USENIX Association

Proceedings of the 5th Symposium on Operating Systems Design and Implementation

Boston, Massachusetts, USA
December 9–11, 2002

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Taming aggressive replication in the Pangaea wide-area file system

Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam
Storage Systems Department, HP Labs, Palo Alto, CA, USA
{ysaito,christos,karlsson,mmallik}@hpl.hp.com

Abstract

Pangaea is a wide-area file system that supports data sharing among a community of widely distributed users. It is built on a symmetrically decentralized infrastructure that consists of commodity computers provided by the end users. Computers act autonomously to serve data to their local users. When possible, they exchange data with nearby peers to improve the system's overall performance, availability, and network economy. This approach is realized by aggressively creating a replica of a file whenever and wherever it is accessed.

This paper presents the design, implementation, and evaluation of the Pangaea file system. Pangaea offers efficient, randomized algorithms to manage highly dynamic and potentially large groups of file replicas. It applies optimistic consistency semantics to replica contents, but it also offers stronger guarantees when required by the users. The evaluation demonstrates that Pangaea outperforms existing distributed file systems in large heterogeneous environments, typical of the Internet and of large corporate intranets.

1 Introduction

Pangaea is a wide-area file system that supports the daily storage needs of a distributed community of users. It is a platform for ad-hoc data sharing—it enables multinational corporations, distributed groups of collaborating users, and content management systems to exchange data efficiently using a file system.

Pangaea builds a unified file system across a federation of up to thousands of widely distributed computers connected by dedicated or virtual private networks. We currently assume that all servers are trusted; relaxing the trust relationship is future work. The system faces continuous reconfiguration, with users moving, companies restructuring, and computers being added or removed. Thus, Pangaea must meet three key goals:

Speed: Hide the wide-area networking latency; file access speed should resemble that of a local file system.

Availability and autonomy: Avoid depending on the availability of any specific node. Pangaea must adapt automatically to server additions, removals, failures and network partitioning.

Network economy: Minimize the use of wide-area networks. Nodes are not distributed uniformly; some nodes are in the same LAN, whereas some others are half way across the globe. Pangaea should transfer data between nodes in physical proximity, when possible, to reduce latency and save network bandwidth.

We argue that a system should follow a *symbiotic* design to achieve these goals in dynamic, wide-area environments. In such a system, each server functions autonomously and allows reads and writes to its files even when disconnected. As more computers become available, or as the system configuration changes, servers dynamically adapt and collaborate with each other, in a way that enhances the overall performance and availability of the system.

Pangaea realizes symbiosis by *pervasive replication.* It aggressively creates a replica of a file or directory whenever and wherever it is accessed. There is no single "master" replica of a file. Any replica may be read or written at any time, and replicas exchange updates among themselves in a peer-to-peer fashion. Pervasive replication achieves high performance by serving data from a server close to the point of access, high availability by letting each server contain its working set, and network economy by transferring data among close-by replicas. The following sections introduce two key strategies used to implement pervasive replication.

1.1 Graph-based replica management

Pangaea's replica management must satisfy three goals. First, it must support a large number of replicas, to maximize availability. Second, it needs to manage the replicas of each file independently, since it is difficult to predict file-access patterns accurately in a wide area. Third, it needs to support dynamic addition and removal of replicas even when some nodes are not available. Pangaea addresses these challenges by maintaining a sparse, yet strongly connected and randomized graph of replicas for each file. The graph is used both to propagate updates and to discover other replicas during replica addition and removal. This design offers three important benefits:

Available and inexpensive membership management: A replica can be added by connecting to a few live replicas that it discovers, no matter how many other replicas are unavailable. Since the graph is sparse, adding or removing a replica involves only a constant cost, regardless of the total number of replicas.

Available update distribution: Pangaea can distribute updates to all live replicas of a file as far as its graph is connected. The redundant and flexible nature of graphs makes them extremely unlikely to be disconnected even after multiple node or link failures.

Network economy: The random-graph design facilitates the efficient use of wide-area network bandwidth, for a system with an aggressive replication policy. Pangaea achieves this by clustering replicas in physical proximity tightly in the graph, and by creating a spanning tree along faster edges dynamically during update propagation.

1.2 Optimistic replica coordination

A distributed service faces two inherently conflicting challenges: high availability and strong data consistency [8, 37]. Pangaea aims at maximizing availability: at any time, users must be able to read and write any replica and the system must be able to create or remove replicas without blocking.

