NFS/M: An Open Platform Mobile File System

John C.S. Lui* Oldfield K.Y. So T.S. Tam
Department of Computer Science & Engineering
The Chinese University of Hong Kong
Hong Kong

Abstract

With the advancement of wireless network and mobile computing, there is an increasing need to build a mobile file system that can perform efficiently and correctly for accessing online information. Previous system research on mobile file system is based on some experimental platforms. In this paper, we describe the design and implementation of a mobile file system on an open platform, the Linux kernel, and at the same time, our mobile file system is compatible to the popular NFS 2.0 protocol. In this paper, we formally define the file semantics of our mobile file system, which we called the NFS/M. We also specify the conditions of object conflict as well as our conflict resolution algorithms. NFS/M supports client side caching, data prefetching, file system service during the disconnected mode, data reintegration and conflict resolution on various file system objects. Since the NFS/M is based on an open platform, it serves as a basic building block for developing future mobile computing applications.

1 Introduction

Recently, the proliferation of portable computers and wireless networking technologies provides the necessary baseline infrastructure for performing *mobile computing*. There has been a lot of attention lately on this type of computing paradigm [2]. In mobile computing, users can communicate with each other and retrieve online information without the constraint of having to work at a fixed location. To realize the benefit of this appealing computing paradigm, there are still many technical challenges [1, 2, 3, 4, 5, 6].

One key requirement of the mobile computing is to enable the mobile users to access information regardless of the location of the user and the state of the communication channel or the data server. Although the conventional distributed file systems, such as the NFS and AFS [8], can provide data access in remote areas, they are not suitable for the mobile computing applications. The reason is that they were designed with the assumption that the communication network is *fast* and *reliable*. Under the mobile computing environment, user can traverse from one place to another so that the cost and the quality of the network, like the communication bandwidth, may vary significantly. Worse yet, network connection may not even available in some areas. Under these situations, any access to the data server may cause the mobile computer to be in an inoperable state and thereby causing the mobile users the incapability of using their local computing resource even if it is operational. The mobile file system must be able to cope with these situations.

*This research is supported in part by the UGC & CUHK Research Grants.

1.1 Related Work

Research work on accessing data from the remote file server and data availability can be traced back to the period of designing distributed file systems. Both the theoretical and the architectural aspect can be found in [13, 14]. Davidson S. B. [17] provided a comprehensive discussion on the tradeoff between data availability and correctness. Based on the optimistic replicate control protocol [15], experimental mobile file system like the Coda [16] and the Ficus [11, 12] were built.

Both the Coda and the Ficus allow access of data during the disconnection period and thereby increasing the data availability upon server or communication network failure. The Coda focuses on long-term disconnection which happens more often in the mobile computing environment while the Ficus focuses on a large scale distributed system in which rapid local access and high data availability are its main concern. Coda supports data reintegration, which is the process to propagate any updates during the disconnection period and integrate the data from client-side back to the server upon the communication channel reestablisment. Note that in Coda, it retains certain read-from relations [9] between file system objects. Based on its transactional file system model, data reintegration is made possible only if all the updates satisfy the one-copy view serializability condition [9]. If the updates cannot satisfy the condition, the whole update transaction in the client side is rejected to avoid data inconsistency. Ficus, on the other hand, uses reconciliation process to synchronize each volume replica. Conflicts are defined on a file-by-file basis and only individual file update conflict is considered. If there is any conflict, different conflict resolution algorithms are applied based on the semantics of the file so that updates from two different network partitions can be merged to a single copy. In general, the Coda and the Ficus were designed based on two different file models. Coda was developed based on the client-server model with the notion of *stateful* server, where certain system information, like call-back promises are kept on the server side. Instead, the Ficus was designed based on the peer to peer model in which there is no clear distinction between the role of different machines.

Although Coda and Ficus are appropriate for mobile computing environment, however, the most common file system nowadays is the NFS [7]. NFS was designed based on the principle of *stateless* server. So, it is difficult to incorporate the Coda or the Ficus onto the NFS platform. Our goal is to design a file system that can work seamlessly with the existing NFS and appropriate for mobile computing applications as well. Our mobile file system,

NFS/M, uses a different approach, as compare to Coda and Ficus system, in resolving partitioned updates during the disconnected mode.

1.2 Our contributions

The contributions of our work[1] are as follow. 1) Our system is implemented based on an open UNIX platform and is based on the industrial standard NFS 2.0 [7]. Therefore, it can work seamlessly with any existing NFS server without server reconfiguration. 2) We have proposed and implemented an efficient client-side module which supports data prefetching, data replacement, file system services emulation during the disconnection period and data conflict detection and resolution during the reintegration period. 3) Our system allows a high degree of concurrency when the network is partitioned, and the system tries to preserve all the updated data upon the communication channel reestablishment. 4) Our system ensures the concept of *access transparency* such that application programming interface will be identical to existing NFS. 5) Our system provides *failure transparency* in the sense that when the NFS server fails or network connection is not available, NFS/M clients can continue their work without interruption. When everything return to normal, NFS/M will propagate any updates during the disconnection period and will try to merge any updates from different network partitions.

