programs of the database management system have a chance to "correctly" transform it. Thus the proper perspective is that operation outside of the correctness boundaries of Figure 10.4 merely adds to the rate of incorrectness of the database. We are making an *end-to-end argument* here: there may be little point in implementing heroic coordination correctness measures in a lower layer if the higher-layer user of our application makes other mistakes, and has procedures in place to correct them anyway.

With that perspective, one can in principle balance heavy-handed but "correct" transaction coordination schemes against simpler techniques that can occasionally damage the data in limited ways. One piece of evidence that this approach is workable in practice is that many existing data management systems offer optional locking protocols called "cursor stability", "read committed", or "snapshot isolation", all of which are demonstrably incorrect in certain cases. However, the frequency of interacting update actions that actually produce wrong answers is low enough and the benefit in increased concurrency is high enough that users find the trade-off tolerable. The main problem with this approach is that no one has yet found a good way of characterizing (with the goal of limiting) the errors that can result. If you can't bound the maximum damage that could occur, then these techniques may be too risky.

An obvious question is whether or not some similar strategy of operating beyond a correctness boundary applies to atomicity. Apparently not, at least in the area of instruction set design for central processors. Three generations of central processor designers (of the main frame processors of the 1950's and 1960's, the mini-computers of the 1970's, and the one-chip microprocessors of the 1980's) did not recognize the importance of all-or-nothing atomicity in their initial design and were later forced to retrofit it into their architectures in order to accommodate the thread switching that accompanies multilevel memory management.

10.5.3 Directions for Further Study

Chapters 9 and 10 have opened up only the first layer of study of atomicity, transactions, durability, replication, and consistency; there are thick books that explore the details. Among the things we have touched only lightly (or not at all) are distributed atomic actions, hierarchically composing programs with modules that use locks, the systematic use of loose or incorrect coordination, the systematic application of compensation, and the possibility of malicious participants.

Implementing distributed atomic actions efficiently is a difficult problem for which there is a huge literature, with some schemes based on locking, others on timestamp-based protocols or version histories, some on combining the two, and yet others with optimistic strategies. Each such scheme has a set of advantages and disadvantages with respect to performance, availability, durability, integrity, and consistency. No one scheme seems ready to dominate and new schemes appear regularly.

Hierarchical composition—making larger atomic actions out of previously programmed smaller ones—interacts in an awkward way with locking as a before-or-after atomicity technique. The problem arises because locking protocols require a lock point for correctness. Creating an atomic action from two previously independent atomic actions is difficult because each separate atomic action has its own lock point, coinciding with its own commit point. But the higher-layer action must also have a lock point, suggesting that the order of capture and release of locks in the constituent atomic action needs to be changed. Rearrangement of the order of lock capture and release contradicts the usual goal of modular composition, under which one assembles larger systems out of components without having to modify the components. To maintain modular composition, the lock manager needs to know that it is operating in an environment of hierarchical atomic actions. With this knowledge, it can, behind the scenes, systematically rearrange the order of lock release to match the requirements of the action nesting. For example, when a nested atomic action calls to release a lock, the lock manager can simply relabel that lock to show that it is held by the higher layer, not-yet-committed, atomic action in which this one is nested. A systematic discipline of passing locks up and down among nested atomic actions thus can preserve the goal of modular composition, but at a cost in complexity.

Returning to the idea suggested by Figure 10.4, the possibility of designing a system that operates in the region of incorrectness is intriguing, but there is one major deterrent: one would like to specify, and thus limit, the nature of the errors that can be caused by miscoordination. This specification might be on the magnitude of errors, or their direction, or their cumulative effect, or something else. Systematic specification of tolerance of coordination errors is a topic that has not been seriously explored.

Compensation is the way that one deals with miscoordination or with recovery in situations where rolling back an action invisibly cannot be accomplished. Compensation is performing a visible action that reverses all known effects of some earlier, visible action. For example, if a bank account was incorrectly debited, one might later credit it for the missing amount. The usefulness of compensation is limited by the extent to which one can track down and reverse everything that has transpired since the action that needs reversal. In the case of the bank account, one might successfully discover that an interest payment on an incorrect balance should also be adjusted; it might be harder to reverse all the effects of a check that was bounced because the account balance was incorrectly too low. Apart from generalizations along the line of "one must track the flow of information output of any action that is to be reversed" little is known about systematic compensation; it seems to be an application-dependent concept. If committing the transaction resulted in drilling a hole or firing a missile, compensation may not be an applicable concept.

