Name space consistency in the Pangaea wide-area file system

HPL SSP Technical Memo No. 2002-12

(This is a preliminary draft. A more complete version will appear as a HPL Technical Report.)

Yasushi Saito and Christos Karamanolis
{ysaito,christos}@hpl.hp.com

December 4, 2002

Abstract

Pangaea is a wide-area file system that enables ad-hoc collaboration in multi-national corporations or in distributed groups of users. This paper describes Pangaea's approach for keeping the file-system's name space consistent and proves its correctness. Maintaining the name space is a simple matter in traditional file systems that store the entire volume in a single node. It is not so in Pangaea, because of the two key techniques it employs to improve performance and availability in a wide area—*pervasive replication* that lets each file be replicated on its own set of nodes on demand from users, and *optimistic replication* that lets updates be issued on any replicas at any time. A naive implementation may leave some files unreachable in the name space or some directory entries pointing to non-existent files.

To detect conflicting updates and inform all affected replicas about the resolution outcome reliably, Pangaea embeds, in each file, a data structure called *backpointer* that authoritatively defines the file's location in the file-system's name space. Conflicting directory operations are detected by a replica of the (child) file as a discrepancy in the value of the backpointer. The replica can then unilaterally resolve conflicts and disseminate the conflict resolution outcome to the the parent directories.

1 Introduction

Pangaea is a wide-area file system that serves storage needs of multi-national corporations or distributed groups of users. This paper describes Pangaea's protocols for maintaining the hierarchical filesystem name space. A more comprehensive description of Pangaea appears in a separate paper [11].

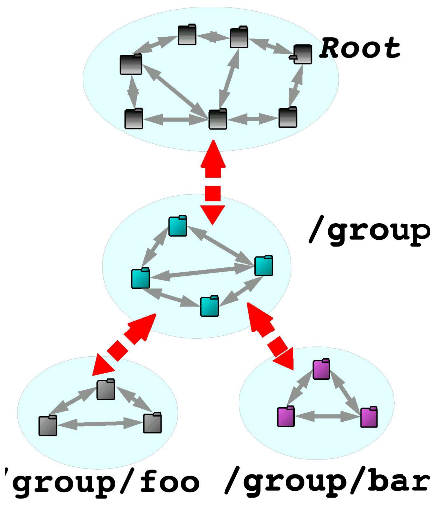

Figure 1: An example of the Pangaea file system. Replicas are added dynamically for each file or directory as users access it from various sites. Thus, different files or directories are replicated on different sets of sites.

1.1 Overview of Pangaea

Pangaea federates computers provided by the users to build a single filesystem.[1] To achieve high performance and availability in a wide area, Pangaea deploys two strategies not found in traditional replicated file systems: *pervasive replication* and *optimistic replication*.

Pangaea aggressively creates a replica of a file or directory[2] whenever and wherever it is accessed. This *pervasive replica-*

[1]We currently assume that servers trust each other; relaxing the trust relationship is future work.

[2]Pangaea treats a directory as a file with special contents. We sometimes use the term "file" for both a regular file and a directory.

tion policy improves performance by serving data from a node close to the point of access, improves availability by naturally keeping many copies of popular data and letting each server contain its working set. Figure 1 shows a sample file system. This policy brings challenges as well: the set of nodes that replicate a file (called the *replica set*) can become different from that of its parent directory or siblings. Such situations complicate detecting and resolving conflicting directory operations, as we discuss further in Section 1.2.

A distributed service faces two conflicting challenges: high availability and strong data consistency [4, 16]. Pangaea aims at maximizing availability and sacrifices strong consistency when unavoidable. It lets any user update any replica at any time, propagates the updates among replicas in the background, and detects and resolves conflicts after they happen. Pangaea thus supports "eventual" consistency, guaranteeing that changes made by a user are seen by another user only in some unspecified future. More specifically, assuming that all nodes can exchange updates with one another, and users cease to issue updates for a long enough period, Pangaea ensures the following properties:

1. For every file, the state of all its replicas will become identical.
2. Every file has valid pathname(s).
3. No directory entry refers to a non-existent file.

Section 7 defines these properties more formally and shows that our protocol actually satisfies them.

1.2 Challenges of replica management in Pangaea

Optimistic replication itself is not a new idea. The first optimistically replicated file system, Locus, was developed in early '80s [9, 15]. The combination of pervasive replication and optimistic replication, however, adds a unique complexity to name-space management, because the system must maintain the integrity of the namespace across multiple files or directories replicated on different sets of nodes.

Consider Example 1. When the dust settles, we want file `foo` to appear either at `/alice/foo1` or at `/bob/foo2`, but not both. However, suppose that node A, on behalf of Alice, adds an entry for `foo1` to `/alice` and sends the change to the replica on node C. If the final result is to move `/foo` to `/bob/foo2`, node C must undo the change, even though node C never receives Bob's update and cannot detect the conflict in the first place. We thus need a mechanism for forwarding conflict-resolution results to replicas that fail to detect conflicts.

1. File `/foo` and directories `/alice` and `/bob` are initially replicated on replica sets $\{A, B\}$, $\{A, C\}$, and $\{B, D\}$, respectively.
2. Alice on node A does `mv /foo /alice/foo1`.
3. Simultaneously, Bob on node B does `mv /foo /bob/foo2`.

Example 1: Example of rename-rename conflict.

Removing a directory (`rmdir`) poses another challenge: a file under a removed directory may be updated by another node concurrently. Consider Example 2. A naive implementation would remove `/foo` but leave file `/foo/bar` without a name. Another implementation would just delete `/foo/bar` when `/foo` is deleted, which at least keeps the file system consistent, but loses Alice's data. In this situation, the system must "revive" directory `/foo` if a live file is found underneath.

1. An empty directory `/foo` is replicated on nodes $\{A, B\}$.
2. Alice on node A creates file `/foo/bar`.
3. Bob on node B does `rmdir /foo`.

Example 2: Example of rmdir-update conflict.

1.3 Overview of Pangaea's replica management protocol

This section overviews Pangaea's four key strategies for addressing the aforementioned challenges. Essentially, a distributed file system like Pangaea can ensure eventual consistency when (1) there is always at least one node that detects any conflicting pair of operations, (2) a conflict-resolution decision is reliably propagated to all files affected by the conflict, and (3) replicas of each file make the same decision regarding conflict resolution. The first property is ensured using a data structure called *backpointer*, described in Section 1.3.1. The second property is ensured by two techniques, called name-space containment and directory resurrection, described in Sections 1.3.2 and 1.3.3. The final property is ensured by the use of a uniform timestamp-based conflict detection and resolution rule, described in Section 1.3.4.

1.3.1 Localizing conflict-resolution decisions using backpointers

Pangaea lets the “child” file have the final say on the resolution of a conflict involving directory operations.[3] For instance, in Example 1, if Bob’s operation is to win over Alice’s, a replica of file `foo` tells the replicas of `/alice` to undo their change.

For this policy to work, each file must be able to determine its location in the filesystem name space, unlike other file systems that let a directory dictate the location of children files. Pangaea achieves this by storing a special attribute, called *backpointers*, in each replica (a file usually has only one backpointer, unless it is hard-linked). A backpointer contains the ID of the parent directory and the file’s name within the directory. We implement directory operations, such as `rename` and `unlink`, as a change to the file’s backpointer(s). In Example 1, operation `mv /foo /alice/foo` changes file `/foo`’s backpointer from $\langle \mathit{fid}_{/}$, `"foo"`$\rangle$ to $\langle \mathit{fid}_{alice}$, `"foo1"`$\rangle$ (fid_x is the ID of the file x). When a replica receives a change to its backpointer, it also reflects the change to its parents by creating, deleting, or modifying the corresponding entries.

1.3.2 Name-space containment

Conflict resolution using backpointers requires that each file can perform a (local or remote) update to a replica of the directory that the backpointer refers to. One approach, adopted in our earlier implementation, is to embed pointers to (some of) the replicas of the parent directory in the backpointer[4] and modify the parent directory using remote procedure calls. This design turned out to be unwieldy: the backpointer is used to initiate a change in the directory, but its directory links must be changed when the directory’s replica set changes. Because of this circular control structure, we could not easily keep the information of the backpointer and the parent directory properly synchronized.

Our current implementation trivializes this problem by requiring that, for every replica of a file, its parent directories be also replicated on the same node. We call this property *name-space containment*, because all intermediate path-name components of every replica are all replicated on the same node.

This policy improves Pangaea’s availability and eases administration by allowing accesses to every replica on a node using ordinary file-access system calls, even when the node is disconnected from the rest of the system—i.e., it naturally offers the benefits of island-based replication [5]. On the other hand, it increases the storage overhead of the system and adds a different sort of complexity to the protocol: every update to a replica potentially involves replicating its parent directories. We describe our solution for maintaining the name-space containment property in Section 4.3 and study the storage overhead in Section 9.

A node must discover and replicate the root directory when starting the Pangaea service for the first time. The locations of the root replicas are maintained using a gossip-based distributed membership service [11].