The organization of our paper is as follows: in section 2.1, we describe the system architecture, phase transitions for various states of the network connectivity and the NFS/M file semantics. In Section 3, we formally define how data conflict can be detected and resolved. In Section 5, we describe the various modules in the NFS/M and their functionalities under different phases. Conclusions and future work are given in Section 6.

2 System Architecture, Phases and File Semantics

In this section, we describe the architecture in which we develop the NFS/M. We also explain the phase transition structure of the NFS/M such that the system can keep track of the states of the connectivity with the NFS server. Lastly, we describe the NFS/M file semantics.

2.1 Architecture

In our design, the NFS/M clients can share the same NFS server with other traditional NFS clients in a cluster of heterogeneous computer systems. The advantage of our system is that it is not necessary to reconfigure the NFS server so as to accommodate the NFS/M. Different PCs/workstations running different operating systems can still connect to the same NFS server for accessing data. This scenario is illustrated in Figure 1.

Within a system which has the NFS/M support, there is a *NFS/M client*, a daemon called *middle*, which is the cache manager as well as the proxy server, a *reintegrator* and a *data prefetcher*. The architecture is illustrated in Figure 2.

For our software platform, we have developed the NFS/M on a Linux 2.0.29 kernel. Since the modification is only made on the client computer which is running Linux

[1]Note that the NFS/M package for the Linux 2.0.29 can be downloaded from the URL: http://eagle.cse.cuhk.edu.hk/yprj2277.

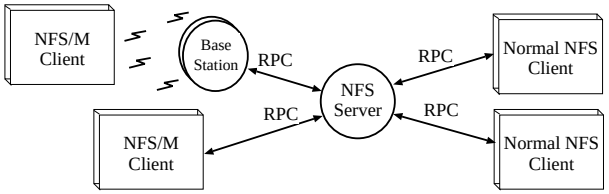

Figure 1: NFS Server with NFS/M and other traditional NFS Clients

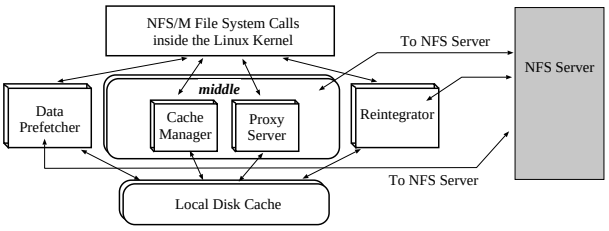

Figure 2: NFS/M Modules

with the NFS/M feature, therefore, the client computer can access data from any conventional NFS server.

2.2 Phase of the NFS/M File System

NFS/M client maintains an internal state which we term as *phase*, which is used to indicate how file system service are provided under different conditions of network connectivity. There are three phases, namely, the *connected phase*, the *disconnected phase* and the *reintegration phase*. In the connected phase, file system service are provided by the NFS server and the local disk cache manager. In the disconnected phase, the communication link between the client and the server is not available and the file system service are provided by the proxy server, which is a module of the NFS/M, to emulate the functionalities of the remote NFS server. In the reintegration phase, the communication link between the NFS server and the client is re-established and file system services can be provided by the NFS server and the cache manager. The propagation of the client updates during the disconnected phase is performed by the reintegrator module.

The *phase* of a NFS/M client can be changed either by external events, e.g. when the network is disconnected, or the phase can be changed internally, e.g. when the reintegration process is finished. The phase transition diagram of the NFS/M is depicted in Figure 3.

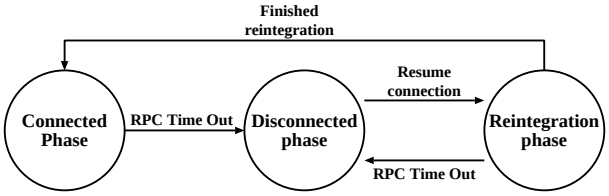

Figure 3: Phase Transition in NFS/M

Upon the initial startup, the NFS/M will be in the connected phase. During the connected phase, the NFS/M can retrieve file system objects, on behalf of the NFS/M client, from the NFS server or from the local disk cache (if the requested data objects are cached). The NFS/M will access the remote NFS server by using the Sun RPC. If the

server does not respond within a certain period of time[2], the NFS/M will switch to the disconnected phase. During the disconnected phase, the NFS/M probe the NFS server on a regular basis[3]. If both the NFS server and the communication channel are available, the NFS/M will switch into the reintegration phase. During reintegration phase, the NFS/M access the NFS server by the Sun RPC just like in the connected phase, and at the same time, propagate the updates made by the NFS/M client during the disconnected phase back to the NFS server. If the server does not respond within a certain period of time[4], the NFS/M will switch back to the disconnected phase and the reintegration process will be suspended. Upon the successful termination of the reintegration process, the NFS/M will switch back to the connected phase.

