5.1 Introduction

Until recently, mobile clients were mainly used as personal digital assistants. However, over the years not only wireless data networks have improved, but also the capabilities of mobile devices have enhanced considerably. So, it is possible to build collaborative applications to run on mobile device. Such applications rely on sharing and accessing information from stable common data repositories. For example, mobile users may share their appointment calendars, access and exchange emails, bibliographic databases, etc. The principal requirements of such applications are to allow mobile clients actively read and write shared data.

With continuous enhancement in capabilities of portable mobile devices and matching increases in data rate and bandwidth of wireless communication, it has become possible to think beyond applications targeting group collaboration just in organization of personal space. Transmission of moderate volume of data at reasonably lower latencies has made mobile computing a reality. Still intrinsic unreliability of wireless network have to be factored in some way or other to provide a stable data sharing and acessing environment so that mobile computing applications can run without requiring much of clever tweaking in design and implementations.

Three different mechanisms, have been explored to facilitate sharing data in mobile environment, namely,

1. distributed objects,
2. replicated storage system, and
3. file system.

Each mechanism is implemented at a different level of softaware stack.

Distributed object based sharing of data is realized through a tools known as Rover toolkit [?]. Rover is meant for building mobility aware applications. It relies on two main mechanisms, namely, relocatable dynamic objects (RDOs) and queued remote procedure call (QRPC). Sharing of data is in the form of units of RDOs. RDOs define a distributed object abstraction. Each RDO an object with a well-defined interface which can hop from one device to another. On the other other hand, QRPC supports non-blocking RPC, hence, allows disconnected operation.

For many of the non-real time collaborative applications such as calendar, email, program development, etc., strong consistency is not important. A replicated storage system which assures different types of weak consistency guarantees can be ideal environment for building such applications. Bayou [?] which was built by reasearchers at Xerox Park provides sharing through a replicated, weakly consistent storage system.

In order to support mobility, data sharing mechanism should be able to cope up with a range of network outages and characteristics. The variations could in many dimensions, from high bandwidth and constant connectivity to low bandwidth, ocassional and highly unreliable connectivity. From the users prospective, the shared data should support highly available, efficient and transparent access even as the users roam around. Some of the problems which arise in this context are:

5.2 Mobility Transparent File System

For many of the collaborative applications, mobile clients need improved data access, sharing and storage supports over potentially unreliable wireless networks which suffer from long latencies. Applications can be either mobility aware or mobility transparent. Mobility transparent applications need greater level of support which hide physical characteristics of wireless communication medium. On the other hand, mobility aware applications have ability to adapt to the dynamic changes in physical characteristics of wireless communication links.

In order to access remote data over a wired network, the applications running on servers or workstations can afford to rely on RPC based file operations such as `open`, `close`, `read` bytes, `write` bytes, etc. However, remote access model is not suitable for the applications running on mobile devices. Resource poor, low power devices which communicate over wireless networks seem to work best with caching/rsynch model. This model is built upon two basic operations, viz., download and upload as illustrated in Figure **??**. A read transfers the entire file to a local device. After read is successful, other operations on the file are performed on the local copy. A store operation causes the file to be uploaded to its remote location. A model for sharing based on caching and remote synchronization is particularly useful for scaling and performance enhancement Therefore, this approach is eminently suitable for mobile distributed systems.

Many issues arise in supporting file operations on mobile clients in a distributed file system. Apart from the inherent technological limitations of wireless networks such as long latency, low bandwidth, mobility of clients in a mobile distributed system introduce two important transparency issues, namely,

- Mobile clients move without informing reconfiguration information to the file system.
- Mobile clients experience temporary disconnections when passing through shadow regions of wireless networks.

So, transparent and efficient file operations on remote files at mobile clients will be possible if the file system design can accomplish twin objectives of managing the client cache for better performance and server replications for high availability.

A file system that can support location-transparent, replication independent operations on remote data is the most desirable solution for providing full range of data services to applications running on mobile devices. However, design difficulties and implementation related problems in building such a distributed file system are far too many. Furthermore, considering the class of applications which typically run on mobile devices, a file system perhaps appears to be an overkill.

Like Coda, Bayou relies on replica management to make the accessing of shared data highly available. But unlike Coda it is not as transparent. The application should be aware of the fact that it is operating on replicated data. There is a difference between mobility aware and mobility transparent applications. Mobility transparent applications are not aware of client's mobility. So, applications can be developed without any special requirements for adaptation. In the case of mobility aware applications, the extreme connectivity modes are handled at application level. In other words, mobility aware applications can initiate appropriate caching and consistency decisions to fully exploit the intermittent connectivity, so that application operate gracefully in periods of disconnection.

5.3 Handling of Disconnection

In a fixed network a disconnection is treated as a failure. However, such an approach in the case of mobile system will not be practical. Because, hosts in a mobile distributed systems often experience disconnection, and intermittent weak connections. Due to inherent characteristics of wireless communication, fluctuations in bandwidth do occur frequently. The problem is compounded further by resource poorness of mobile terminals. So, treating a disconnection as a failure will mean that a computation can never progress in a mobile distributed system. Therefore, to enable meaningful mobile applications to run, disconnection of various degrees should be handled differently from failures or crashes. Often a disconnection is elective in nature, this it allows a mobile terminal to plan ahead for an impending disconnection. Similarly, when a mobile enters the state of weak connection from the state of a strong connection and vice versa, its transitions can be handled by a weak disconnection protocol.