1.3.3 Avoiding file losses by resurrecting directories

Pangaea addresses the problem of rmdir-update conflicts, shown in Example 2, by “resurrecting” deleted directories when necessary. When a node receives a request to create a replica for file F for which its parent directory, say D, does not exist (because it is “rmdir”ed by another concurrent update), the node schedules a special procedure to be called later.[5] Similarly, the node schedules the procedure to be called when it deletes directory D with an entry to a live file F. This procedure, when executed, checks if F is still live and D is still dead; if so, it recreates D and adds F’s entry to D.

To resurrect a dead directory, when a node is requested to delete a replica, it removes the replica’s contents but retains the last backpointer the replica has had. This “dead backpointer” determines the location in the namespace the directory is to be resurrected.

This procedure potentially works recursively all the way up to the root, resurrecting directories along the way. For instance, if Bob on node B does `rm -rf /a/b/c` and Alice on node A does `touch /a/b/c/foo` simultaneously, then directories `c`, `b`, and `a` are resurrected in order to create a place for file `foo`.

1.3.4 Uniform conflict resolution using last-writer-wins policy and full-state transfer

Achieving eventual consistency requires that all replicas of a particular file make the same decision when confronted with a conflict. For the contents of a regular file, we use a version vector [9] to detect conflicts and let the user fix the conflict manually [11].

[3] Resolution is “localized” to a specific file, but not to a specific node; conflicts are still resolved by any replica of the file.

[4] In practice, we embedded the gold-replica set of the parent directory; see Section 2.

[5] The wait is needed, because it is quite likely that the node will receive a request to remove F in the near future. We discuss this issue further in Section 8.2.

Conflicts regarding the structure or attribute of the file system, such as backpointers or access permissions, are amenable to automatic resolution because of their well-defined semantics. We use the combination of the "last writer wins" rule [6, 14] and full state transfer to resolve such conflicts. A high-level file system operation—e.g., `write` or `unlink`—is assigned a unique timestamp by the issuing node. When a replica discovers two conflicting updates, it picks the one with the newer timestamp and overwrites the older one (if the old update was already applied). With full state transfer, each update completely overwrites the contents and attributes of a replica.[6] By applying the update with the newest timestamp, replicas will eventually converge to common state.

A timestamp is generated using the node's real-time clock. Thus, using the last-writer-wins policy, the clocks of nodes must at least be loosely synchronized to respect the users' intuitive sense of update ordering. This is usually not a problem, as modern protocols (e.g., NTP [8]) can synchronize clocks within about 100 milliseconds even over a wide-area network.

1.4 Related work

Many file systems replicate at a volume granularity and build a unified name space by mounting a volume underneath another (e.g., LOCUS [15], Coda [7], and Roam [10]). Because these systems need not support cross-volume directory operations, a single replica can locally detect and resolve conflicting updates. Pangaea, in contrast, replicates at a file or directory granularity to support wide-area ad-hoc collaboration. Pangaea must run a distributed protocol for name-space maintenance because every directory operation in Pangaea crosses a replication boundary.

Several file systems replicate data at a finer granularity. FARSITE [1] replicates at the unit of a "directory group", which resembles a volume, but with a dynamically defined boundary. It supports file renaming across directory groups using a Byzantine-fault-tolerant consensus protocol that coordinates nodes in a lock-step manner. Slice [2] replicates files and directories independently over a cluster of servers and uses two-phase commits to coordinate nodes. Pangaea, in contrast, coordinates nodes optimistically to improve availability and performance in a wide area, but it must detect and resolve conflicting updates.

Data structures similar to backpointers are used in several file systems. DiFFS [17] places files and directories independently on a cluster of servers. It uses backpointers to implement at-least-once remote procedure calls (RPCs) for directory operation. DiFFS does not support replication. S4 [13] is a file system with a security auditing capability. It keeps a backpointer-like data structure to reconstruct a file's full path name from its inode number and chooses a security policy based on the path name. S4's backpointers are used only for auditing and not for replication.

```
proc UpdateReplica
  $r_2$: Replica // New replica contents.
preconditions:
  $\forall \langle pfid, fname \rangle : r_2.bptrs \bullet pfid \in \mathrm{dom}(DISK)$
    and $IsLive(r_2) \Rightarrow IsLive(DISK(pfid))$
postconditions:
  $r_2.bptr \neq \{\}$ $^{\langle 3 \rangle}$
--------------------------------------------------
  ...
```

Listing 1: Example of algorithm description

1.5 Notational conventions

Listing 1 shows an example of algorithm description. UpdateReplica is the name of the procedure with one parameter, r_2. Label "**preconditions:**" shows the condition that must be ensured by the caller, and label "**postconditions:**" shows the condition that this procedure ensures on return. A code block is demarcated by indentation, as in Occam or Python. Label "$\langle 3 \rangle$" is a marker used to refer to the algorithm in the paper.

We use several primitive functions without showing their implementations. Function Newtimestamp generates a globally unique timestamp. A timestamp is a tuple $\langle clock, nodeid \rangle$, where *clock* is the value of the real-time clock, and *nodeid* is the node that generated the timestamp. The latter value is used only to break ties. Function Newfileid generates a globally unique File ID.[7] Function Deepcopy creates an object that is structurally identical to the old object but does not share memory with the old object. In addition, we use several mathematical symbols borrowed from the Z notation [12]:

- "$\langle \mathrm{val}_1, \ldots, \mathrm{val}_n \rangle$" represents a tuple of values.
- "$\mathbb{P}$ **type**" represents a (possibly empty) set of **type**. "$\mathbb{P}_1$ **type**" represents a nonempty set of **type**. "**Key** $\rightarrowtail$ **Val**" represents a one-to-many mapping from type **Key** to **Val**.
- "dom(F)" returns the domain of function (or mapping) F, and "ran(F)" returns the range of F. For instance,

$$\mathrm{dom}(\{1 \mapsto 3, 2 \mapsto 8, 4 \mapsto 3\}) \quad = \quad \{1,2,4\},$$

[6] In practice, full-state transfer happens only during conflict resolution. Pangaea uses "deltas" to reduce update propagation overhead in the absence of conflicts [11].

[7] In practice, Pangaea currently uses a timestamp as a file ID. Thus, Newfileid is an alias for `Newtimestamp`.

$$\mathrm{ran}(\{1 \mapsto 3, 2 \mapsto 8, 4 \mapsto 3\}) \quad = \quad \{3, 8\}.$$

- "X $\oplus$Y" substitutes a part of mapping X by Y. E.g.,

$$\{1 \mapsto 3, 2 \mapsto 1\} \oplus \{1 \mapsto 5, 3 \mapsto 4\}$$
$$= \{1 \mapsto 5, 2 \mapsto 1, 3 \mapsto 4\}.$$

- "X $\lhd$Y" means function-domain restriction. E.g.,

$$\{2\} \lhd \{1 \mapsto 3, 2 \mapsto 8, 4 \mapsto 6\} = \{1 \mapsto 3, 4 \mapsto 6\}.$$

- "$\forall$*var: set* $\bullet$*expr*" means that *expr* holds for *var* in *set*. E.g.,

$$\forall\, n : \{11, 13, 17\} \bullet \mathrm{IsPrime}(n).$$

- "$\Diamond$ *expr*" means that *expr* holds eventually.
- "$\{$*var: set* $\bullet$*expr*$\}$" means set comprehension. E.g.,

$$\{x : \{1, 2, 3\} \bullet x^2\} = \{1, 4, 9\}.$$

2 Structure of the Pangaea file system

Pangaea creates replicas of a file whenever and wherever requested. It distinguishes two types of replicas: *gold* and *bronze*. They can both be read and written by users at any time, and they both run an identical update-propagation protocol. Gold replicas, however, play an additional role in maintaining the hierarchical name space. First, gold replicas act as starting points during path-name traversal (i.e., in UNIX kernel procedure `namei`). Their locations are thus registered in file's parent directory. Second, gold replicas perform several tasks that are hard to do in a completely distributed way, such as maintaining a minimum replication factor for a file. Currently, Pangaea designates replicas created during initial file creation as gold and fixes their locations unless some of them fail permanently.

Bronze replicas are created in response to user demands. To manage them cheaply and without a single point of failure, Pangaea builds a distributed graph of replicas independently for each file; a newly created replica joins the system by spanning edges to a few existing (gold or bronze) replicas. These edges are used both to discover the replica and propagate updates to the file. Pangaea provides reliable protocols for keeping the graph strongly connected and broadcasting updates efficiently over graph edges. These protocols are described in more detail in [11]. This paper focuses on the maintenance of the gold-replica set and links between directories and their children.

Listing 2 shows the structure of a replica. The descriptions of the attributes follow.

$\langle 1 \rangle$ The globally unique ID of the file that the replica represents. The file ID is fixed once a replica is created.

$\langle 2 \rangle$ Graph edges to some bronze replicas of the file [11].

$\langle 3 \rangle$ The set of gold replicas of the file. The replica is gold if the node that stores the replica is in *gpeers*; otherwise, the replica is bronze.

$\langle 5 \rangle$ This field is either null, or it records the last backpointer the replica has had just before the file was deleted (Section 1.3.3). Section 4.2.

$\langle 6 \rangle$ Shows the freshness of the replica (Section 1.3.4). The attributes in the replica, including *gpeers* and *bptrs*, are serialized by this timestamp.

$\langle 7 \rangle$ The contents of a regular file.

