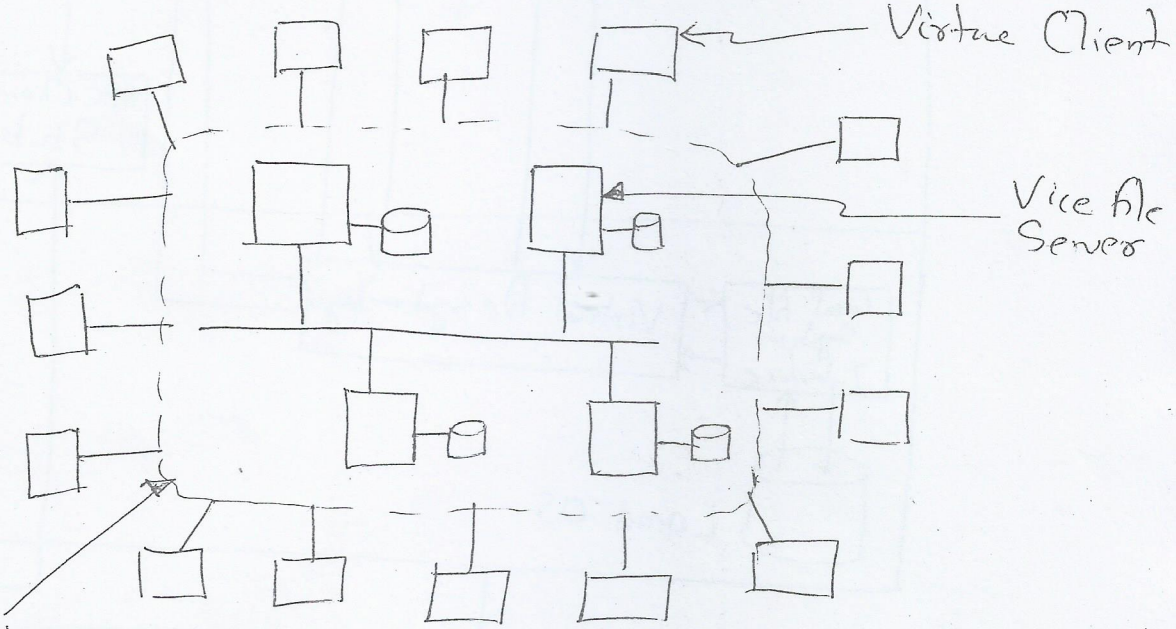
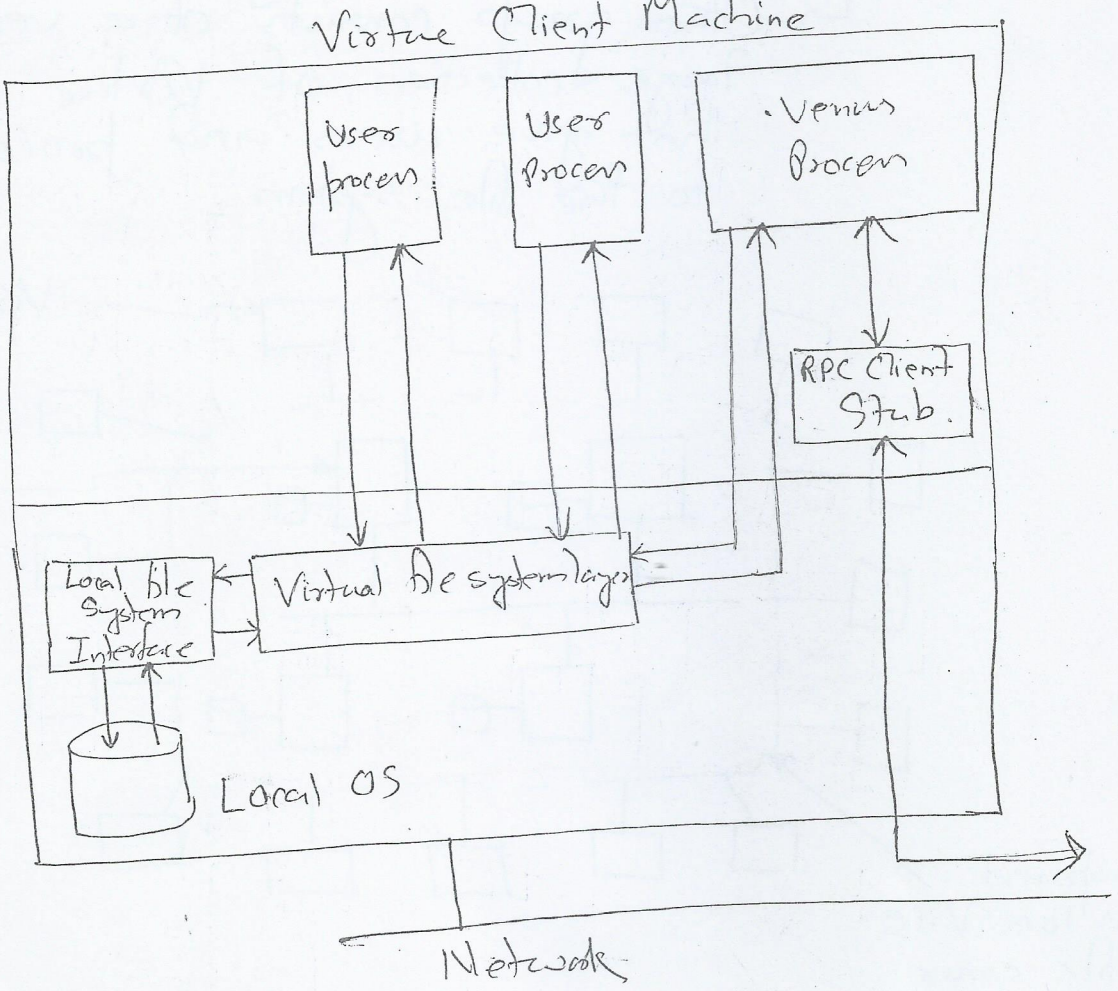
Overview of Coda