2.3 NFS/M File Semantics

Let us formally describe the NFS/M file semantics so that we can understand how data conflicts can be detected as well as under what situations these data conflict can be resolved. The following definitions give us a clear understanding so as to devise a procedure to resolve some of these data conflicts. In the NFS/M, we view a file system as a collection of files and directories. Formally, we have

Definition 1 *A file system is denoted by a set $\mathcal{S}$ where $\mathcal{S} = \{\mathcal{S}_f \cup \mathcal{S}_d\}$. The set $\mathcal{S}_f$ denotes all the file objects f_i in the file system while the set $\mathcal{S}_d$ denotes all the directory objects d_j in the file system.*

Normally, $\mathcal{S}$ is stored in the NFS server. Any client has the capability to apply any NFS file system operations to any files or directories in $\mathcal{S}$. In general, we translate any standard NFS file operations into our NFS/M file semantics which are based on the following four primitive file operations: 1) $write(f_i)$, 2) $read(f_i)$, 3) $write(d_j)$ and 4) $read(d_j)$. Before we clarify this claim, let us have the following definition.

Definition 2 *If a file object f_i is immediately under the directory object d_j, then we denote this relationship by $f_i \prec d_j$. Similarly, if a directory object d_k resides immediately under another directory object d_j, we denote the relationship by $d_k \prec d_j$.*

An immediate implication of the above definition is that some NFS file operations (e.g., creating a new file, delete a file ... etc) may require a write operation to the directory where the file is immediately under. The mapping between the original NFS file operations to the NFS/M read/write semantics is illustrated in Table 1. For example, the fifth entry in the table indicates that creating a file f_i under directory d_j requires a write to d_j for updating the file entry and a write to a new file f_i for allocating an inode. Similarly, the sixth entry in the table indicates that deleting a file f_i under directory d_j requires a write to the file f_i (for example, to update the link count and if it is zero, deallocate the associated inode) as well as an update operation to d_j for updating the file entry.

If all the users can directly access the data from the server, we said that all users are connected within one network partition. However, if there is any communication link failure, or if some mobile users are disconnected from the network, we said that these users are residing in different network partitions. For the mobile computing environment, it is very often that mobile users may be disconnected from the server. In order to provide the computing service and the capability of accessing the data, object caching is used so that even during the disconnection period, the mobile user can access the cached data objects. This way, we can enhance the data availability. However, data caching introduces the data consistency problem. For example, if two mobile clients reside in different network partitions and they modify the same data object, data conflict will occur. Because of this reason, we need a systematic way to detect any data conflict and at the same time classify which data conflicts can be resolved. This will be discussed in the following section.

Original NFS semantics	NFS/M Read/Write semantics
getattr f_i	$read(f_i)$
setattr f_i	$write(f_i)$
read f_i	$read(f_i)$
write f_i	$write(f_i)$
create f_i where $f_i \prec d_j$	$write(f_i), write(d_j)$
remove f_i where $f_i \prec d_j$	$write(f_i), write(d_j)$
rename f_i to f_i' where $f_i \prec d_j$, $f_i' \prec d_j'$	$write(f_i)$, $write(d_j)$, $write(f_i')$, $write(d_j')$
create $d_j \prec d_k$	$write(d_j), write(d_k)$
remove d_j where $d_j \prec d_k$	$write(d_j), write(d_k)$
rename d_k to d_k' where $d_k \prec d_j$, $d_k' \prec d_j'$	$write(d_k)$, $write(d_j)$, $write(d_k')$, $write(d_j')$

Table 1: From the Original NFS semantics to the NFS/M read-write semantics

3 Conflict Detection

In general, a NFS/M client can cache a subset of files from the NFS server during the connected period. Let S be the set of file system objects on the NFS server. Let S' be the set of file system objects on the local disk cache of the NFS/M client. All the objects in S' were selected for caching during the connected period of the mobile users. It is obvious that $S' \subseteq S$. In general, there are two kinds of objects in a file system, namely, the file object and the directory object. We use f_i and d_j to denote any arbitrarily file and directory object respectively. For notational convenience, let us use o_i to denote any file system object, which may either be a file or a directory.

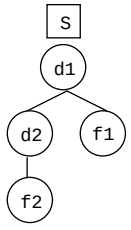

Figure 4: Example of file system objects in S

Now, consider a NFS/M client, which has already cached a subset S' of the file system objects from S, When the phase changes from the connected to the disconnected phase, the NFS/M proxy server module starts to emulate

[2]The default is 100ms and it is adjustable

[3]The default is 1000ms and it is adjustable

[4]The default is 100ms and again, it is adjustable in the mobile client

the NFS server so as to provide file system service. During the period of disconnection, objects in S' may be modified, e.g. $f_i \in S'$ may be changed to f_i' (either by modifying the file f_i or by renaming the file f_i to f_i'). In addition, there is a possibility that the subset S' may grow or shrink, which corresponds to the file system operation - create or remove. To make this more precise, two new notations are introduced. We let C be a creation set and R be a removal set, where C is the set of file system objects which were created or were removed during the disconnected phase, respectively. We also note that the removal set only deals with the objects which are originally in S', i.e. $R \subseteq S'$. For the case where a user has deleted a file which was created in the disconnected phase, i.e. a delete operation on file $f_i \in C$, we do not consider f_i in R but rather representing this case by removing a file f_i from C (or simply assuming that there was no creation of f_i during the disconnected phase). It follows that, from the above convention, the set of file system objects on the NFS/M local disk cache at the end of the disconnected phase (or at the beginning of the reintegration phase) can be denoted by $(S' - R) \cup C$. Note that the NFS/M has enough information to distinguish the sets S', R and C from the local disk cache via a log facility which was constructed during the disconnection period. The log is basically a sequence of file operations that occur during the disconnection period, with the exception that it does not contain any read-only operations, e.g., like nfs_read, nfs_readdir, etc (detail description of the log creation is given in Section 5.2).