Preserving data consistency is the key to disconnection management. In disconnected mode, an application may continue to execute in stand-alone mode at a mobile host. The execution is made possible by pre-fetching (known as hoarding) requisite data in the mobile host before an impending disconnection. But the quality of data consistency that can be maintained with a strong connection will progressively deteriorate in disconnected mode as modifications and updates get executed by an application over the cached data. With program control it is possible to exercise control over the deviation of the cached copy (called a quasi copy) from the corresponding master copy. However, the degree of allowable deviations will depend on the application's level of tolerance. For example, in a portfolio management of stocks, it may be perfectly alright to work with a quasi copy dated by 24 hours.

The problem of cache consistency and its implications on applications is best understood by case studies. Although the quality requirements for maintaining data consistency will vary with applications, the issues typical to maintenance of data consistency are same across applications. We, therefore, focus on data consistency issues through operations on a distributed shared file system for mobile environment. Our motivation in choosing this case study is two fold, namely, to understand (i) how data consistency issue is handled in disconnected mode, (ii) how storage of data is handled in mobile environments.

5.4 Mobile Distributed File Systems

A Distributed File System (DFS) allows location-transparent sharing of data among workstations. A user can access the data from any workstation, but the accessed data is stored only at one or more computers known as servers. A DFS subsumes fast and reliable network connection and makes the remote files appear as local at the workstations connected to the network. The sharing of the files in a DFS is achieved by separating the roles of two sets of computers, namely,

1. The *clients* who access the files, and
2. The *servers* which store the latest copies of files.

A server receives the updates to files from the clients and supplies these according to requests of clients. The servers are computers with large storage space. A client normally does not have much storage. Therefore, the clients are inexpensive. A DFS consists of very few servers and a large number of clients. Each server can serve requests of many client concurrently. The sharing of a common data repository facilitates collaborative computing among the clients.

So, the fundamental goal of a DFS is to support efficient transparent access of a shared file system while maintaining data consistency. While the advantages are too many, there are also many problems in a DFS implementation. For example, the server may receive several interleaved requests for supply of data from a large number of clients. This puts heavy load on the servers and the performance at the server side suffers. The data is shipped back and forth, the servers supplying files and receiving updates from the clients, potentially exposing data to sniffers on the network. The network outages render clients completely useless. Similarly, a server crash cripples the clients. There are also serious semantic problem related to data consistencies as several clients working on the same file may provide conflicting updates to a server. In this case, a server has no mechanism to arbitrate on the updates to the same file from a number of clients and decide which ones to accept. The problem occurs when the updates are not available in real time.

But the biggest advantage of a DFS is location-transparent access to the file system. Therefore, by augmenting a DFS to handle problems like:

- A range of network disconnections,
- Scalability,
- Inconsistency of data, and

- Conflicting updates by clients,

it should be possible to design an efficient, mobility transparent DFS for mobile distributed systems (MDS). Due to inherent difficulties of maintaining connection in wireless networking a mobile client may experience range of network connectivity, from full disconnection to intermittent weak connection. Furthermore, due to the poor state of battery, on its own volition a mobile client may choose to disconnect from network. Scaling is an important issue in a MDS, because unlike conventional DFS (where computers are connected by wired network) it is not possible to control the number of clients attempting to access a shared file system. A problem closely related to scalability data availability. Though availability problem can be solved by replicating file system, replica maintenance is challenging. Network partitioning, delayed and conflicting updates from different clients may lead to problem of data consistency. Consistency problem also arises due to the fact that disconnected mobile clients update cached local copies of the files and attempt to upload conflicting updates of file copies to the servers. Applications can not work properly unless some consistency guarantee is available from the shared file system. However, if file system can guarantee even certain relaxed consistency, mobile applications may be engineered appropriately.

Summarizing the discussion on DFS for mobile environment, we find that the basic solution paradigm for augmenting DFS to mobile environment comprises of

1. Replication of data (cloning, copying and caching), and
2. Data collection (hoarding/pre-fetching) in anticipation of future use.

Therefore, much of the efforts in design of DFS that allows mobile transparent access to shared files, will be in maintaining replications and guaranteeing weak data consistency across the copies held by clients and the replicas at servers.

5.5 Coda

The major motivation in Coda file system was to build a DFS that is resilient to frequent network outages. It was a prototype implementation for highly available distributed file system for disconnected operation of clients. But the idea of putting a client-side cache manager coupled with the concept of hoarding actually turned out to be extremely well-suited for mobile computing. Coda project was conceived and implemented by Satyanarayanan and his group [1, 4, 5] at CMU and began around late 80's.

The disconnected mode operation supported by Coda file system is actually an initial step towards sharing and collaborative computation in a mobile computing environment. Two other storage systems Bayou [7], and Rover [3] were announced subsequently to mainly support client mobility and weak network connectivity. Both employ the similar strategies, namely, replications, caching, and weak data consistency. Bayou's focus was on supporting application specific mechanism to detect conflicts and resolve them. Therefore, the approach to tackle disconnection is application-aware. On the other hand, Rover supports building of both mobility-transparent applications and mobility-aware applications. Coda's initial focus was fully application-transparent mobility. Our discussion is mainly centered around Coda file system as it seems to be a precursor of almost all storage systems supporting client mobility and collaborative computing in presence of weak or no network connectivity.

5.5.1 Overview of Coda

Coda provides a common name space for all the files that clients share. Typically Coda running on a client shows a file system of type `coda` mounted under `/coda`. All files provided by any of the servers are available to a client under the Coda directory. In contrast the NFS file systems are available on per server basis with additional requirement that the file system should be explicitly exported to a client. In order to access NFS file systems a client has to have mount commands on per server basis e.g., mounting entries can be specified in `/etc/fstab` file. Each file system should be exported by the corresponding server. However, for accessing a file under Coda file system, a client simply need to know the mount point of only `coda`. Coda system running in a client will fetch the required file from the appropriate servers and make it available to the client.