- It was designed ~~file system~~ 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 file system.
- Coda nodes are partitioned into two groups
  - ↳ One group consists of a relatively small number of dedicated Vice file servers, which are centrally administered.
  - ↳ Other group consists of a very much larger collection of Virtue workstations that give users and processes access to the file system



process called Venus. ~~clause~~ ~~rate~~

- A Venus process is responsible for providing access to the files that are maintained by the Vice file servers.
- Venus is also responsible for allowing the client to continue operation even if access to the file servers is (temporarily) impossible.
- The internal architecture of a Virtue workstation is:



+ Venus runs as a user-level process
+ Virtual file System (VFS) layer intercepts all calls from client applications, and forwards these calls either to the local file system or to Venus.

+ 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

UDP
At-most-once semantics

— There are three different server-side processes.
  ↳ Vice file server — responsible for maintaining a local collection of files.
    ↳ A file server runs as a user-level process.
  ↳ Trusted Vice machines are allowed to run an authentication server.
  ↳ Update processes are used to keep meta-information on the file system consistent at each Vice server.

— Coda provides a globally shared name space that is maintained by the Vice servers.
  ↳ Client 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.

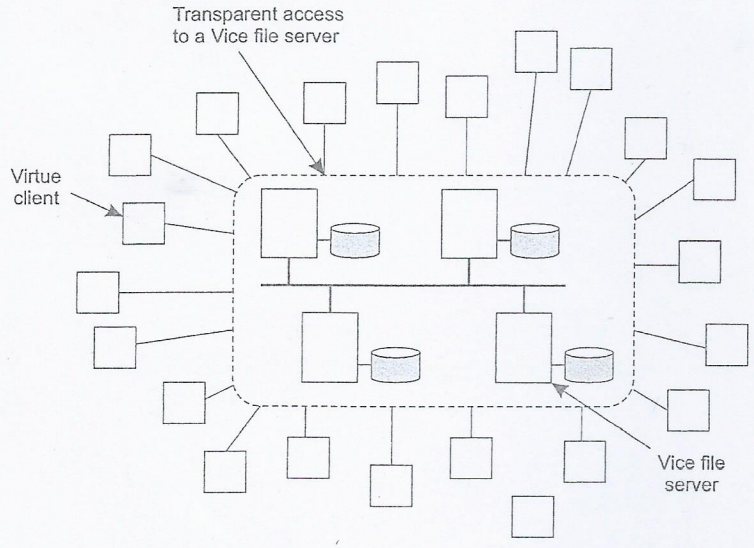


Figure 10-1. The overall organization of AFS.

