Type of user	Description
Owner	The owner of a file
Group	The group of users associated with a file
Everyone	Any user or process
Interactive	Any process accessing the file from an interactive terminal
Network	Any process accessing the file via the network
Dialup	Any process accessing the file through a dialup connection to the server
Batch	Any process accessing the file as part of batch job
Anonymous	Anyone accessing the file without authentication
Authenticated	Any authenticated user or process
Service	Any system-defined service process

Figure 10-20. The various kinds of users and processes distinguished by NFS with respect to access control.

10.2 THE CODA FILE SYSTEM

Our next example of a distributed file system is Coda. **Coda** has been developed at Carnegie Mellon University (CMU) in the 1990s, and is now integrated with a number of popular UNIX-based operating systems such as Linux. Coda is in many ways different from NFS, notably with respect to its goal for high availability. This goal has led to advanced caching schemes that allow a client to continue operation despite being disconnected from a server. Overviews of Coda are described in (Satyanarayanan et al., 1990; Kistler and Satyanarayanan, 1992). A detailed description of the system can be found in (Kistler, 1996).

10.2.1 Overview of Coda

Coda was designed to be a scalable, secure, and highly available distributed file system. An important goal was to achieve a high degree of naming and location transparency so that the system would appear to its users very similar to a pure local file system. By also taking high availability into account, the designers of Coda have also tried to reach a high degree of failure transparency.

Coda is a descendant of version 2 of the **Andrew File System** (**AFS**), which was also developed at CMU (Howard et al., 1988; Satyanarayanan, 1990), and inherits many of its architectural features from AFS. AFS was designed to support the entire CMU community, which implied that approximately 10,000 workstations would need to have access to the system. To meet this requirement, AFS nodes are partitioned into two groups. One group consists of a relatively small number of dedicated **Vice** file servers, which are centrally administered. The other group consists of a very much larger collection of **Virtue** workstations that give users and processes access to the file system, as shown in Fig. 10-21.

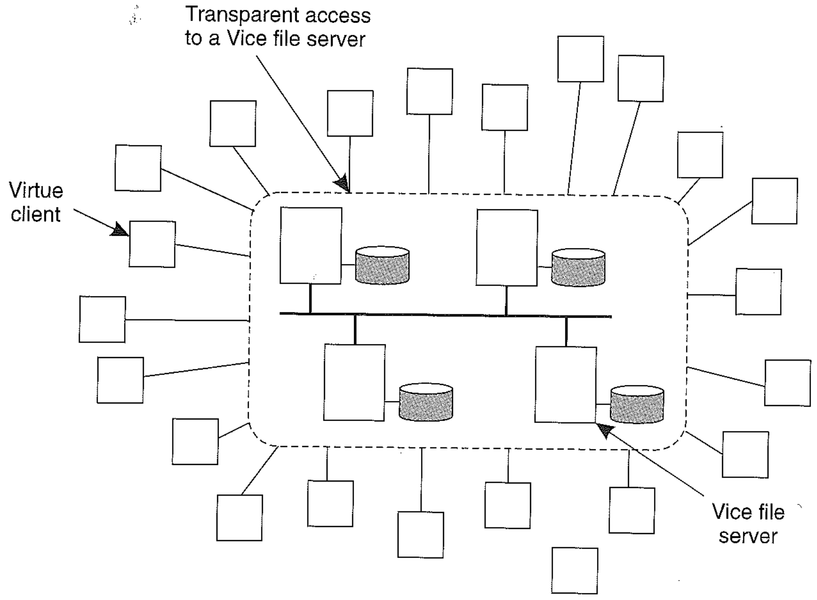

Figure 10-21. The overall organization of AFS.

Coda follows the same organization as AFS. Every Virtue workstation hosts a user-level process called **Venus**, whose role is similar to that of an NFS client. A Venus process is responsible for providing access to the files that are maintained by the Vice file servers. In Coda, Venus is also responsible for allowing the client to continue operation even if access to the file servers is (temporarily) impossible. This additional role is a major difference with the approach followed in NFS.

The internal architecture of a Virtue workstation is shown in Fig. 10-22. The important issue is that Venus runs as a user-level process. Again, there is a separate Virtual File System (VFS) layer that intercepts all calls from client applications, and forwards these calls either to the local file system or to Venus, as shown in Fig. 10-22. This organization with VFS is the same as in NFS. Venus, in turn, communicates with Vice file servers using a user-level RPC system. The RPC system is constructed on top of UDP datagrams and provides at-most-once semantics.

There are three different server-side processes. The great majority of the work is done by the actual Vice file servers, which are responsible for maintaining a local collection of files. Like Venus, a file server runs as a user-level process. In addition, trusted Vice machines are allowed to run an authentication server, which we discuss in detail later. Finally, update processes are used to keep meta-information on the file system consistent at each Vice server.

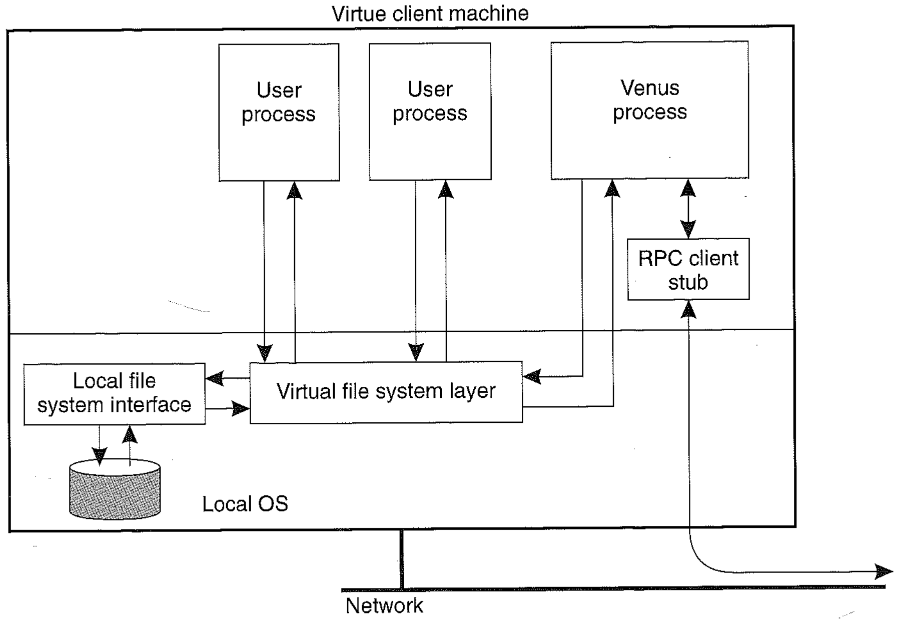

Figure 10-22. The internal organization of a Virtue workstation.

