Designing a Robust Namespace for Distributed File Services

Zheng Zhang and Christos Karamanolis

Hewlett-Packard Laboratories
1501 Page Mill Rd, Palo Alto, CA 94304, USA
{zzhang,christos}@hpl.hp.com

Abstract

A number of ongoing research projects follow a partition-based approach to provide highly scalable distributed storage services. These systems maintain namespaces that reference objects distributed across multiple locations in the system. Typically, atomic commitment protocols, such as 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. Furthermore, they use conservative recovery procedures and may considerably restrict the concurrency of overlapping operations in the system.

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

1. Introduction

These A number of ongoing research projects follow a partition-based approach to achieve high scalability for access to distributed storage services. They address the inherent scalability problems of traditional cluster file systems, which are due to contention for the globally shared resources. Instead, they partition the storage resources in the system; shared access is controlled on a per-partition basis. A major requirement of all these systems is to maintain namespaces that reference objects that reside in multiple partitions. Typically, the namespace in these environments is distributed itself.

For example, DiFFS, an experimental distributed file service currently under development in HP Labs, follows the partitionable approach [10]. File system objects, such as files and directories, can be placed in different partitions, which may be geographically distributed. Each partition is controlled by one *partition server*, which coordinates operations that may affect the state of the resources it owns (allocate or de-allocate blocks, for example). Objects are placed and may be migrated and/or replicated according to locality of access, type of content, reliability and numerous other parameters. This policy-driven distribution of objects gives DiFFS the flexibility required for a wide range of deployment options.

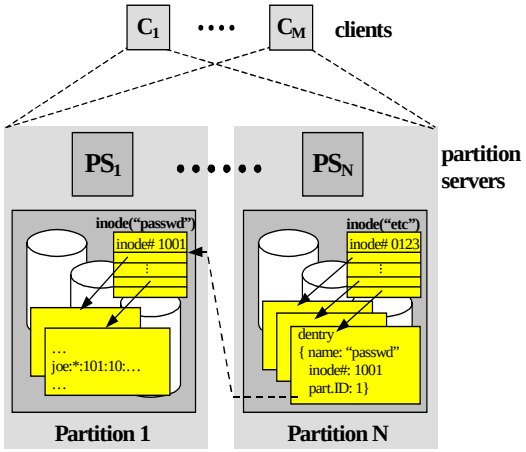


Figure 1: Cross-partition references in a DiFFS namespace.

The namespace in DiFFS is implemented by means of directories that may also be placed in any partition, not necessarily the same with their children in the namespace, as depicted in Figure 1; a file with inode number 1001, which resides in partition 1, is referenced with the name "*passwd*" from a directory in partition N. Other systems that follow a similar approach include *Slice* from Duke University [1] and *Archipelago* from Princeton [9].

While the intention of this report is to investigate protocols for building robust namespaces in the context of DiFFS, the problem is more generic. It can be broadly stated as: *maintaining a consistent namespace over a collection of distributed objects efficiently*.

Changes to the global namespace take the form of one of two classes of operations: *link*, which inserts a reference to a possibly newly created object and *unlink*, which removes a reference to an object. Any of the above operations potentially spans more than one site in a distributed system. The site containing the directory (namespace object) and the one containing the referenced

object can be physically apart. Slice and Archipelago use 2-phase commit to implement distributed namespace operations [1, 9]. Atomic commitment protocols are known to have a high computational cost [16, 5]. They impose a high overhead to failure-free execution, due to synchronous logging in the critical path of the operations. Additionally, they lock system resources across all the sites involved in the protocol for the duration of the multi-phase protocol execution, thus worsening the problem of contention for resources, e.g., free block lists and block allocation maps. Lastly, atomic commitment protocols follow a conservative approach for recovery from failure; in the presence of failure, incomplete operations are typically aborted.

This paper proposes a set of lightweight protocols for implementing the two main classes of operations in distributed namespaces. The main requirement for the protocols' design is to minimize the overhead imposed to failure-free execution. This is achieved by reducing the number of synchronous I/O in the critical path of the operation execution. Additionally, they avoid distributed resource locking; serialization of operations on each partition suffices. The protocols are robust against both communication and host failures. They use aggressive recovery techniques to re-play incomplete operations, in most failure scenarios. The protocols are compared with typical 2-phase commit implementations and are shown to be superior in all critical performance factors: communication round-trips, synchronous I/O, operation concurrency. These benefits come at the price of additional data structures associated with the distributed objects: *back pointers*—references back to the namespace.