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.

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. 10-0 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

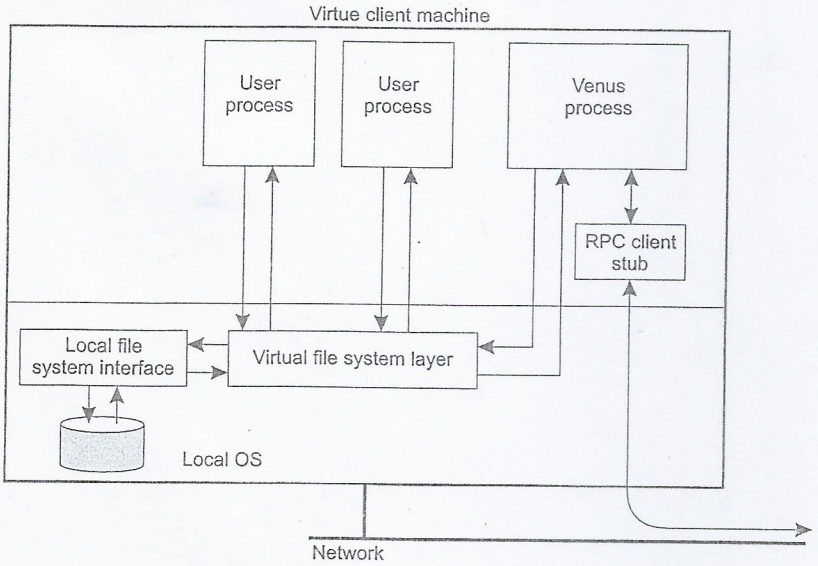


Figure 10-2. The internal organization of a Virtue workstation.

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. 10-3.

Another feature of RPC2 that makes it different from other RPC systems, is

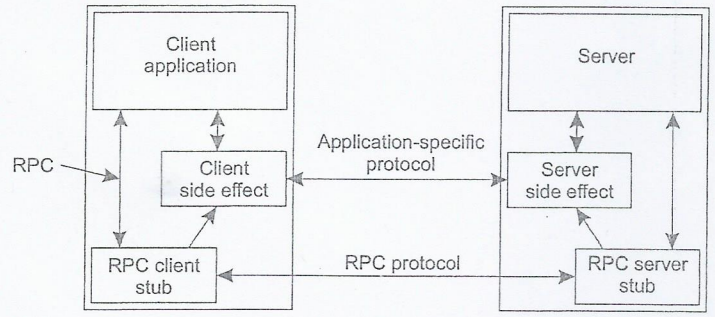


Figure 10-3. Side effects in Coda's RPC2 system.

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. 10-4(a).

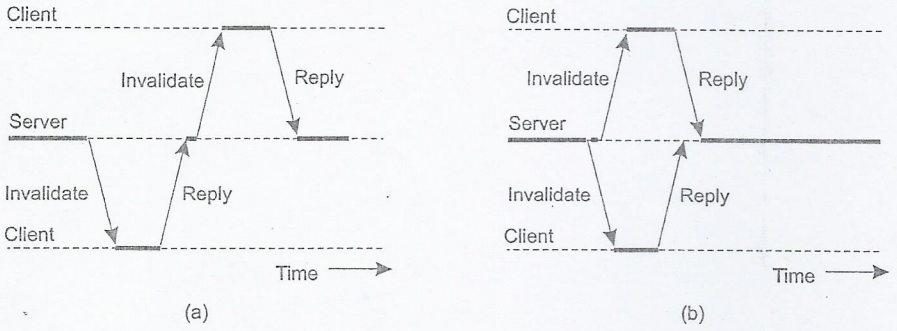


Figure 10-4. (a) Sending an invalidation message one at a time. (b) Sending invalidation messages in parallel.