To address this challenge, Pangaea uses two techniques for replica management. First, it pushes updates to replicas rather than invalidating them, since the former achieves higher availability in a wide area by keeping up-to-date data in more locations. This approach may result in managing unnecessary replicas, wasting both storage space and networking bandwidth. To ameliorate this problem, Pangaea lets each node remove inactive replicas, as discussed in Section 4.4.

Second, Pangaea manages replica contents optimistically. It lets any node issue updates at any time, propagates them among replicas in the background, and detects and resolves conflicts after they happen. Thus, Pangaea supports only "eventual" consistency, guaranteeing that a user sees a change made by another user in some unspecified future time. Recent studies, however, reveal that file systems face very little concurrent write sharing, and that users demand consistency only within a window of minutes [31, 35]. Pangaea's actual window of inconsistency is around 5 seconds in a wide area, as we show in Section 7.6. In addition, Pangaea provides an option that synchronously pushes updates to all replicas and gives users confirmation of their update delivery (Section 5.3). We thus believe that Pangaea's consistency semantics are sufficient for the ad-hoc data sharing that Pangaea targets.

Pangaea does not support applications that require strong consistency such as open-close consistency, that use locks, or that synchronize using directory operations (i.e., "lock files").

2 Related work

Traditional local-area distributed file systems do not meet our goals of speed, availability, and network economy. Systems such as xFS [2] and Frangipani [33] rely on tight node coordination for replica management and cannot overcome the non-uniform networking latencies and frequent network partitioning that are typical in wide-area networks.

Pervasive replication resembles the persistent caching used in client-server file systems such as AFS [13], Coda [20], and LBFS [21]. Pangaea, however, can harness nodes to improve the system's robustness and efficiency. First, it provides better availability. When a server crashes, there are always other nodes providing access to the files it hosted. Updates can be propagated to all live replicas even when some of the servers are unavailable. The decentralized nature of Pangaea also allows any node to be removed (even permanently) transparently to users. Second, Pangaea improves efficiency by propagating updates between nearby nodes, rather than between a client and a fixed server and, creating new replicas from a nearby existing replica. In this sense, Pangaea generalizes the idea of Fluid replication [16] that utilizes surrogate Coda servers placed in strategic (but fixed) locations to improve the performance and availability of the system.

Pangaea's replication follows an optimistic approach similar to that of mobile data-sharing services, such as Lotus Notes [15], TSAE [10], Bayou [32], and Roam [25]. These systems lack replica location management and rely on polling, usually by humans, to discover and exchange updates between replicas. Pangaea keeps track of replicas automatically and distributes updates proactively and transparently to all the users. Most of these systems replicate at the granularity of the whole database (except Roam, which supports subset replicas). In contrast, Pangaea's files and directories are replicated independently, and some of its operations (e.g., "rename") affect multiple files, each replicated on a different set of nodes. Such operations demand a new protocol for ensuring consistent outcome after conflicts, as we discuss in Section 5.2. Pangaea offers a simple conflict resolution policy similar to that of Roam, Locus [36], or Coda [18]. We chose this design over more sophisticated approaches (as in Bayou), because Pangaea can make no assumptions about the semantics of file-system operations.

FARSITE [1] and Pangaea both build a unified file system across a federation of nodes, but they have different objectives. FARSITE's goal is to build a reliable service on top of untrusted nodes using Byzantine consensus protocols, and it is designed primarily for local-area networks. Pangaea assumes trusted servers, but it dynamically replicates files at the edge to minimize the use of wide-area networks.

Recent peer-to-peer data sharing systems, built on top of load-balanced, fault-tolerant distributed hash tables, share many properties with Pangaea. Systems such as CFS [6] and PAST [27] employ heuristics to exploit physical proximity when locating data, but they do not support concurrent in-place updates of hierarchically structured data. Pangaea, unlike these systems, provides extra machinery for conflict detection and resolution, as we discuss in Section 5.2. Oceanstore [17] builds a file system with strong consistency by routing updates through a small "core" of replicas. Pangaea, instead, allows in-place updating of any replica without centralized coordination to maximize availability. Ivy [22] is a peer-to-peer file system that lets data be updated, any time, anywhere, by exchanging operation logs between replicas. Because the replicas poll remote logs frequently, it supports stronger consistency than Pangaea. Its log-based update propagation also allows for more versatile conflict resolution than in Pangaea. However, because Ivy forces each user to read the logs of all other writers, it can only support a small file system with a small number of writers.