The remaining of the paper is organized as follows: Section 2 provides a concise definition of the problem space. Section 3 outlines the system model assumptions for the proposed protocols. Section 4 is the core of the paper describing the details of protocols for distributed namespace operations, including recovery and conflict resolutions issues. An analysis of alternative 2-phase commit (2PC) implementations is given in section 5; it is shown that the protocols proposed in this paper outperform their 2PC counterparts in all critical performance factors. The paper is concluded with discussion of related work in section 6 and final remarks in section 7.

2. Problem Abstraction

A *namespace* provides a mapping between names and physical objects in the system (e.g., files). Usually, a user refers to an object by a textual name. The latter is mapped to a lower-level *reference* that identifies the actual object, including location and object identity. The namespace is implemented by means of directories, special files that are persistent repositories of ⟨*Name, reference*⟩ pairs. The namespace may be distributed—directories may be placed in any location in the system. In this context, the requirement for consistency of the namespace can be formalized in terms of four properties, as depicted in Table 1.

1. ***One name is mapped to exactly one object[1].***
2. ***One object may be referenced by one or more names.***
3. ***If there exists a name that references an object, then that object exists.***
4. ***If an object exists, then there is at least one name in the namespace that references it.***

Table 1 . Requirements for namespace consistency.

The two fundamental namespace operations are *link* and *unlink*:

link: a new reference, pointing to a possibly newly created object, is inserted into the namespace.

unlink: a reference pointing to an already existing object is removed from the namespace. If all references to an object are removed, the object itself is garbage collected.

Other namespace operations can be either reduced to or composed by these two primitives. For more details refer to [17]. As an example, Table 2 shows how NFS namespace operations are mapped to these two primitives.

File Service operation	***Namespace primitive(s)***
create/mkdir	obtain a new object + ***link***
link	***link***
remove/rmdir/unlink	***unlink***
rename	***link*** *(to_dir)*+ ***unlink****(from_dir)*

Table 2: Using the two fundamental namespace primitives.

This paper is based on the observation that by imposing a certain order on the execution of namespace operations, we can guarantee that all possible inconsistencies in the namespace are reduced to instances of "orphan" objects. An *orphan* is an object that physically exists in the system, but is not referenced by any name in the namespace. The required execution order can be generalized to the following three steps:

[1] We consider that replicas of an object correspond to one logical object.

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.

The above principle applies to every distributed namespace operation [17]. In particular, the results of the ordering principle in the case of the two fundamental primitives are as follows:

link: add the reference to the namespace at the last stage of the execution.

unlink: remove the reference from the namespace is the very first stage of the execution.

In either case, the only possible inconsistency due to failures is that the target object is not referenced by any name in the namespace. We claim that handling orphan objects (violation of property 4 in Table 1) is easier than handling invalid references (violation of property 3 in Table 1).

3. System Model and Failure Assumptions

The design and correctness of the protocols discussed in this paper is based on the following assumptions about the failure model in the system:

- 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. For example, the use of TCP as the communication protocol between hosts guarantees this property.
- Consistency of the object-store at each partition is guaranteed, despite failures. This property is ensured by mechanisms of the physical file system, such as *journaling* [14], *soft updates* [6] or recovery procedures (*fsck*) [3].

The required behavior by the client that performs operations in the namespace is "at-most-once" semantics. This is consistent with the semantics provided by traditional client-server distributed file systems, such as NFS [4], CIFS [12] and AFS [8]. When the client application receives a reply to a request to the file service, it is guaranteed that the request has been performed exactly once. If no response is received, the client cannot know whether the request was performed or not.

4. Protocols

The execution of the *link* and *unlink* operations is initiated by a client which invokes a request to the site where the affected directory resides (namespace site). The requests are parameterized with the data required for the execution of the corresponding protocols, as shown in Table 3. The rest of this section introduces the key data structures used in the protocols and then describes in detail the protocols and corresponding recovery procedures.

• **link(P,N,O)** • **unlink(P,N)**	**P**: the parent directory's reference: ⟨*site, inode#*⟩ (*site* is the namespace site where the request is sent). **N**: the *name* assigned to the object (string). **O**: the object's reference: ⟨*site, inode#*⟩.

Table 3: The link and unlink operation requests.