A full path name for a file in Coda may span over several mount points across servers. Each mount point would correspond to a partial subtree of the file system in a Coda server. Files are grouped into collections known as volumes at Coda server end. Usually, the files of a volume are associated with a single user. A mount point at a client's end is the place holder where a server volume can be mounted. Figure **??** shows that a Coda file system at a

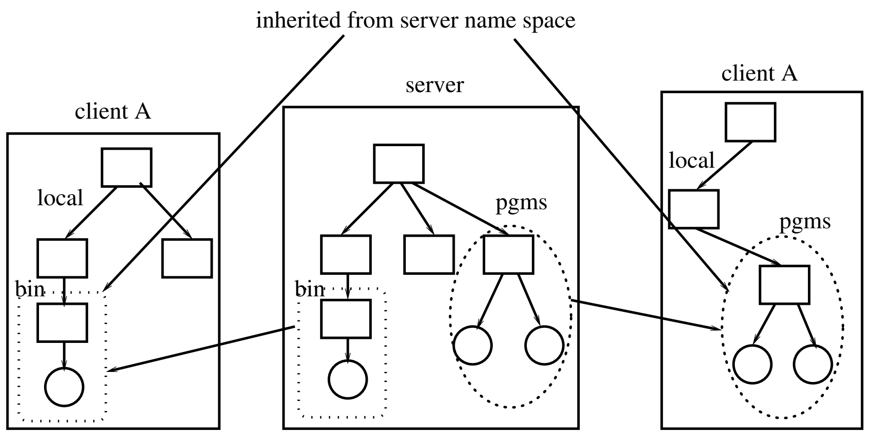
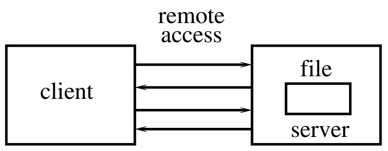
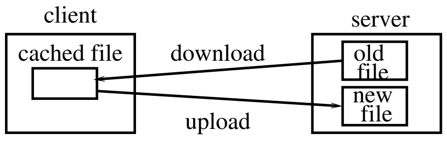


Figure 5.1: Coda name space in clients.



(a) RPC based access.



(b) Upload/download access.

Figure 5.2: Remote access of shared files system.

client is composed of files from two servers. On the server side a volume can be viewed as a disk partition in Linux and corresponds to a partial subtree of the file system. So, the Coda name space at client can be constructed by mounting volumes on mount points. A leaf node of a volume may be the root of another volume. Thus, an entire name space may span across different volumes located in one or different servers.

There are two way to operate on files in shared file system by the contending clients. In the approach followed by NFS, the copy of the file is held at the server, and the clients operate on files through RPC as indicated by Figure 5.2(a). Obviously, this approach is not suitable for scaling. The alternative approach, illustrated by Figure 5.2(b), shows that a client downloads a local copy and operates on local copy. The old copy is held by the server till the time the client commits the updates by uploading the new file. RPC is basically a request-response based mechanism. The latency of response may

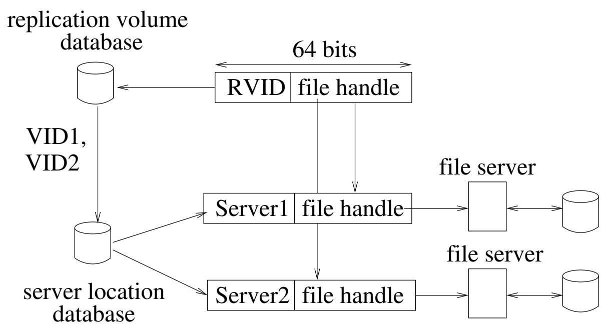


Figure 5.3: Fetching file in Coda

depend on several factors including load at server for processing requests, network latency or failures. So, the first approach does not scale up. Coda uses the second approach. The problem with this approach is that many clients which have copies of the same file may report conflicting updates. The conflicts must be resolved at the server. However, Coda relies on Unix semantics for operation on files. Typically, in a Unix file system only one user has write access while those sharing may have read accesses. In case write access rights are held by many, the applications using the shared files should be designed to handle the updates by any client. When some updates to a file in Coda file system is made, the server takes responsibilities to inform its clients.

A file in Coda can be accessed from one of the many files servers which hold a copy. A file handle consists of two parts, Replication Volume ID (RVID) and a file handle. The volume replication database indexed by RVID provides the Volume IDs (VIDs) in which replicas of the required file is available. A query to volume location database with VIDs provide location of file servers which store the replicas of the file. File IDentifier (FID) is constructed by prepending the Server ID to the file handle. The process of fetching file as described above is illustrated by figure 5.3.

The interactions of various Coda components are depicted by figure **??**.

Any operation that requires access of objects in shared file system in Coda is in granularity of files. Suppose a user wants to display the contents of a file in `/coda` directory, e.g., by issuing a `cat`. The user may be connected to network or operating in disconnected mode. The `cat` command generates a sequence of system calls like `open`, `read` and then `write` on display, etc., in