$\langle 8 \rangle$ Directory entries. An entry is identified by pair $\langle$fileid, filename$\rangle$. That is, Pangaea allows duplicate filenames as far as they refer to different files. This design simplifies handling of the situation in which two users create two files with the same name. Section 8.1 discusses a strategy for presenting a more natural user interface on top of this design.

$\langle 9 \rangle$ Shows whether this entry is live. In Pangaea, deleted entries are not removed from the directory immediately. They are just marked invalid using this field and kept in the directory to disambiguate update/delete conflicts (i.e., invalid entries are used as death certificates [3].)

$\langle 10 \rangle$ Shows the last time either *bptrs* or *gpeers* of the child file has changed. This timestamp is used to serialize other fields in **Dentry**.

$\langle 11 \rangle$ Points to the gold replicas of the child file.

The values of attributes *fid*, *gpeers*, *ts*, *bptrs*, and *deadBptr* will be the same on all replicas of a file in the absence of outstanding updates, but the value of *peers* differs between replicas, as it is used to construct the file's graph.

Two key attributes connect a file and its parent directories. Attribute *gpeers* $\langle 11 \rangle$ in the parent directory entry point to the file's gold-replica set $\langle 3 \rangle$. The backpointers of the file $\langle 4 \rangle$ point back to the parent directories. These attributes reciprocally link each other in the absence outstanding updates. Figure 2 illustrates the relationships between the replica's attributes.

Listing 3 shows persistent variables kept on each node. *DISK* stores the set of replicas. *CLOG* records the set of replicas whose state may be inconsistent with other replicas of the same file. A replica stays in *CLOG* until all the neighboring replicas in the graph acknowledge its update. *ULOG* stores the set of files whose backpointers have changed but whose parent

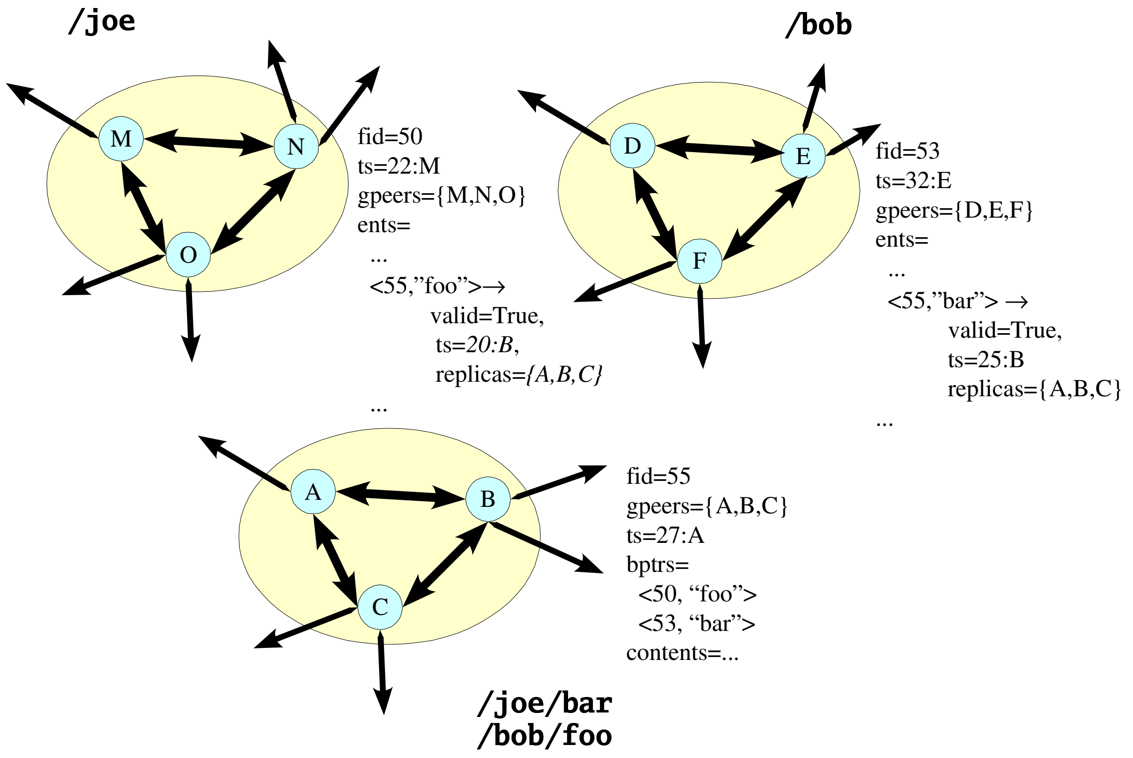

Figure 2: Example of a file system. Directories `/joe` and `/bob` have the FileIDs of 50 and 53, respectively. A file with the ID of 55 is hard-linked to the two directories, one as `/joe/bar` and the other as `/bob/foo`. Attribute "ts=22:M" of `/joe` shows that this directory's timestamp is 22:M, i.e., it is issued by node M at time 22 (timestamps are in fact generated using the node's real-time clock, but we clock values as small integers for brevity.). A circle around a letter indicates a gold replica. For instance, directory `/joe` has three gold replicas, on nodes M, O, and N. Arrows denote edges created through attributes *gpeers* and *peers*. Bronze replicas are not shown in this picture, but the thin arrows emanating from the gold replicas indicate links to them.

```
type Replica = record
   fid⟨1⟩ : FileID
   peers⟨2⟩ : $\mathbb{P}$ NodeID
   gpeers⟨3⟩ : $\mathbb{P}_1$ NodeID
   bptrs⟨4⟩ : $\mathbb{P}$ Backptr
   deadBptr⟨5⟩ : Backptr
   ts⟨6⟩ : Timestamp
type RegularReplica inherits Replica =
   contents⟨7⟩ : Data
   Invariants:
      $\neg$IsLive(r) $\Rightarrow$ contents = {}
type DirReplica inherits Replica =
   ents⟨8⟩ : ⟨FileID,String⟩ $\rightarrowtail$ DEntry
   Invariants:
      $\neg$IsLive(r) $\Rightarrow$ ents = {}
type Backptr = ⟨FileID, String⟩
type Dentry = record
   valid⟨9⟩ : bool
   ts⟨10⟩ : Timestamp
   gpeers⟨11⟩ : $\mathbb{P}_1$ NodeID
proc IsLive(r)
   return r is the root or r.bptrs $\neq$ {}
```

Listing 2: Structure of a replica

directories have not received the change. It maps a file ID to the set of backpointers deleted from the replica.[8]

```
DISK: FileID $\rightarrowtail$ Replica
CLOG: FileID $\rightarrowtail$ $\mathbb{P}_1$ NodeID
ULOG: FileID $\rightarrowtail$ $\mathbb{P}$ Backptr
Invariants::
   // Updates are only for existing replicas.
   dom(CLOG) $\cup$ dom(ULOG) $\subseteq$ dom(DISK) ⟨12⟩
```

Listing 3: Persistent data structures kept at each node.

3 High-level name-space operations

This section describes the implementation of high-level name-space operations. Listing 4 shows how a file is created. To create a file, a node must already store a replica of the parent directory (*d*). Otherwise, the node must create a new bronze replica of the directory by calling the procedures described in Section 4.3. This requirement applies to every directory operation described in this section. Procedure Create itself only creates a local replica of the file. A generic procedure UpdateReplica, described in Section 4, actually adds an entry to the parent directory and propagates the changes to other nodes.

Other name-space operations are implemented in a similar fashion. Listings 5, 6 and 7 implement removing (`unlink`), hard-linking (`link`), and renaming (`rename`), respectively.

[8]Backpointers added to the replica are not recorded in *ULOG*, as they can are stored from the replica's *bptr*.

Listing 8 shows how a file's contents can be updated (say, by `write`). Unlink does not delete the replica object even after its backpointers become empty. Attributes such as *ts*, *peers*, and *gpeers* are kept on disk so that it can reject stale updates that arrive in the future (e.g., because of message reordering) and resurrect the directory, if needed, to maintain the name space's consistency (Section 1.3.3). We call a replica without a backpointer a "death certificate" [3]. Death certificates are removed by a background garbage-collection process that runs nightly, as described in Section 6.

```
proc Create
   d: DirReplica // The local replica of the parent directory.
   fname: string // The name of the new file in d
   gpeers: $\mathbb{P}_1$ NodeID // The placement of the replicas of the file.
preconditions:
   IsLive(d) ⟨13⟩
--------------------------------------------------
   r ← Newreplica()
   r.fid ← Newfileid()
   r.gpeers ← gpeers
   r.ts ← Newtimestamp()
   r.peers ← {}
   r.bptrs ← {⟨d.fid, fname⟩}
   r.contents ← None
   UpdateReplica(r)
```

Listing 4: File creation procedure.

```
proc Unlink
   f: Replica // The file to be unlinked.
   d: DirReplica // The directory the file belongs to.
   fname: string // f's name in d.
preconditions:
   IsLive(d)
   f is a directory $\Rightarrow$ f.ents = {}
   ⟨d.fid,fname⟩ $\in$ f.bptrs
--------------------------------------------------
   f' ← Deepcopy(f)
   f'.bptrs ← f.bptrs \ {⟨d.fid,fname⟩}
   if f'.bptrs = {} then
      f'.deadBptr ← ⟨d.fid,fname⟩
   f'.ts ← Newtimestamp()
   UpdateReplica(f')
```

Listing 5: Unlink and rmdir.