4.1. Data structures

Directory

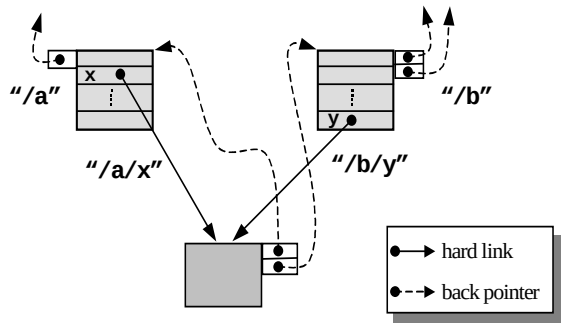
It is a special object in the file service, which is used as a repository of ⟨*name, reference*⟩ pairs, called *directory entries*. Directories are accessed by lookup procedures to locate objects by name. An object *reference* consists of two parts: 1) the *site* where the object resides, and 2) a unique *identifier* of the object in that location, such as the *inode number*. For the protocols described here, the directory entry contains a monotonically increasing number, *generation #*, that uniquely identifies a specific name to object binding.

object name	***object reference***		***generation#***
	site	*inode#*	

Table 4. Directory entry structure.

Back pointer

In traditional file systems, objects are assigned a property known as "link-count". This is an integer representing the number of references (hard-links) to the object in the namespace. For the protocols presented here, the notion of link-count is extended by means of "back pointers", i.e. references back to the parent directories of the object, as shown in Figure 2(a). The back pointer consists of two parts (Figure 2(b)): 1) the *reference* (site and inode#) of the parent directory; 2) the *name* and *generation#* of the corresponding link. Back pointers are required to guarantee namespace consistency, in the presence of conflicting operation execution and/or operation recovery, as discussed in later in this section. Back pointers can be either part of the inode structure or be implemented as separate files. A more detailed discussion on back pointers can be found in [17].



(a) Example of back pointers.

Parent dir reference		*link name*	*link generation#*
site	*inode#*		

(b) Back pointer structure.

Figure 2. Hard links and back pointers.

Log record

The proposed protocols make use of *intention logs* to record execution state in persistent storage. The structure of a log record is shown in Table 2. The fields refer to the name to object binding that is to be created or removed, in the case of *link* and *unlink* respectively. The creation and reclamation of a log record mark the beginning and the end of the execution of an operation. A open log record implies that the operation has not been completed. In the case of recovery from failure, the contents of the log record are used for the recovery procedure.

Operation type (link/ unlink)	*namespace object (directory) ref*		*object name*	*object reference*		*genera-tion #*
	site	*inode#*		*site*	*inode#*	

Table 5 . Log record structure.

4.2. Failure-free protocols

There are two sites involved in the execution of protocols for *link* and *unlink*: the *namespace site*, where the referencing directory resides; the *object site*, where the referenced object resides. In the general case, these two sites are remote from each other and the protocol execution involves message transmission between the two. Table 6 provides a legend for the message diagrams used to describe protocol execution, in the following paragraphs. In order to keep the discussion simple, all disk operations other than log accesses are assumed to be synchronous.[2]

[2] In fact, the requirement for such operations is less strict: to be performed on stable storage, before the next log operation or message transmission in the flow of control.

[act]	An atomic operation on stable storage.
→	A communication message across sites.
Xn	A potential failure position. A failure at this point may affect anything after the immediately preceding atomic action.
[Log+/-]$_{S/A}$	Creation (or update) / reclamation of a log record; synchronous (force write) or asynchronous (lazy), respectively.
D+/-	Creation / removal of a ⟨name, reference⟩ pair (directory entry).
Bptr+/-	Creation / removal of a back pointer.

Table 6. Legend for the protocol message diagrams.

Link

The protocol for *link* is depicted in Figure 3. The execution follows the ordering principle laid out in section 2—changes on the object are performed first (adding a back pointer) followed by changes in the namespace (adding the new directory entry). The execution is initiated with the synchronous creation of a log record in the namespace site. The protocol requires one message round trip between the namespace and object sites. It involves two synchronous accesses to storage, one on the object site to add the back pointer and one on the namespace site to create the directory entry. Additionally, it requires two accesses to the log for the creation and reclamation of a record, with the former being synchronous. That is, a reply to the client can be sent as soon as the directory entry (*D* in pseudo-code) is added. The "add bptr" message carries a payload, which is used to create the back pointer on the object site.

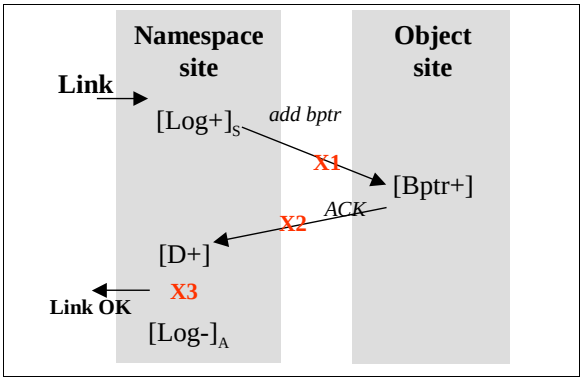


Figure 3. Failure-free execution of operation link.