A number of companies are active in the field of wide-area collaborative data sharing, including FileFish, Scale8, WebFS, and Xythos. They offer a uniform, seamless interface for sharing files in a wide-area network, independent of the physical locations of users and data. Some of them provide features such as intelligent location of the cached copy closest to the user. However, they all use a centralized database to keep track of the location of files and replicas. Thus, their design does not meet Pangaea's goals of availability and autonomy.

3 Pangaea: a structural overview

This section overviews the structure of a server and the major data structures it maintains. Pangaea's design follows a symmetrically distributed approach. A Pangaea *server* handles file-access requests from users. We assume that a user uses a single server during a log-in session (lasting, say, a few hours), so that on-demand replication improves file-access latency; the user may move between servers over time. Each server maintains local hard disks, used to store replicas of files and directories. Servers interact with each other in a peer-to-peer fashion to provide a unified file-system.

3.1 Definitions

We use the terms *node* and *server* interchangeably. Nodes are automatically grouped into *regions*, such that nodes within a region have low round-trip times (RTT) between them ($<$5ms in our implementation). Pangaea uses region information to optimize replica placement and coordination.

Pangaea replicates data at the granularity of files and treats directories as files with special contents. Thus, we use the term *file* to refer to a regular file or a directory. An *edge* represents a known connection between two replicas of a file; updates to the file flow along edges. The replicas of a file and the edges between them comprise a strongly connected graph. The set of replicas of a file is called the file's *replica set*.

3.2 Structure of a server

The Pangaea server is currently implemented as a user-space NFSv3 loopback server (Figure 1). The server consists of four main modules:

NFS protocol handler receives requests from applications, updates local replicas, and generates requests for the replication engine. It is built using the SFS toolkit [19] that provides a basic infrastructure for NFS request parsing and event dispatching.

Replication engine accepts requests from the NFS protocol handler and the replication engine running on other nodes. It creates, modifies, or removes replicas, and forwards requests to other nodes if necessary. It is the largest part of the Pangaea server.

Log module implements transaction-like semantics for local disk updates via redo logging. The server logs all the replica-update operations using this service, allowing them to survive crashes.

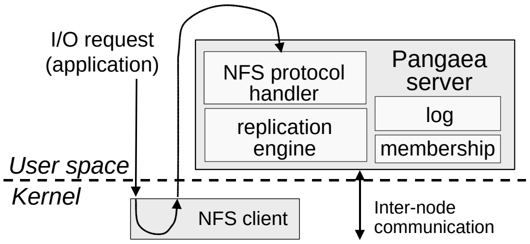

Figure 1: *The structure of the Pangaea server.*

Membership module maintains the status of other nodes, including their liveness, available disk space, the locations of root-directory replicas, the list of regions in the system, the set of nodes in each region, and a round-trip time (RTT) estimate between every pair of regions.

This module runs an extension of van Renesse's gossip-based protocol [34]. Each node periodically sends its knowledge of nodes' status to a random node chosen from its live-node list; the recipient merges this list with its own. A few fixed nodes are designated as "landmarks" and they bootstrap newly joining nodes. The protocol has been shown to disseminate membership information quickly with low probability of false failure detection.

The region and RTT information is gossiped as part of the membership information. A newly booted node obtains the region information from a landmark. It then polls a node in each existing region to determine where it belongs or to create a new singleton region. In each region, the node with the smallest IP address elects itself as a leader and periodically pings nodes in other regions to measure the RTT.

This membership-tracking scheme, especially the RTT management, is the key scalability bottleneck in our system—its network bandwidth consumption in a 10,000-node configuration is estimated to be 10K bytes/second/node. We plan to use external RTT-estimation services, such as IDMaps [9], once they become widely available.

3.3 Structure of a file system

Pangaea decentralizes both the replica-set and consistency management by maintaining a distributed graph of replicas for each file. Figure 2 shows an example of a system with two files. Pangaea 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 from which bronze replicas are found during path-name traversal. To this end, the directory entry of a file lists the file's gold replicas. Second, gold replicas perform several tasks that are hard to do in a completely distributed way. In particular, they are used as pivots to keep the graph connected after a permanent node failure, and to maintain a minimum replication factor for a file. They form a clique in the file's graph so that they can monitor each other for these tasks. These issues are discussed in more detail in Section 4. Currently, Pangaea designates replicas created during initial file creation as gold and fixes their locations unless some of them fail permanently.