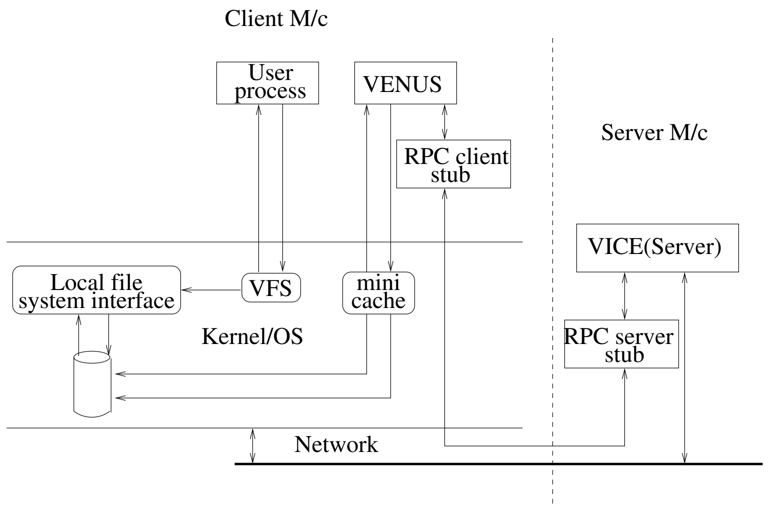
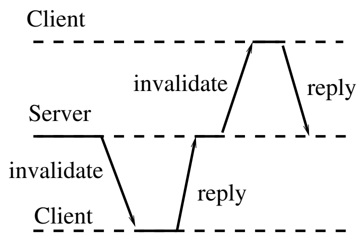


Figure 5.4: Interaction of Coda components

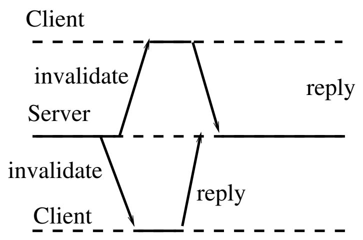
respect of required files. All system calls are executed in *kernel* mode. The kernel at a client requires the services of the Virtual File System (VFS) to open a file before it can service `cat`. VFS recognizes that the file argument to be a Coda resource and informs the kernel module for Coda file system. Coda module works basically as an in-kernel mini-cache. It keeps a cache of recently answered requests for Coda files from VFS. If the file is locally available as a container file under the cache area within the local Coda file system, then VFS can service the call. In case the file is not locally available the mini-cache passes the request for the Coda file to a user level program called *Venus* running at the client. Venus checks the local disk, and in case of a cache miss contacts the server to locate the file. When the file has been located it responds to kernel (mini-cache) which in turn services the request of the client program.

Communication in Coda

Coda uses RPC2 which is a reliable variant of RPC with side-effects which supports multiple RPCs in parallel. Side effects provide hooks for pre and post RPC processing. The hooks are not part of RPC2 but are active parameters. RPC-2 starts a new thread for each request. The server periodically keeps the client informed that its request is being serviced. As side effects,



(a) Sequential invalidation.



(b) Invalidation by broadcast.

Figure 5.5: Invalidation notification using RPC and RPC2.

application specific protocols are allowed for asynchronous transfer of files between the client and the server or streaming of video to the client. RPC2 also has multicast support. Figure **??** illustrates the RPC2 mechanism and communication between a Coda server and a client.

RPC2 allows clients to register callbacks with servers, and when a break in callback promise occurs, the server multicasts invalidation notifications to the connected clients. On getting invalidation notification, the clients become aware of updates in cached files and may choose to fetch the updates which allow them to maintain data consistency as needed by the applications. The advantage of RPC2's multicast based notification over RPC's sequential notification protocol is depicted in Figure 5.5. In multicast based notification the server does not worry if an acknowledgement from a client does not arrive. This is in contrast to the case where each invalidation message is sent in sequence (as in RPC) and the server waits for the reply from a client before processing the next invalidation message.

5.5.2 Design of Coda

As mentioned earlier, a client views Coda as a single location independent Unix like file system. At a file server level, Coda is mapped to a granularity of subtrees. A volume, which is a set of files and directories located at one server, forms a partial subtree of the shared name space. The volume mappings are cached by Venus at the individual clients. Coda uses replications and callbacks to maintain high availability and data consistencies. The set of sites where replicas of a file system volume are placed called *Volume Storage Group* (VSG). A subset of a VSG accessible to a client, which is referred to the client's *Accessible* VSG (AVSG).

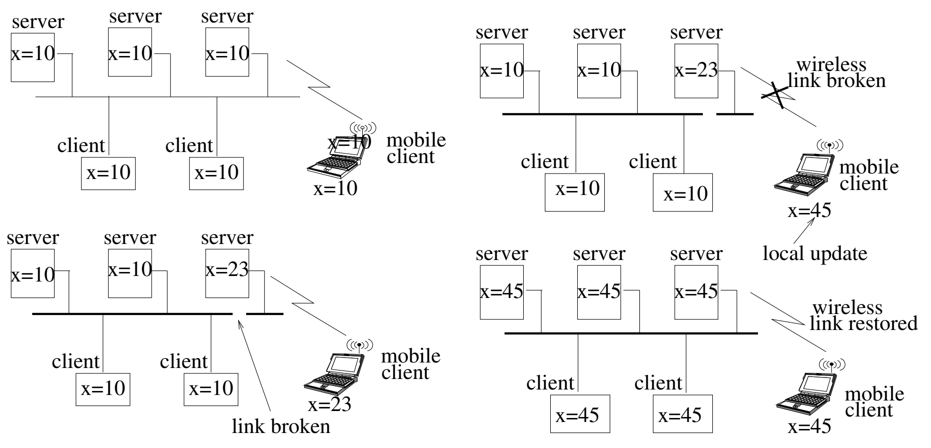
The overhead for maintaining replications at client is controlled by call-back based cache coherence protocol. Basically, Coda distinguishes between the replicas cached at clients as *second class* and those at the server as *first class*. If there is any change in a first class replica then the server sends an invalidation notification to all the connected clients having a cached copy. The notification is referred to as a *callback break*, because it is equivalent to breaking a promise that the file copy held by the client is no longer correct. On receiving the notification the client can re-fetch the updated copy from the server. The updates in Coda are propagated to the servers to the AVSGs and then on to missing VSGs (those which are not accessible to the client).