Namespace site:

```
Link(P,N,O) {
  if dir-entry D does not exist then
      r := {"link",P,O,N,new_gen#());
      Log+S(r);
      Link_body(r);
  else
      Reply to client (error);
}

Link_body(r) {
  info := r;
  send "add bptr"+ info to Obj site;
  Wait until reply received or Timeout;
  if reply=ACK then
      if D does not exist then
          D+;
          reply to client (success);
          Log-A(r);
      else /* LL(1) */
          unlink execute; //as in unlink op
          Log-A(r);
          reply to client (error);
  else if reply=NACK then
      reply to client (error);
      Log-A(r);
}
```

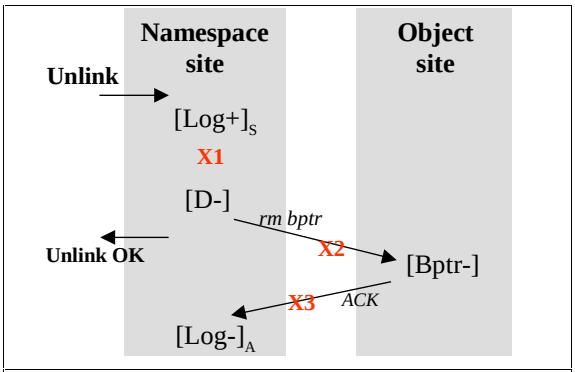
Object site:

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

Figure 3 (cont'd). Failure-free execution of operation link.

Unlink

Figure 4 describes the protocol for *unlink*. Again, the execution follows the ordering principle of section 2—the reference is first removed from the namespace (directory), before this fact is reflected on the referenced object (back-pointer removal). The log record that is synchronously created on the namespace site contains all the necessary information for recovery execution in the case of failure. The protocol for *unlink* requires one message roundtrip. However, unlike *link*, a reply to the client can be sent as soon as the reference has been removed from the namespace and before an ACK is received from the object site. The operation requires two accesses to the log, only the first one (record creation) being synchronous.



Namespace site:

```
Unlink(P,N) {
  if dir-entry D does exist then
      r := {"unlink",P,D.O,N,D.gen#);
      Log+S(r);
      D-;
      Reply to client (success);
      Unlink_body(r);
  else
      Reply to client (error);
}

Unlink_body(r) {
  info := r;
  Send "remove-bptr"+info to Obj site;
  Wait until reply received or Timeout;
  if reply received(ACK or NACK) then
      Log-A(r);
}
```

Object site:

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

Figure 4. Failure-free execution of operation unlink.

4.3. Recovery protocols

Recovery techniques for traditional transactional protocols fall into two general classes, *conservative* and *aggressive* [2]. In our case, conservative recovery implies that the partial results of the original operation 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 reclamation of the log record. With aggressive recovery, the aim is to complete a partially performed operation and bring the namespace and object sites in mutually consistent states, as far as that operation is concerned. This paper focuses on aggressive recovery techniques aiming at stronger quantitative semantics for operation completion.

On recovery from failure, either host or communication, the namespace site traverses the log for records indicating incomplete operations. The basic idea behind the recovery processes proposed here is to re-execute those operations without creating a new log record and, in the case of *link*, without generating a new generation#. In this way, operation re-execution and the corresponding messages are indistinguishable from their failure-free counter-parts. The main advantage of this approach is that, in multi-operation conflict analysis, one needs to consider potential conflicts among only failure-free operations, without explicitly considering recovery processes.

```
on_timeout_for_record (r) {
      replay_link/unlink (r);
}

total_recovery {
      for all records r in log do
            replay_link/unlink (r);
}
```

Figure 5. Starting recovery process.

Recovery is initiated by the namespace site, in either of two ways:

- When the communication with a specific host (where object-site operations are pending) timeouts; implemented by routine "on_timeout_for_record(r)" in Figure 5.
- When the namespace site recovers from a crash; implemented by routine "total_recovery" in Figure 5.

Recovery protocol for link

There are three possible points where the execution of the *link* protocol may be interrupted due to failures, as shown in Figure :

- Point ***X1***: Just the log record is created; the back pointer has not been added and no other following step has been executed.
- Point ***X2***: The back pointer is added, but the namespace has not been updated.
- Point ***X3***: Both the object and namespace are updated, but the log record has not been reclaimed.