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. 10-4(b). Instead of invalidating each copy one-by-one, the server sends an invalidation message to all clients in parallel. As 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. 10-5. 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.

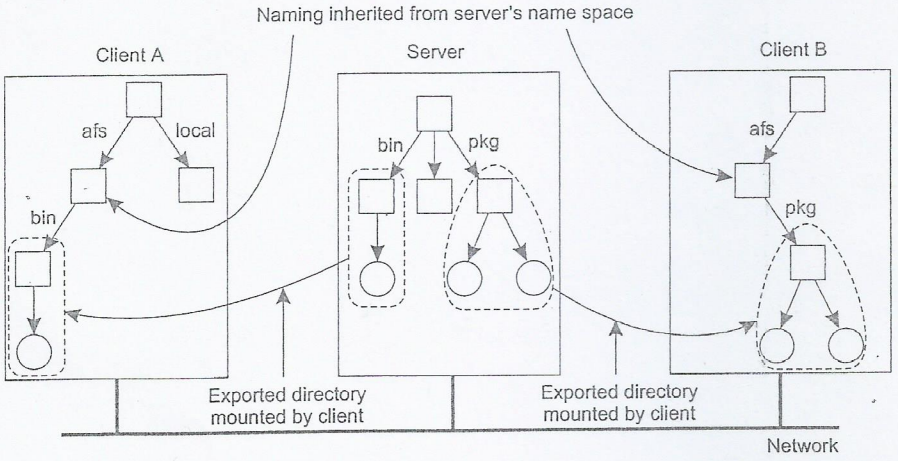


Figure 10-5. Clients in Coda have access to a single shared name space.

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.

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. 10-6.

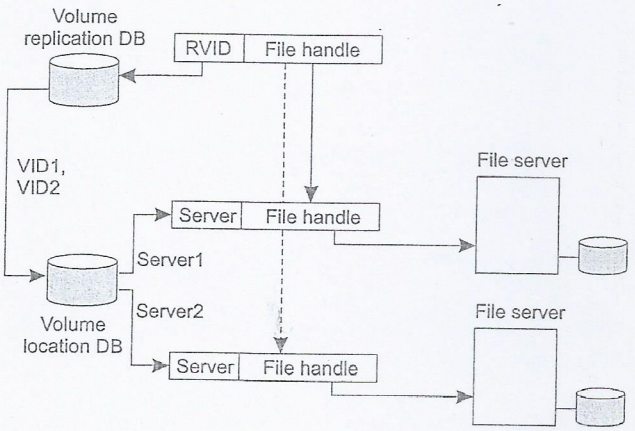


Figure 10-6. The implementation and resolution of a Coda file identifier.

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

Synchronization

- 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

+ When a client succesfully opens a file 'f', an entire copy of 'f' is transfered to the client's machine.
+ The server records that the client has a copy of 'f'.
+ Suppose client 'A' has opened file 'f' for writing.
  - ↳ When another client 'B' wants to open 'f' as well, it will fail.

Server has record. { Client 'A' might have already modified 'f'

↳ An attempt by 'B' to get a copy from server for reading would succeed.

↳ An attempt by 'B' to open for writing would succeed as well.

+ Several copies of 'f' will be stored locally at various client.

↳ BUT only one client will be able to modify 'f'.

↓

If the 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.

