Cross-Partition Protocols in a Distributed File Service

Zheng Zhang, Christos Karamanolis, Mallik Mahalingam,
Daniel Muntz
Computer Systems and Technology Laboratory
HP Laboratories Palo Alto

HPL-2001-129
May 23rd , 2001*

distributed file system, distributed namespace, fault tolerance, Storage Area Network (SAN)

A number of ongoing research projects follow a partition-based approach in order to achieve high scalability for access to the distributed storage service. These systems maintain a namespace that references objects distributed across multiple locations in the system. Typically, atomic commitment protocols (e.g., 2-phase commit) are used for updating the namespace, in order to guarantee its consistency even in the presence of failures. Atomic commitment protocols are known to impose a high overhead to failure-free execution. In addition, they use conservative recovery processes and may considerably restrict the concurrency of overlapping operations in the system.

This report proposes a set of new protocols for the implementation of the fundamental operations in a distributed namespace. The protocols use intention logs to impose a minimal overhead to failure-free execution. They are robust against both communication and host failures, and they use aggressive recovery procedures to re-execute incomplete operations. The proposed protocols are compared with their 2-phase commit counterparts and are shown to be strictly better in all critical performance factors: communication round-trips, synchronous I/O, operation concurrency.

* Internal Accession Date Only

1. Introduction

DiFFS is a distributed file service architecture designed for storage area networks [1]. DiFFS achieves high scalability by following a partitioning approach to sharing storage resources. The architecture is robust against failures and unfavorable access patterns. It is independent of the physical file system(s) used for the placement of data; multiple file systems can co-exist in a DiFFS system.

Much of DiFFS scalability is attributed to its unique partitioning approach. Each partition is controlled by one *partition server*, which coordinates non-idempotent operations that may affect the state of the resources (allocate or de-allocate blocks, for example) of the SAN partition under its responsibility. Other operations can bypass the partition server and directly access the SAN storage (Figure 1-a)

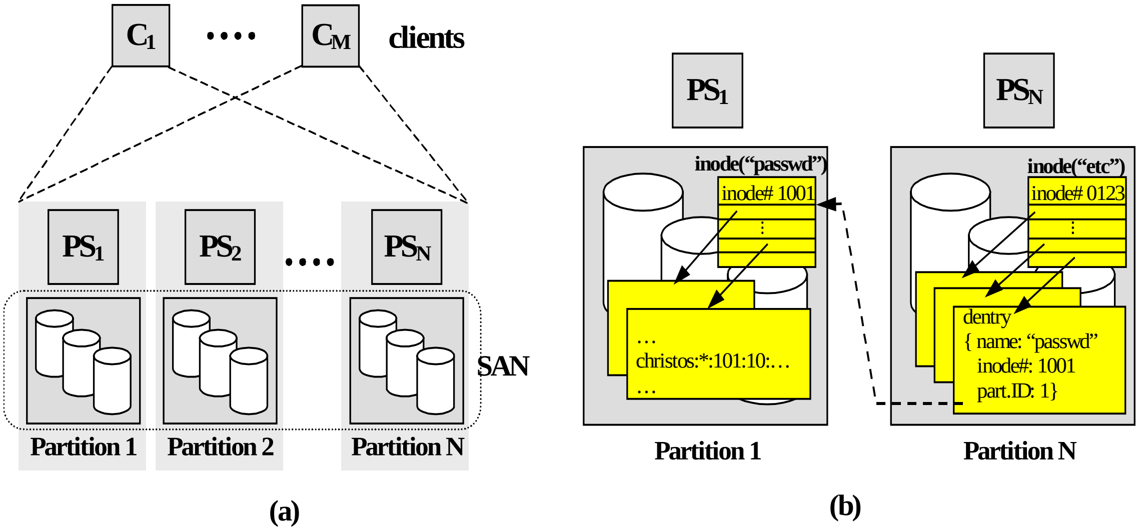


Figure 1: (a) DiFFS overall architecture and (b) Cross-partition reference in DiFFS namespace

DiFFS partitions work together to present one uniform filesystem space to the end clients. As such, directories may reside in arbitrary partitions, and a file's parent directory can be on a different partition. Directories are stitched together with *cross*-partition references. In essence, the traditional filesystem inode information is augmented with the partition ID of the pointed project. The challenge is then how to make sure this global namespace is kept valid even in presence of failures, and how to do this in the most efficient way.

While the intention of this report is to investigate protocols for cross-partition operations in the context of DiFFS, the problem here is generic. It can be broadly stated as: *how to maintain global namespace consistency over a collection of distributed objects*. Changes to the global namespace take one of two forms:

- ***Insertion***: a new reference, pointing to a possibly newly created object, is inserted into the namespace. File system operations such as *Create*, *Link*, *Rename* ("to" directory) belong to this category.

- ***Removal***: a reference pointing to an already existed object is removed from the namespace. *Remove* and *Rename* ("from" directory) are primary examples of these operations. If all references to an object are removed, the object itself is garbage collected.

The remaining of the paper is organized as follows: Section 2 covers general assumptions including failure model and client-side semantic guarantees. Section 3 discusses how we reduce the problem to "orphan" objects and never corrupt the global namespace. Section 4 is the main body of the paper; it describes the protocol of each DiFFS namespace operation in detail as well as the recovery procedures. Section 5 discusses issues for the efficient implementation of the. Section 6 compares our algorithms with an approach using the conventional 2-phase commitment protocol and shows why and how our algorithms are indeed more lightweight. Lastly, we conclude in Section 7 by discussing related work in this area.

2. General assumptions

2.1 System model and failure assumptions

We assume the following failure model:

- Hosts fail by crashing; they do not exhibit malicious (Byzantine) behavior.
- Messages may be not sent or not delivered due to host crashes. Also, messages may be lost due to network partitioning. On recovery from any such failure, the communication session between two hosts is re-established. Messages delivered during the same communication session between two hosts are always delivered in order. This condition is guaranteed by using TCP as the communication protocol.
- Consistency of the local object-store is guaranteed, despite failures. This property is ensured by mechanisms of the physical file system, such as journaling [2], soft updates [3] or recovery procedures (fsck) [4].
- Log entries are written synchronously and atomically.