Figure 6 describes the recovery protocol for the *link* operation. The "if" clause distinguishes failures that occur at point ***X3*** from failures at ***X1*** or ***X2***. In the latter case, the main body of the link operation. ("Link_body (r)" defined in Figure) is re-executed, *without* creating a new log record. If the failure occurred at point ***X3*** ("else" clause), the recovery protocol just reclaims the log record of the original execution; the rest of the operation has been completed.

If objects were annotated with traditional link-count attributes, the above procedure would risk unnecessarily incrementing the link-count of the target object. The use of back pointers, which uniquely reference parent directories and the operation that created the link, guarantees that *Link* operations can be safely re-executed in the presence of failures. Even if failures occur during the recovery procedure, the procedure can be re-initiated without risking causing any inconsistencies at either the object or the namespace site.

```
replay_link (r) {
    if dir-entry D does not exist then
        Link_body (r);
        // same as in the failure-free case
    else
        Log-A;
}
```

Figure 6. Recovery process for operation link.

Recovery protocol for unlink

There are three possible points where the execution of the *ulink* protocol may be interrupted, as shown in Figure 4:

- Point ***X1***: The log is created but no other step has been performed.
- Point ***X2***: The namespace is updated, but the back pointer has not been removed at the object site.
- Point ***X3***: Both the namespace and the object (back pointer) are updated, but the log has not been reclaimed.

```
replay_unlink (r) {
   if dir-entry D exists && gener# matches then
       D-;
       reply to client (success);
       Unlink_body (r);
       // same as in the failure-free case
       Log-A;
}
```

Figure 7. Recovery process for operation unlink.

Figure 7 describes the recovery protocol for *unlink*. The "if" clause distinguishes failures that occur at point ***X1*** from failures at ***X2*** or ***X3***. In the latter case, the main body of the unlink operation ("Unlink_body(r)" defined in Figure 4) is re-executed, *without* creating a new log record. If the failure occurred at point ***X1***, then only the log record is reclaimed. Again, the use of back pointers guarantees that *unlink* operations and the recovery protocol can be safely re-executed in the presence of failures, without risking inconsistencies in the system.

4.4. Multi-operation conflicts

A major requirement for the design of the protocols proposed here is to facilitate maximum concurrency of

operation execution. The idea is *not* to lock resources across the involved sites for the duration of the protocol execution, as it is the case with transactional protocols. As a result, we have to explicitly address issues of conflicting operations in the system.

Link/link conflicts

There are two cases of potential conflicts of *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.

In case (1), the first operation to successfully set the back pointer is the one that eventually succeeds, even if recovery takes place and either of the *link* operations is re-executed. When a *link* operation is executed at the object site and a back pointer for the referenced name entry already exists, one of two cases applies:

i) The generation# in the back pointer matches the generation# in the payload of "add bptr" (*LL(2)* in Figure). This implies that this operation has already been completed successfully at the object site. An ACK is returned to the namespace.

ii) The two generation#'s do not match. A NACK is returned indicating that the back pointer has been already added by another *link* operation (*LL(3)* in Figure).

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

Unlink/unlink conflicts

The only possible case of conflicting *unlink* operations occurs when they refer to the same namespace entry. Irrespectively of the interleaving of executions, only one operation succeeds in removing the directory entry. In other words, this class of conflicts is easily resolved by operation serialization at the directory entry.

Link/unlink conflicts

Link/unlink conflicts are not an issue in the absence of failures, because the operations are serialized at the directory entry. When failures result in incomplete operation execution, there are two cases of conflicts to be considered.

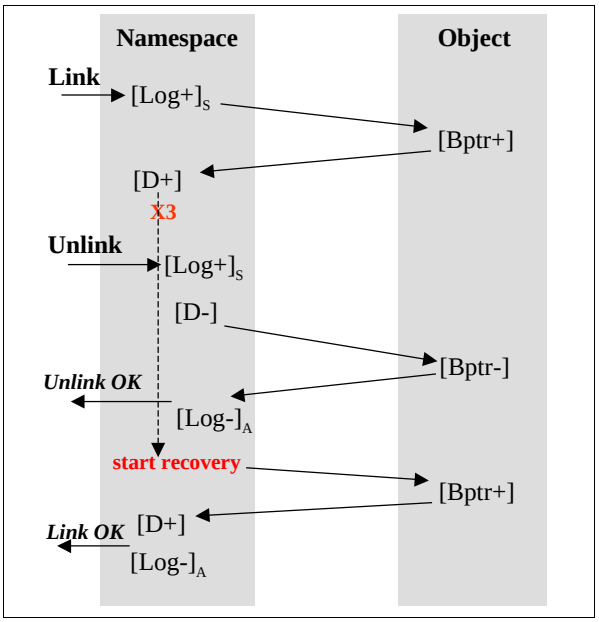


Figure 8. A link/unlink conflict scenario.