Coda appears to its users as a traditional UNIX-based file system. It supports most of the operations that form part of the VFS specification (Kleiman, 1986), which are similar to those listed in Fig. 10-3 and will therefore not be repeated here. Unlike NFS, Coda provides a globally shared name space that is maintained by the Vice servers. Clients have access to this name space by means of a special subdirectory in their local name space, such as */afs*. Whenever a client looks up a name in this subdirectory, Venus ensures that the appropriate part of the shared name space is mounted locally. We return to the details below.

10.2.2 Communication

Interprocess communication in Coda is performed using RPCs. However, the **RPC2** system for Coda is much more sophisticated than traditional RPC systems such as ONC RPC, which is used by NFS.

RPC2 offers reliable RPCs on top of the (unreliable) UDP protocol. Each time a remote procedure is called, the RPC2 client code starts a new thread that sends an invocation request to the server and subsequently blocks until it receives an answer. As request processing may take an arbitrary time to complete, the server regularly sends back messages to the client to let it know it is still working on the request. If the server dies, sooner or later this thread will notice that the messages have ceased and report back failure to the calling application.

An interesting aspect of RPC2 is its support for side effects. A **side effect** is a mechanism by which the client and server can communicate using an application-specific protocol. Consider, for example, a client opening a file at a video server. What is needed in this case is that the client and server set up a continuous data stream with an isochronous transmission mode. In other words, data transfer from the server to the client is guaranteed to be within a minimum and maximum end-to-end delay, as explained in Chap. 2.

RPC2 allows the client and the server to set up a separate connection for transferring the video data to the client on time. Connection setup is done as a side effect of an RPC call to the server. For this purpose, the RPC2 runtime system provides an interface of side-effect routines that is to be implemented by the application developer. For example, there are routines for setting up a connection and routines for transferring data. These routines are automatically called by the RPC2 runtime system at the client and server, respectively, but their implementation is otherwise completely independent of RPC2. This principle of side effects is shown in Fig. 10-23.

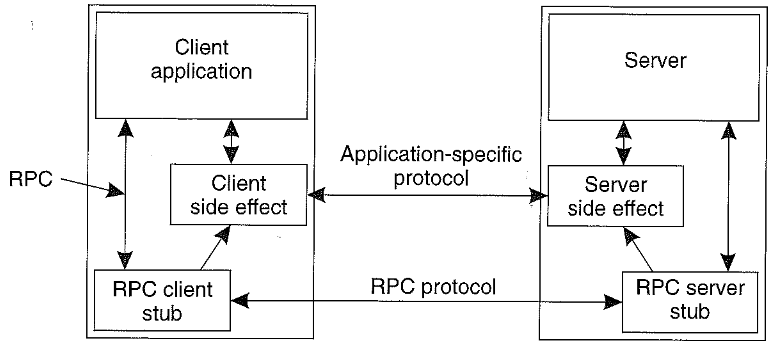

Figure 10-23. Side effects in Coda's RPC2 system.

Another feature of RPC2 that makes it different from other RPC systems, is its support for multicasting. As we explain in detail below, an important design issue in Coda is that servers keep track of which clients have a local copy of a file. When a file is modified, a server invalidates local copies by notifying the appropriate clients through an RPC. Clearly, if a server can notify only one client at a time, invalidating all clients may take some time, as illustrated in Fig. 10-24(a).

The problem is caused by the fact that an RPC may fail. Invalidating files in a strict sequential order may be delayed considerably because the server cannot reach a possibly crashed client, but will give up on that client only after a relatively long expiration time. Meanwhile, other clients will still be reading from their local copies.

A better solution is shown in Fig. 10-24(b). Instead of invalidating each copy one-by-one, the server sends an invalidation message to all clients in parallel. As

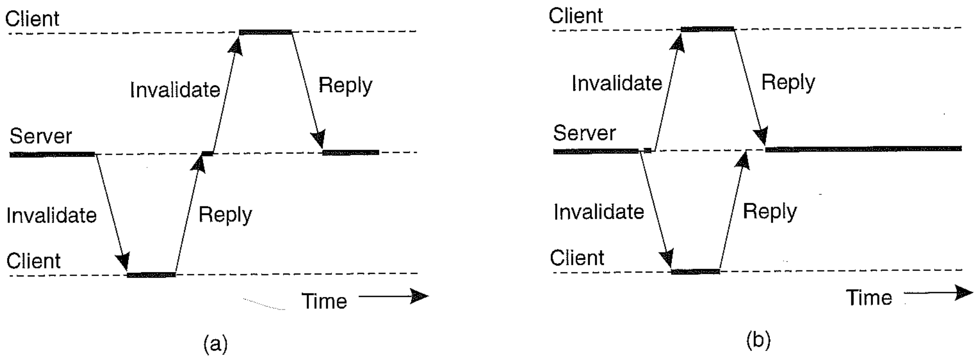

Figure 10-24. (a) Sending an invalidation message one at a time. (b) Sending invalidation messages in parallel.

a consequence, all nonfailing clients are notified in the same time as it would take to do an immediate RPC. Also, the server notices within the usual expiration time that certain clients are failing to respond to the RPC, and can declare such clients as being crashed.