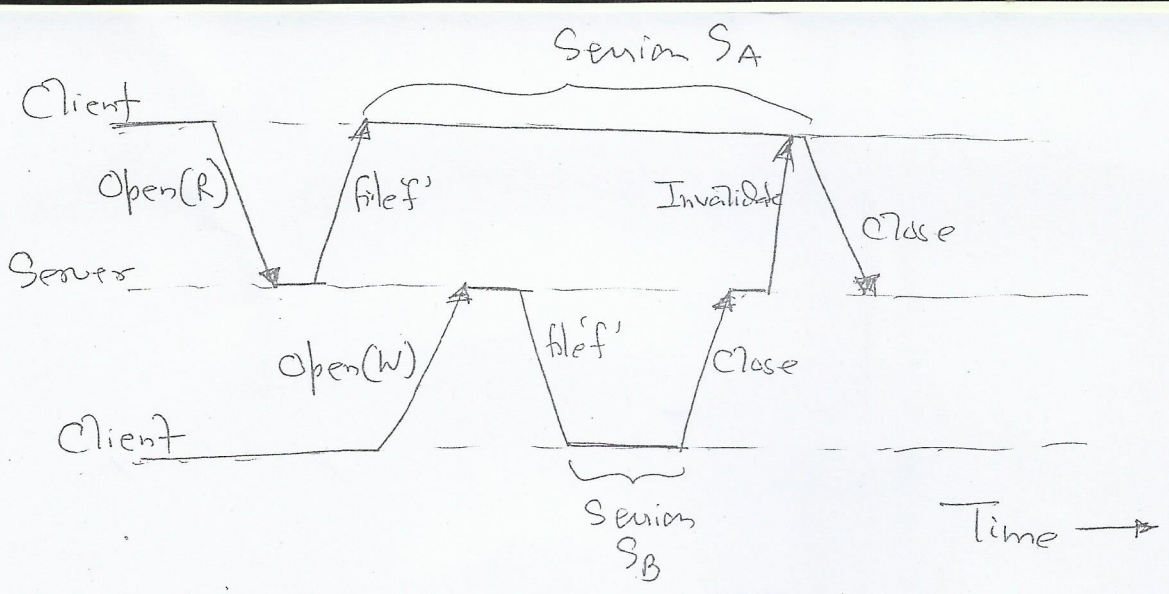
↳ The reason for this apparently inconsistent behaviour, is that a session is treated as a transaction in Coda.

+ figure shows time line for two processes 'A' and 'B'.

'A' → Opened file 'f' for reading

↓

leading to session 'S_A'



Client 'B' → Opened file 'f' for writing
↓
Session 'S_B'

→ When 'B' closes session 'S_B', it transfer the updated version of 'f' to the server
↓ in turn
Send invalidation message to 'A'
↳ 'A' will know that it is reading old data (older version of 'f')
↓
Session 'S_A' will close.

Caching & Replication

— These two approaches are fundamental for achieving the goal of high availability.

Client Caching

— Client side caching is crucial to the operation of Coda for two reasons
  - ↳ Caching is done to achieve scalability.
  - ↳ 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.

— 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.

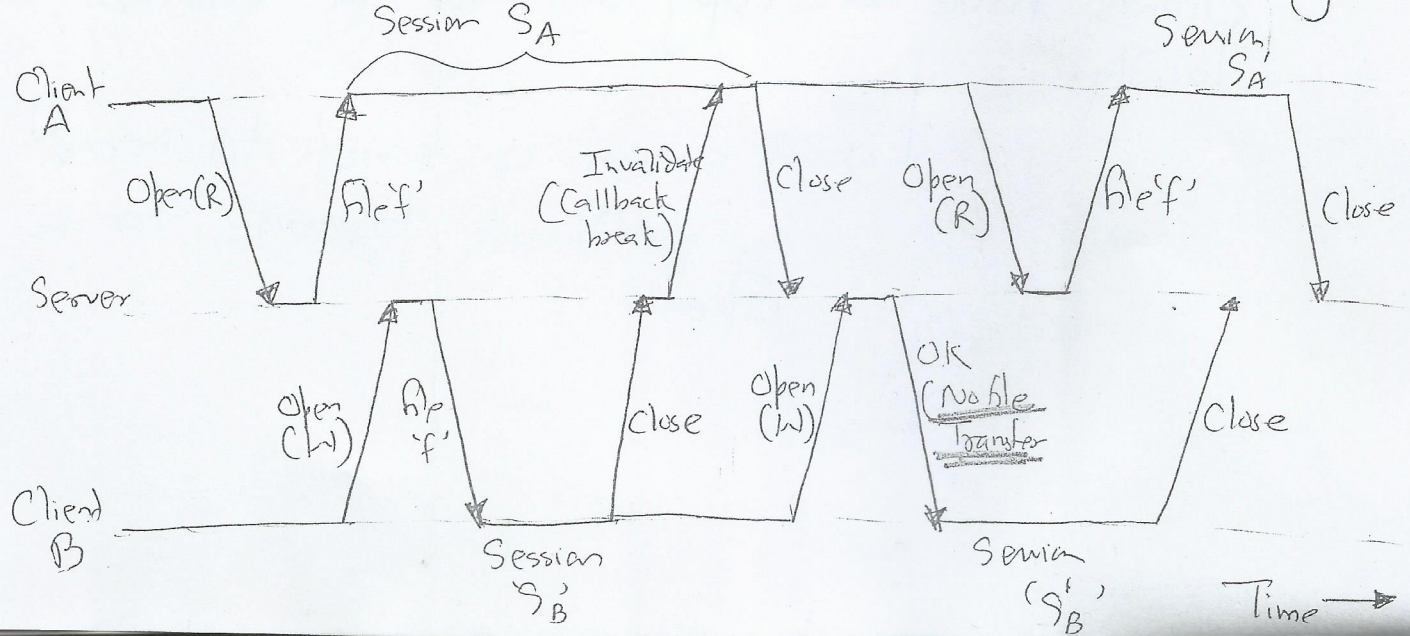
file for the first time, it notifies the server, which, in turn, sends an invalidation message to the other clients.