The client operates in disconnected mode when there is no network connection to any site or server in the fixed network. Equivalently, it implies that the AVSG for the client is a null set. In order to make the file system highly available, the clients should be allowed to operate with the cached copies even when the AVSG is empty. Under this circumstance, Venus (cache manager at client end) services the file system calls by relying solely on its cache. When a client is operating in disconnected mode and the file being accessed is not available in cache, a cache miss occurs. The cache misses cannot be masked or serviced. So, they appear as failures to the application program. The updates made by a client on the cached copies are propagated to the server when the client is re-connected. After propagation of the updates, Venus re-charges cache with the server replication.

Figure 5.6 illustrate the transitions between cache and replication at the clients in a typical Coda environment. Figure 5.6(a) shows that all the clients including the mobile client are connected to network. Therefore, the value of some object, say x, is identical across the copies held by the servers and the clients when all are connected. The same holds in a network partition though it does not hold across the partitions. Figure 5.6(b) indicates that a client operating in disconnected mode can update the value of the object in its cache, but this update will not be visible to server or to other clients even inside the same network partition. But when the disconnected client reconnects, the value of the object in it or at other nodes becomes same.

The high level design goals of Coda file system are:

- To design a highly available and scalable file system,
- To tackle a range of failures from disconnection to weak connection,



(a) Connected client. (b) Disconnected client.

Figure 5.6: Transition between cache and replication at client.

- To provide a location-transparent Unix-like single shared name space for the files.

5.5.3 Scalability

A small set of server serve as a storage repository for all shared data. A large number of untrusted clients access and share the data. The requirement of scalability in this context is ensure that the ratio of the clients to the servers should be as large as possible. This necessitates the transfer of most of the functionalities to clients. Therefore, adding more number of clients does not increase the load on the servers. The OS running on each client can filter out all the system calls that do not need interaction with the servers. Only `open` and `close` are forwarded to cache manager Venus running on the client. After a file is opened `read`, `write`, `seek` can bypass Venus. Since the servers are not unduly loaded to support the clients functionalities, and they are trusted, they can bear all the responsibilities for preserving the integrity and the security of data.

The most obvious approach to increase scalability in the use of shared data by large number clients is by employing caching at the clients. But maintaining the consistency of the replica cache at clients as the corresponding server replicas receive updates, has to be resolved cleverly. The updates are usually generated at the client side. So, the problem is much more complex than just finding a mechanisms to maintain consistency of client-side cache. Specially, since the clients may also use same replicas simultaneously and without each other's knowledge, there will be potential conflicts in client-generated updates.

Another approach of Coda is to disallow any rapid system-wide change. So, the additional update overheads like voting, quorum strategies, etc., which involve interactions of a number of servers and/or clients are eliminated completely. The approach dispenses efficiency in preference to consensus! In summary, the important Coda attributes that attempts maximize scalability and yet preserve the data integrity are as follows:

- The Coda uses the model of untrusted clients and trusted servers.
- The clients cache the data in granularity of entire files.
- Cache coherence is maintained by callbacks.

Once a file is opened successfully, it is assured to be available in local cache. The subsequent operations on the cached files at the client are protected from network failures. The idea of caching file in whole also compatible with disconnected mode of operation by the clients. A client is required to register a callback with the server in order to get notifications of about the updates on the cached files. This approach prevents system-wide propagation of changes. Another advantage of callback based cache coherence protocol is that files can be easily moved between servers, all that clients are required to know is a map of files to servers and cache this dynamically changing map.

5.5.4 Disconnection and failures

The main idea is to make the client as much autonomous as possible. So, the client becomes resilient to a range failures. If every client is made completely autonomous then there is no sharing at all. This is a degenerate case of personal computers operating in isolation from one another. Therefore, the goal in Coda is two-fold, namely, (i) using shared storage repository, and (ii) continuance of client operation even in case of network failures.

Caching can provide limited autonomy to clients with sharing. Coda stipulates caching in granularities of *whole* files. No partial caching is allowed. This approach simplifies the failure model and makes disconnected operation transparent to the applications running in a client. A cache miss can occur only at `open`. There will be no cache miss for `read`, `write`, `seek` or `close`. So, an application running on a client operating in disconnected mode will be unaware of disconnection when the required file replicas are locally cached. Coda uses another additional technique called *hoarding* to pre-cache the application specific files from the server in anticipation of a future disconnection. Hoarding is essentially a large grain prefetching and discussed separately.

Server replication is one more strategy employed by Coda to increase availability of shared data. Copies of same file are stored in multiple servers. It provides greater level of availability of shared storage repository. When a client is connected to at least one server it depends on the server replication. Since many server replications are permitted the period of disconnection can be minimized. The moment client gets disconnected it can switch to cached copy of shared data. While replication strategy improves the availability of data it introduces the difficult problem of preserving data consistency across network.

5.5.5 Replica control strategy

As stated earlier clients are portable and untrusted. A client has a small storage, data can be willfully tampered or corrupted. It is unrealistic to assume a user will backup the data. Thus, the integrity of data in the replication held by a client cannot be trusted. In contrast the server have big capacities, more secure and carefully monitored by professional administrative staff. This naturally provides the guarantee of quality of data at the servers. Therefore, it is appropriate to distinguish between two types of replications. The replicas at servers represent the *first class* replication. The client replicas represent the *second class* replication. The first class replicas are of higher quality compared to that of the second class replicas. The utility of second class replicas is limited to the clients where they are located. Furthermore, a second class replica is not available system-wide. However, since the correctness of execution of a client application depend second class replicas, these replicas must be periodically synchronized with the corresponding first class replicas. The basic purpose of the second class replicas is to make the shared data highly available to clients in presence of frequent, as well as prolonged network failures.

Figure 5.6 shows that when a client operates in disconnected mode there is actually a network partition between the second class replication held by the client and the all the corresponding first class replications. The replica control strategy may be either

- *pessimistic*, or
- *optimistic*.