2.2 Client-side failure semantics

It is important to discuss the failure semantics presented to the clients. The idea is to provide no stronger and no weaker semantics than those of traditional client-server models (e.g., NFS [5] or CIFS [6]). The client application can receive an error, but the operation may or may not have been carried out. For example, when the application creates a new file and receives an error, it may find later that the file has actually been created in the system. The requirement is for the operations to satisfy "at-most-once" semantics.

These semantics are consistent with what traditional client-server protocols offer. For example, in NFS, the server can crash before the RPC request is received, or network partition prevents the request to arrive at the server at the first place. In this scenario, the operation will not be carried out at the server. On the other hand, even when the operation is carried out, the server may crash before the RPC response is sent, or network partitioning may prevent the response to be delivered to the client. In this case, the operation has already been performed on the server, but the application running on the client gets an error nonetheless. If any failures occur, we simply do not respond back to the client. This is equivalent to failure of the client-server communication.

3. Problem abstraction

3.1 The two basic operations

DiFFS has the freedom to implement its own namespace operations – utilize a standard set such as NFS is an option rather than a requirement. Also, all we care about are non-idempotent operations. From DiFFS namespace point of view, the two basic primitives are *link*, which inserts a reference to an object into the namespace and *unlink*, which removes a reference to an object. The rest of distributed file system operations can be mapped to these two primitives, as shown in the following table:

Table 1: De-composition of distributed file system operations.

NFS operations	***DiFFS primitives***
Create/mkdir	Obtain a free file + *Link*
Link	*Link*
Remove/rmdir	*Unlink*
Rename	*Link (to_dir) + unlink(from_dir)*
Migration support	*Relink = unlink(old_obj) + link (new_obj)*

Each partition server owns a pool of pre-allocated, “free” files. Thus, the DiFFS *create* operation is implemented by 1) obtaining a file from the pool of the target partition and 2) performing a *link* primitive in the namespace. Details on optimizations of the create operation are discussed in the implementation section (Section 5).

3.2 Reducing the problems due to failures to “orphan” objects

Existing systems that follow a partition-based approach suggest the use of transactional semantics [7, 8] (for example 2-phase-commit protocol) for the execution of such operations [9-11]. These protocols are expensive and affect the performance of operations in the failure-free case [12, 13] (for a more detailed comparison, see Section 6). A more lightweight approach is used in DiFFS. By imposing a strict order on the execution of such operations, we can guarantee that all possible inconsistencies are reduced to an instance of “orphan” objects. An orphan is an object that physically exists in some partition but is not referenced from any point in the DiFFS namespace.

The required execution order for cross-partition operations can be abstracted to the following three steps:

1. Remove reference from the namespace, if necessary.

2. Perform changes of the target object, if any.

3. Insert reference in the namespace, if necessary.

So, the problem of namespace integrity is reduced to garbage collection of orphan files:

- In the case of *link*: the reference is added to the namespace at the last stage of the execution. Thus, in the worst case, the target object may not be referenced by any point in the namespace.

- In the case of *unlink*: removing the reference from the namespace is the very first stage of the execution. Again, in the worst case, the target object may not be referenced by any point in the namespace.

4. Protocols

4.1 Data Structures

There are two fundamental data structures in the DiFFS namespace organization and they complement each other:

1. **Directory**:

   Each directory consists of multiple directory entries. The directory entry contains essentially the reference to an object in the system. These references are typically used by lookup procedures to locate the object in the file system. Specifically, in DiFFS, the directory entry includes the name, partition ID (PID), inode# as well as a monotonically increasing number (trans#), the transaction number of the operation that created the directory entry.

2. **Back-pointers**:

   In traditional file systems, such as Ext2 of Linux [4], system objects (e.g. files and directories) are assigned a property known as “link-count”. This property is an integer representing the number of hard-links that reference the object. However, for a distributed filesystem such as DiFFS, link-count is not a sufficient property to maintain information regarding the references to an object. The reason is that the integrity of the link-count may be compromised in the presence of multiple concurrent operations that involve the same object and in the presence of failure where partially completed operations are re-executed. To address these problems, we extend the notion of link-count by using “back-pointers”. The back-pointers associated to an object reference the directories that contain hard-links to the object. They are attributes associated with each object, if it has been linked into the DiFFS namespace. Back pointer contains the PID of the parent inode, the parent inode #, the name under which it appears in the directory, and the trans# of the *link* operation that inserts this object into the namespace.

   In fact, the back pointer resembles a directory entry—it contains a reference to an object in the system. The difference is that in the case of back pointer, this is a “parent” directory of the object the pointer is associated to. The uniqueness of the back pointer content is what makes the operations *idempotent*, something that plain link-count is unable to guarantee. The use of back-pointers will become clear in the section where recovery and conflict resolution protocols are discussed.

4.2 Failure-free protocols

This section describes the protocols for failure-free execution of the *link* and *unlink* operations, in DiFFS. The following is an index of the notations used in the diagrams:

- [act]: an atomic operation, either done or not done.

- ->: cross-partition message (or flow of control).
- **X*n***: denotes an error position. This error can affect anything *after* the immediately proceeding atomic action.
- Log+/-: log record insertion /deletion. The log record contains sufficient information to replay a given operation.
- D+/-: addition/deletion of a directory entry.
- Bptr+/-: back pointer addition/deletion.