Parallel RPCs are implemented by means of the **MultiRPC** system (Satyanarayanan and Siegel, 1990), which is part of the RPC2 package. An important aspect of MultiRPC is that the parallel invocation of RPCs is fully transparent to the callee. In other words, the receiver of a MultiRPC call cannot distinguish that call from a normal RPC. At the caller's side, parallel execution is also largely transparent. For example, the semantics of MultiRPC in the presence of failures are much the same as that of a normal RPC. Likewise, the side-effect mechanisms can be used in the same way as before.

MultiRPC is implemented by essentially executing multiple RPCs in parallel. This means that the caller explicitly sends an RPC request to each recipient. However, instead of immediately waiting for a response, it defers blocking until all requests have been sent. In other words, the caller invokes a number of one-way RPCs, after which it blocks until all responses have been received from the nonfailing recipients. An alternative approach to parallel execution of RPCs in MultiRPC is provided by setting up a multicast group, and sending an RPC to all group members using IP multicast.

10.2.3 Processes

Coda maintains a clear distinction between client and server processes. Clients are represented by Venus processes; servers appear as Vice processes. Both type of processes are internally organized as a collection of concurrent threads. Threads in Coda are nonpreemptive and operate entirely in user space. To account for continuous operation in the face of blocking I/O requests, a separate thread is used to handle all I/O operations, which it implements using low-level asynchronous I/O operations of the underlying operating system. This thread effectively emulates synchronous I/O without blocking an entire process.

10.2.4 Naming

As we mentioned, Coda maintains a naming system analogous to that of UNIX. Files are grouped into units referred to as **volumes**. A volume is similar to a UNIX disk partition (i.e., an actual file system), but generally has a much smaller granularity. It corresponds to a partial subtree in the shared name space as maintained by the Vice servers. Usually a volume corresponds to a collection of files associated with a user. Examples of volumes include collections of shared binary or source files, and so on. Like disk partitions, volumes can be mounted.

Volumes are important for two reasons. First, they form the basic unit by which the entire name space is constructed. This construction takes place by mounting volumes at mount points. A mount point in Coda is a leaf node of a volume that refers to the root node of another volume. Using the terminology introduced in Chap. 4, only root nodes can act as *mounting* points (i.e., a client can mount only the root of a volume). The second reason why volumes are important, is that they form the unit for server-side replication. We return to this aspect of volumes below.

Considering the granularity of volumes, it can be expected that a name lookup will cross several mount points. In other words, a path name will often contain several mount points. To support a high degree of naming transparency, a Vice file server returns mounting information to a Venus process during name lookup. This information will allow Venus to automatically mount a volume into the client's name space when necessary. This mechanism is similar to crossing mount points as supported in NFS version 4.

It is important to note that when a volume from the shared name space is mounted in the client's name space, Venus follows the structure of the shared name space. To explain, assume that each client has access to the shared name space by means of a subdirectory called */afs*. When mounting a volume, each Venus process ensures that the naming graph rooted at */afs* is always a subgraph of the complete name space jointly maintained by the Vice servers, as shown in Fig. 10-25. In doing so, clients are guaranteed that shared files indeed have the same name, although name resolution is based on a locally-implemented name space. Note that this approach is fundamentally different from that of NFS.

File Identifiers

Considering that the collection of shared files may be replicated and distributed across multiple Vice servers, it becomes important to uniquely identify each file in such a way that it can be tracked to its physical location, while at the same time maintaining replication and location transparency.

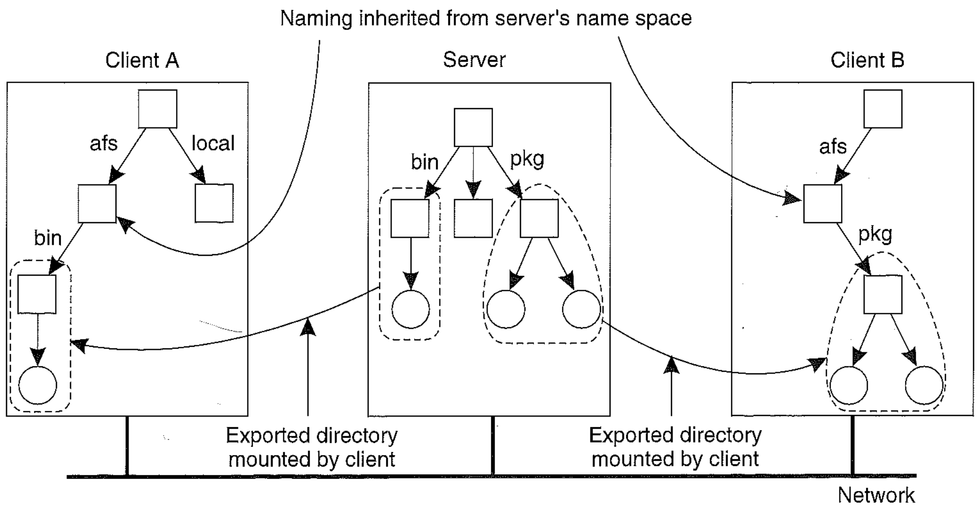
Naming inherited from server's name space
Client A
Server
Client B
afs
local
bin
bin
pkg
afs
pkg
Exported directory mounted by client
Exported directory mounted by client
Network

Figure 10-25. Clients in Coda have access to a single shared name space.

Each file in Coda is contained in exactly one volume. As we mentioned above, a volume may be replicated across several servers. For this reason, Coda makes a distinction between logical and physical volumes. A logical volume represents a possibly replicated physical volume, and has an associated **Replicated Volume Identifier (RVID)**. An RVID is a location and replication-independent volume identifier. Multiple replicas may be associated with the same RVID. Each physical volume has its own **Volume Identifier (VID)**, which identifies a specific replica in a location independent way.

The approach followed in Coda is to assign each file a 96-bit file identifier. A file identifier consists of two parts as shown in Fig. 10-26.