In pessimistic control strategy update is allowed only by one node of a unique partition. So, prior to switching into disconnected mode, a client acquires exclusive lock on the cached object and relinquishes the lock only after re-connection. Acquiring exclusive lock disallows read or write at any other replica. Such an approach is not only too restrictive, but also infeasible in context of mobile clients. The reasons can be summarized as follows.

- A client can negotiate for exclusive lock only in case of a planned dis-connection.
- Duration of disconnection is unpredictable.
- Reconnection of a client holding an exclusive lock can not be forced.

In case of a involuntary disconnection there is no opportunity for a client negotiate an exclusive lock for any shared object. Of course, one possible approach may be to play safe and allow lock to a client. However, several clients sharing the same set of files may want to acquire locks and many of these get disconnected at the same time. In a situation like this the system has to come up with sophisticated arbitration mechanism for allowing exclusive lock to one of the contending clients. The client which finally gets the lock will not even know that as it could be disconnected by the time arbitration process completes. Possibly exclusive lock on some objects only for a brief period of disconnection is acceptable. But if the period of disconnection is long then the retention of exclusive lock is detrimental to the availability of shared of data. It is also not possible to force a re-connection of the client holding the exclusive lock. Whereabouts of the client may not be known, and quite likely the client may not even be present in the cell. The whole set of client population may potentially be under the mercy of single lock holder operating in disconnection mode. A timeout associated with exclusive lock can partly solve the problem of availability. The lock expires after a timeout. So other client can take turns to have lock access to the objects. But once the *lease lock* expires the disconnected client loses the control of the lock. Any other pending request lock can be granted. However, the concept of *lease lock* contrary to the purpose of exclusive lock and making file system highly available due to following reasons.

- Disconnected client is in midst of update when it loses the lease of the lock.

- Other client may not interested for the object when lease on lock expires.

If a disconnected client loses lock control, all updates made by it have to be discarded. One simple idea that may alleviate the problem is the lazy release locks. A client continues to hold the locks even after the expiry of the lease unless some other client explicitly places a request at the server. But then the exact timing of the lock release can never be synchronized to avoid overlap with update activity at the client.

Optimistic replica control mechanism does not allows any exclusive lock. The clients update their cached objects with the hope its update will not conflict with updates by other clients. It means several clients operating with same set of cached objects in disconnected mode can report conflicting updates on reconnection. Therefore, it requires the system to be sophisticated enough to detect and resolve the update conflicts. But making the conflict resolution fully automatic is extremely difficult and not possible. However, it is possible to localize the conflicts and preserve evidences for a manual repair in extreme cases.

The objective of choosing an optimistic strategy is to provide a uniform model for every client when it operates in disconnected mode. It allows concurrency as a number of clients can use a replica of the same object simultaneously. The argument in favour of optimistic replica control also comes from the fact that Coda's cached objects are in granularity of files. In an Unix like shared file system, usually the write access is controlled by only one user while other users typically provided with read accesses. So, a client having write access can update the cached file without any creating update conflicts. Only the clients having read access may possibly get slightly outdated values. But then applications can be tuned to handle such eventualities. In the case of many applications it will not even matter if the data is bit outdated.

Another approach may have been to employ optimistic control strategy for disconnected mode and pessimistic strategy in connected mode. While such an approach is possible, it will require the user to be adaptive to the two anomalous situations depending on whether operating in disconnected or connected mode. While a user would be able to do updates in disconnected mode s/he would be unable to do so in connected state as some other client has obtained an exclusive/shared lock prior to the request of this user. Therefore, a uniform model of replica control is preferred.

5.5.6 Visibility of updates

The design of Coda was initially geared to provide user a fail-safe, highly available shared file system like Unix. Under Unix semantics any modification to a file becomes immediately visible to all users who access it. Obviously, preserving this semantics is not only incompatible for scalability but also impractical when users are allowed to operate in disconnected mode. So, Coda implementors settled for a diluted form of the visibility requirement under Unix file operation semantics. The difference between a cached copy of file available at a disconnected client from its latest copy depends on the difference of time of the last update of cached copy and the current time. Hence, a time stamp (`store id`) on replica may be taken as parameter for the distance function that measures the difference between cached replica and its server state. Since Coda allows data sharing at the granularity of file revalidating cached file at the time of opening, and propagating updates at the time closing file should be adequate for most applications. A successful `open` means the contents of the file being accessed is the latest replication available with the server from where the client fetched it. It is possible that the server itself may have a stale copy. Therefore, the revalidation of cached copy at the client at the time of opening a file actually means it is synchronized with the latest replica available with the server site. It may be possible to increase the currency guarantee by querying the replica custodians on the fixed network when the file opening is attempted at a client. The query can be initiated by the server on which Venus dockets the open call.

5.5.7 Currency guarantee

Coda provides a weaker currency guarantee with respect to updates. This because the cache revalidation is callback-based. By establishing a callback with a server, the client gets a notification of cache invalidation when server replica is updated. The event of notification is referred to as breaking a callback promise by the server. The callback has to be re-established with the server by updating the cached object to the server's copy. But the detection of callback break and a subsequent re-establishment callback may be possible only when network connection is alive. If a callback break occurs during a network failure then the notification cannot be delivered to the client. Such an event is termed as a lost callback. Because of a lost callback the client may continue to work on an outdated cached copy till time another callback break occurs due to an updates by another client. The callback registration is allowed on one server. Therefore, currency guarantee can be associated with a time parameter, that is the cached copy at client is same as the server copy at sometime in the past. A client will not receive any notification if the preferred server, on which the client registered callback, is down. Once a call back notification is sent the server is freed from obligation to send another notification unless clients registers a new callback. So the client has to fetch a fresh copy from the from the server if it is needed for the applications.