To illustrate, figures 4 and 5 shows the relationship between the sets S, S', R and C. Figure 4 is an example of the set S of file system objects on the NFS server. Objects d_1, d_2 are the directories, where d_1 is the immediate parent of d_2, denoted by $d_2 \prec d_1$. At the same time the file objects f_1, f_2 are under the directories d_1 and d_2 respectively, i.e., $f_1 \prec d_1$ and $f_2 \prec d_2$. These objects reside in the NFS server, and we denote the set of file system objects by S, where $S = \{d_1, d_2, f_1, f_2\}$.

Now, suppose there is a NFS/M client, and it cached S completely. The cached set is called S', assume that it is fully cached, we have $S' = S = \{d_1, d_2, f_1, f_2\}$. Then the NFS/M client is disconnected from the network and the situation is that there are two replicates of S, one is S and the other is S'. Since the NFS/M allows the user to continue to work on the cached subset S', and Figure 5 shows an example of changes to S'. A NFS/M client first issues a $write(f_1)$ operation, and then $remove(f_2)$, $create(f_3)$. Recall that from the NFS/M semantics, the $remove(f_2)$ can be reduced to two write operations, $write(f_2)$ and $write(d_2)$, and similarly $create(f_3)$ is reduced to $write(f_3)$, $write(d_2)$. Since d_2 , f_3 and f_1 are modified[5] , their changes should be propagated back to the NFS server when the communication link is re-established. Notice that at the end of the disconnected period, $d_1, f_1, d_2 \in S'$, $f_2 \in R$ and $f_3 \in C$, and the objects need to be propagated back to the server are: f_1, d_2 and f_3.

According to the above discussion, the nature of reintegration is to propagate the changes in the set $(S' - R) \cup C$ (from the NFS/M client) to the set S (on the NFS server). Since there is a possibility that an object, either a file or a directory, $o_i \in S'$ and $o_i \in S$ is modified during the disconnection period, a data conflict is possible. A natural

[5]changes in d_2 include a deletion of f_2 and a creation of f_3

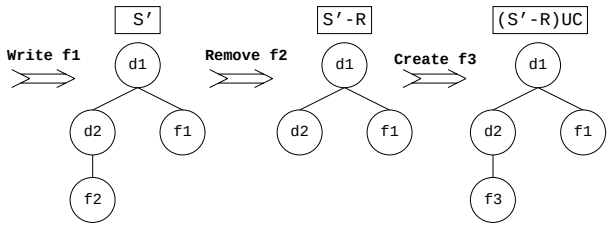

Figure 5: Example of file system objects in S', R, C

question is how the system can detect any data conflict. We answer this question by first giving the following definition.

Definition 3 *Let τ be the "last modified time" of an object o_i that was fetched from the NFS server upon caching o_i in the local disk cache. Therefore, $o_i \in S'$ and $o_i \in S$ before the disconnection period. At the end of the disconnected period, a conflict is detected if the modified time of the object o_i in the server is greater than τ, or if the object o_i does not exist in the server ($o_i \notin S$).*

Since the "last modified time" τ of an object is stored in the NFS/M upon caching of that object, the system can use this information to detect any data conflict and at the same time, determine whether the conflict can be resolved or not. The above definition is the core algorithm for detecting conflict.

Theorem 1 *Let user B be the mobile user and during his disconnected period, B issued a $write(o_i)$, where $o_i \in S'$. If there was no write operation to object o_i issued by other users during the disconnection period of B, then there is no data conflict for the object o_i.*

Proof: This can be observed easily from the definition of data conflict. Since there was no user who issued a $write(o_i)$, therefore, during the reintegration period of user B, the modified time of o_i in the server is less than or equal to τ, so there will not be any data conflict. ∎

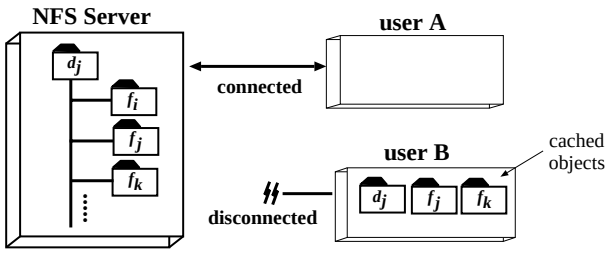

Figure 6: Example of data conflict.

Remarks: The implication of the above theorem is that the NFS/M allows simultaneous reading of files, simultaneous reading and writing of different files. Using the example illustrated in Figure 6, some of the read/write operations which were issued during the disconnection period of B will not cause a data conflict can be: (1) A issued $read(f_i)$ and B issued $read(f_i)$, (2) A issued $read(f_i)$ and B issued $read(f_j)$, (3) A issued $read(f_i)$ and B issued $write(f_i)$, (4) A issued $read(f_i)$ and B issued $write(f_j)$, (5) A issued $write(f_i)$ and B issued $read(f_j)$ and (6) A issued $write(f_i)$ and B issued $write(f_j)$.