The first case occurs when a *link* operation fails at point ***X3***; before recovery is initiated, an *unlink* operation is started for the same entry. This is demonstrated in Figure 8. The recovery of *link* is initiated after the successful completion of *unlink*. As a result of the *link* re-execution, a back pointer is added at the object and an entry inserted in the namespace. Eventually, the namespace is consistent, but overall this scenario may present unacceptable semantics for the clients.

Such scenarios can occur only in the presence of a crash of the namespace site. To address them, the namespace site does not serve any new operations upon recovery from a crash, until all pending operations in the log are re-started (not necessarily completed). In the example of Figure 8, the *unlink* operation is not initiated until the partially complete *link* has already been re-started.

The second case occurs when 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, as illustrated in Figure 9. *Link* successfully adds a new back pointer at the object site and inserts an entry at the namespace site. The recovery procedure for *unlink* is later initiated; it compares the contents of *unlink*'s log record against the existing directory entry with the same name; the fields of the two do not match and thus the *unlink* re-execution is aborted and the log is reclaimed.

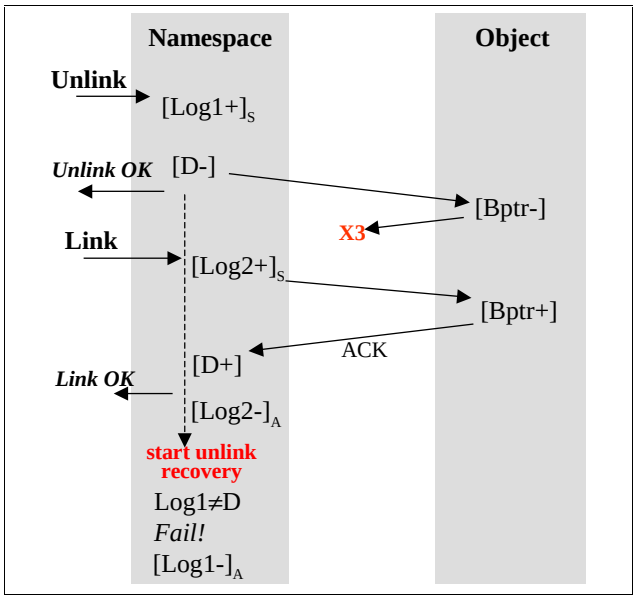


Figure 9. An unlink/link conflict scenario.

5. Comparison with 2-phase Commit

The approach followed by other research projects [1, 9] is to use *2-phase commit* (2PC) for the implementation of distributed namespace operations. To facilitate a comparison with the proposed protocols, a 2PC-based implementation of the *link* and *unlink* operations is discussed in this section.

Figure 10 illustrates scenarios that result in committing *link* and *unlink* executions, using the typical "presumed nothing" (PrN) variation of 2PC [7]. The coordinator of 2PC is chosen to be the namespace site. Back pointers are not necessary. *Link-counts* at the object site suffice, since the target object properties are kept "locked" between the "prepare" and "commit" phases. Back pointers are not required for scenarios of conflicting *link* and/or *unlink* operations either. It is sufficient to make sure that upon recovery of the namespace site, pending transactions are restarted before handling any new client requests. Updating the link-count of an object still requires a synchronous disk access. The 2PC implementations have the following disadvantages in comparison with the protocols of Figures 2 and 3:

- Overhead on failure-free operation execution. They require *two* message round-trips and *three* synchronous writes to the log, as opposed to just *one* round trip and *two* synchronous writes[3] in the proposed protocols.

[3] Both approaches require the same synchronous access to persistent storage to update the namespace (directory) and the object properties (back pointer or link-count, respectively).