4 Name-Space management and conflict resolution

Listing 9 defines the central procedure, UpdateReplica, for fixing name-space inconsistencies. It is called by both local

```
proc Hardlink
  f: RegularReplica // The replica of the file.
  d: DirReplica // The directory to which f will be linked to.
  fname: string // The filename within d.
preconditions:
  IsLive(d)
------------------------------------------------------------
  f' ← Deepcopy(f)
  f'.bptrs ← f.bptrs ∪ {⟨d.fid, fname⟩}
  f'.ts ← Newtimestamp()
  UpdateReplica(f')
```

Listing 6: Hard linking.

```
proc Rename
  f: Replica // The file to be moved.
  $d_F$: DirReplica // The origin dir.
  $d_T$: DirReplica // The destination dir.
  $fname_F$: string // The filename in $d_F$
  $fname_T$: string // The filename in $d_T$
preconditions:
  IsLive($d_F$) and IsLive($d_T$)
  $\langle d_F.fid, fname_F \rangle \in f.bptrs$
------------------------------------------------------------
  f' ← Deepcopy(f)
  $f'.bptrs \leftarrow f.bptrs \setminus \{\langle d_F.fid, fname_F \rangle\} \cup \{\langle d_T.fid, fname_T \rangle\}$
  f'.ts ← Newtimestamp()
  UpdateReplica(f')
```

Listing 7: Renaming.

```
proc Write
  f: RegularReplica
  newcontents: Data
------------------------------------------------------------
  f' ← Deepcopy(f)
  f'.contents ← newcontents
  f'.ts ← Newtimestamp()
  UpdateReplica(f')
```

Listing 8: Updating contents of a regular file.

high-level directory operations (Section 3) and remote update requests (Listing 10). It takes new replica contents (r_2), copies them to the local replica, changes the parent directory entry if necessary, and schedules the update to be pushed to other replicas.

4.1 Propagating updates

Procedure IssueCupdate, shown in Listing 10, propagates a change to other replicas of the same file ("C" stands for "contents"). This paper describes only the most basic propagation mechanism that transfers the entire replica state, even when just a byte is modified. A separate paper [11] introduces two techniques, *delta propagation* and *harbingers*, that drastically reduce the update propagation overhead.

Updates are propagated to other replicas periodically in the background by PropagateCupdate. Processing a remote update is similar to applying a local update: the local replica is updated, if needed, and the change is forwarded to the neighboring replicas in the file's graph. The only difference is that the receiving site must ensure the name-space containment property (Section 1.3.2) before applying the update. The node thus replicates all intermediate directories in the file's path by calling CreateReplica recursively, as described in Section 4.3.

4.2 Repairing name-space inconsistencies

Procedure IssueUupdate is called by UpdateReplica when a file's backpointer is possibly inconsistent with the corresponding directory ("U" stands for "uplink"). This procedure only logs the request for later execution. Procedure ProcessUupdate actually updates the parent directory to match the file's backpointer. On exit, this procedure guarantees that there is a directory entry for every backpointer, and that there is no entry in directories to which the file lacks backpointers.

We usually must wait before files added to *ULOG* are treated by ProcessUupdate, because executing it immediately will waste both the disk and network bandwidth and sometimes undo the update against the user's expectation. Section 8.2 discusses this problem in more detail and introduces a strategy for choosing the waiting period.

4.3 Creating a bronze replica

Pangaea dynamically creates a bronze replica on two occasions. First is when the user accesses a file on a node for the first time (this operation is not described in the paper). Second is when the node is asked to create a gold replica of a file, but it

```
proc UpdateReplica
  $r_2$: Replica // New replica contents.
preconditions:
  // All parent directories are stored locally.
  // Moreover, if $r_2$ is live, then parent must also be live.
  $\forall \langle pfid, fname \rangle$: $r_2.bptrs \bullet pfid \in \mathrm{dom}(DISK)$
    and $IsLive(r_2) \Rightarrow IsLive(DISK(pfid))$ ⟨14⟩
------------------------------------------------------------
if $r_2.fid \notin \mathrm{dom}(DISK)$ then
  // The replica isn't locally stored yet.
  $DISK \leftarrow DISK \cup \{ r_2.fid \mapsto r_2 \}$
  IssueCupdate($r_2$)
  return

$r_1 \leftarrow DISK(r_2.fid)$

if File is regular then
  Do some application-specific stuff.
  We can potentially use version vectors here.
else
  // Union dir entries, taking ones with newer timestamps on conflict.
  for $(key \mapsto e) \in r_2.ents$
    if $key \notin \mathrm{dom}(r_1.ents)$ or $r_1.ents(key).ts < e.ts$
      $r_1.ents \leftarrow r_1.ents \oplus \{key \mapsto e\}$
    for each added or deleted entry $\langle fid, fname \rangle$ in $r_1.ents$
      // Entry $\langle fid, fname \rangle$ is potentially inconsistent. Fix up later.
      if $fid \in \mathrm{dom}(DISK)$ then
        IssueUupdate($DISK(fid)$, $\{\}$)⟨15⟩

if $r_2.ts > r_1.ts$ then⟨16⟩
  // The file's attributes are to be updated.
  $r_1.ts \leftarrow r_2.ts$
  if $r_1.gpeers \neq r_2.gpeers$ then
    $r_1.gpeers \leftarrow r_2.gpeers$
    // When the replica's gold-peer set changes, I must reflect the
    // change to the parent dir entry.
    IssueUupdate($r$, $\{\}$)

  // Resolve potential conflicts on back pointers
  if $r_1.bptrs \neq r_2.bptrs$ or $r_1.deadBptr \neq r_2.deadBptr$ then
    IssueUupdate($r_1$, $r_1.bptrs \setminus r_2.bptrs$)⟨17⟩
    $r_1.bptrs \leftarrow r_2.bptrs$
    $r_1.deadBptr \leftarrow r_2.deadBptr$

    // If the last link to the replica is gone, erase the contents.
    if $\neg$ IsLive($r_1$) then
      if $r_1$ is a regular file then
        $r_1.contents \leftarrow None$
      else
        for $e \in r_1.ents \bullet e.valid$
          IssueUupdate($DISK(e.fid)$, $\{\}$)⟨18⟩
        $r_1.ents \leftarrow \{\}$

if Any of $r_1$'s attributes has changed then
  IssueCupdate($r_1$)⟨19⟩
```

Listing 9: Applying updates to a replica.

```
proc IssueCupdate
  $r$: Replica // The replica of the file being updated.
------------------------------------------------------------
$CLOG(r.fid) \leftarrow r.gpeers \cup r.peers$
```

```
proc PropagateCupdate // Runs periodically in the background
------------------------------------------------------------
for $(fid \mapsto targets) \in CLOG$
  $r \leftarrow DISK(fid)$ // See ⟨12⟩.
  // Send the location of the parent dirs so that the target can
  // replicate them to ensure name-space containment.
  $pDirs \leftarrow \{p: r.bptrs \bullet \langle p.fid, DISK(p.fid).gpeers \rangle\}$ // See ⟨14⟩.
  for $n \in targets$
    send $\langle CUPDATE, r, pDirs \rangle$ to $n$.
when receive $\langle CUPDATE\text{-}REPLY, ts \rangle$ from node $n$
  if $CLOG(fid).ts = ts$
    $CLOG(fid) \leftarrow CLOG(fid) \setminus \{n\}$
    Remove $fid$ from $CLOG$ when $CLOG(fid)$ becomes empty
```

```
when receive $\langle CUPDATE, r, pDirs \rangle$
  $r$: Replica // New replica contents.
  $pDirs$: $\mathbb{P}\ \langle$FileID, $\mathbb{P}$ NodeID$\rangle$ // Name and location of parent dirs.
------------------------------------------------------------
for $\langle pfid, ppeers \rangle \in pDirs$
  CreateReplica($pfid$, $ppeers$)
  ResurrectDirectory($pfid$)
UpdateReplica($r$)
send$\langle CUPDATE\text{-}REPLY, r.ts \rangle$
```

Listing 10: Issuing, propagating, and receiving an update.

```
proc IssueUupdate
  $r$: Replica // The replica of the file
  $del$: $\mathbb{P}\ \langle$FileID,String$\rangle$ // Backpointers deleted from the replica.
------------------------------------------------------------
if $r.fid \in \mathrm{dom}(ULOG)$ then
  $del \leftarrow del \cup ULOG(r.fid)$
$ULOG \leftarrow ULOG \oplus \{r.fid \mapsto del\}$
```

```
proc ProcessUupdate // Called periodically in the background.
------------------------------------------------------------
for $(fid \mapsto del) \in ULOG$
  $r \leftarrow DISK(fid)$ // See ⟨12⟩.
  for $pfid, fname \in del \cup r.bptrs$
    ResurrectDirectory($pfid$)
    $d \leftarrow DISK(pfid)$
    $valid = pfid \in r.bptrs$ // Is this entry to be added?
    $new = (\langle fid, fname \rangle \mapsto Dentry(valid, r.ts, r.gpeers))\}$
    $d.ents \leftarrow (\langle fid, fname \rangle \lhd d.ents) \cup new$
    if $d.ents$ has changed
      $d.ts \leftarrow$ Newtimestamp()
      IssueCupdate($d$)
$ULOG \leftarrow \{\}$
```