Corollary 1 *When two or more users who resided in different network partitions issued a write(f) where $f \in \mathcal{S}$, a file conflict will be detected during the communication channel re-establishment.*

Proof: Because a *write*(f) operation induces a change in the modified time, the modify time of f in the server will be greater than the "stored modified time" τ. Therefore, base on the definition of data conflict, this conflict can be detected. ∎

```
procedure CONFLICT_DETECTION
begin
    for each co_i ∈ S' which is modified (according to
    the log)
        if so_i ∉ S then return conflict
        else
        if (so_i.modified_time > co_i.stored_modified_time)
        then return conflict
        else return no conflict
end
```

Figure 7: Conflict detection algorithm

Let co_i and so_i be file system objects in the NFS/M client and the NFS server, respectively. The conflict detection algorithm is given in Figure 7. The conflict detection algorithm returns either *conflict* or *no conflict*. For the case of *no conflict*, the corresponding object is written back to the server, that is, the changes of the object in the mobile client's local disk cache is propagated back to the server. On the other hand, if data conflict is detected, this object will be passed to the conflict resolution module, which will be discussed next.

4 Conflict Resolution

In general, we can classify data conflict into two classes, namely, file conflict and directory conflict. Each of them can be dealt with differently. In this section, we present these two classes of conflict resolution algorithms.

4.1 File Conflict Resolution

File conflict to an object f_i occurs when multiple write operations were issued by two or more users which are residing in different network partitions. The nature of file conflict prohibited us to merge two changes into one, unless we know exactly the file format (e.g., if it is a database file, we may merge the two writes on two different tuples back into a database file). Because of this, we adopt an approach of maintaining two versions of the file object, one is from the NFS server and the other is from the cache of NFS/M client. For example, the system detects a file conflict for file f_i, then the NFS/M creates a new file on the NFS server using the original file name of f_i, together with a suffix. Then the content of the NFS/M client copy is written to this new file. For details, the suffix we chose is the machine name of the NFS/M client. Once this is done, the system will inform the NFS/M client who issued the file operation that caused the file conflict. Note that based on the scheme described above, some interesting mobile utilities can be developed so as to merge some well-known format files, as discussed in [11].

4.2 Directory Conflict Resolution

Directory conflict can be resolved by using the semantics information on the structure of the directory data. A directory can be viewed as a well-structured file, which contains a list of directory entries where each entry consists of a filename and a fileid.

Before we discuss about the resolution method, let us define the following notation:

- o_i: A file system object, either a file or a directory which is in both S' (for client side) and S (for server side). This implies $o_i \in S'$ (since $S' \subset S$).
- o_i': A file system object, either a file or a directory, which is in either C (for client side) or S (for server side) or both.
- d_s: A directory object in the NFS server, or $d_s \in S$.
- d_c: Replicated copy of a directory object which is stored in the NFS/M client, or $d_c \in (S' - R)$.
- $o_i \prec d_c$: indicates that o_i is immediate under the directory d_c.
- $o_i \not\prec d_c$: indicates that o_i is not immediate under the directory d_c.
- $o_i' \prec d_c$: indicates that o_i' is a directory under d_c, and note that $o_i' \in C$.
- $o_i' \not\prec d_c$: indicates that o_i' is not under a directory d_c, and in addition $o_i' \notin C$.
- $o_i \prec d_s$ (or $o_i' \prec d_s$): indicates that o_i (or o_i') is immediate under the directory d_s, according to the file system set S on the server.
- $o_i \not\prec d_s$ (or $o_i' \not\prec d_s$): indicates that o_i (or o_i') is *not* immediate under the directory d_s, according to the file system set S on the server.

The resolution algorithm is as follow: d_s and d_c are the corresponding server and client copies referring to the conflicting directory. Assume that there are n_s directory entries in d_s and n_c directory entries in d_c. Therefore, there are $n = n_s + n_c$ objects that are possibly under d_c and d_s. Note that some of them may be the same, but in any case, the system examines these n objects one by one. For each system object o_i within those n objects, the system executes the conflict resolution procedures as indicated in Table 8. After all n objects are examined, the resolution algorithm terminates.

Situation upon reconnection	Resolution strategy
1. $o_i \prec d_c$ & $o_i \prec d_s$	Do Nothing
2. $o_i \prec d_c$ & $o_i \not\prec d_s$	Remove o_i from d_c
3. $o_i \not\prec d_c$ & $o_i \prec d_s$	Remove o_i from d_s
4. $o_i' \prec d_c$ & $o_i' \prec d_s$	Name clash resolution
5. $o_i' \prec d_c$ & $o_i' \not\prec d_s$	Create o_i' in d_s
6. $o_i' \not\prec d_c$ & $o_i' \prec d_s$	Create o_i' in d_c

Figure 8: Directory Conflict Resolution Strategy

Here, we give a comprehensive presentation on each entry of the table.

- Entry 1 represents the case that the object exists in both copies (client and server) of the directory, this implies that both the server and the NFS/M client have the consistent view of o_i, therefore no action is necessary.