The first part is the 32-bit RVID of the logical volume that the file is part of. To locate a file, a client first passes the RVID of a file identifier to a **volume replication database**, which returns the list of VIDs associated with that RVID. Given a VID, a client can then look up the server that is currently hosting the particular replica of the logical volume. This lookup is done by passing the VID to a **volume location database** which returns the current location of that specific physical volume.

The second part of a file identifier consists of a 64-bit file handle that uniquely identifies the file within a volume. In reality, it corresponds to the identification of an index node as represented within VFS. Such a **vnode** as it is called, is similar to the notion of an inode in UNIX systems.

10.2.5 Synchronization

Many distributed file systems, including Coda's ancestor, AFS, do not provide UNIX file-sharing semantics but instead support the weaker session semantics. Given its goal to achieve high availability, Coda takes a different approach and

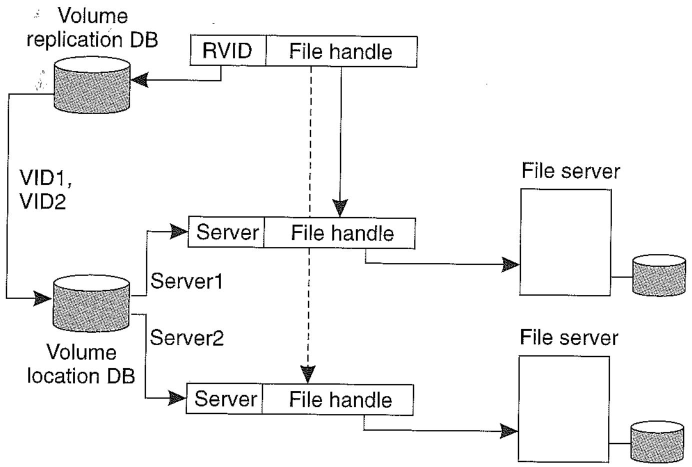

Figure 10-26. The implementation and resolution of a Coda file identifier.

makes an attempt to support transactional semantics, albeit a weaker form than normally supported by transactions.

The problem that Coda wants to solve is that in a large distributed file system it may easily happen that some or all of the file servers are temporarily unavailable. Such unavailability can be caused by a network or server failure, but may also be the result of a mobile client deliberately disconnecting from the file service. Provided that the disconnected client has all the relevant files cached locally, it should be possible to use these files while disconnected and reconcile later when the connection is established again.

Sharing Files in Coda

To accommodate file sharing, Coda uses a special allocation scheme that bears some similarities to share reservations in NFS. To understand how the scheme works, the following is important. When a client successfully opens a file *f*, an entire copy of *f* is transferred to the client's machine. The server records that the client has a copy of *f*. So far, this approach is similar to open delegation in NFS.

Now suppose client *A* has opened file *f* for writing. When another client *B* wants to open *f* as well, it will fail. This failure is caused by the fact that the server has recorded that client *A* might have already modified *f*. On the other hand, had client *A* opened *f* for reading, an attempt by client *B* to get a copy from the server for reading would succeed. An attempt by *B* to open for writing would succeed as well.

Now consider what happens when several copies of *f* have been stored locally at various clients. Given what we have just said, only one client will be able to modify *f*. If this client modifies *f* and subsequently closes the file, the file will be transferred back to the server. However, each other client may proceed to read its local copy despite the fact that the copy is actually outdated.

The reason for this apparently inconsistent behavior, is that a session is treated as a transaction in Coda. Consider Fig. 10-27, which shows the time line for two processes, *A* and *B*. Assume *A* has opened *f* for reading, leading to session S_A. Client *B* has opened *f* for writing, shown as session S_B.

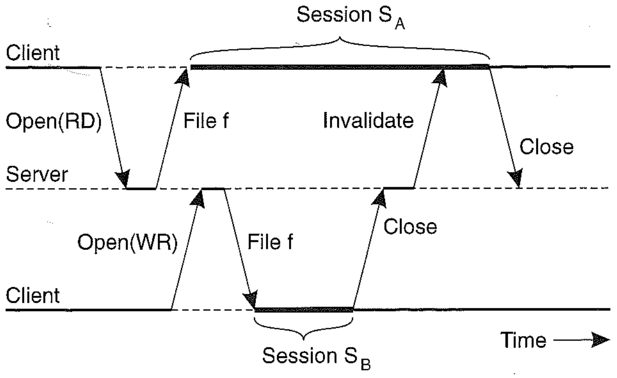

Figure 10-27. The transactional behavior in sharing files in Coda.

When *B* closes session S_B, it transfers the updated version of *f* to the server, which will then send an invalidation message to *A*. *A* will now know that it is reading from an older version of *f*. However, from a transactional point of view, this really does not matter because session S_A could be considered to have been scheduled before session S_B.

Transactional Semantics

In Coda, the notion of a network partition plays a crucial role in defining transactional semantics. A **partition** is a part of the network that is isolated from the rest and which consists of a collection of clients or servers, or both. The basic idea is that series of file operations should continue to execute in the presence of conflicting operations across different partitions. Recall that two operations are said to conflict if they both operate on the same data, and at least one is a write operation.

Let us first examine how conflicts may occur in the presence of network partitions. Assume that two processes *A* and *B* hold identical replicas of various shared data items just before they are separated as the result of a partitioning in the network. Ideally, a file system supporting transactional semantics would implement **one-copy serializability**, which is the same as saying that the execution of operations by *A* and *B*, respectively, is equivalent to a joint serial execution of those

operations on nonreplicated data items shared by the two processes. This concept is discussed further in (Davidson et al., 1985).

We already came across examples of serializability in Chap. 5. The main problem in the face of partitions is to *recognize* serializable executions after they have taken place within a partition. In other words, when recovering from a network partition, the file system is confronted with a number of transactions that have been executed in each partition (possibly on shadow copies, i.e., copies of files that were handed out to clients to perform tentative modifications analogous to the use of shadow blocks in the case of transactions). It will then need to check whether the joint executions can be serialized in order to accept them. In general, this is an intractable problem.