Listing 11: Fixing name space inconsistencies.

lacks the replica of file's parent directories (Listing 10). Listing 12 describes the algorithm for creating a bronze replica. Procedure CreateReplica works recursively from the given directory and ensures that all intermediate directories, up to the root directory, are replicated locally. It does not, however, guarantee that these files are live (i.e., has a non-empty backpointer) or that each directory has an entry that correctly points to the child. ResurrectDirectory, described in Section 4.4, ensures these properties.

```
proc CreateReplica
  fid: FileID // The ID of the file
  peers: $\mathbb{P}_1$ NodeID // The known set of gold peers of the file
postconditions:
  fid $\in$ dom(DISK)
----------------------------------------
  if fid $\in$ dom(DISK) then
    return
  send ⟨SEND-CONTENTS, fid⟩ to random node n $\in$ peers
  Wait until receive ⟨CONTENTS, r, pDirs⟩ from n
  for ⟨pfid, ppeers⟩ $\in$ pDirs
    CreateReplica(pfid, ppeers)
  UpdateReplica(r)
  Add edges between r and random existing replicas.
```

```
when receive ⟨SEND-CONTENTS, fid⟩ from node n
----------------------------------------
  r $\leftarrow$ DISK(fid)
  pDirs $\leftarrow$ {p: r.bptrs • ⟨p.fid, DISK(p.fid).gpeers⟩} // See ⟨14⟩.
  send ⟨CONTENTS, r, pDirs⟩ to n.
```

Listing 12: Creating a bronze replica. A more detailed description about the construction of the graph appears in [11].

4.4 Resurrecting directories

Both c- and u-update processing (Listings 10 and 11) requires that a file's parent directories are live and with valid pathnames. Procedure ResurrectDirectory, shown in Listing 13, is used in conjunction with CreateReplica to resurrect a dead directory and re-create an entry in its parent. This procedure is also recursive—it ensures that all the intermediate directories also are live and with valid pathnames.

5 Examples of conflict resolutions

This section describes how Pangaea resolves several common types of conflicts. We use a tabular form, shown in Figure 3, to display the state of the system.

This example shows the state of two replicas stored on node A. Label $fid_/$ represents the file ID of the replica of the directory initially located at "/". Replica $fid_/$ has the timestamp ⟨6⟩

```
proc ResurrectDirectory
  fid: FileID
preconditions:
  fid $\in$ dom(DISK)
  fid is a directory
postconditions:
  IsLive(DISK(fid))
----------------------------------------
  r $\leftarrow$ DISK(fid)
  if IsLive(r) then
    return

  ResurrectDirectory(r.deadBptr.pfid)
  r.bptrs $\leftarrow$ { r.deadBptr }
  r.ts $\leftarrow$ Newtimestamp()
  IssueCupdate(r)
  let ⟨pfid, fname⟩ = r.deadBptr •
    d $\leftarrow$ DISK(pfid)
    d.ents(⟨fid,fname⟩) $\leftarrow$ Dentry(true, r.ts, r.gpeers) ⟨20⟩
    d.ts $\leftarrow$ Newtimestamp()
    IssueCupdate(d)
```

Listing 13: Filling missing name space components.

A	$fid_/$	5:A	ents={⟨fid_{alice}, "alice", 12:A⟩}
	fid_{alice}	12:A	ents={*⟨fid_{bar}, "bar", 13:B⟩}

Figure 3: Example of state descriptions.

of 5:A[9], and its entries contain directory "alice" at fid_{alice}, with timestamp 12:A. A directory entry marked "*" is invalid (i.e., *ent.valid = false* in Listing 2).

5.1 Scenario: rename-rename conflict

Let us first revisit Example 1. We just show state transitions on nodes A and B, as nodes C and D only passively receive updates from nodes A and B.

1. Initially, the replicas are consistent.

A	$fid_/$	5:A	ents={⟨fid_{foo}, "foo",...⟩}
	fid_{alice}	6:A	ents={}
	fid_{foo}	8:A	bptrs=⟨$fid_/$, "foo"⟩
B	$fid_/$	5:A	ents={⟨fid_{foo}, "foo",...⟩}
	fid_{bob}	7:B	ents={}
	fid_{foo}	8:A	bptrs=⟨$fid_/$, "foo"⟩

2. Alice does `mv /foo /alice/foo1`.

A	$fid_/$	10:A	ents={*⟨fid_{foo},"foo",...⟩}
	fid_{alice}	11:A	ents={⟨fid_{foo}, "foo1", 12:A⟩}
	fid_{foo}	12:A	bptrs=⟨fid_{alice}, "foo1"⟩
B	$fid_/$	5:A	ents={⟨fid_{foo}, "foo",...⟩}
	fid_{bob}	7:B	ents={}
	fid_{foo}	8:A	bptrs=⟨$fid_/$, "foo"⟩

[9]

3. Bob does `mv /foo /bob/foo2`.

A	$fid_{/}$	10:A	ents={*⟨fid_{foo},“foo”,...⟩}
	fid_{alice}	11:A	ents={⟨fid_{foo}, “foo1”, 12:A⟩}
	fid_{foo}	12:A	bptrs=⟨fid_{alice}, ”foo1”⟩
B	$fid_{/}$	9:B	ents={*⟨fid_{foo}, “foo”,...⟩}
	fid_{bob}	11:B	ents={⟨fid_{foo}, “foo2”, 12:B⟩}
	fid_{foo}	12:B	bptrs=⟨fid_{bob}, ”foo2”⟩

4. Let us assume node B's node ID is larger than A's; that is, timestamps are ordered in the following manner:

$$12{:}B > 12{:}A > 11{:}B > 11{:}A.$$

Node B sends the update to file fid_{foo} to node A. Node A first replicates directory `/bob` (by copying the state from node B) to ensure the name-space containment property (Listing 10). Node A then changes replica fid_{foo}'s back-pointer and removes the fid_{foo}'s entry in `/alice`.

A	$fid_{/}$	10:A	ents={*⟨fid_{foo},“foo”,...⟩}
	fid_{alice}	14:A	ents={*⟨fid_{foo}, “foo1”, 12:B⟩}
	fid_{bob}	11:B	ents={⟨fid_{foo}, “foo2”, 12:B⟩}
	fid_{foo}	12:B	bptrs=⟨fid_{bob}, ”foo2”⟩
B	$fid_{/}$	9:B	ents={*⟨fid_{foo}, “foo”,...⟩}
	fid_{bob}	11:B	ents={⟨fid_{foo}, “foo2”, 12:B⟩}
	fid_{foo}	12:B	bptrs=⟨fid_{bob}, ”foo2”⟩

5. Node A sends update to file $fid_{/}$ to B. Node B applies the change, but it actually leave the directory's contents intact. Node A also sends update to file fid_{foo} to B, but B discards the update, because B already has applied this update.

6. In the end, the file `/foo` will move to `/bob/foo2`. The state of the nodes looks like below:

A	$fid_{/}$	10:A	ents={*⟨fid_{foo},“foo”,...⟩}
	fid_{alice}	14:A	ents={*⟨fid_{foo}, “foo1”, 12:B⟩}
	fid_{bob}	11:B	ents={⟨fid_{foo}, “foo2”, 12:B⟩}
	fid_{foo}	12:B	bptrs=⟨fid_{bob}, ”foo2”⟩
B	$fid_{/}$	10:A	ents={*⟨fid_{foo}, “foo”,...⟩}
	fid_{bob}	11:B	ents={⟨fid_{foo}, “foo2”, 12:B⟩}
	fid_{foo}	12:B	bptrs=⟨fid_{bob}, ”foo2”⟩

5.2 Scenario: delete-update conflict

In this example, directory `/` and file `/foo` are both initially replicated on two nodes, {A, B}. Alice on node A deletes `/foo`, while Bob on node B edits and updates `/foo`.

1. Initially, the replicas are consistent:

A	$fid_{/}$	5:A	ents={⟨fid_{foo}, “foo”, 6:A⟩}
	fid_{foo}	6:A	bptrs={⟨fid_{foo}, “foo”⟩}
B	$fid_{/}$	5:A	ents={⟨fid_{foo}, “foo”, 6:A⟩}
	fid_{foo}	6:A	bptrs={⟨fid_{foo}, “foo”⟩}

2. Alice deletes file `/foo`.

A	$fid_{/}$	10:A	ents={*⟨fid_{foo}, “foo”, 11:A⟩}
	fid_{foo}	11:A	bptrs={}
B	$fid_{/}$	5:A	ents={⟨fid_{foo}, “foo”, 6:A⟩}
	fid_{foo}	6:A	bptrs={⟨fid_{foo}, “foo”⟩}

3. B edits file `/foo`.

A	$fid_{/}$	10:A	ents={*⟨fid_{foo}, “foo”, 11:A⟩}
	fid_{foo}	11:A	bptrs={}
B	$fid_{/}$	5:A	ents={⟨fid_{foo}, “foo”, 6:A⟩}
	fid_{foo}	11:B	bptrs={⟨fid_{foo}, “foo”⟩}

Consider two cases, depending on whose update timestamp is larger.