5.5.8 Venus and its Operations

The main focus is on the ability of the clients to work in disconnected mode or in the face of various network failures. So, we first discuss the client side support needed for Coda. Then we describe the server side support.

Figure 5.4 gives a high-level decomposition of Coda components support at a client. Venus is implemented as a user-level program. This the approach made it portable, and easy to debug. The Coda mini-cache which is a part of client's kernel supports only for servicing system calls for the file objects in Coda file system if it is already cached and open has been successful. Kernel uniformly forwards file system calls – whether it is for a Coda resource or a local resource – to VFS. The process of servicing file system calls is as follows.

1. If the call is for a Coda file, VFS forwards that call to mini-cache control.
2. Which in turn determines if the file is already cached, and services the request.
3. In case it is not in cache then the call is forwarded to Venus.

4. Venus actions are as follows:
   (a) Contacts the servers, fetches the required object from one of them.
   (b) Caches the file in local disk and returns to kernel via mini-cache.
   (c) Mini-cache can remember some recently serviced calls and caches these.

The state of mini-cache changes after Venus reports successful fetching and caching of objects from the custodian server, occurrences of callback breaks, and revalidation of client cache afterwards. Since Venus is a user-level program, mini-cache is critical to good performance.

The major task of Venus is fetching objects from accessible servers holding the replicas of the requested file objects. But it has to perform another complicated task, namely, propagating updates to the servers. It operates in three states

1. emulation
2. hoarding
3. reintegration

The states and transitions between Venus states have been depicted in figure 5.7. The states are represented by labeled circles. The transitions from

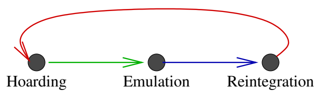


Figure 5.7: Venus state transitions

emulation to reintegration represents transition from disconnected to connected state, and transition from hoarding to emulation represents the reverse. So, hoarding and reintegration both are connected states while emulation is the disconnected state.

Hoarding

Hoarding is merely a pre-fetching in large granules. The success of pre-fetching depends on the ability to capture locality with respect to immediate future. In a mobile client a session of disconnected operation can continue for few minutes to hours or even for days. Therefore, it impossible to implement a hoarding strategy which will allow a client to operate in disconnected mode without failures. It has to take into account following factors:

- Mobile user's file reference behaviour in near to distant future.
- Involuntary disconnection.
- The cost of cache miss during disconnection.
- The physical limitation of cache.

The decision on hoarding becomes easy if a user specifies a profile of references. Disconnection and reconnection are purely random in time. Therefore, involuntary disconnections are hard to handle. However, as a preparatory to voluntary disconnection, it may be possible to hoard the data needed for disconnected operation. Still it is hard to estimate the cost of a cache miss. So, the decision to cache certain objects and the decision not cache certain other objects becomes hard. The combined decision should have twin objectives:

1. Minimizes the cost of not caching certain objects.
2. Maximizes the benefit in cost of caching certain objects.

Since, the caching decision will depend on the criticality of applications running on the client, the cost model should be defined appropriately. Based on the cost model, an object's importance for caching can be represented by a priority factor. Then a hoarding algorithm can perform a hoard walk according to the priority of the object. It is also convenient to define the cache equilibrium based. The cache equilibrium signifies that the mobile user's data requirements are mostly met from the cached objects. The cache size limitation disturbs the equilibrium as a result of normal activities. Some cache objects may have to make way for other objects, also some objects may be used more often than others at different instance of time. Therefore, Venus needs to periodically perform a hoard walk for restoring cache equilibrium. Coda's implementation fixes the interval of hoard walk as ten minutes.

However, Coda also allows users to specify the inclusion of a provision for triggering hoard walk at the time of voluntary disconnection in the hoard profile.

Emulation.

In emulation state Venus assumes the role of the proxy for a server at the client. It performs following important functions.

- Checks the access control.
- Generates temporary file identifiers (*tfid*s) for newly created objects.
- Evaluates priorities of mutating objects and applies these to cache these objects.
- Reports or blocks a failures when a cache miss occurs.
- Logs information to let the update activities during disconnected operation to be *replayed* on reintegration.

The tfid of newly created object is replaced by a permanent file identifiers (pfid) at the time of reintegration. The removal of a mutated object incurs more cost compared to that of a non-mutated object. Therefore, the hoarding priorities of updated objects should be very high. None of the deleted objects need be cached. So the lowest priority is assigned to these objects. A cache miss can not be masked unless the user has explicitly requested for masking it until reconnection. So, the default behavior of Venus is to report a failure on a cache miss.

The log is kept as a per volume log. A log entry consist of a copy of the corresponding system call, its arguments as well as the version state of the all objects referenced by the system call.

Replay log and its optimization

All the operations during emulation state are logged. The objective of logging is that the operations can be replayed at the server during the reintegration. Only updates are logged. Therefore, it is also alternatively referred to as Change Modify Log (CML). The length of the replay log can be reduced by applying a number of optimizations. Optimizing CML is critical for mobile environment as transfer of log requires both bandwidth and time. Any operation which is followed by its inverse can be purged from CML. For example, if a `mkdir` operation is followed by a `rmdir` operation with same argument then both operations need not be logged. Though CML typically occupies a few percent of the cache size, considering the criticality of cache during emulation state, it is desirable to apply optimizations to reduce the size of CML as much as possible.