↳ Such invalidation message is called a callback break.

↳ Broz the server will then discard the callback promise it held for the client it just sent an invalidation.

+ As long as a client knows it has an outstanding callback promise at the server, it can access the file locally. (safely)

+ If the client needs some file and the file is in its cache. 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 again.



+ When client 'A' starts session 'S_A'
  ↳ Server records a callback promise.
+ Same with client 'B'.
+ When 'B' closes 'S_B', the server breaks its promise to callback client 'A' by sending callback break message.
  ↳ Client 'A' closes 'S_A'
+ When 'A' later wants to open session 'S'_A', it will find its local copy of 'f' to be invalid, so that it will fetch the latest version from the server.

↳ When 'B' opens session S'_B, it will notice that server still has an outstanding callback promise implying that 'B' can simply re-use the local copy it still has from session 'S_B'.

→ Coda allows file servers to be replicated.

→ Unit of replication is a volume

→ The collection of servers that have a copy of a volume, are known as that volumn's Volume Storage Group (VSG).

→ In the presence of failures, a client may not have access to all servers in a volume's VSG.

↳ A client's Accessible VSG (AVSG) for a volume consists of those servers in that volume's VSG that the client can contact.

If AVSG is empty → The client is said to be disconnected.

→ Coda uses a replicated-write protocol to maintain consistency of a replicated volume.

→ 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.

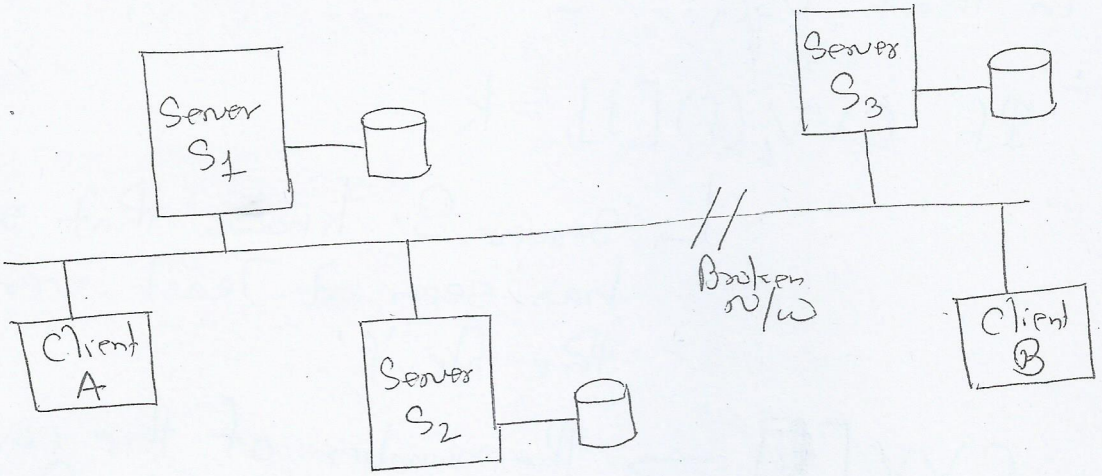
→ When closing a ~~file~~ session on an updated file, the client transfers it in parallel to each member in the AVSG.