- Entry 2, $o_i \not\prec d_s$ means that the o_i has been removed from d_s on server side. Therefore, the algorithm removes the corresponding o_i under d_c at client side to preserve consistency.
- Entry 3, this is similar to entry 2. The object o_i has been removed from d_c. So the corresponding o_i under d_s needs to be removed to preserve consistency.
- Entry 4 is classified as the name clash problem. Since $o_i' \in S$ and $o_i' \in C$, it implies that the users have created an object o_i' with the same filename under the same directory in different network partition. Therefore, the NFS/M resolves this by splitting a single copy into two, the procedure is similar to the file conflict resolution, in which the NFS/M adds a reasonable suffix to the filename of the copy $o_i' \in C$.
- Entry 5 is the case that object o_i' is created on the client side only. Since there is no name clash problem (as oppose to entry 4), therefore, we can safely create o_i' on the NFS server.
- Entry 6 represents an object o_i' is created on the server partition but not in the NFS/M client. Therefore, the client side should also include the object o_i' in d_c.

Note that one of the entries on the table states the action as *name clash resolution*. This resolution method is similar to the file conflict resolution, which is to add a suffix on one of the copy of the file f_i, such that the system maintains two versions of the object.

5 NFS/M Modules and Functionalities

Briefly speaking, the core of the NFS/M is the daemon called *middle*, which traps all RPC traffic that comes from the NFS/M client to the NFS server. Two other important modules include the Reintegrator (RI) and the Data Prefetcher (DP). The module *middle* can be further subdivided into two logical modules, namely, 1) the Cache Manager (CM) and 2) the Proxy Server (PS). These modules work independently in different phases of the NFS/M. During the connected phase, all the NFS requests from the local client are serviced by the CM, which fetches data from the NFS server and at the same time, decides which data item needs to be cached in the cache so as to improve the read/write performance. During the disconnected phase, the PS module emulates the NFS server to provide file services. Therefore, local client can still read/write a file (if it is cached) as well as to create new files or new directories. During the reintegration phase, the RI performs reintegration so as to propagate changes to file system objects which were inside the cache back to the NFS server. These three modules work independently to provide all necessary functionalities of the NFS/M.

5.1 Cache Manager (CM)

All the file system operations to any cached objects in the local disk cache are managed by the Cache Manager (CM). The CM services all the requests defined in the NFS 2.0 protocol [7], and it functions only in the connected phase. We describe the CM in two aspects: 1) the data structure which defines the cache format, and 2) the operations related to any object in the cache.

5.1.1 Cache Data Structure

The local disk cache consists of files and directories, which are the fundamental objects of file systems. Each file system object has two parts, the data and the metadata. For the data part, the system stores the data that associated with the file, and for a directory, it contains the list of file entries inside it, i.e. all file names that are immediately under that directory. In the metadata part, the system stores the information about the particular object. Figure 9 depicts the information that NFS/M stores for a file or a directory object.

Metadata Structure	
directory	file
`file attributes`	`file attributes`
`new_bit`	`new_bit`
`modified_bit`	`modified_bit`
`accessed_bit`	`accessed_bit`
`wholefile_bit`	`wholefile_bit`
`server_modified_time` (τ)	`server_modified_time` (τ)
	`validation bitmap`

Figure 9: Metadata Structure

The detail explanation of each item in the metadata structure is described below:

- `file attributes` are the typical attributes for the UNIX files.
- `new_bit` is a flag that indicates a cached object which is newly created during the disconnected phase.
- `modified_bit` is a binary flag that indicates a cached object which has been modified during the disconnected phase.
- `accessed_bit` is a binary flag that indicates whether cached object which has been accessed (read/write) during the disconnected phase.
- `wholefile_bit` is a binary flag that indicates whether a cached object is partially or fully cached. This flag is true if all the validation bits are true, and it is false otherwise. See below for the definition of validation bit.
- `server_modified_time` (τ) is the last known modified time just before disconnected phase comes. It is an important metadata which the system uses for detecting any data conflict, and this metadata is used by the PS and the RI.
- `Validation bitmap` is a set of bits such that each bit representing whether the corresponding block of a file is cached or not. The system also uses this information to derive the wholefile_bit.

5.1.2 Cache Operations

Upon a NFS `read` operation, the CM looks for the requested object to see whether it exists in the local disk cache and test whether it is valid or not. We say that an object is valid if its `server_modified_time`, τ, is the same as the "modified time" of the corresponding object in the NFS server. To obtain the "modified time" of an object, the system uses the `nfs_getattr` to request from the NFS server the file attribute, which contains the modified time. If the modified time matches with τ, the CM returns the data blocks stored in the cache. Note that this kind of operation reduces the communication overhead caused by transferring large amount of data block from the NFS server to the NFS/M client. Since the NFS/M only needs to fetch the file attribute, which is much smaller as compare to the data blocks, so the response time of accessing

any cached data object is reduced. As for the read request to any non-cached object, the CM will fetch the data block from the server, and at the same time, the CM will store the data block in the cache.

For a NFS `write` operation, the CM uses the write-through mechanism. It writes the data both to the NFS server and the cache, and then updates the value of τ in the cache so that it is same as the modified time on the server. Besides updating the value of τ, it also updates the validation bits of the corresponding object.