At times a certain sequence of operations may be interpreted to mean a single logical operation. In that case, it may be possible to record just one equivalent logical operation in the log in place of sequence of a number of operations. For example, a `close` after an `open` installs a completely new file. So, instead of individually logging an `open`, the sequence of intervening `write`s, and finally, a `close`, all these can be effectively represented by a single `store` operation for a file. Note that logically `store` does not mean entire contents of the file is logged. It merely points to the cached copy of the file. Therefore, a `store` invalidates preceding `store`s if any. It makes sense only to record the latest `store`.

Recoverable virtual memory

In emulation state Venus must provide persistence guarantees for the following

- A disconnected client to restart after a shutdown and continue from where it left off.
- In the event of a crash during disconnected state, the amount of data loss should not be more than that if an identical crash occurred in a connected state.

To ensure persistence of the kind mentioned above, Venus keeps the cache and other important data structures in non-volatile storage at the client.

The important data structures or the *meta-data* used by Venus consists of

- Cached directories and symbolic link contents,
- Status blocks of cached objects of all types,
- Replay logs, and

- Hoard database (*hoard profile*) for the client.

All the meta data are mapped into the address space of Venus as the *recoverable virtual memory* (RVM). But the contents cached files are stored as ordinary Unix files on client's local disk. Only transactional access is allowed to meta data. The transactional access ensures that the transition of meta data is always from one consistent state to another. RVM hides the recovery associated complications from Venus. RVM supports local non-nested transactions. An application may choose to reduce commit latency by labelling *commits* as *noflush*. But the commits still have to be flushed periodically for providing *bounded persistence* guarantee. Venus can exploit the capabilities of RVM to provide good performance with an assured level of persistence. For example, during hoarding (non emulating state) the log flushes are done infrequently, as a copy is available on the server. However, during emulation the server is not accessible. Therefore, Venus can resort to frequent flushes to ensure minimum amount of data loss.

Resource exhaustion

Since non-volatile storage has a physical limit, Venus may encounter storage problem during emulation state. Two possible scenarios where the resource exhaustion may have an impact on performance are:

1. Disk space allocated for file cache is full.

2. Disk space allocated for RVM full.

When Venus encounters file cache full problem, possibly storage can be reclaimed by either truncating or deleting some of the cached files. However, when Venus runs out of space for RVM nothing much can be done. The only way to alleviate this problem will be to block all mutating operations until reintegration takes place. However, non mututaing operations can still be allowed. One uniform alternative to tackle the problem of space could be to apply compressions on file caches and RVM contents. A second alternative could be to allow the user to undo updates selectively and release storage. The third viable approach could be free storage by backing up the parts of file cache and RVM on a removable disks.

5.5.9 Reintegration

Reintegration is an intermediate state of Venus between the time it switches form the role of *proxy server* to *cache manager* at the client. All the updates made by clients are propagated to the server during this state. After that, the cache at a client is synchronized with the server's state. Reintegration is performed on one volume at a time in a transactional step. All update activities in the volume remain suspended until the completion of reintegration. The actual task of reintegration is performed by replaying the update log stored by Venus.

Replay algorithm

The Coda implementation views the reintegration activity as execution of replay algorithm at a volume server. For reintegration Venus switches its role from pseudo-server to cache manager for the client. The updates made during emulation state is propagated to the server. Reintegration is done per volume by replaying the logged operations directly on the server. The execution of replay algorithm consist of four phases:

1. Log parsing,
2. Validity check and execution,
3. Back-fetching, and
4. Lock release.

In phase 1 log is parsed, and operations with required arguments are identified. Phase 2 consists of various types of checks and validations. These include integrity, protection and disk space checks. Following the checks, conflict detection is carried out; and all the operations in the replay log are performed at the server in a single transaction. The conflict detection is a bit complicated process. In order to determine update conflicts, the store IDs of objects in replay log are compared with store IDs of the corresponding objects at the server. If the store IDs match then the operations in the log are performed locally on server. If store IDs do not match then the actions depend on the operations being validated. In the case of a file, the transaction for reintegration is aborted. However, in case of a directory object, conflicts occur

- If the name of newly created object collides with an existing name,
- If the object updated at the client has been deleted at the server or vice versa,
- If directory attributes at the client and the server has been modified.

After determining conflicts all operation except `store` are executed at the server. For `store` operation a shadow file is created for the actual file. Then in phase 3 backfetching is excuted. Backfetching step transfers the updated files from the client to the server and the fetched files replace the corresponding shadow files created in phase 2. Finally phase 4 commits the transaction and releases all the locks.

Bibliography

[1] Braam, P. The coda distributed file system. *Linux Journal*, 50 (June 1998), 46–51.

[2] Joseph, A. D., deLespinasse, A. F., Tauber, J. A., Gifford, D. K., and Kaashoek, M. F. Rover: A toolkit for mobile information access. In *Proceedings of 15th ACM Symposium on Operating Systems Principles* (1995), pp. 156–171.

[3] Joseph, A. D., Tauber, J. A., and Kaashoek, M. F. Mobile computing with rover toolkit. *IEEE Transaction on Computers 46*, 3 (1997), 337–352.

[4] Kistler, J. J., and Satyanarayan, M. Disconnected operation in coda file system. *ACM Transaction on Computer Systems 10*, 1 (1992), 3–25.

[5] Mummert, L., Ebling, M., and Satyanarayan, M. Exploiting weak connectivity for mobile access. In *Proceedings of 15th ACM Symposium on Operating Systems* (1995), pp. 143–155.

[6] Terry, D. B., Demers, A. J., Peterson, K., Spreitzer, M. J., Theimer, M. M., and Welch, B. B. Session guarantees for weakly consistent replicated storage data. In *Proceedings of IEEE Conference on Parallel and Distributed Information Systems (PDIS)* (1994), pp. 140–149.

[7] Terry, D. B., Theimer, M. M., Peterson, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of 15th ACM Symposium on Operating Systems Principles* (1995), pp. 173–183.