The approach followed in Coda is to interpret a session as a transaction. A typical session starts with explicitly opening a file by a call to open. Hereafter, a series of read and write operations will generally follow, after which the session is terminated by closing the file with a call to close. Most UNIX system calls constitute a single session on their own and are also considered by Coda to be independent transactions.

Coda recognizes different types of sessions. For example, each UNIX system call is associated with a different session type. More complex session types are the ones that start with a call to open. The type of a session is automatically deduced from the system calls made by an application. An important advantage of this approach is that applications that use the standardized VFS interface need not be modified. For each session type, it is known in advance which data will be read or modified.

As an example, consider the **store** session type, which starts with opening a file *f* for writing on behalf of a specific user *u*, as explained above. Fig. 10-28 lists the meta-data associated with *f* and user *u* that are affected by this session type, and whether they are only read or also modified. For example, it will be necessary to read the access rights that user *u* has with respect to file *f*. By explicitly identifying the metadata that are read and modified for a specific session, it becomes much easier to recognize conflicting operations.

File-associated data	Read?	Modified?
File identifier	Yes	No
Access rights	Yes	No
Last modification time	Yes	Yes
File length	Yes	Yes
File contents	Yes	Yes

Figure 10-28. The metadata read and modified for a *store* session type in Coda.

From this point on, session processing is fairly straightforward. Let us start by considering a series of concurrent sessions that take place within a single network partition. To simplify matters, assume that there is a single server contained in the partition. When a client starts a session, the Venus process on the client machine fetches all data contained in the session's read set and write set from the server, provided that the rules for file sharing as explained above are not violated. In doing so, it effectively acquires the necessary read and write locks for those data.

There are two important observations to make at this point. First, because Coda can automatically infer the type of a session from the (first) system call made by an application, and knows the metadata that are read and modified for each session type, a Venus process knows which data to fetch from a server at the start of a session. As a consequence, it can acquire the necessary locks at the start of a session.

The second observation is that because file sharing semantics are effectively the same as acquiring the necessary locks in advance, the approach followed in this case is the same as applying a two-phase locking (2PL) technique as explained in Chap. 5. An important property of 2PL, is that all resulting schedules of read and write operations of concurrent sessions are serializable.

Now consider processing sessions in the face of partitions. The main problem that needs to be solved is that conflicts across partitions need to be resolved. For this purpose, Coda uses a simple versioning scheme. Each file has an associated version number that indicates how many updates have taken place since the file was created. As before, when a client starts a session, all the data relevant to that session are copied to the client's machine, including the version number associated with each data element.

Assume that while one or more sessions are being executed at the client, a network partition occurs by which the client and server are disconnected. At that point, Venus will allow the client to continue and finish the execution of its sessions as if nothing happened, subject to the transactional semantics for the single-partition case described above. Later, when the connection with the server is established again, updates are transferred to the server in the same order as they took place at the client.

When an update for file f is transferred to the server, it is tentatively accepted if no other process has updated f while the client and server were disconnected. Such a conflict can be easily detected by comparing version numbers. Let V_{client} be the version number of f acquired from the server when the file was transferred to the client. Let $N_{updates}$ be the number of updates in that session that have been transferred and accepted by the server after reintegration. Finally, let V_{now} denote the current version number of f at the server. Then, a next update for f from the client's session can be accepted if and only if

$$V_{now} + 1 = V_{client} + N_{updates}$$

In other words, an update from a client is accepted only when that update would lead to the *next* version of file f. In practice, this means that only a single client will eventually win having the effect that the conflict is resolved.

Conflicts arise if f is being updated in concurrently executed sessions. When a conflict occurs, the updates from the client's session are undone, and the client is, in principle, forced to save its local version of f for manual reconciliation. In terms of transactions, the session cannot be committed and conflict resolution is left to the user. We return to this issue below.

10.2.6 Caching and Replication

As should be clear by now, caching and replication play an important role in Coda. In fact, these two approaches are fundamental for achieving the goal of high availability as set out by the developers of Coda. In the following, we first take a look at client-side caching, which is crucial in the face of disconnected operation. We then take a look at server-side replication of volumes.

Client Caching

Client-side caching is crucial to the operation of Coda for two reasons. First, and in line with the approach followed in AFS (Satyanarayanan, 1992), caching is done to achieve scalability. Second, caching provides a higher degree of fault tolerance as the client becomes less dependent on the availability of the server. For these two reasons, clients in Coda always cache entire files. In other words, when a file is opened for either reading or writing, an entire copy of the file is transferred to the client, where it is subsequently cached.

Unlike many other distributed file systems, cache coherence in Coda is maintained by means of callbacks. For each file, the server from which a client had fetched the file keeps track of which clients have a copy of that file cached locally. A server is said to record a **callback promise** for a client. When a client updates its local copy of the file for the first time, it notifies the server, which, in turn, sends an invalidation message to the other clients. Such an invalidation message is called a **callback break**, because the server will then discard the callback promise it held for the client it just sent an invalidation.

The interesting aspect of this scheme is that as long as a client knows it has an outstanding callback promise at the server, it can safely access the file locally. In particular, suppose a client opens a file and finds it is still in its cache. It can then use that file provided the server still has a callback promise on the file for that client. The client will have to check with the server if that promise still holds. If so, there is no need to transfer the file from the server to the client again.

This approach is illustrated in Fig. 10-29, which is an extension of Fig. 10-27. When client A starts session S_A, the server records a callback promise. The same happens when B starts session S_B. However, when B closes S_B, the server breaks its promise to callback client A by sending A a callback break. Note that due to the transactional semantics of Coda, when client A closes session S_A, nothing special happens; the closing is simply accepted as one would expect.