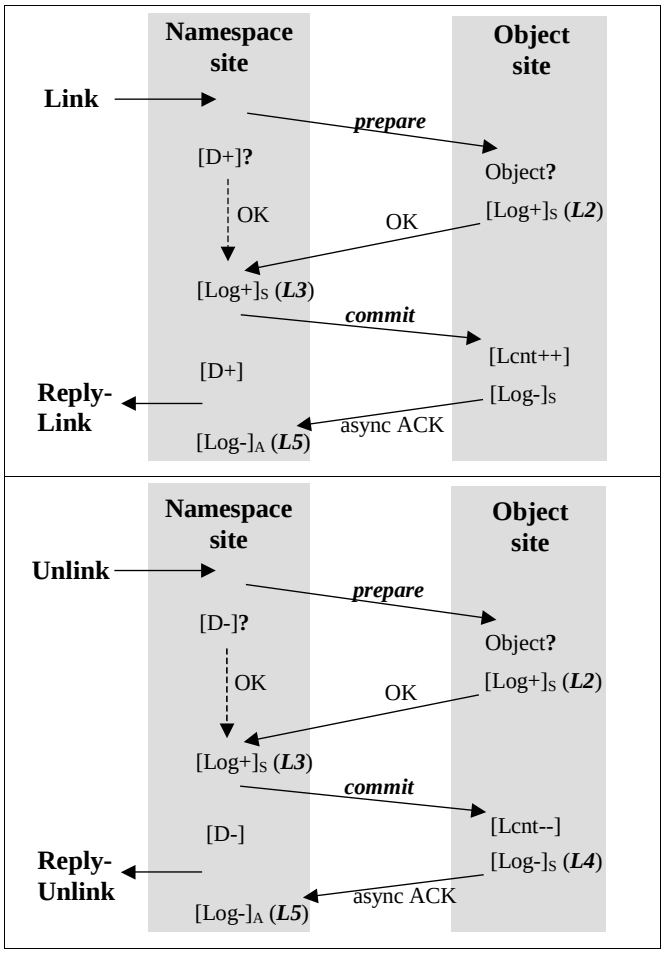


Figure 10. An implementation of link and unlink using 2-phase commit.

- Lower degree of concurrency. The object site data structure (link-count) is locked for the duration of the operation execution; no other operation that involves the same object can be performed concurrently, even from another namespace site. The phase-2 message (commit or abort) is required before that resource is unlocked. In the presence of failures, such as network partitioning, there is no upper time bound for that message to be delivered. This is an occurrence of the blocking problem of 2PC.
- Weaker execution semantics to the client. When the client receives back a reply to a *link* or *unlink* request, it is guaranteed that the namespace has been updated, but it is *not* guaranteed that the object properties (link-count) have been updated. The latter would require a *synchronous* ACK to the commit decision, before a reply is sent back to the client. The protocols of this paper provide the stronger guarantee without requiring a second round-trip latency in the critical path of the operation.

	Required object attributes	Total log accesses	Sync log accesses in critical path	Total message round trips	Round trips in critical path	Recovery approach	Degree of concurrency
2PC	Simple: *Link-counter*	Link: 4 Unlink: 4	Link: 3 Unlink: 3	Link: 2 Unlink: 2	Link: 1 Unlink: 1	Conservative	Low
DiFFS	Complex: *Back-pointer*	Link: 2 Unlink: 2	Link: 2 Unlink: 1	Link: 1 Unlink: 1	Link: 1 Unlink: 0	Aggressive	High (with conflict resolution)

Table 7: Comparison with 2PC protocols.

Table 7 summarizes the comparison between the 2PC PrN and the protocols introduced in this paper. Although there exist 2PC optimizations, such as the "presumed commit" (PrC) variation [13, 11], the important performance features, including the number of synchronous log accesses and message round-trips on the critical path would still be the same.

6. Related Work

DiFFS is an architecture designed to provide a widely distributed file service [10]. Much of the architecture's scalability and flexibility is due to its partition-based approach to storage distribution. DiFFS introduces a novel design of distributed namespaces, where the physical location of files is independent of their position in the namespace hierarchy. This facilitates policy-driven allocation of files to resources, and transparent support for various file types arbitrarily dispersed throughout the namespace. The granularity of the distribution is an individual object. This is in contrast with approaches followed by AFS [8] and NFSv4 [15], where the distribution granularity is an entire volume. Preliminary performance results for distributed namespace operations based on the protocols of this paper are reported in [10]. They demonstrate that the operations scale well with the number of servers, since they introduce a constant performance overhead factor, irrespectively of the size of the system: two servers involved for all operations (except *rename*, which involves three servers).

Fine-grain distribution is not new. Slice [1] and Archipelago [9], in particular, use hash-based approaches to place groups of objects to certain partitions. This approach restricts the flexibility of these systems. For example, breaking hotspots that occur inside a single hashed group is difficult; the same is true when system reconfiguration is required, e.g., adding/removing partitions. Despite their differences, all these systems (DiFFS, Slice and Archipelago) have one common requirement: maintain a consistent distributed namespace that references objects arbitrarily distributed in the system. Slice and Archipelago employ 2-phase commit protocols for cross-partition namespace operations. While 2PC is a mature technique, it is known to 1) impose a high overhead to failure-free execution, and 2) reduce execution concurrency due to resource locking. Although there exist optimized versions of 2PC, they often come with a price that may render them impractical for our purposes. An optimized version of "presumed commit" (PrC) [11], for example, retains a permanent record for all aborted transactions, which may grow arbitrarily with the number of aborts. Thus, the general challenge is how to maintain consistent namespace over a collection of distributed objects, in an *efficient* way.