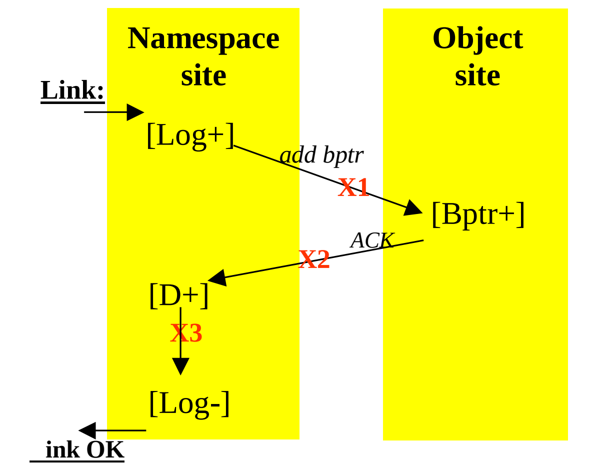
4.2.1 *Link*

The first fundamental operation in the DiFFS namespace is *link*, which refers to insertion to the namespace. The algorithm for the execution of *link* is shown in Figure 2.

Namespace site:

```
Link {
   if D does not exist then
        Log+;
        Execute Link main body;
   else
        Reply to client (error);
}
Link execute {
   Send “add bptr”+ “info” to Obj site;
   Wait until reply received or Timeout;
   if reply=ACK then
        if D does not exist then
             D+;
             Log-;
             reply to client (success);
        else /* ELINK(1) */
             unlink execute; //as in unlink op
             Log-;
             reply to client (error);
   else if reply=NACK then
        Log-;
        reply to client (error);
}
```

Message diagram:



Object site:

```
Add back-pointer {
   if back-pointer exists
      (compare info vs. back-poiner) then
        if same transaction# then /* ELINK(2) */
             send ACK back to namespace;
        else /* ELINK(3) */
             send NACK back to namespace site;
   else
        Bptr+;
        Send ACK back to namespace;
}
```

Figure 2. Failure-free execution of operation link.

The *link* operation takes one round trip to be accomplished. In addition, there are three synchronous I/O’s to storage, two of which to the log, and another to add the back pointer. We insert the directory entry after the back pointer is added at the object-side, strictly following the ordering rule laid out earlier. The object-side back pointer contains sufficient bookkeeping information in order to handle recovery and perform garbage collection. The use of back pointers will become clear in the sections where recovery and multi-operation conflicts are discussed.

The opening and reclamation of a log record in the name-space site mark the beginning and the end of the *link* operation respectively. A log record being open implies that the operation has not been completed. This may indicate the existence of failure and is used to initiate the recovery process.

Notes:

- The “add bptr” message carries an “info” record as its payload. This record includes the location details (PID, inode#) of the parent directory as well as the transaction# of the current *link* operation. The back-pointer has to contain all this information to resolve conflicting operations and perform garbage collection.
- The *ELINK(*)* points are handling special conflict cases, as will be described later in the “multi-operation conflicts” section.

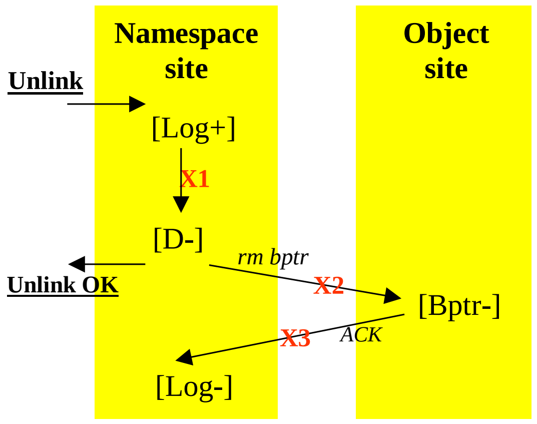
4.2.2 *Unlink*

The second fundamental operation in the DiFFS namespace is *unlink*, which removes a reference of an object from the namespace. The algorithm for the execution of *unlink* is shown in Figure 3.

Namespace site:

```
Unlink {
   if D does exist then
        Log+;
        D-;
        Reply to client (success);
        Unlink execute;
   else
        Reply to client (error);
}
Unlink execute {
   Send “remove bptr” + “info” to Obj. site;
   Wait until reply received or Timeout;
   if reply received (either ACK or NACK) then
        Log-;
}
```

Message diagram:



Object site:

```
Remove back-pointer {
   if back-pointer does not exist
     /*EUNLINK(1)*/
     (“info” not equal to bptr) then
       send NACK back to namespace site;
   else
       Bptr-;
       Send ACK back to namespace;
}
```

Figure 3. Failure-free execution of operation unlink.

Notes:

- The “remove bptr” message carries an “info” record as its payload. This record includes the location details (PID, inode#) of the parent directory as well as the transaction# of the *link* operation that inserted the object into the dentry (available in the dentry). This information is required to ensure that the *unlink* operation removes the back-pointer related to the dentry just removed from the namespace and not the back-pointer created by some other (conflicting) link operation.
- The EUNLINK(*) points are handling special conflict cases, as will be described later in the “multi-operation conflicts” section.

The interesting thing to observe is that the *unlink* operation takes one round trip to be accomplished, but unlike *link*, a reply to the client can be sent as soon as the directory entry is deleted from the namespace. There are also three synchronous I/Os required for the execution of the *unlink*, two of which to the log and another one to remove the back pointer. We delete the directory entry first, after which the back pointer is removed at the object-side, strictly following the ordering rule laid out earlier.

4.3 Recovery protocols

4.3.1 General properties of recovery protocols

Recovery falls into two general classes, namely *conservative* and *aggressive* [7]. In the case of conservative recovery, partial results of the original operations execution are undone in both the namespace and object sites. In the worst-case scenario, conservative recovery unrolls the results of an operation that was successful apart from its last part, the reclaiming of the log record. In the case of aggressive recovery, the aim is to complete a partially performed operation and bring the namespace and object sites in consistent state as far as that operation is concerned.

In this report, we focus on aggressive recovery. Aggressive recovery takes the form of picking up an open log record and re-executing the corresponding operation (without opening another log record, we will see details later). This way, all recovery transactions and messages are indistinguishable from failure-free operations. The advantage is that in multi-operation conflict analysis, we only need to consider potential conflicts among failure-free operations without worrying about interference with recovery processes. Also, we are able to reuse the majority of the routines of the failure-free operations, reducing overall complexity. A side effect of this is that we now provide the stronger semantics, quantitatively, for operation completion (informally speaking, there are better chances that an operation will terminate successfully).

```
on timeout for record r {
    replay link/unlink operation (r);
}
recover host {
    for all records r in log do
        replay link/unlink operation (r);
}
```

Figure 4. Starting recovery process.