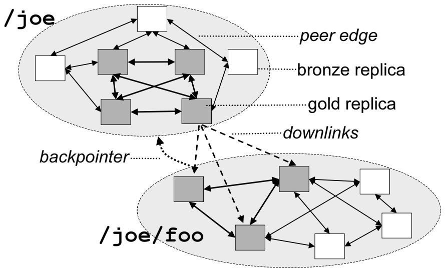

Figure 2: *An example of a directory* `/joe` *and file* `/joe/foo`*. Each replica of* `joe` *stores three pointers to the gold replicas of* `foo`*. Each replica of* `foo` *keeps a backpointer to the parent directory. Bronze replicas are connected randomly to form strongly connected graphs. Bronze replicas also have uni-directional links to the gold replicas of the file, which are not shown here.*

Each replica stores a *backpointer* that indicates its location in the file-system name space. A backpointer includes the parent directory's ID and the file's name within the directory.[1] It is used for two purposes: to resolve conflicting directory operations (Section 5.2), and to keep the directory entry up-to-date when the gold replica set of the file changes (Section 6.2).

Figure 3 shows the key attributes of a replica. The timestamp (*ts*) and the version vector (*vv*) [23] record the last time the file was modified. Their use is described in more detail in Section 5. *GoldPeers* are uni-directional links to the gold replicas of the file. *Peers* point to the neighboring (gold or bronze) replicas in the file's graph.

4 Replica set management

In Pangaea, a replica is created when a user first accesses the file, and it is removed when a node runs out of disk space or finds a replica to be inactive. Because these operations are frequent, they must be carried out efficiently and

[1] A replica stores multiple backpointers when the file is hard-linked. A backpointer need not remember the locations of the parent-directory replicas, since a parent directory is always found on the same node due to the namespace-containment property (Section 4.3).

```
struct Replica
    fid: FileID                   // 96 bit globally unique file ID
    ts: TimeStamp                 // Pair of ⟨physical clock, IP addr⟩.
    vv: VersionVector             // Maps IP addr ↦ TimeStamp
    goldPeers: Set⟨NodeID⟩ // Set of IP addresses
    peers: Set⟨NodeID⟩
    backptrs: Set⟨FileID, String⟩ // Pair of ⟨dirID, fname⟩
    . . .
end
struct DirEntry
    fname: String
    fid: FileID
    downlinks: Set⟨NodeID⟩
    ts: Timestamp
end
```

Figure 3: *Key attributes of a replica.*

without blocking, even when some nodes that store replicas are unavailable. This section describes algorithms based on random walks that achieve these goals.

4.1 File creation

We describe the interactions between the modules of the system and the use of various data structures using a simple scenario—a user on server *S* creates file *F* in directory *D*.

For the moment, assume that *S* already stores a replica of *D* (if not, *S* creates one, using the protocol described in Section 4.2.) First, *S* determines the location of *g* initial replicas of *F*, which will become the gold replicas of the file (a typical value for *g* is 3). One replica will reside on *S*. The other $g-1$ replicas are chosen at random from different regions in the system to improve the expected availability of the file. Second, *S* creates the local replica for *F* and adds an entry for *F* in the local replica of *D*. *S* then replies to the client, and the client can start accessing the file.

In the background, *S* disseminates two types of updates. It first "floods" the new directory contents to other directory replicas. It also floods the contents of *F* (which is empty, save for attributes such as permissions and owner) to its gold-replica nodes. In practice, as we describe in Section 5, we deploy several techniques to reduce the overhead of flooding dramatically. As a side effect of the propagation, the replicas of *D* will point to *F*'s gold replicas so that the latter can be discovered during future path-name lookups.

4.2 Replica addition

The protocol for creating additional replicas for a file is run when a user tries to access a file not present in her local node. Say that a user on node *S* wants to read file *F*. A read or write request is always preceded by a directory lookup (during the `open` request) on *S*. Thus, to create a replica, *S* must replicate the file's parent directory. This recursive step may continue all the way up to the root directory. The locations of root replicas are maintained by the membership service (Section 3.2).