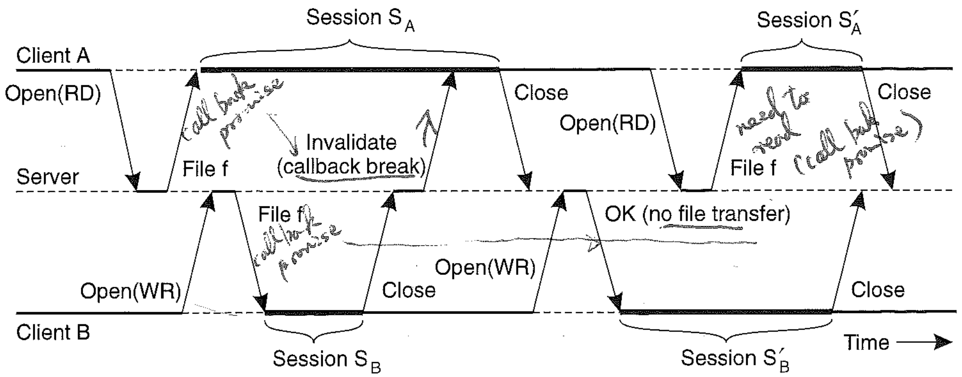

Figure 10-29. The use of local copies when opening a session in Coda.

The consequence is that when A later wants to open session S'_A, it will find its local copy of f to be invalid, so that it will have to fetch the latest version from the server. On the other hand, when B opens session S'_B, it will notice that the server still has an outstanding callback promise implying that B can simply re-use the local copy it still has from session S_B.

Server Replication

Coda allows file servers to be replicated. As we mentioned, the unit of replication is a volume. The collection of servers that have a copy of a volume, are known as that volume's **Volume Storage Group**, or simply **VSG**. In the presence of failures, a client may not have access to all servers in a volume's VSG. A client's **Accessible Volume Storage Group (AVSG)** for a volume consists of those servers in that volume's VSG that the client can contact. If the AVSG is empty, the client is said to be **disconnected**.

Coda uses a replicated-write protocol to maintain consistency of a replicated volume. In particular, it uses a variant of Read-One, Write-All (ROWA), which was explained in Chap. 6. When a client needs to read a file, it contacts one of the members in its AVSG of the volume to which that file belongs. However, when closing a session on an updated file, the client transfers it in parallel to each member in the AVSG. This parallel transfer is accomplished by means of multiRPC as explained before.

This scheme works fine as long as there are no failures, that is, for each client, that client's AVSG of a volume is the same as its VSG. However, in the presence of failures, things may go wrong. Consider a volume that is replicated across three servers S_1, S_2, and S_3. For client A, assume its AVSG covers servers S_1 and S_2 whereas client B has access only to server S_3, as shown in Fig. 10-30.

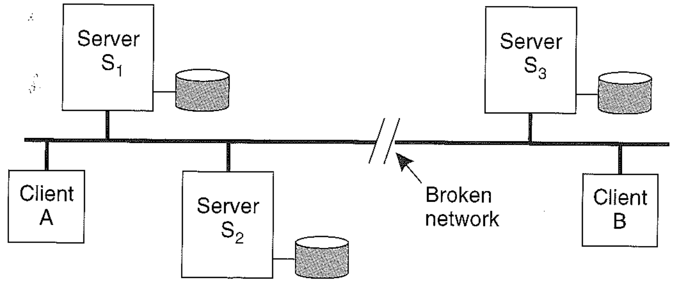

Figure 10-30. Two clients with a different AVSG for the same replicated file.

Coda uses an optimistic strategy for file replication. In particular, both A and B will be allowed to open a file, f, for writing, update their respective copies, and transfer their copy back to the members in their AVSG. Obviously, there will be different versions of f stored in the VSG. The question is how this inconsistency can be detected and resolved.

The solution adopted by Coda is an extension to the versioning scheme discussed in the previous section. In particular, a server S_i in a VSG maintains a **Coda version vector** $CVV_i(f)$ for each file f contained in that VSG. If $CVV_i(f)[j] = k$, then server S_i knows that server S_j has seen at least version k of file f. $CVV_i[i]$ is the number of the current version of f stored at server S_i. An update of f at server S_i will lead to an increment of $CVV_i[i]$. Note that version vectors are completely analogous to the vector timestamps discussed in Chap. 5.

Returning to our three-server example, $CVV_i(f)$ is initially equal to $[1,1,1]$ for each server S_i. When client A reads f from one of the servers in its AVSG, say S_1, it also receives $CVV_1(f)$. After updating f, client A multicasts f to each server in its AVSG, that is, S_1 and S_2. Both servers will then record that their respective copy has been updated, but not that of S_3. In other words,

$$CVV_1(f) = CVV_2(f) = [2,2,1]$$

Meanwhile, client B will be allowed to open a session in which it receives a copy of f from server S_3, and subsequently update f as well. When closing its session and transferring the update to S_3, server S_3 will update its version vector to $CVV_3(f)$=[1,1,2].

When the partition is healed, the three servers will need to reintegrate their copies of f. By comparing their version vectors, they will notice that a conflict has occurred that needs to be repaired. In many cases, conflict resolution can be automated in an application-dependent way, as discussed in (Kumar and Satyanarayanan, 1995). However, there are also many cases in which users will have to assist in resolving a conflict manually, especially when different users have changed the same part of the same file in different ways.

10.2.7 Fault Tolerance

Coda has been designed for high availability, which is mainly reflected by its sophisticated support for client-side caching and its support for server replication. We have discussed both in the preceding sections. An interesting aspect of Coda that needs further explanation is how a client can continue to operate while being disconnected, even if disconnection lasts for hours or days.

Disconnected Operation

As we mentioned above, a client is said to be disconnected with respect to a volume, if its AVSG for that volume is empty. In other words, the client cannot contact any of the servers holding a copy of the volume. In most file systems (e.g., NFS), a client is not allowed to proceed unless it can contact at least one server. A different approach is followed in Coda. There, a client will simply resort to using its local copy of the file that it had when it opened the file at a server.

Closing a file (or actually, the session in which the file is accessed) when disconnected will always succeed. However, it may be possible that conflicts are detected when modifications are transferred to a server when connection is established again. In case automatic conflict resolution fails, manual intervention will be necessary. Practical experience with Coda has shown that disconnected operation generally works, although there are occasions in which reintegration fails due to unresolvable conflicts.