Recovery is initiated in either of two ways, as shown in Figure 4:

1) When the communication with a specific host (where object-site operations are pending) timeouts; routine “on timeout for record r” in Figure 4.

2) When the namespace host recovers from total failure (crash); routine “recover host” in Figure 4.

4.3.2 Recovery protocols for *link*

Consider all possible failure points and refer to the three “***X***” failure positions in Figure 2:

1) Point **X1** in Figure 2: then back pointer is not added, steps following “Bptr+” will not be executed either.

2) Point **X2** in Figure 2: back pointer is added, but directory entry is not inserted, steps following “D+” will not be executed either.

3) Point **X3** in Figure 2: directory entry is inserted, but the log is not reclaimed.

Figure 5 describes the algorithm of the recovery process for the *link* operation. The “if” clause distinguishes failures that occur at point **X3** from failures at points **X1** or **X2**. In the latter case we simply re-execute the main body of the original operation. This is performed by calling routine “link execute (r)”, defined in Figure 2, which does not create a new log record. Note that due to the use of back-pointers (which reference parent directories), we are not risking unnecessarily incrementing the link-count of the target object. In other words, due to back-pointers, *link* is an *idempotent* operation. *Link* operations can be safely re-executed in the presence of failures at points **X1** or **X2**.

```
replay link operation (r) {
    if dir-entry D does not exist then
        link execute (r);  // same as in the failure-free case
    else
        Log-;
}
```

Figure 5. Recovery process for operation link.

If failures occur during the recovery procedure (before it is complete), the procedure can be re-initiated without risking causing any inconsistencies.

4.3.3 Recovery protocols for *unlink*

Consider all possible failure points and refer to the three “X” failure position in Figure 3:

1) Point **X1** in Figure 3: server crashes right after the log is opened but before anything else is done.

2) Point **X2** in Figure 3: directory entry is removed but nothing else is done.

3) Point **X3** in Figure 3: back pointer is removed but server crashed before the log is reclaimed.

The recovery procedure is shown in Figure 6. The “if” clause distinguishes failures that occur at point **X1** from failures at points **X2** or **X3**. In the latter case, we simply re-execute the main body of the original operation. This is performed by calling routine “unlink execute”, defined in Figure 3, which does not create a new log record. Note that due to the use of back-pointers, *unlink* also becomes an *idempotent* operation. *Unlink* operations can be safely re-executed in the presence of failures at points **X2** or **X3**, without risking unnecessarily decrementing the link-count of the object.

```
replay unlink operation (r) {
    if dir-entry D exists then
        D-;
        reply to client (success);
        unlink execute;  // same as in the failure-free case
    Log-
}
```

Figure 6. Recovery process for operation unlink.

If failures occur during the recovery procedure (before it is complete), the procedure can be re-initiated without risking causing any inconsistencies.

4.4 Multi-operation conflicts

4.4.1 General properties

4.4.2 *Link/link* conflicts

There are two cases of potential conflicts of two *link* operations: 1) they refer to the same name entry and to the same object; 2) they refer to the same name entry but to different objects.

1. In case (1), the first operation to successfully set the back-pointer at the object is the one that eventually succeeds. This is the case, even if recovery has to take place and either of the *link* operations has to be re-played. When a *link* operation is re-played at the object site and a back-pointer for the referenced entry already exists, an ACK is returned (implying that this operation has completed successfully at the object site) only if the transaction# in the back-pointer matches the transaction# of the operation (*ELINK(2)* in Figure 2). Otherwise, a NACK is returned, because the back-pointer has been already added but that was due to another *link* operation (*ELINK(3)* in Figure 2).

2. In case (2), success depends on which operation enters the directory entry (D+) first. To achieve this, on return of an ACK for a *link* operation, the namespace is checked again for the entry. If the entry already exists (inserted by another *link* operation and is referencing another object), the *link* operation fails and its results have to be undone on the object site. This is the case for *ELINK(1)* in Figure 2. The functionality of the *unlink* operation is re-used for this purpose. Note that the two objects may reside in different hosts and therefore there are no guarantees for the delivery order of the ACKs for the two conflicting *link* operations.

4.4.3 *Unlink/unlink* conflicts

There is only one possible case of two conflicting *unlink* operations. This is when they both refer to the same namespace entry. Irrespectively of the interleaving of executions, only one operation succeeds in removing the namespace entry. In other words, this class of conflicts is easily resolved by serializing the operations at the name entry.

4.4.4 *Link/Unlink* conflicts

There are two possible cases of *link/unlink* conflicts:

1) A *link* operation fails at point X3; before recovery is initiated, an *unlink* operation is started for the same entry.

2) An *unlink* operation fails at points X2 or X3; before recovery is initiated, a *link* operation is started for the same entry (and same object?).

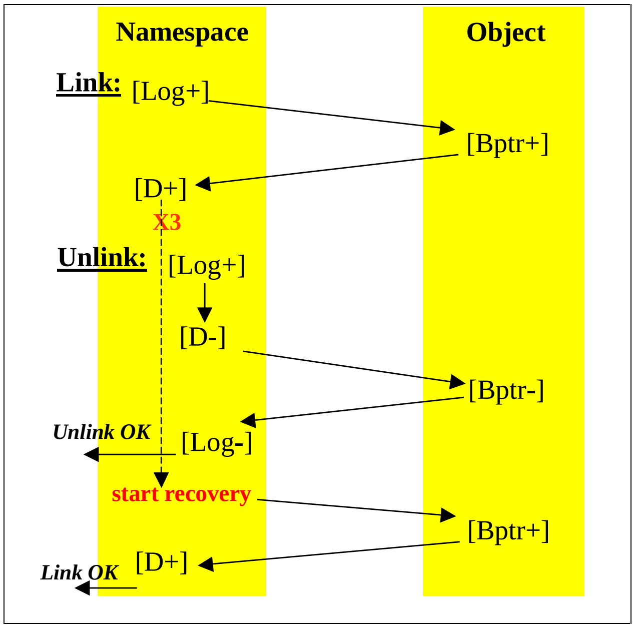


Figure 7. A link/unlink conflict scenario.