Note that since the disk cache has a limited size[6]. Due to this size limit, the CM must perform cache replacement when the cache space is exhausted. The cache replacement algorithm that we are using is basically the least-recently-used (LRU) algorithm, with one exception. Since our data prefetcher provides an option for the user to specify some files which are more *favorable* for caching, therefore, these files will be given a higher priority to stay in the cache, even though these files may not be used recently.

5.2 Proxy Server (PS)

The NFS/M daemon *middle* serves as the *proxy server* in the disconnected phase, it emulates the functionalities of the remote NFS server by using the cached file system objects in the local disk cache and at the same time, supports new file/directory creation. Optimistic replica control [16] has been employed among different network partitions so as to increase the availability of file system objects.

Since the cache contains only a *subset* of the file system objects of the NFS server, this implies that the PS can only provide file system services to the disconnected client when the referred file system objects are within this subset. All the references to any uncached object will result in a "`file not found`" error.

When the NFS/M client is first disconnected from the NFS server, it contains a subset of the file system objects from the server. However, this situation changes when the user continue to modify, create and delete objects in the local disk cache. For example, when the user creates a new file during the disconnected period, this file will not appear in the NFS server until the end of the *reintegration* period. This introduces a problem for object identification. Under the NFS protocol, each file system object is identified by an unique *object handle*, and all handles are generated by the NFS data server. During the disconnected period, PS is responsible for generating new object handles for requests from the NFS/M clients. At the same time, the NFS server is able to generate new object handle for other NFS clients. Since the network is partitioned, and there is no communication between the NFS server and the PS, it may turn out that the handle generated by these two parties is the same. The question is how to maintain the consistency of the handle generated by PS.

Since the handle generation pattern by any NFS server varies depending on different implementations. Therefore, we choose not to modify the server side code to maintain the compatibility. Since the PS has all the information about the file system objects within the cache, it follows that the PS is able to generate an object handle which is unique within the scope of the cache. We have implemented an algorithm to generate a unique handle with respect to the local disk cache.

[6]the default size of the cache is 10M and the disk cache size can be dynamically adjusted by the NFS/M client.

Although it is possible to generate an unique handle with respect to the cache, it does not guarantee that the handle is unique with respect to the NFS server. When the phase changes from the disconnection to the reintegration phase, there is a possibility that two handles generated by two parties (NFS server and PS) will cause a *handle clash*. Because of this reason, the system takes care of the handle clash problem in the Reintegrator module, which we will discuss in a later section.

To enable the propagation of updates of the disconnected client to the NFS server after reconnection, the PS will keep track of all the *update* operations to the cache, it includes *write, create, remove, rename, mkdir, rmdir, setattr, symlink* and *link* in the *log*. The *log* is a list of entries that follows the chronological order of *update* operations during the disconnection period. Each log entry consists of the operation types, i.e. write, create, remove, etc., and other parameters such as the object handles and the filenames. e.g. a log entry looks like:

```
[write, object handle];
[create, dir object handle, new filename];
```

Notice that for the write operation, the system only considers the final write operation to a file. If there are several write operations upon a file f_i, the system only keeps the final version of f_i after those write operations. This is because the cache keeps a single copy of f_i only for the efficient usage of the disk space. Therefore, for the log entry the system only keeps the last write operation to f_i.

5.3 Reintegrator (RI)

The Reintegrator (RI) is responsible for propagating the changes of the data objects in the local disk cache performed during the disconnected period back to the NFS server. Since file operations are allowed during the disconnected period, there exists the possibility of conflicts among different updates to the cached data objects. There are three tasks for the RI, namely, 1) conflict detection, 2) update propagation, and 3) conflict resolutions. Given the log produced by the PS during the disconnected period, the RI checks each entry in the log, to see whether the propagation of the objects in the disconnected client back to the NFS server will cause a conflict or not (e.g. using the conflict detection algorithm described in section 3). If the propagation will cause a conflict, the file system operation in the log is termed as "invalid". If the file system operation is valid, the RI propagates the changes back to the NFS server, otherwise, the RI tries to resolve the conflicting operations using the conflict resolution algorithm described in the section 4.

Lastly, it is important to mention the case of file handle clash. Recall that it is possible for the NFS/M client to create a new file during the disconnected period. For example, there is a file, with handle H_1, which was created during disconnected period. To propagate the new file back to the NFS server, the NFS server may create a new file handle, H_2, for this newly created file. Therefore, RI needs to change the file handle information of the file in the local disk cache as well as the entries in the log file from H_1 to H_2.

5.4 Data Prefetcher (DP)

Data prefetching is an effective technique for improving data access performance [18, 19]. In a mobile file system, it

also increases the availability of data during the disconnection period by collecting data objects in the cache. Also, it reduces the latency during the *weakly connected period* by selectively decide which data need to be cached and which data need to be written back to the NFS server [10].

Data prefetching techniques can be classified into two categories, they include the *informed prefetching* and the *predictive prefetching*. With informed prefetching, the users and the application programmers can specify which files are needed ahead of time and thereby preload them to the cache. With predictive prefetching, the system tries to predict which objects will be referred and try to preload these objects. In general, prediction is based on user's file traces or access patterns.