Pangaea performs a *short-cut* replica creation to transfer data from a nearby existing replica. To create a replica of *F*, *S* first discovers the file's gold replicas in the directory entry during the path-name lookup. *S* then requests the file contents from the gold replica closest to *S* (say *P*). *P* then finds a replica closest to *S* among its own graph neighbors (say *X*, which may be *P* itself) and forwards the request to *X*, which in turn sends the contents to *S*. At this point, *S* replies to the user and lets her start accessing the replica. This request forwarding is performed because the directory only knows *F*'s gold replicas, and there may be a bronze replica closer to *P* than the gold ones.

The new copy must be integrated into the file's replica graph to be able to propagate updates to and receive updates from other replicas. Thus, in the background, *S* chooses *m* existing replicas of *F*, adds edges to them, and requests them to add edges to the new replica in *S*. The selection of *m* peers must satisfy three goals:

- Include gold replicas so that they have more choices during future short-cut replica creation.
- Include nearby replicas so that updates can flow through fast network links.
- Be sufficiently randomized so that, with high probability, the crash of nodes does not catastrophically disconnect the file's graph.

Pangaea satisfies all these goals simultaneously, as a replica can have multiple edges. *S* chooses three types of peers for the new replica. First, *S* adds an edge to a random gold replica, preferably one from a different region than *S*, to give that gold replica more variety of regions in its neighbor set. Second, it asks a random gold replica, say *P*, to pick the replica (among *P*'s immediate graph neighbors) closest to *S*. Third, *S* asks *P* to choose $m-2$ random replicas using random walks that start from *P* and perform a series of RPC calls along graph edges. This protocol ensures that the resulting graph is *m* edge- and node- connected, provided that it was *m*-connected before.

Parameter *m* trades off availability and performance. A small value increases the probability of graph disconnection (i.e., the probability that a replica cannot exchange updates with other replicas) after node failures. A large value for *m* increases the overhead of graph maintenance and update propagation by causing duplicate update delivery. We found that $m=4$ offers a good balance in our prototype.

4.3 Name-space containment

The procedures for file creation and replica addition both require a file's parent directory to be present on the same node. Pangaea, in fact, demands that for every file, all intermediate directories, up to the root, are always replicated on the same node. This *name-space-containment* requirement yields two benefits. First, it naturally offers the availability and autonomy benefits of island-based replication [14]. That is, it enables lookup and access to every replica even when the server is disconnected and allows each node to take a backup of the file system locally. We quantify these benefits in Section 7.8. Second, it simplifies the conflict resolution of directory operations, as we discuss in Section 5.2.

On the other hand, this requirement increases the system-wide storage overhead by 1.5% to 25%, compared to an idealized scheme in which directories are stored on only one node [28].[2] We consider the overhead to be reasonable, as users already pay many times more storage cost by replicating files in the first place.

4.4 Bronze replica removal

This section describes the protocol for removing bronze replicas. Gold replicas are removed only as a side effect of a permanent node loss. We discuss the handling of permanent failures in Section 6.

A replica is removed for two possible reasons: because a node has run out of disk space, or the cost of keeping the replica outweighs its benefits. To reclaim disk space, Pangaea uses a randomized GD-Size algorithm [24]. We examine 50 random replicas kept in the node and calculate their merit values using the GD-Size function that considers both the replica's size and the last-access time [5]. The replica with the minimum merit is evicted, and five replicas with the next-worst merit values are added to the candidates examined during the next round. The algorithm is repeated until it frees enough space on the disk.

Optionally, a server can also reclaim replicas not worth keeping. We currently use a competitive updates algorithm for this purpose [12]. Here, the server keeps a per-replica counter that is incremented every time a replica receives a remote update and is reset to zero when the replica is read. When the counter's value exceeds a threshold (4 in our prototype), the server evicts the replica.

To remove a replica, the server sends notices to the replica's graph neighbors. Each neighbor, in turn, initiates a random walk starting from a random gold replica[3] and uses the protocol described in Section 4.2 to establish a replacement edge with another live replica. Starting the walk from a live gold replica ensures that the graph remains strongly connected. A similar protocol runs when a node detects another node's permanent death, as we describe in Section 6.

4.5 Summary

The graph-based pervasive replication algorithms described in this section offer some fundamental benefits over traditional approaches that have a fixed set of servers manage replica locations.