The scenario of Figure 7 demonstrates case (1). Operation *link(x)* is performed. The operation successfully adds a back-pointer to the object, an ACK is received back, the entry is added in the namespace, but the log is not reclaimed because of a failure (crash of the namespace site). Before recovery of the *link* operation is initiated, another client does a lookup, discovers the new entry *x* and invokes a conflicting *unlink(x)* operation at the namespace site. The *unlink* is successfully completed and only then the recovery of *link* is initiated. This will re-play all the *link* operation: a back-pointer will be added at the object and an entry inserted in the namespace. Note the namespace is intact, but this may present unacceptable semantic from clients' point of view.

Such pathological scenarios can occur only in the presence of total failure (host crash) of the namespace site. So, to avoid these problems, on recovery, the namespace site does not serve any new operations until all pending operations in the log are re-started. In the example of Figure 7, the *unlink* operation is not started until the partially complete *link* operation has already been started. Our *link* recovery procedure simply reclaims the log. After which the *unlink* operation can proceed as usual.

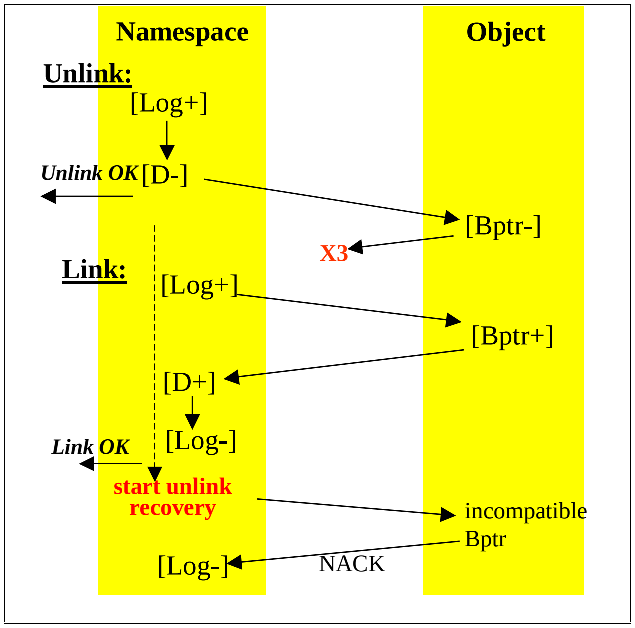


Figure 8. An unlink/link conflict scenario.

The scenario of Figure 8 demonstrates case (2). Operation *unlink(x)* is performed. The operation successfully removes the back-pointer at the object site but no ACK is received back. In the meanwhile, before recovery is initiated, a conflicting operation *link(x)* is started at the namespace site. The *link* is successfully completed by adding a (new) back-pointer at the object site and inserting an entry at the namespace site. After that, the recovery process is initiated for the *unlink* operation and tries to re-do the object site execution. This cannot compromise the back-pointer that the successful *link* created, because the transaction # of the back-pointer is different than the transaction # of the *unlink* operation. So, the re-execution of *unlink* fails at the object site and a NACK is returned back to the namespace, where the log is reclaimed. This scenario demonstrates another use of the back-pointers at the object site for guaranteeing the integrity of the namespace.

4.5 DiFFS rename protocol and recovery

In DiFFS, *rename* is a compound operation made up by two primitives:

- *Link(fh_to_dir, fh_obj, name),* and then
- *unlink(fh_from_dir, name)*

These two primitives and their recovery procedures have been described extensively. Therefore, the main issue to consider is the recovery procedure for the composite operation. The compositional approach followed here for the implementation of *rename* simplifies the design of its protocol and recovery process.

This compositional approach may result in some performance loss against an alternative monolithic implementation of the operation. However, we believe that the reduction of protocol complexity justifies such a loss, especially given that *rename* is an operation that occurs rarely in typical workloads.

The recovery procedure is as follows. Note that failure/success here refer to the outcome of the composite operation. We have the option to either abort by rolling back or to replay *rename* by re-starting the process:

- If *link* fails, there are two sub-cases:
    - *Link* is later verified as successful (i.e., the dentry is inserted into the *to_dir*): Either do a *unlink(fh_to_dir, name)* to roll back and then abort, or continue at the point of *unlink(fh_from_dir, name)*.
    - If *link* fails and the dentry is verified as not being inserted. Either restart from the beginning or abort.
- If *link* succeeds, and *unlink* fails, there are two sub-cases:
    - *unlink* is later verified as success. Simply exit.
    - *unlink* is later verified as failure. Either do a *unlink(fh_to_dir, name)* to roll back and abort, or repeat the *unlink(fh_from_dir, name)* to continue.
- If both steps succeed, *rename* is completed.

4.6 DiFFS relink protocol and recovery

Relink is an operation required in DiFFS to support object migration. In the case of migration, a new instance of the object is created, and the namespace needs to be updated to point to the new instance, while releasing the reference to the old instance.

One way to implement *relink* is similar to that used for *rename*, i.e. design it as a compound operation:

- *unlink(fh_dir, old_obj_name)*, followed by
- *link(fh_dir, fh_new_obj)*

As in the case of *rename*, we are not concerned with each primitive's recovery. *Relink* is executed as if it was a user-level process and the failure/success carries a different meaning. Note that unlike *rename*, after the first step is completed successfully, we do not have the option of rolling back, because the object may have now been garbage collected already. The recovery process for the *relink* operation is as follows:

- If *unlink* fails, there are two sub-cases:
    - *Unlink* is later verified as successful (i.e., the dentry is removed from the directory): Either do a *link(fh_dir, fh_old_obj)* to roll back and then abort, or continue at the point of *link(fh_dir, fh_new_obj)*.
    - If *unlink* fails and the dentry is verified as still existing. Either restart from the beginning or abort.
- If *unlink* succeeds, and *link* fails, there are two sub-cases:
    - *Link* is later verified as success. Simply exit.
    - *Link* is later verified as failure. Repeat the *link(fh_dir, fh_new_obj)* to continue.