Note that both of these techniques have their advantages and the performance gain is applicational dependent. For a rarely accessed file, such as the telephone number database file, which may sometimes be very critical for the mobile user. In this case, the informed prefetching works well because this kind of file access cannot be derived by past accesses. While in other cases, the users themselves may not know exactly what files are needed in the future. One good example is the UNIX password file. Without this file, no user is allowed to login to the system. In this case, predictive prefetching has an advantage over the informed prefetching for it is totally transparent to the application programs and users.

In NFS/M, we have implemented both techniques. For the informed prefetching, the users can specify which files are preferred to be prefetched[20]. We have also implemented a simple predictive prefetching technique such that if a particular block of a file is fetched, the rest of the blocks will be fetched as well. Since it is very likely that the user may want to access the rest of the file when he/she starts accessing a block of the file. Therefore, it is a good idea to fetch the remaining blocks into the data cache in advance.

6 Conclusion

In this paper, we present the design and implementation of a new mobile file system, NFS/M, on the Linux platform and it is based on the NFS protocol. We also present the formal definition for the NFS/M file semantics, a conflict detection algorithm and conflict resolution algorithms. We also illustrate how the NFS/M system can keep track of the state of the network connectivity. We have implemented various modules such that the NFS/M can support client-side data caching (so as to improve the system performance), file services support during the disconnected period and data reintegration when the communication channel is re-established. Finally, experiments show the NFS/M actually improves the overall system performance though client-side caching, as well as increased the availability of file system objects during disconnected period (Please refer [21] for details of experiments). Future work on NFS/M is focused on creating a set of file conflict resolution utilities based on the semantic information of different applications.

Acknowledgments

The authors would like to thank the anonymous referees for their insightful and helpful comments.

References

[1] Rafael Alonso and Henry F. Korth. Database System Issues in Nomadic Computing, SIGMOD RECORD, Vol 22, 1993.

[2] George H. Forman and John Zahorjan. The Challenges of Mobile Computing, IEEE Computer, April, 1994.

[3] Tomasz Imielinski and B. R. Badrinath Data Management for Mobile Computing, SIGMOD RECORD, Vol 22, 1993.

[4] Henning Koch and Lars Krombholz and Oliver Theel. A Brief Introduction into the World of Mobile Computing, THD-BS-1993-03, Department of Computer Science, University of Darmstadt.

[5] Cedric C.F. Fong, John C.S. Lui , Man Hon Wong. Quantifying Complexity and Performance Gains of Distributed Caching in a Wireless Network Environment. 13^{th} International Conference on Data Engineering (ICDE '97), Birmingham, England, 1997.

[6] Cedric C.F. Fong, John C.S. Lui , M.H. Wong, E.Silva. Performance Analysis of Mobile Terminals Tracking Algorithms, 18^{th} IFIP TC7 Conference on System Modeling and Optimization, Detroit, 1997

[7] Bill Nowicki, RFC 1094: NFS 2.0 Protocol Specification, Sun Microsystems, Inc., March 1989

[8] George Coulouris, Jean Dollimore, Tim Kindberg, Distributed Systems - Concepts and Design (2nd Ed.), Addison-Wesley

[9] J. J. Kistler, Disconnected Operation in a Distributed File System, PhD thesis, School of Computer Science, Carnegie Mellon University. May 1993

[10] Lily B. Mummert, Maria R. Ebling, M. Satyanarayanan, Exploiting Weak Connectivity for Mobile File Access, August 1995

[11] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, Gerald Popek, Resolving File Conflicts in the Ficus File System, Proceedings of the 1994 Summer Usenix Conference

[12] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier, Implementation of the Ficus replicated file system, USENIX Conference Proceedings June 1990, pages 63-71

[13] Fischer, M. J., and Michael, A., Sacrificing serializability to attain high availability of data in an unreliable network, Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, May., In ACM, New York, pp. 70-75. 1982.

[14] Parker, D.S., and Ramos, R.A., A distributed file system architecture supporting high availability. Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks, pp. 161-183, 1982. Lawrence Berkeley Laboratory, University of California, Berkeley.

[15] Davidson S. B., An optimistic protocol for partitioned distributed database systems, Doctoral dissertation, Dept. of Electrical Engineering and Computer Science, Princeton Univ., Princeton, N.J. (Oct.) 1982.

[16] J.J. Kistler, M. Satyanarayanan, Disconnected Operation in the Coda File System, ACM Transactions on Computer Systems, Feb. 1992, Vol. 10, No. 1, pp. 3-25,

[17] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen, Consistency in partitioned networks, ACM Computing Surveys, 17(3):341-370, September 1985.

[18] Hui Lei and Dan Duchamp, An Analytical Approach to File Prefetching, 1997 USENIX Annual Technical Conference, Anaheim CA, January 1997

[19] Geoffrey H. Kuenning, Gerald J. Popek, Peter L. Reiher, An Analysis of Trace Data for Predictive File Caching in Mobile Computing, 1994 Summer USENIX Conference, April 1994

[20] NFS/M System Manual on Linux 2.0.x, NFS/M Project Group, September 1997

[21] John C.S. Lui, K.Y. So, T.S. Tam, NFS/M: An Open Platfrom Mobile File System, Technical Report CS-TR-97-14, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Oct 1997.