↓

multiRPC ←

- This scheme works fine when client's AVSG of a volume is the same as its VSG.

- Consider a volume that is replicated across three servers S_1, S_2 and S_3.



for client 'A', assume its AVSG covers S_1 and S_2

Client 'B' has access only to server S_3.

+ Coda uses an optimistic strategy for file replication
  - Both client '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.
    - There will be different version of 'f' stored in the VSG.

version vector & $CVV_i(f)$ for each file 'f' contained in that VSG.

If $CVV_i(f)[j] = k$

↳ Server S_i knows that server S_j has seen at least version 'k' of the file 'f'.

$CVV_i[i]$ → 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 $CVV_i[i]$.

+ for server S_1, S_2, & S_3

$CVV_i(f)$ 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.

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

+ Meanwhile, client 'B' will be allowed to open a session in which it receives a copy of 'f' from 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 and that needs to be repaired.

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. 10-11.

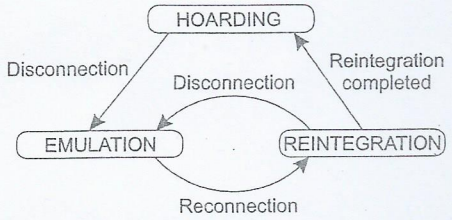


Figure 10-11. The state-transition diagram of a Coda client with respect to a volume.

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).

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. 10-11, 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.

Communication

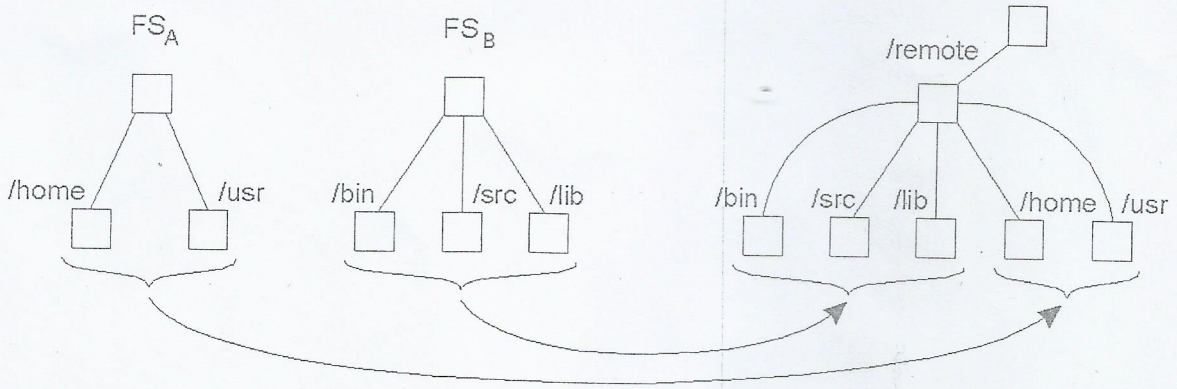
- For communications Plan 9 uses the 9P protocol and network interfaces are represented as directories.

File	Description
ctl	Used to write protocol-specific control commands
data	Used to read and write data
listen	Used to accept incoming connection setup requests
local	Provides information on the caller's side of the connection
remote	Provides information on the other side of the connection
status	Provides diagnostic information on the current status of the connection

Files associated with a single TCP connection in Plan 9.

Naming

- A client can mount multiple name spaces at the same mount point composing a **union directory**.
- The mounting order is maintained in file search.



A union directory in Plan 9.