- If both steps succeed, *relink* is completed.

An alternative, more efficient implementation is the following. When an object is migrated, its associated back pointers are moved together with the object's content to the new location. In this case, the *relink* operation is simply an atomic transaction updating the directory entry and is local to the namespace site.

5. Implementation and optimizations

This section is dedicated to various alternatives for efficient implementation of the proposed protocols.

5.1 Optimizing the Create operation in DiFFS

Create is one of the most frequent metadata operations in any kind of filesystem. In DiFFS, as mentioned briefly in Section 3.1, *create* is a compound operation made up of obtaining an object and then linking it into DiFFS namespace. To speed up the *create* operation, each partition preallocate idle objects (files) and advertise them to all other partitions.

To eliminate any race conditions, sets of objects advertised to different partition are disjoint. Each partition server has a map that records what idle objects are available from other partitions. This map can be simply a file local to each partition server. If a partition is to *create* an object resides from another partition but its map indicates that it have used up all that were advertised to it, it requests the target partition for more. This process can be proactive as well. When each partition server finds that its repository of idle objects are running low, it goes ahead and creates more to meet the projected demand and subsequently advertise them to other partitions.

5.2 Optimizing the link and unlink primitives

From the previous discussion, we see that cross-partition operations where namespace and object change do not reside in one place carry their price tag. There are 3 synchronous I/O to be performed at the object and namespace side altogether.

The synchronous I/Os are essentially of two types: the intention log (at either the object or the namespace site) whose failing to be reclaimed triggers garbage collection process, and the additional bookkeeping such as back pointers necessary to facilitate the detection of garbage. We look at them in turn.

5.2.1 Intention Log

The purpose of the intention log is to facilitate garbage detection and collection in face if *any* failure. As such we have to make sure that the log is persistent on the storage before the very first outbound message. Consider *link*. If an object has been created and the insertion message to the namespace left before the log has reached stable storage, a crash at the object-side will leave us no clue as how to start the garbage collection upon recovery. The case for *remove* is similar.

Writing a single log is not expensive since log updates are sequential. Constantly writing and deleting log record are costly, however. This is so because this pattern simply breaks the sequential access pattern to the disk. To solve this problem, we reclaim log entry in a lazy fashion. When an operation closes, we write to the intention log a record indicating which transaction is closed. This way, the intention log is efficiently updated sequentially at all time. In addition, In addition, a background process creates checkpoint periodically. When checkpoint occurs, a new log is used instead. In parallel, the background process scans through the old intention log to collect entries and identify if any intention log is left open and, if so, engage the garbage collection process.

Note we can afford to do the reclaiming late precisely because our overall execution order guarantees that the namespace is never corrupted. Before an intention log is successfully reclaimed, the object to which the log belongs is still in question as far as its legitimacy in the namespace is concerned. So how long can we delay the reclamation of the log? In fact, there is no real constraint. This is so because if the object is garbage, it's an orphan and therefore we can pick it up any time we want.

5.2.2 Back pointers

Back pointers are object-side properties. Therefore by definition it can be completely out of the critical path for namespace *unlink* operations.

Back pointer addition is on the critical path for *link*. Pure *link* operation is not frequent. More often is DiFFS *link* gets invoked as a result of "create." Compare with NFS create, however, DiFFS creates goes faster because it's simply a *link* with a pre-allocated idle file. In contrary, traditional NFS create will involve multiple disk IOs to modify the physical filesystem resource, read and change bitmap etc. None of these IOs are necessary with pre-allocated files. Each partition server can allocate idle files periodically and assign quotas to other partition servers and advertise them. In this way, the chance of running out idle, remove files when "create" is to be carried out is minimal.

Back pointer can be implemented in a number of different ways:

1. Per-object shadow file.

   In this case, assume */diffs/foo* is the object, then */diffs/shadow/foo* can be the shadow file that contains the back pointers. The shadow file is created locally when the object (*/diffs/foo*) is created.

2. Consolidated shadow file.

   Per-file shadow/back pointer file can be consolidated into one, or multiple, shadow files. The inode # of */diffs/foo* can be hashed by a deterministic function to locate a corresponding shadow file where its back pointer is kept. Modification to this shadow file, in the form of insert or delete back pointers, are journaled to keep the integrity of the content. DiFFS' principle of implementing directories as files require this journaling service to be in place already.

In both cases above, adding or removing a back-pointer involves a synchronous I/O to the shadow file. In the case of *link*, this synchronous I/O is on the critical path of the operation. Optimizing the performance of back-pointer operations is thus an important implementation issue. We suggest the use of a journaling service (the same used for the logging on the namespace site) for improving performance. Back-pointer additions/removals are recorded synchronously to a record of a log file at the object site. Log/journal files are fine tuned for very efficient synchronous I/Os. The contents of the record (addition or removal of a back-pointer for an object) are then propagated to the shadow file (per-object or consolidated) in a lazy manner, outside the critical path of operations.

6. Comparison with other approaches

6.1 2-phase-commit

Other research projects [9, 11] suggest using traditional transaction mechanisms for updates of the global namespace. In particular, *2-phase-commit* (2PC) is recommended for the implementation of operations that span more than one site. In this report, we make two claims:

1. 2-phase-commit introduces an overhead to the failure-free execution of operations. Failure-free operation overhead is avoided by the lightweight protocols proposed here.
2. 2-phase-commit may lock system resources for extensive periods of time reducing the concurrency of operation execution in the system. Our protocols do not lock any system resources and allow for maximum concurrency of operation execution, as discussed in section 4.4.