Case 1: 11:B > 11:A: The update for file fid_{foo} is sent from node B to A. Node A revives file fid_{foo} and schedules procedure ProcessUupdate to be called. This procedure will revive fid_{foo}'s entry in directory $fid_{/}$. The change to $fid_{/}$ is sent to node B, which accepts it.

A	$fid_{/}$	12:A	ents={⟨fid_{foo}, “foo”, 11:B⟩}
	fid_{foo}	11:B	bptrs={⟨fid_{foo}, “foo”⟩}
B	$fid_{/}$	12:A	ents={⟨fid_{foo}, “foo”, 11:B⟩}
	fid_{foo}	11:B	bptrs={⟨fid_{foo}, “foo”⟩}

Case 2: 11:A > 11:B. Node B's update to fid_{foo} is sent to A, but is ignored. Node A's update to fid_{foo} is sent to B. Node B removes fid_{foo} and its entry in $fid_{/}$. B sends its update to `/` back to A. In the end, the state of the nodes looks like below:

A	$fid_{/}$	12:B	ents={*⟨fid_{foo}, “foo”, 11:A⟩}
	fid_{foo}	11:A	bptrs={}
B	$fid_{/}$	12:B	ents={*⟨fid_{foo}, “foo”, 11:A⟩}
	fid_{foo}	11:A	bptrs={}

5.3 Scenario: rmdir-update conflict

This section shows how Pangaea resolves Example 2.

1. Initially, all the replicas are consistent:

A	$fid_{/}$	5:A	ents={⟨fid_{foo}, “foo”, 6:A⟩}
	fid_{foo}	6:A	ents={}, bptrs={⟨$fid_{/}$,“foo”⟩}
B	$fid_{/}$	5:A	ents={⟨fid_{foo}, “foo”, 6:A⟩}
	fid_{foo}	6:A	ents={}, bptrs={⟨$fid_{/}$,“foo”⟩}

2. Alice creates file `/foo/bar`.

A	$\mathit{fid}_{/}$	5:A	ents={⟨fid_{foo}, “foo”, 6:A⟩}
	fid_{foo}	10:A	ents={⟨fid_{bar}, “bar”, 11:A⟩}, bptrs={⟨$\mathit{fid}_{/}$,“foo”⟩}
	fid_{bar}	11:A	bptrs={⟨fid_{foo},“bar”⟩}
B	$\mathit{fid}_{/}$	5:A	ents={⟨fid_{foo}, “foo”, 6:A⟩}
	fid_{foo}	6:A	ents={}, bptrs={⟨$\mathit{fid}_{/}$,“foo”⟩}

3. Bob removes directory fid_{foo}.

A	$\mathit{fid}_{/}$	5:A	ents={⟨fid_{foo}, “foo”, 6:A⟩}
	fid_{foo}	10:A	ents={⟨fid_{bar}, “bar”, 11:A⟩}, bptrs={⟨$\mathit{fid}_{/}$,“foo”⟩}
	fid_{bar}	11:A	bptrs={⟨fid_{foo},“bar”⟩}
B	$\mathit{fid}_{/}$	8:B	ents={*⟨fid_{foo}, “foo”, 10:B⟩}
	fid_{foo}	10:B	ents={}, bptrs={}

Consider two cases, depending on whose update timestamp is larger.

Case 1: 11:B $>$ 11:A $>$ 10:B $>$ 10:A.

(a) Updates to $\mathit{fid}_{/}$ and fid_{dir} are sent from node B to A. Node A deletes `/dir` but notices that file fid_{foo} has become an orphan and puts it in *ULOG* (Listing 11).

A	$\mathit{fid}_{/}$	8:B	ents={*⟨fid_{dir}, “dir”, 10:B⟩}
	fid_{dir}	10:B	ents={}, bptrs={}
	fid_{foo}	11:A	bptrs={⟨fid_{dir},“foo”⟩}
B	$\mathit{fid}_{/}$	8:B	ents={*⟨fid_{dir}, “dir”, 10:B⟩}
	fid_{dir}	10:B	ents={}, bptrs={}

(b) Update to fid_{dir} is sent from node A to B, but is ignored by B.

(c) Later, node A runs ProcessUupdate. A resurrects directory fid_{dir}. The update for fid_{dir} is sent to node B, and B applies this update. The final state on both the nodes will become like below:

A	$\mathit{fid}_{/}$	12:A	ents={⟨fid_{dir}, “dir”, 13:A⟩}
	fid_{dir}	13:A	ents={⟨fid_{foo}, “foo”, 11:A⟩}
	fid_{foo}	11:A	bptrs={⟨fid_{dir},“foo”⟩}
B	$\mathit{fid}_{/}$	12:A	ents={⟨fid_{dir}, “dir”, 13:A⟩}
	fid_{dir}	13:A	ents={⟨fid_{foo}, “foo”, 11:A⟩}

Case 2: 11:A $>$ 11:B $>$ 10:A $>$ 10:B.

(a) The update to $\mathit{fid}_{/}$ is sent to A, which accepts it. The update to fid_{dir} is sent to A, which rejects it. At this moment, A will notice that directory fid_{dir} has an orphan and puts fid_{dir} in *ULOG*.

A	$\mathit{fid}_{/}$	8:B	ents={*⟨fid_{dir}, “dir”, 10:B⟩}
	fid_{dir}	10:A	ents={⟨fid_{foo}, “foo”, 11:A⟩}, bptrs={⟨$\mathit{fid}_{/}$,“dir”⟩}
	fid_{foo}	11:A	bptrs={⟨fid_{dir},“foo”⟩}
B	$\mathit{fid}_{/}$	8:B	ents={*⟨fid_{dir}, “dir”, 10:B⟩}
	fid_{dir}	10:B	ents={}, bptrs={}

(b) The update to fid_{dir} is sent from node A to B, which it accepts. Node B puts fid_{dir} in *ULOG*, because the directory entry corresponding to fid_{dir}'s backpointer is missing.

A	$\mathit{fid}_{/}$	8:B	ents={*⟨fid_{dir}, “dir”, 10:B⟩}
	fid_{dir}	10:A	ents={⟨fid_{foo}, “foo”, 11:A⟩}, bptrs={⟨$\mathit{fid}_{/}$,“dir”⟩}
	fid_{foo}	11:A	bptrs={⟨fid_{dir},“foo”⟩}
B	$\mathit{fid}_{/}$	8:B	ents={*⟨fid_{dir}, “dir”, 10:B⟩}
	fid_{dir}	10:A	ents={⟨fid_{foo}, “foo”, 11:A⟩}, bptrs={⟨$\mathit{fid}_{/}$,“dir”⟩}

(c) Node A fixes the inconsistency between fid_{dir} and $\mathit{fid}_{/}$. A sends the update to $\mathit{fid}_{/}$ to B.

A	$\mathit{fid}_{/}$	13:A	ents={⟨fid_{dir}, “dir”, 10:A⟩}
	fid_{dir}	10:A	ents={⟨fid_{foo}, “foo”, 11:A⟩}, bptrs={⟨$\mathit{fid}_{/}$,“dir”⟩}
	fid_{foo}	11:A	bptrs={⟨fid_{dir},“foo”⟩}
B	$\mathit{fid}_{/}$	13:A	ents={⟨fid_{dir}, “dir”, 10:A⟩}
	fid_{dir}	10:A	ents={⟨fid_{foo}, “foo”, 11:A⟩}, bptrs={⟨$\mathit{fid}_{/}$,“dir”⟩}

6 Periodic recovery and garbage collection

The protocol described so far never removes the replica object even after Unlink—it removes the replica contents but keeps attributes, such as *ts*, *gpeers*, and *deadBptr*, as death certificates. Death certificates must be removed eventually, or the disk will become filled with junk. Pangaea runs a garbage-collection module periodically (every three nights by default) to improve the "health" of the system by culling old tombstones and mending a file's replica graph. Listing 14 shows the garbage collection algorithm. We assume a reliable failure detection here. Specifically:

1. A node's permanent death can accurately be detected. All live nodes, within a fixed time period, agree on which nodes have died permanently.
2. If, for any reason, a node declared permanently dead (by other nodes) comes back, it must wipe the disk out, assume a new node ID, and join the system from scratch.