Finally, all of the before-or-after atomicity schemes we explored assume that the various participants are all trying to reach a consistent, correct result. Another area of study explores what happens if one or more of the workers in a multiple-site coordination task decides to mislead the others, for example by sending a message to one site reporting it has committed, while sending a message to another site reporting it has aborted. (This possibility is described colorfully as the *Byzantine Generals'* problem.) The possibility of adversarial participants merges concerns of security with those of atomicity. The solutions so far are based primarily on extension of the coordination and recovery protocols to allow achieving consensus in the face of adversarial behavior. There has been little overlap with the security mechanisms that will be studied in Chapter 11[on-line].

One reason for exploring this area of overlap between atomicity and security is the concern that undetected errors in communication links could simulate uncooperative behavior. A second reason is increasing interest in peer-to-peer network communication, which frequently involves large numbers of administratively independent participants who may, either accidentally or intentionally, engage in Byzantine behavior. Another possible source of Byzantine behavior could lie in outsourcing of responsibility for replica storage.

Exercises

10.1 You are developing a storage system for a application that demands unusually high reliability, so you have decided to use a three-replica durable storage scheme. You plan to use three ordinary disk drives D1, D2, and D3, and arrange that D2 and D3 store identical mirror copies of each block stored on D1. The disk drives are of a simple design that does not report read errors. That is, they just return data, whether or not it is valid.

10.1a. You initially construct the application so that it writes a block of data to the same sector on all three drives concurrently. After a power failure occurs during the middle of a write, you are unable to reconstruct the correct data for that sector. What is the problem?

10.1b. Describe a modification that solves this problem.

10.1c. One day there is a really awful power glitch that crashes all three disks in such a way that each disk corrupts one random track. Fortunately, the system wasn't writing any data at the time. Describe a procedure for reconstructing the data and explain any cases that your procedure cannot handle.

1994–3–2

10.2 What assumptions does the design of the RECONCILE procedure of Section 10.4.2 make with respect to concurrent updates to different replicas of the same file?

A. It assumes that these conflicts seldom happen.
B. It assumes that these conflicts can be automatically detected.
C. It assumes that all conflicts can be automatically resolved later.
D. It assumes that these conflicts cannot happen.

1999–3–04

10.3 Mary uses RECONCILE to keep the files in her laptop computer coordinated with her desktop computer. However, she is getting annoyed. While she is traveling, she works on her e-mail inbox, reading and deleting messages, and preparing replies, which go into an e-mail outbox. When she gets home, RECONCILE always tells her that there is a conflict with the inbox and outbox on her desktop because while she was gone the system added several new messages to the desktop inbox, and it dispatched and deleted any messages that were in the desktop outbox. Her mailer implements the inbox as a single file and the outbox as a single file. Ben suggests that Mary switch to a different mailer, one that implements the inbox and outbox as two directories, and places each incoming or outgoing message in a separate file. Assuming that no one but the mail system touches Mary's desktop mailboxes in her absence, which of the following is the most accurate description of the result?

A. RECONCILE will still not be able to reconcile either the inbox or the outbox.
B. RECONCILE will be able to reconcile the inbox but not the outbox.
C. RECONCILE will be able to reconcile the outbox but not the inbox.
D. RECONCILE will be able to reconcile both the inbox and the outbox.

1997–0–03

10.4 Which of the following statements are true of the RECONCILE program of Figure 10.3?

A. If RECONCILE finds that the content of one copy of a file differs from the other copy of the same file, it indicates a conflict.

B. You create a file X with content "a" in file set 1, then create a file X with content "b" in file set 2. You then delete X from host 1, and run RECONCILE to synchronize the two file sets. After RECONCILE finishes you'll see a file X with content "b" in file set 1.

C. If you accidentally reset RECONCILE's variable named *last_reconcile_time* to midnight, January 1, 1900, you are likely to need to resolve many more conflicts when you next run RECONCILE than if you had preserved that variable.