In order to justify these claims, consider the implementation of the *link* and *unlink* operations, using a typical 2PC protocol. The protocol coordinator can be either the client that initiates the operation or one of the sites that are involved in the execution of the operation. To facilitate a straightforward comparison of the 2PC protocols with our protocols, in the following diagrams, the coordinator of 2PC is chosen to be the name-space site.

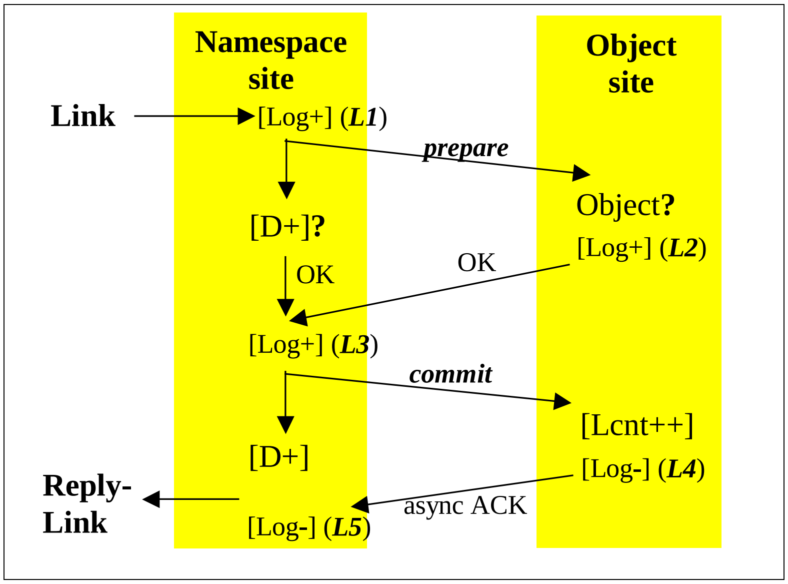


Figure 9. An implementation of link using 2-phase-commit.

Figure 9 illustrates a 2PC-based implementation for operation *link*. Figure 10 illustrates a 2PC-based implementation of operation *unlink*. As it is the case with the protocols of Figure 2 and Figure 3, there is one round-trip over the network on the critical path of each operation. In the case of 2PC, the use of back-pointers is not necessary. Link-counts at the object site suffice, since the target object properties are kept “locked” between phase-1 (delivery of “prepare”) and completion of phase-2. Back-pointers are not required for scenarios of conflicting *link* and/or *unlink* operations (similar to those of Figure 7 and Figure 8). It is sufficient to make sure that on recovery of the namespace site, pending transactions are restarted before handling any new client requests. The use of link-counts instead of back-pointers does not improve the performance of the 2PC protocols; updating the link-count of an object still requires a synchronous IO. In addition, the 2PC protocols have the following disadvantages in comparison to the protocols proposed in this report, as discussed below.

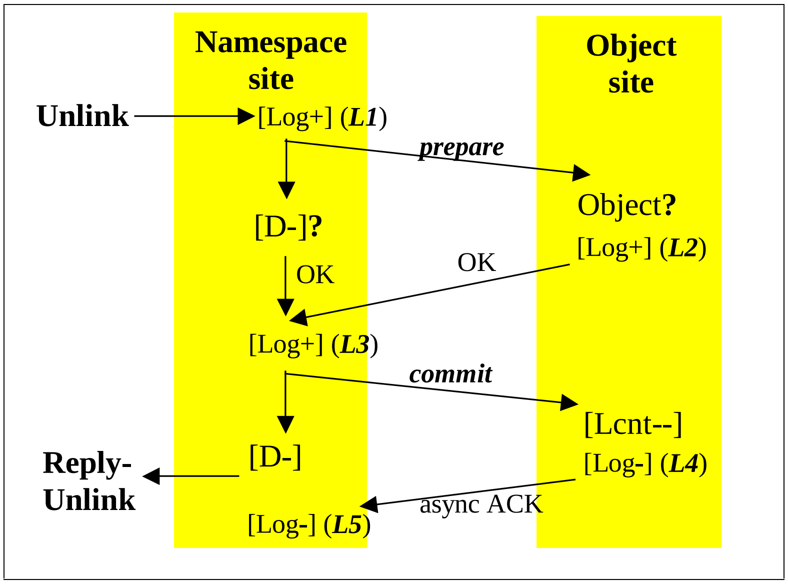


Figure 10. An implementation of unlink using 2-phase-commit.

- The 2PC protocols have *three* accesses to log files in the critical path of the operations, as opposed to just one in our protocols. The log record at the object site (L2) is required to make sure that the back-pointer is not changed by another transaction while the outcome of the current transaction is pending. The second log record in the namespace site (L3) is required to ensure that the outcome of the phase-1 of 2PC (“prepare”) is recorded, so that the operation can be re-played if a failure occurs during phase-2 of 2PC.

- 2PC protocols require *two* messages over the network for the back-pointer to be updated, as opposed to just one message in our protocols.

- In the presence of host or communication failure anywhere up to log point L3, the partial results of the 2PC execution are un-done on recovery. In our protocol, the operation results are re-done as long as the initial log record has been created in the namespace. The latter difference does not affect the overall failure semantics as perceived by the clients, but in practice, the protocols of Figure 2 and Figure 3 in combination with the recovery process of Figure 5 and Figure 6 provide better probability for the successful implementation of the operations in the presence of failures.

- In the case of the 2PC protocol, when a client receives back a reply to its *link* or *unlink* operation, we are guaranteed that the namespace has been updated ([D+] or [D-]), but we are *not* guaranteed that the link-count of the object has been updated (incremented or decremented respectively). The latter would require a second synchronous round-trip interaction—an ACK to the commit operation, before a reply is sent back to the client. In the case of our algorithm, the stronger guarantee is provided without requiring a second round-trip latency in the critical path of the operation.

- Another issue with the 2PC protocols is related to resource management at the object site. After point L2, the log record at the object site has to be kept until completion of phase-2 (commit or abort). However, in the presence of failure (e.g. network partitioning), there is no guarantee for the time it takes for the phase-2 message to be delivered (if delivered at all). So, the issue is how long has a log record to be kept and whether it can be reclaimed if no phase-2 communication is received. This is an important problem, because the related object link-count is in fact locked for the duration of the log record—no other transaction can access it to change its value (even from another namespace site).