Simple and efficient recovery from failures: Graphs are, by definition, flexible—spanning edges to *any* replica makes the graph incrementally more robust and efficient. Moreover, using just one type of edges both to locate replicas and to propagate updates simplifies the recovery from permanent failures and avoids any system disruption during graph reconfiguration.

Decoupling of directories and files: Directory entries point only to gold replicas, and the set of gold replicas is typically stable. Thus, a file and its parent directory act mostly independently once the file is created. Adding or removing a bronze replica for the file does not require a change to the directory replicas. Adding or removing a gold or bronze replica for the directory does not require a change to the file replicas. These are key properties for the system's efficiency.

5 Propagating updates

This section describes Pangaea's solutions to three challenges posed by optimistic replication: efficient and reliable update propagation, handling concurrent updates, and the lack of strong consistency guarantees.

5.1 Efficient update flooding

The basic method for propagating updates in Pangaea is *flooding* along graph edges, as shown in Figure 4. Whenever a replica is modified on a server, the server pushes the entire file contents to all the graph neighbors, which in turn forward the contents to their neighbors, and so on, until all the replicas receive the new contents. This simple flooding

[2] Due to the lack of wide-area file system traces, we analyzed the storage overhead using a fresh file system with RedHat 7.3 installed. The overhead mainly depends on the spatial locality of accesses, i.e., the degree to which files in the same directory are accessed together. We expect the overhead in practice to be much closer to 1.5% than 25%, because spatial locality in typical file-system traces is usually high.

[3] The gold-replica set is kept as a part of the replica's attributes; see Figure 3.

```
r: Replica being updated.
when Update is newly issued
   Log ⟨r.fid, r.vv⟩.
   Send ⟨r.fid, r.vv, r.data⟩ to nodes in r.peers
   Unlog ⟨r.fid, r.vv⟩ after all the neighbors reply.
when Update ⟨fid, vv, data⟩ is received from node n.
   if this update has already been applied then Reply to n
   else Log and apply the update.
        Reply to n
        Forward the update to r.peers - {n}.
        Unlog after all the neighbors reply.
```

Figure 4: *A simple flooding algorithm to distribute updates. This code assumes that updates are issued one at a time; the handling of concurrent updates is discussed in Section 5.2.*

algorithm guarantees reliable update delivery as long as the replica graph is strongly connected. The following three sections introduce techniques for improving the efficiency of the basic flooding algorithm.

5.1.1 Optimization 1: delta propagation

A major drawback of flooding is that it propagates the entire file contents even when only one byte has been modified. Delta propagation improves the propagation efficiency while maintaining the logical simplicity of flooding. Here, whenever a portion of a file is changed (e.g., adding an entry to a directory), Pangaea propagates only a small, semantic description of the change, called a *delta*. Deltas, in general, must be applied in the same order to every replica to produce the same result. We ensure this by having each delta carry two timestamps: the *old timestamp* that represents the state of the replica just before the change, and the *new timestamp* that shows the state of the replica after the change [15]. A replica applies a delta only when its current timestamp matches the delta's old timestamp. Otherwise, it resorts to full contents transfer, with potential conflict resolution as described in Section 5.2. In practice, updates are handled almost exclusively by deltas, and full-state transfer happens only when there are concurrent writes, or when a node recovers from a crash.

Pangaea further reduces the size of updates by delta merging, akin to the feature implemented in Coda [20]. For example, when a file is deleted right after it is modified (which happens often for temporary files), the server quashes the modification if it has not yet been sent to other replicas. Delta merging is transparent to users because it adds no delay to propagation.

5.1.2 Optimization 2: harbingers

Flooding guarantees reliable delivery by propagating updates (deltas or full contents) over multiple links at each step of the algorithm. Thus, it consumes *m* times the optimal network bandwidth, where *m* is the number of edges per replica. Harbingers eliminate redundant update deliveries.