7. Conclusions

This paper introduces namespace protocols with the same robustness characteristics as that of 2PC. We show that all namespace operations can be decomposed into just two primitive operations: *link* and *unlink*. We discuss protocols and recovery procedures for these two operations, taking under consideration all possible failure scenarios as well as conflicts that may occur. We claim that these protocols impose low overhead to failure-free execution and, in general, are more lightweight than traditional atomic commitment. To justify this claim, we conduct a detailed comparison with protocols that are based on 2PC. We demonstrate that the proposed protocols outperform 2PC in all critical performance factors; that is, communication round-trips and synchronous I/O on the critical path of operations. In addition, they facilitate higher concurrency for operation execution and 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 properties that do not exist in traditional file systems.

In particular, back pointers to all “parent” directories must be associated with every object in the system.

While atomic commitment protocols are generic techniques for implementing distributed transactions, they may not provide optimal solutions in certain cases. For example, for the problem of maintaining a consistent distributed namespace, the decision on whether an operation execution can proceed is not “distributed” but it is rather determined by the namespace site alone. It is this “one-sided” decision making that allows the design of lightweight protocols that have the same robustness as 2PC. In fact, many distributed computing problems may be of similar nature. In such cases, we expect that more efficient solutions can be developed in place of generic atomic commitment protocols.

8. Acknowledgements

We are indebted to Svend Frolund and Dejan Milojicic for their valuable feedback on earlier drafts of this paper. Mallik Mahalingam and Dan Muntz contributed considerably in the formulation of the presented ideas. We would also like to thank John Wilkes for providing the motivation to explore this area.

9. References

[1] Anderson, D., Chase, J., and Vadhat, A. "Interposed Request Routing for Scalable Network Storage", in *Proc. of the Usenix OSDI*. San Diego, CA, USA.

[2] Bernstein, P.A., Goodman, N., and Hadzilacos, V., *Concurrency Control and Recovery in Distributed Databases.* 1987.

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

[4] Callaghan, B., *NFS Illustrated*. Addison-Wesley Professional Computing Series, 2000: Adison-Wesley.

[5] Cheung, D. and Kameda, T. "Site-Optimal Termination Protocols for a distributed Database under Networking Partitioning", in *Proc. of the 4th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. Minaki, Ontario, Canada, August 1985.

[6] Gagner, G. and Patt, Y. "Metadata Update Performance in file Systems", in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 49-60, November 1994.

[7] Gray, J. and Reuter, A., *Transaction Processing: Concepts and Techniques*, 1993: Morgan Kaufman.

[8] Howard, J., *et al.*, *Scale and Performance in a Distributed File System.* ACM Transactions on Computer Systems, Vol. **6**(1): pp. 51-81, 1988.

[9] Ji, M., Felten, E.W., Wang, R., and Singh, J.P. "Archipelago: An Island-Based File System for Highly Available and Scalable Internet Services", in *Proc. of the 4th USENIX Windows Systems Symposium*, August 2000.

[10] Karamanolis, C., *et al.*, “An Architecture for Scalable and Manageable File Services”, Hewlett-Packard Labs, Palo Alto, Technical Report, HPL-2001-173, July 2001.

[11] Lampson, B. and Lomet, D. "A New Presumed Commit Optimization for Two Phase Commit", in *Proc. of the 19th VLDB Conference*. Dublin, Ireland, 1993.

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

[13] Mohan, C., Lindsay, B., and Obermarck, R., *Transaction Management in the R* Distributed Data Base Management System.* ACM Transactions on Database Systems, Vol. **11**(4): pp. 378-396, 1986.

[14] Preslan, K., *et al.* "Implementing Journaling in a Linux Shared Disk File System", in *Proc. of the 8th NASA Goddard Conference on Mass Storage Systems and Technologies*, March 2000.

[15] Shepler, S., *et al.*, “NFS version 4 Protocol”, RFC 3010, 2000.

[16] Skeen, D. "Nonblocking Commit Protocols", in *Proc. of the ACM SIGMOD*, pp. 133--142, 1981.

[17] Zhang, Z., Karamanolis, C., Mahalingam, M., and Muntz, D., “Cross-Partition Protocols in a Distributed File Service”, Hewlet-Packard Labs, Palo Alto, Technical Report, HPL-2001-129, May 23, 2001.