The following table summarizes the comparison between 2-phase-commit protocols and the protocols proposed in this paper.

	Required object attributes	**Total log accesses**	**Log accesses in critical path**	**Total round trips**	**Round trips in critical path**	**Recovery approach**	**Degree of concurrency**
2PC	Simple: Link counter	Link: 5 Unlink: 5	Link: 3 Unlink: 3	Link: 2 Unlink: 2	Link: 1 Unlink: 1	Conservative	Low
DiFFS	Complex: Back pointers	Link: 2 Unlink: 2	Link: 2 Unlink: 1	Link: 1 Unlink: 1	Link: 1 Unlink: 0	Aggressive	High (with conflict resolution)

7. Related Work and Conclusions

DiFFS is an architecture designed to provide a widely distributed file service [1]. Much of DiFFS scalability and flexibility is due to its partition-based approach to storage distribution. This report proposes robust and lightweight cross-partition protocols for operations in the DiFFS namespace.

The report claims that the cross-partition protocols of DiFFS are instances of a more general problem: *how to maintain consistent global namespace over a collection of distributed objects*. We show that all namespace operations can be decomposed into just two primitive operations: *link* and *unlink*. The main issue with cross-partition operations is how to maintain namespace consistency even in the presence of communication and/or host failures. We prove that by imposing a specific execution order to these operations, we can reduce the problem to instances of “orphan” objects in the system; the global namespace is never corrupted. The report discusses protocols and recovery procedures for the two primitive operations, taking under consideration all possible scenarios of conflicts. We claim that the proposed protocols impose minimal overhead to failure-free execution and, in general, they are more lightweight than traditional atomic commitment protocols. To justify this claim, we conduct a detailed comparison with protocols that are based on 2-phase-comit. We demonstrate that the proposed protocols are strictly better in all critical performance factors (communication round-trips, synchronous I/O). In addition, they facilitate higher concurrency for operation execution; they provide better probabilistic characteristics for successful completion of cross-partition operations in the presence of failures. The price for those desirable characteristics is that objects must be annotated with additional properties than in traditional file systems. In particular, the back pointers to all “parent” directories must be associated with every object in the system.

Slice is a research system from Duke University [11], which also follows a partition-based approach to achieve scalability for a distributed file service. Slice uses two possible mechanisms for object distribution, MKDIR SWITCHING and NAME HASHING, the details of which are outside the scope of this report. Irrespectively of the distribution mechanism used, namespace operation in Slice may cross more than one site. Slice suggests using 2-phase-commit for logging and recovery across sites. However, the performance implications of such protocols are not clear from the published results for Slice.

Archipelago is another research system from Princeton University [9, 14] that suggests 2-phase-commit as the mechanism to guarantee namespace consistency in a distributed file system. Archipelago proposes the deployment of a distributed file system across the nodes of a cluster of mutually trusted file servers. A variety of physical file systems may be supported in the cluster. The basic principle of the system is to distribute files arbitrarily across the cluster servers using hashing of object names (pathnames). Namespace objects (directories) are also distributed using the same mechanism, but, in addition, they are replicated across the cluster. The degree of replication of a directory depends on the usage of the directory—the more it is used, the higher its degree of replication is. The current Archipelago prototype uses 2-phase-commit for the implementation of cross-server operations, such as *CreateDir* and *RemoveDir*. The difference with DiFFS is that in this case, 2-phase-commit is used not only for namespace consistency but also for achieving consistency between the replicas of the directory.

8. References

[1] C. Karamanolis, M. Mahalingam, D. Muntz, and Z. Zhang, “DiFFS: a Scalable Distribtued File System,” HP Laboratories, Palo Alto, Technical Report HPL-2001-19, January 24 2001.

[2] K. Preslan, A. Barry, J. Brassow, R. Cattelan, A. Manthei, E. Nygaard, S. V. Oort, D. Teigland, M. Tilstra, M. O'Keefe, G. Erickson, and M. Agarwal, “Implementing Journaling in a Linux Shared Disk File System,” presented at 8th NASA Goddard Conference on Mass Storage Systems and Technologies, 2000.

[3] G. Gagner and Y. Patt, “Metadata Update Performance in file Systems,” presented at USENIX Symposium on Operating Systems Design and Implementation (OSDI), 1994.

[4] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 1st ed: O'Reilly, 2001.

[5] B. Callaghan, *NFS Illustrated*: Adison-Wesley, 2000.

[6] P. Leach and D. Perry, “CIFS: A Common Internet File System,” *Microsoft Interactive Developer*, 1996.

[7] P. A. Bernstein, N. Goodman, and V. Hadzilacos, “Concurrency Control and Recovery in Distributed Databases,” , 1987.

[8] J. N. Gray, “Notes on Database Operating Systems,” in *Operating Systems: An Advanced Course*, 1978.

[9] M. Ji, E. W. Felten, R. Wang, and J. P. Singh, “Archipelago: An Island-Based File System for Highly Available and Scalable Internet Services,” presented at 4th USENIX Windows Systems Symposium, 2000.

[10] B. White, M. Walker, M. Humphrey, and A. Grimshaw, “A Secure and Scalable File System Infrastructure,” Computer Science Department, University of Virginia, Charlotesville, Tecnical Report 2000.

[11] D. Anderson, J. Chase, and A. Vadhat, “Interposed Request Routing for Scalable Network Storage,” presented at Usenix OSDI, San Diego, CA, USA, 2000.

[12] D. Cheung and T. Kameda, “Site-Optimal Termination Protocols for a distributed Database under Networking Partitioning,” presented at 4th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, 1985.

[13] D. Skeen, “Nonblocking Commit Protocols,” presented at ACM SIGMOD, 1981.

[14] M. Ji and E. W. Felten, “Design and Implementation of an Island-based File System,” Department of Computer Science, Princeton University, Technical Report 610-99, October 1999.