2008–3–2

10.5 Here is a proposed invariant for a file reconciler such as the program RECONCILE of Figure 10.3: At every moment during a run of RECONCILE, every file has either its original contents, or its correct final contents. Which of the following statements is true about RECONCILE?

A. RECONCILE does not attempt to maintain this invariant.

B. RECONCILE maintains this invariant in all cases.

C. RECONCILE uses file creation time to determine the most recent version of a file.

D. If the two file sets are on different computers connected by a network, RECONCILE would have to send the content of one version of each file over the network to the other computer for comparison.

Additional exercises relating to Chapter 10 can be found in problem sets *40* through *42*.

Glossary for Chapter 10

backup copy—Of a set of replicas that is not written or updated synchronously, one that is written later. Compare with *primary copy* and *mirror*. [Ch. 10]

cache coherence—Read/write coherence for a multilevel memory system that has a cache. It is a specification that the cache provide strict consistency at its interface. [Ch. 10]

close-to-open consistency—A consistency model for file operations. When a thread opens a file and performs several write operations, all of the modifications weill be visible to concurrent threads only after the first thread closes the file. [Ch. 4]

coheerence—See *read/write coherence* or *cache coherence*.

consensus—Agreement at separated sites on a data value despite communication failures. [Ch. 10]

consistency—A particular constraint on the memory model of a storage system that allows concurrency and uses replicas: that all readers see the same result. Also used in some professional literature as a synonym for *coherence*. [Ch. 10]

constraint—An application-defined invariant on a set of data values or externally visible actions. Example: a requirement that the balances of all the accounts of a bank sum to zero, or a requirement that a majority of the copies of a set of data be identical. [Ch. 10]

eventual consistency—A requirement that at some unspecified time following an update to a collection of data, if there are no more updates, the memory model for that collection will hold. [Ch. 10]

fingerprint—Another term for a *witness*. [Ch. 10]

incremental backup—A backup copy that contains only data that has changed since making the previous backup copy. [Ch. 10]

invalidate—In a cache, to mark "do not use" or completely remove a cache entry because some event has occurred that may make the value associated with that entry incorrect. [Ch. 10]

logical copy—A replica that is organized in a form determined by a higher layer. An example is a replica of a file system that is made by copying one file at a time. Analogous to logical locking. Compare with physical copy. [Ch. 10]

master—In a multiple-site replication scheme, the site to which updates are directed. Compare with *slave*. [Ch. 10]

page fault—See *missing-page exception*.

pair-and-spare—See *pair-and-compare*.

physical copy—A replica that is organized in a form determined by a lower layer. An example is a replica of a disk that is made by copying it sector by sector. Analogous to *physical locking*. Compare with *logical copy*. [Ch. 10]

prepaging—An optimization for a multilevel memory manager in which the manager predicts which pages might be needed and brings them into the primary memory before the application demands them. Compare with *demand algorithm*.

presented load—See *offered load*.

primary copy—Of a set of replicas that are not written or updated synchronously, the one that is considered authoritative and, usually, written or updated first. (Compare with *mirror* and *backup copy*.) [Ch. 10]

quorum—A partial set of replicas intended to improve availability. One defines a read quorum and a write quorum that intersect, with the goal that for correctness it is sufficient to read from a read quorum and write to a write quorum. [Ch. 10]

reconciliation—A procedure that compares replicas that are intended to be identical and repairs any differences. [Ch. 10]

replicated state machine—A method of performing an update to a set of replicas that involves sending the update request to each replica and performing it independently at each replica. [Ch. 10]

slave—In a multiple-site replication scheme, a site that takes update requests from only the master site. Compare with *master*. [Ch. 10]

snoopy cache—In a multiprocessor system with a bus and a cache in each processor, a cache design in which the cache actively monitors traffic on the bus to watch for events that invalidate cache entries. [Ch. 10]

strict consistency—An interface requirement that temporary violation of a data invariant during an update never be visible outside of the action doing the update. One feature of the read/write coherence memory model is strict consistency. Sometimes called *strong consistency*. [Ch. 10]

witness—A (usually cryptographically strong) hash value that attests to the content of a file. Another widely used term for this concept is **fingerprint**. [Ch. 10]

Index of Chapter 10

Design principles and hints appear in *underlined italics*. Procedure names appear in SMALL CAPS. Page numbers in **bold face** are in the chapter Glossary.