In practice, these conditions can easily be satisfied. One solution is simply to have the system administrator declare a node's decommissioning manually. Alternatively, we can use standard heart-beat techniques with an extremely large timeout value, such as a month; the usual cause of inaccurate failure detection, such as network partitioning and slow nodes, cannot persist for a month in practice. The second condition can be maintained by node checking its clock on reboot and reinitializing itself if it has been down for longer than a month.

```
// Called every third night for every replica on the node.
proc GarbageCollection
   LiveNodes: $\mathbb{P}_1$ NodeID // Set of live nodes.
   r: Replica // Replica to be inspected.
   EXPIRE: integer // Dead-replica expiration period, e.g., a month.
--------------------------------------------------------------------
// Remove old tombstones. Removing after EXPIRE seconds is safe
// because we cannot receive any new update with timestamp older
// than EXPIRE after removing r.
if $\neg$ IsLive(r) and $r.ts < Newtimestamp - EXPIRE$ then
   $DISK \leftarrow \{r.fid\} \lhd DISK$
   return

$r' \leftarrow$ Deepcopy(r)

// Remove dead entries in the directory
if r' is a directory then
   for $(key \mapsto val)$: r'.ents •
      if $\neg$ val.valid and $val.ts <$ Newtimestamp() - EXPIRE then
         $r'.ents \leftarrow \{key\} \lhd r'.ents$
   if at least one entry has been removed from r'.ents then
      $r'.ts \leftarrow$ NewTimestamp()
      UpdateReplica(r)

// If some gold peers are found dead, recreate one elsewhere
if $me \in r.gpeers$ and $r.gpeers \not\subseteq Livenodes$ then
   $newNodes \leftarrow$ Pick $\|r.gpeers \setminus Livenodes\|$ random live nodes.
   $r'.gpeers \leftarrow r.gpeers \setminus Livenodes \cup newNodes$
   $r'.ts \leftarrow$ NewTimestamp()
   UpdateReplica(r)

// If we find graph edges to dead nodes, re-span it.
for $n \in r.peers \setminus Livenodes$
   Add edges between r and a random replica.
   $r.peers \leftarrow r.peers \cap Livenodes$
```

Listing 14: Periodic recovery from permanent failures. The failure detection service supplies the list of live nodes in variable *LiveNodes*.

7 Correctness

We defined Pangaea's consistency guarantee informally in Section 1.1: (1) the state of all replicas of a file converges, (2) every file has a valid path name, and (3) no directory entry points to a non-existent file. Notice that we do not guarantee that executing high-level operations (e.g., Unlink) actually produces the results expected by the user. Such a guarantee is ultimately impossible when conflicts happen. Moreover, our protocol may undo a `rmdir` operation on rare occasions to maintain the consistency of name-space operations (Section 1.3.3). Pangaea does try to minimize the possibility of lost updates by letting nodes wait in certain situations, as we discuss in Section 8.2.

Criterion (1) is ensured when all replicas of a file receive every update and make identical decisions regarding conflicts. That every replica receives every update is ensured by our update-flooding protocol described in [11]. The use of *CLOG* guarantees that update propagation is fault tolerant. That replicas make identical decisions is guaranteed by having a replica's state identified by a globally unique timestamp $\langle 6 \rangle$, and by having all replicas pick the update with the largest timestamp $\langle 16 \rangle$.

In the following, recall that by "live" replica we mean that it has a non-empty list of backpointers or that it is a replica of the root directory. By "a valid entry" we mean that the entry is valid for long enough.

Criteria (2) and (3) can be proven using the following properties.

(i) If a file replica is live, then there is a live local replica of every directory referenced by a a valid backpointer of the file replica (namespace containment).

```
$\forall$ f: ran(DISK) and IsLive(f) •
( $\langle d.fid, fname \rangle \in f.bptrs$
   $\Rightarrow$ $d \in$ dom(DISK) and IsLive(d) )
```

(ii) If a directory points to a file and a replica of the file does not have a backpointer to the directory, then eventually the corresponding directory entry is removed.

```
$\forall$ d: ran(DISK) and IsLive(d) •
( f: ran(DISK) and IsLive(f) •
  $(\langle d.fid, fname \rangle \mapsto ent) \in d.ents$ and ent.valid
    and $\langle d.fid, fname \rangle \notin f.bptr$
  $\Rightarrow \Diamond\ \langle f.fid, fname \rangle \mapsto ent) \notin d.ents$ )
```

(iii) If a live directory replica contains a valid entry that points to a file, then there exist live replicas of the file.

```
$\forall$ d: ran(DISK) and IsLive(d) •
( $\langle f.fid, fname \rangle \mapsto ent) \in d.ents$ and ent.valid
   $\Rightarrow$ f: ran(DISK') and IsLive(f) )
```

The namespace containment property (property (i)) implies that every file with live replicas has a valid path name—it is eventually referenced by directories with replicas on the same nodes as the file replicas. Thus, criterion (2) holds. Properties (ii) and (iii) state that directory antries and backpointers are eventually mutually consistent—a directory entry exists if and only if there exist live replicas of a file that have back pointers to the directory. Thus, criterion (3) also holds.

Proof of property (i): A change to the backpointer is initiated by high-level directory operations (Section 3) or by remote update processing (Listing 10).

The precondition of the user-initiated operations (e.g., condition $\langle 13 \rangle$ in Listing 4) demands that the target directory be locally stored and live. Thus, when a backpointer is created in some replica by one of these operations, a local replica of the directory already exists.

Remote update processing specifically creates all the parent directories of the new replica contents using CreateReplica and ResurrectDirectory before calling UpdateReplica. Thus, when the backpointer is created, a local replica of the directory already exists.□

Proof of property (ii): Assume that file f is pointed to by a valid entry $d.ents$ of directory d, but there is no backpointer from a replica of f back to d.

This is possible when the backpointer is removed. There are two cases where the backpointer is removed, while a directory entry still exists.

Case 1: During an `unlink` operation (or the unlink part of a `rename` operation), the backpointer of the file replica is removed before UpdateReplica is called (listing 5). IssueCupdate propagates the backpointer removal to the other replicas of the file $\langle 19 \rangle$. Criterion (1) guarantees that the coresponding update is disseminated to all file replicas. Any backpointer changes are also recorded in *ULOG*. ProcesUupdate eventually processes every record of *ULOG* and any backpointer removals are reflected on the local directory replica, i.e., the corresponding entry is removed. The new directory entries are propagated to the other directory replicas through IssueCupdate. Criterion (1) again guarantees that the coresponding update is disseminated to all directory replicas.

Case 2: When a potential conflict with the backpointers is detected and resolved $\langle 17 \rangle$. Similarly to case 1, IssueUupdate records the resolved backpointer change (which may remove a backpointer) in *ULOG*.When the *ULOG* is processed, any removed backpointers are reflected on removed entries in the replicas of the corresponding directories.

In any case where a backpointer is created (`Create`, `Link`, `Rename`, CreateReplica), the backpointer is created before the corresponding directory entry.□

Proof of property (iii):

If there was no live replica of the file, that means that there would be no backpointer corresponding to this directory entry. Given property (ii), the directory entry would have been evetnually removed from all directory replicas. Thus, for the directory entry to remain valid, there must be at least one active file replica that has a backpointer to the file and that backpointer prevails any potential confilct resolutions. Eventually, the backpointer becomes valid in all replicas of the file.□

8 Implementation considerations

8.1 Implementing directories

Directory entries in Pangaea are identified by tuple $\langle$FileID, filename$\rangle$ to simplify conflict resolution (Section 2). This design raises two issues. First, it would be slow if implemented naively, because most directory operations find files by names. Second, it must let the user distinguish two different files with the same name.

The first problem is solved by implementing a directory as a mapping from a string (filename) to a set of directory entries. In absence of conflicting updates, a single filename is mapped to a single entry, achieving the same level of efficiency as the traditional file system's.

The second problem is solved by converting filenames during a directory-scanning request (`readdir`). When Pangaea finds a filename with multiple entries during `readdir`, it disambiguates them by appending (a textual representation of) their file IDs. For example, when filename `foo` is shared by two entries, one for a file with ID of 0x83267 and another with ID of 0xa3c28, the user will see two filenames, `foo@@83267` and `foo@@a3c28`. Future file-related system calls, such as `open` and `rename`, will locate a unique entry from given one of these filenames. The separator between the original filename and the suffix ("`@@`") should be a string that usually does not appear in a filename—otherwise, a user cannot create a file with a name that contains the separator.

8.2 Choosing U-update timeout periods

ProcessUupdate is the central procedure that fixes inconsistency between a file's backpointer and the corresponding directory entry. For two reasons, it is a good idea to wait before fixing the inconsistency.

Immediate execution often results in a storm of updates. When name-space inconsistency is found, ProcessUupdate will be scheduled on every node that replicates that file. If executed immediately, these nodes will broadcast the same resolution result to one another, thereby wasting the disk and network bandwidth. By adding a random delay, however, there is a high chance that one replica resolves the problem and tells other replicas to accept the resolution. On all other replicas, when ProcessUupdate runs, it merely confirms that the name-space is consistent and exits.

For instance, suppose that file `/foo/bar` is replicated on nodes {A, B}, and Alice on node A deletes file `/foo/bar`. Node A pushes the update to `bar` to node B, and node B puts the file's replica in *ULOG*. In this situation, node B should wait for some period and let node A execute ProcessUupdate, update `/foo`, and propagate the change to B.

Immediate execution may undo an operation against the user's expectation. IssueUupdate is called when a directory is removed, but some live files are found under it (steps ⟨15⟩ and ⟨18⟩). For example, suppose that directory `/foo` and file `/foo/bar` are replicated on nodes {A,B}, and Alice on node A does `rm -rf /foo`. The update to `/foo` may arrive at node B before the update to `bar`, in which case node B will register `/foo` in *ULOG* (because file `bar` is still live). If node B executes ProcessUupdate before receiving the update to `foo`, it will end up undoing Alice's change. Rather, node B should wait for awhile, in the hope that update to `bar` will arrive during the wait.

On the other hand, delaying executing ProcessUupdate for too long will prolong the state of inconsistency. We thus set the following guidelines for the waiting period.