The success of the approach followed in Coda is mainly attributed to the fact that, in practice, *write-sharing* a file hardly occurs. In other words, in practice it is rare for two processes to open the same file for writing. Of course, sharing files for only reading happens a lot, but that does not impose any conflicts. These observations have also been made for other file systems (see, e.g., Page et al., 1998, for conflict resolution in a highly distributed file system). Furthermore, the transactional semantics underlying Coda's file-sharing model also makes it easy to handle the case in which there are multiple processes only reading a shared file at the same time that exactly one process is concurrently modifying that file.

The main problem that needs to be solved to make disconnected operation a success, is to ensure that a client's cache contains those files that will be accessed during disconnection. If a simple caching strategy is followed, it may turn out that a client cannot continue as it lacks the necessary files. Filling the cache in advance with the appropriate files is called **hoarding**. The overall behavior of a client with respect to a volume (and thus the files in that volume) can now be summarized by the state-transition diagram shown in Fig. 10-31.

Normally, a client will be in the *HOARDING* state. In this state, the client is connected to (at least) one server that contains a copy of the volume. While in this state, the client can contact the server and issue file requests to perform its work. Simultaneously, it will also attempt to keep its cache filled with useful data (e.g., files, file attributes, and directories).

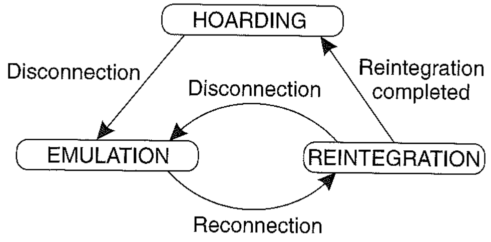

Figure 10-31. The state-transition diagram of a Coda client with respect to a volume.

At a certain point, the number of servers in the client's AVSG will drop to zero, bringing it into an *EMULATION* state in which the behavior of a server for the volume will have to be emulated on the client's machine. In practice, this means that all file requests will be directly serviced using the locally cached copy of the file. Note that while a client is in its *EMULATION* state, it may still be able to contact servers that manage other volumes. In such cases, disconnection will generally have been caused by a server failure rather than that the client has been disconnected from the network.

Finally, when reconnection occurs, the client enters the *REINTEGRATION* state in which it transfers updates to the server in order to make them permanent. It is during reintegration that conflicts are detected and, where possible, automatically resolved. As shown in Fig. 10-31, it is possible that during reintegration the connection with the server is lost again, bringing the client back into the *EMULATION* state.

Crucial to the success of continuous operation while disconnected is that the cache contains all the necessary data. Coda uses a sophisticated priority mechanism to ensure that useful data are indeed cached. First, a user can explicitly state which files or directories he finds important by storing pathnames in a **hoard database**. Coda maintains such a database per workstation. Combining the information in the hoard database with information on recent references to a file allows Coda to compute a current priority on each file, after which it fetches files in priority such that the following three conditions are met:

1. There is no uncached file with a higher priority than any cached file.
2. The cache is full, or no uncached file has nonzero priority.
3. Each cached file is a copy of the one maintained in the client's AVSG.

The details of computing a file's current priority are described in (Kistler, 1996). If all three conditions are met, the cache is said to be in **equilibrium**. Because the current priority of a file may change over time, and because cached files may need to be removed from the cache to make room for other files, it is clear that cache equilibrium needs to be recomputed from time to time. Reorganizing the cache such that equilibrium is reached is done by an operation known as a **hoard walk**, which is invoked once every 10 minutes.

The combination of the hoard database, priority function, and maintaining cache equilibrium has shown to be a vast improvement over traditional cache management techniques, which are generally based on counting and timing references. However, the technique cannot guarantee that a client's cache will always contain the data the user will need in the near future. Therefore, there are still occasions in which an operation in disconnected mode will fail due to inaccessible data.

Recoverable Virtual Memory

Besides providing high availability, the AFS and Coda developers have also looked at simple mechanisms that help in building fault-tolerant processes. A simple and effective mechanism that makes recovery much easier, is **Recoverable Virtual Memory** (**RVM**). RVM is a user-level mechanism to maintain crucial data structures in main memory while being ensured that they can be easily recovered after a crash failure. The details of RVM are described in (Satyanarayanan et al., 1994).

The basic idea underlying RVM is relatively simple: data that should survive crash failures are normally stored in a file that is explicitly mapped into memory when needed. Operations on that data are logged, similar to the use of a writeahead log in the case of transactions. In fact, the model supported by RVM is close to that of flat transactions, except that no support is provided for concurrency control.

Once a file has been mapped into main memory, an application can perform operations on that data that are part of a transaction. RVM is unaware of data structures. Therefore, the data in a transaction is explicitly set by an application as a range of consecutive bytes of the mapped-in file. All (in-memory) operations on that data are recorded in a separate writeahead log that needs to be kept on stable storage. Note that due to the generally relatively small size of the log, it is feasible to use a battery power-supplied part of main memory, which combines durability with high performance.

10.2.8 Security

Coda inherits its security architecture from AFS, which consists of two parts. The first part deals with setting up a secure channel between a client and a server using secure RPC and system-level authentication. The second part deals with controlling access to files. We will not examine each of these in turn.

Secure Channels

Coda uses a secret-key cryptosystem for setting up a secure channel between a client and a server. The protocol followed in Coda is derived from the Needham-Schroeder authentication protocol discussed in Chap. 8. A user is first required to obtain special tokens from an authentication server (AS), as we explain shortly. These tokens are somewhat comparable to handing out a ticket in the Needham-Schroeder protocol in the sense that they are used to subsequently set up a secure channel to a server.

All communication between a client and server is based on Coda's secure RPC mechanism, which is shown in Fig. 10-32. If Alice (as client) wants to talk to Bob (as server), she sends her identity to Bob, along with a challenge R_A, which is encrypted with the secret key $K_{A,B}$ shared between Alice and Bob. This is shown as message 1.