Pangaea uses a two-phase protocol to propagate updates that exceed a certain size (1KB). In phase one, a small message that only contains the timestamps of the update, called a *harbinger*, is flooded along graph edges. The update bodies are sent, in phase two, only when requested by other nodes. When a node receives a new harbinger, it asks the sender of the harbinger (the immediate upstream replica in the flooding chain) to push the update body. Simultaneously, it forwards the harbinger to other neighbors in the graph. When a node receives a duplicate harbinger without having received the update body, it asks its sender to retry later. This is required because the sender of the earliest harbinger may crash before sending the update body. If a node receives a harbinger after having received the update body, it tells the sender to stop sending the update. We chose the harbinger threshold of 1KB, because we found that delta sizes follow a bimodal distribution—one peak around 200 bytes representing directory operations, and a flatter plateau around 20KB representing bulk writes.[4]

This harbinger algorithm not only saves network usage, but also shrinks the effective window of replica inconsistency. When a user tries to read a file for which only a harbinger has been received, she waits until the actual update arrives. Since harbinger-propagation delay is independent of the actual update size, the chance of a user seeing stale file contents is greatly reduced.

5.1.3 Optimization 3: exploiting physical topology

Harbingers have another positive side effect. They favor the use of fast links, because a node requests the body of an update from the sender of the first harbinger it receives. However, unpredictable node or link load may reduce this benefit. A simple extension to the harbinger algorithm improves the data propagation efficiency, without requiring any coordination between nodes. Before pushing (or forwarding) a harbinger over a graph edge, a server adds a delay proportional to the estimated speed of the edge ($10*$RTT in our implementation). This way, Pangaea dynamically builds a spanning tree whose shape closely matches the physical network topology. Figure 5 shows an example. In Section 7.6, we show that this technique drastically reduces the use of wide-area networks when updating shared files.

[4] Pangaea batches NFS write requests and flushes data to disk and other replicas only after a "commit" request [4]. Thus, the size of an update can grow larger than the typical "write" request size of 8KB.

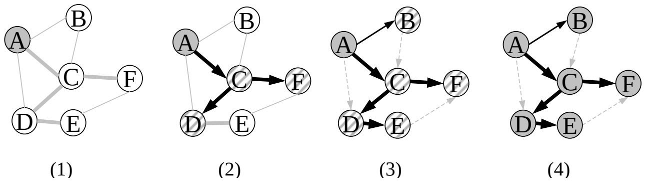

Figure 5: *An example of update propagation for a file with six replicas, A to F. Thick edges represent fast links. (1) An update is issued at A. (2) A sends a harbinger via the fat edge to C. C forwards the harbinger to D and F quickly. (3) D forwards the harbinger to E. After some time, A sends the harbinger to B, and a spanning tree is formed. Links not in the tree are used as backups when some of the tree links fail. (4) The update's body is pushed along the tree edges. In practice, steps 2-4 proceed in parallel.*

5.2 Conflict resolution

With optimistic replication, concurrent updates are inevitable, although rare [35, 31]. We use a combination of version vectors and the last-writer-wins rule to resolve conflicts.

First, recall that when delta timestamps mismatch, servers revert to full-state transfer. We then use version vectors [23] to separate true conflicts from other causes (e.g., missing updates) that can be fixed simply by overwriting the replica. This simplifies conflict resolution.

For conflicts on the contents of a regular file, we currently offer users two options. The first is the "last-writer-wins" rule using update timestamps (attribute *ts* in Figure 3). In this case, the clocks of servers should be loosely synchronized, e.g., using NTP, to respect the users' intuitive sense of update ordering. The second option is to concatenate two versions in the file and let the user fix the conflict manually. Other options, such as application-specific resolvers [36, 18, 32], are certainly possible, but we have not implemented them yet.

Conflicts regarding file attributes or directory entries are more difficult to handle. They fall into two categories. The first is a conflict between two directory-update operations; for example, Alice does "`mv /foo /alice/foo`" and Bob does "`mv /foo /bob/foo`" concurrently. In the end, we want one of the updates to take effect, but not both. The second category is a conflict between "rmdir" and any other operation; for example, Alice does "`mv /foo /alice/foo`" and Bob does "`rmdir /alice`". These problems are difficult to handle, because files may be replicated on different sets of nodes, and a node might receive only one of the conflicting updates and fail to detect the conflict in the first place.