- For a change that happens as a direct result of local filesystem operations, ProcessUupdate should be executed immediately, because the user expects that. In particular, procedures Create, Unlink, and Rename all calls UpdateReplica, which in turn call IssueUupdate. In these situations, ProcessUupdate should be executed immediately.
- For IssueUupdate called as a result of remote update, ProcessUupdate should wait for fairly long, e.g., 3 minutes.

9 Assessing Pangaea's storage overhead due to name-space containment

The name-space containment property increases the storage demand by forcing each node to store directories that it will not actually use. This section evaluates the overhead of this requirement.

Due to the lack of wide-area file system trace, we take a Redhat Linux distribution (workstation version 7.3) and analyze the storage overheads of the system statically, assuming that a distributed group of users stores the Redhat-7.3 files in Pangaea servers. We measure the storage overhead by the percentage of 512-byte disk blocks used by directories, averaged over all nodes in the system. The storage overhead is determined by four parameters:

- *Number of gold replicas per file*. When a user creates a file, a fixed number of gold replicas are placed on nodes chosen semi-randomly. A node may therefore store a gold replica without its local users never accessing it. For each gold replica, all the intermediate directories in its path must also be replicated on the same node. Having more gold replicas will thus increase the space overhead. We vary this parameter from two to four.
- *Gold-replica placement policy.* We experiment with two policies. The *random* policy chooses gold-replica sets for each file or directory uniformly randomly. The *dir* policy chooses gold-replica sets uniformly randomly for a directory, but for regular files in the directory, it chooses the set the same as the directory's. This policy, similar to Archipelago's [5] and Slice's [2], helps directories to be concentrated on fewer nodes and lower the space overhead.
- *Average number of bronze replicas per file*. Bronze replicas impose the same storage overhead as gold replicas. Bronze replicas, however, are created only when the users wants to access it, and we can expect some access locality that improves the storage overhead. We discuss the locality issue below. We change this parameter from 0 to 100.
- *Degree of file-access locality.* In general, we expect some locality in the file-access pattern of a user. In other words, when a user accesses a file, he or she will also access other files in the same directory. We model this behavior via the *degree of file-access locality*. For example, if the value of this parameter is 10%, then 10% of files in a directory are stored on the same node as bronze replicas. We change this parameter from 10% to 100%.

The storage overhead is independent of the number of nodes or users in the system, as an additional node will only increase the size of directories and files proportionally. As it stands, Redhat-7.3 stored on a local file system—i.e., the one-gold-replica, zero-bronze-replica configuration—uses 0.3% of the disk blocks for directories; this configuration sets the lower bound.

Figure 4 shows the result of analysis. Graph (a) shows the storage overhead with the "dir" placement policy. When the number of bronze replicas is small, all the configurations have storage overhead of about 2%. The number of gold replicas plays little role here, because most of the directories will be shared by files below them in the name-space. As the number of bronze replicas grow with low access locality, the overhead grows, since that forces nodes to create a separate directory hierarchy for each replica.

Graph (b) shows storage overhead with the "random" placement. Overall, the random placement shows higher overhead than "dir" placement, since it forces replicating many directories used only to look up the replica of a single file. As more bronze replicas are created, the overhead will be determined by their number and access locality, because the storage space will be dominated by bronze replicas.

Overall, the system uses at most 25% more space than the optimal. Given that we already accepted using 2x to 4x more space to replicate files, we believe that additional 25% of overhead is reasonable. However, the system should try to consolidate the placement of gold replicas from the same directory, since it dramatically lowers storage overhead with no adversarial side effect.

10 Conclusion

This paper described Pangaea's protocol for maintaining the consistency of the filesystem's name space. Because of Pangaea adopts a pervasive and optimistic replication policy, it must run a distributed protocol to inform the conflict-resolution decision by one replica to its parent directories.

Our protocol embeds backpointers in each file to define its position in the name space authoritatively. Directory operations do not directly modify directory entries—they merely change the file's backpointer and let a generic conflict-resolution routine to reflect the change back to the directory. We proved that this protocol guarantees the consistency of the file system—namely, that all replicas of a file become identical, each file has a valid path name, and every directory entry points to a valid file.

Our protocol demands that for every file, all intermediate directories up to the root directory are stored on the same node. We showed overhead caused by this requirement is somewhere between 3% and 25%. Given that we already accepted using 2x to 4x more space to replicate files, we believe that using additional 25% of space is reasonable.

References

[1] Atul Adya, William J. Bolosky, Miguel Castro, Ronnie Chaiken, Gerald Cermak, John R. Douceur, John Howell, Jacob R. Lorch, Marvin Theimer, and Roger Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *5th Symp. on Op. Sys. Design and Impl. (OSDI)*, Boston, MA, USA, December 2002.

[2] Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Interposed Request Routing for Scalable Network Storage. In *4th Symp. on Op. Sys. Design and Impl. (OSDI)*, pages 259–272, San Diego, CA, USA, October 2000.

[3] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. In *6th Symp. on Princ. of Distr. Comp. (PODC)*, pages 1–12, Vancouver, BC, Canada, August 1987.

[4] Armando Fox and Eric A. Brewer. Harvest, yield, and scalable tolerant systems. In *6th Workshop on Hot Topics in Operating Systems (HOTOS-VI)*, pages 174–178, Rio Rico, AZ, USA, March 1999. `http://www.csd.uch.gr/~markatos/papers/hotos.ps`.

[5] M. Ji, E. Felten, R. Wang, and J. P. Singh. Archipelago: an island-based file system for highly available and scalable Internet services. In *USENIX Windows Systems Symposium*, August 2000.

[6] Paul R. Johnson and Robert H. Thomas. RFC677: The maintenance of duplicate databases. http://www.faqs.org/rfcs% -/rfc677.html, January 1976.

[7] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. on Comp. Sys. (TOCS)*, 10(5):3–25, February 1992.

[8] David L. Mills. RFC1305: Network Time Protocol (version 3). http://www.faqs.org/rfcs/rfc1305.html, March 1992.

[9] D. Scott Parker, Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce Walker, Evelyn Walton, Johanna Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. on Software Engineering*, SE-9(3):240–247, 1983.

[10] David H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, UC Los Angeles, 1998. Tech. Report. no. UCLA-CSD-970044.

[11] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *5th Symp. on Op. Sys. Design and Impl. (OSDI)*, Boston, MA, USA, December 2002.

[12] Michael Spivey. *The Z notation: A Reference Manual*. Prentice Hall, 1992. Online copy available at http://spivey.oriel.ox.ac.uk/ mike/zrm/.

[13] John D. Strunk, Garth R. Goodson, Adam G. Pennington, Craig A.N. Soules, and Gregory R. Ganger. Intrusion detection, diagnosis, and recovery with self-securing storage. Technical Report CMU-CS-02-140, CMU SCS, May 2002.

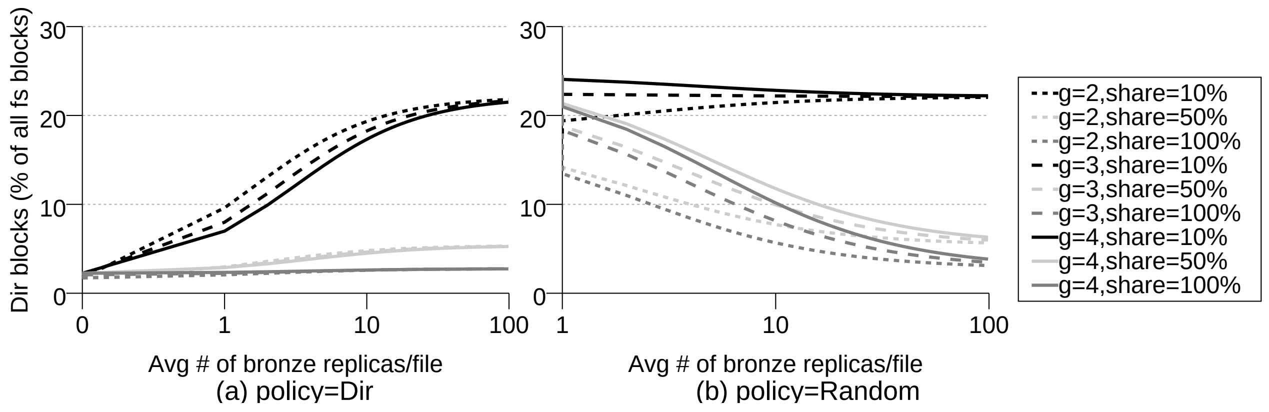

Figure 4: The percentage of disk space occupied by directories. Label "g=" shows the number of gold replicas, and Label "share=" shows the degree of access locality.

[14] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Sys. (TODS)*, 4(2):180–209, June 1979.

[15] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *9th Symp. on Op. Sys. Principles (SOSP)*, pages 49–70, Bretton Woods, NH, USA, October 1983.

[16] Haifeng Yu and Amin Vahdat. The Costs and Limits of Availability for Replicated Services. In *18th Symp. on Op. Sys. Principles (SOSP)*, pages 29–42, Lake Louise, AB, Canada, October 2001.

[17] Zheng Zhang and Christos Karamanolis. Designing a robust namespace for distributed file services. In *20th Symp. on Reliable Dist. Sys (SRDS)*, New Orleans, LA, USA, October 2001.