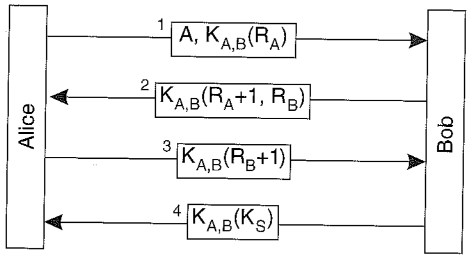

Figure 10-32. Mutual authentication in RPC2.

Bob responds by decrypting message 1 and returning $R_A + 1$, proving that he knows the secret key and thus that he is Bob. He returns a challenge R_B (as part of message 2), which Alice will have to decrypt and return as well (shown as message 3). When mutual authentication has taken place, Bob generates a session key K_S that can be used in further communication between Alice and Bob.

Secure RPC in Coda is used only to set up a secure connection between a client and a server; it is not enough for a secure login session that may involve several servers and which may last considerably longer. Therefore, a second protocol is used that is layered on top of secure RPC, in which a client obtains authentication tokens from the AS as briefly mentioned above.

An **authentication token** is somewhat like a ticket in Kerberos. It contains an identifier of the process that obtained it, a token identifier, a session key, timestamps telling when the token becomes valid and when it expires. A token may be cryptographically sealed for integrity. For example, if T is a token, K_{vice} a secret key shared by all Vice servers, and H a hash function, then $[T, H(K_{vice}, T)]$ is a cryptographically sealed version of T. In other words, despite the fact that T is sent as plaintext over an insecure channel, it is impossible for an intruder to modify it such that a Vice server would not notice the modification.

When Alice logs into Coda, she will first need to get authentication tokens from the AS. In this case, she does a secure RPC using her password to generate the secret key $K_{A,AS}$ she shares with the AS, and which is used for mutual authentication as explained before. The AS returns two authentication tokens. A **clear token** $CT = [A, TID, K_S, T_{start}, T_{end}]$ identifying Alice and containing a token identifier *TID*, a session key K_S, and two timestamps T_{start} and T_{end} indicating when the token is valid. In addition, it sends a **secret token** $ST = K_{vice}([CT]^{*}_{K_{vice}})$, which is *CT* cryptographically sealed with the secret key K_{vice} that is shared between all Vice servers, and encrypted with that same key.

A Vice server is capable of decrypting *ST* revealing *CT* and thus the session key K_S. Also, because only Vice servers know K_{vice}, such a server can easily check whether *CT* has been tampered with by doing an integrity check (of which the computation requires K_{vice}).

Whenever Alice wants to set up a secure channel with a Vice server, she uses the secret token *ST* to identify herself as shown in Fig. 10-33. The session key K_S that was handed to her by the AS in the clear token is used to encrypt the challenge R_A she sends to the server.

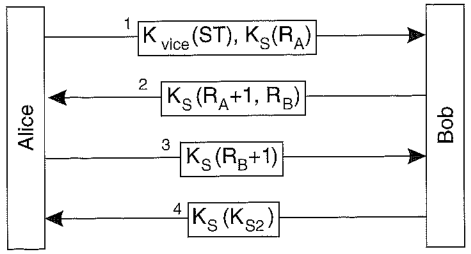

Figure 10-33. Setting up a secure channel between a (Venus) client and a Vice server in Coda.

The server, in turn, will first decrypt *ST* using the shared secret key K_{vice}, giving *CT*. It will then find K_S, which it subsequently uses to complete the authentication protocol. Of course, the server will also do an integrity check on *CT* and proceed only if the token is currently valid.

A problem occurs when a client needs to be authenticated before it can access files, but is currently disconnected. Authentication cannot be done because the server cannot be contacted. In this case, authentication is postponed and access is tentatively granted. When the client reconnects to the server(s), authentication takes place before entering the *REINTEGRATION* state.

Access Control

Let us briefly consider protection in Coda. As in AFS, Coda uses access control lists to ensure that only authorized processes have access to files. For reasons of simplicity and scalability, a Vice file server associates an access control list only with directories and not with files. All normal files in the same directory (i.e., excluding subdirectories) share the same protection rights.

Coda distinguishes access rights with respect to the types of operations shown in Fig. 10-34. Note that there is no right with respect to executing a file. There is simple reason for this omission: execution of files takes place at clients and is thus out of the scope of a Vice file server. Once a client has downloaded a file there is no way for Vice to even tell whether the client is executing it or just reading it.

Operation	Description
Read	Read any file in the directory
Write	Modify any file in the directory
Lookup	Look up the status of any file
Insert	Add a new file to the directory
Delete	Delete an existing file
Administer	Modify the ACL of the directory

Figure 10-34. Classification of file and directory operations recognized by Coda with respect to access control.

Coda maintains information on users and groups. Besides listing the rights a user or group has, Coda also supports the listing of negative rights. In other words, it is possible to explicitly state that a specific user is *not* permitted certain access rights. This approach has shown to be convenient in the light of immediately revoking access rights of a misbehaving user, without having to first remove that user from all groups.

10.3 OTHER DISTRIBUTED FILE SYSTEMS

There are many more distributed file systems than the ones we have discussed so far, although most have many similarities with NFS or Coda. In this section, we take a brief look at three systems that each are different in one or more respects. We first discuss Plan 9, in which every resource is treated as a file. XFS is an example of a serverless file system. Our last example is SFS, a file system in which file names also contain security information.

In contrast to the description of the NFS and Coda, we deliberately leave out many details of the three systems discussed in this section. Instead, we concentrate mainly on the general principles underlying each of the systems with the purpose of giving an impression of alternative approaches to distributed file systems.

10.3.1 Plan 9: Resources Unified to Files

Starting in the 1980s, there has been a trend to replace the centralized timesharing systems with networks of powerful workstations, leading to the network operating systems as we discussed in Chap. 1. One of the problems with this