We only outline our solution here, as it is fully described in [28]. Our principle is always to let the child file ("`foo`" in our example), rather than its parent ("`alice`" or "`bob`"), dictate the outcome of the conflict resolution using the "last-writer-wins" rule. We thus let the file's backpointer (Section 3.3) *authoritatively* define the file's location in the file-system namespace. We implement directory operations, such as "mv" and "rm", as a change to the file's backpointer(s). 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.[5] The parent directory will, in turn, flood the change to its replicas. In practice, we randomly delay the directory-entry patching and subsequent flooding, because there is a good chance that other replicas of the file will do the same. Figure 6 illustrates how Pangaea resolves the first conflict scenario. The same policy is used to resolve the mv-rmdir conflict: when a replica detects the absence of the directory entry corresponding to its backpointer, it re-creates the entry, which potentially involves re-creating the directory itself and the ancestor directories recursively, all the way to the root.

A directory in Pangaea is, in effect, merely a copy of the backpointers of its children. Thus, resolving conflicts on directory contents is done by applying the "last-writer-wins" rule to individual entries. If a file is to be removed from a directory, the directory still keeps the entry but marks it as "dead" (i.e., it acts as a "death certificate" [7]), so that we can detect when a stale change to the entry arrives in the future.

5.3 Controlling replica divergence

The protocols described so far do not provide hard guarantees for the degree of replica divergence—consistency is achieved only eventually.

To alleviate this problem, Pangaea introduces an option, called the "red button", to provide users confirmation of update delivery. The red button, when pressed for a particular file, sends harbingers for any pending updates to neighboring replicas. These harbingers (and corresponding updates) circulate among replicas as described in Section 5.1.2. A replica, however, does not acknowledge a harbinger until all the graph neighbors to which it forwarded the harbinger acknowledge it or time out (to avoid deadlocking, a replica replies immediately when it receives the same harbinger twice). The user who pressed the red button waits until the operation is fully acknowledged or some replicas time out, in which case the user is presented with the list of unavailable replicas.

[5] The replica can always find a replica of the parent directory in the same node, because of the name-space-containment property.

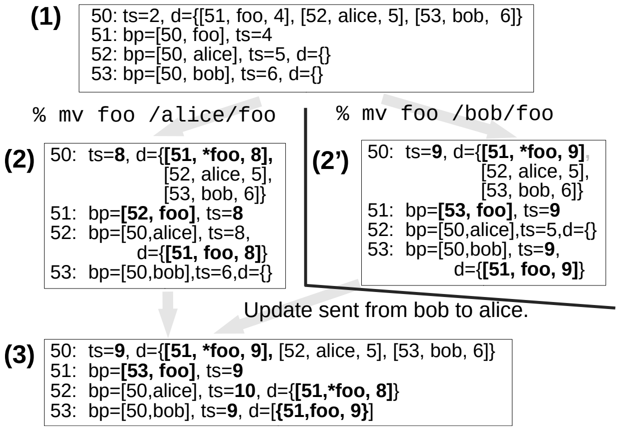
(1)
50: ts=2, d={[51, foo, 4], [52, alice, 5], [53, bob, 6]}
51: bp=[50, foo], ts=4
52: bp=[50, alice], ts=5, d={}
53: bp=[50, bob], ts=6, d={}
% mv foo /alice/foo
% mv foo /bob/foo
(2)
50: ts=8, d={[51, *foo, 8], [52, alice, 5], [53, bob, 6]}
51: bp=[52, foo], ts=8
52: bp=[50,alice], ts=8, d={[51, foo, 8]}
53: bp=[50,bob],ts=6,d={}
(2')
50: ts=9, d={[51, *foo, 9], [52, alice, 5], [53, bob, 6]}
51: bp=[53, foo], ts=9
52: bp=[50,alice],ts=5,d={}
53: bp=[50,bob], ts=9, d={[51, foo, 9]}
Update sent from bob to alice.
(3)
50: ts=9, d={[51, *foo, 9], [52, alice, 5], [53, bob, 6]}
51: bp=[53, foo], ts=9
52: bp=[50,alice], ts=10, d={[51,*foo, 8]}
53: bp=[50,bob], ts=9, d=[{51,foo, 9}]

Figure 6: *Example of conflict resolution involving four files, "/" (FileID=50), "/foo" (FileID=51), "/alice/" (FileID=52), and "/bob/" (FileID=53). "ts=2" shows the replica's timestamp. "bp=[50,foo]" shows that the backpointer of the replica indicates that the file has the name "foo" in the directory 50 ("/"). "d={[51,foo,4]}" means that the directory contains one entry, a file "foo" with ID of 51 and timestamp of 4. Bold texts indicate changes from the previous step. Entries marked "*foo" are death certificates. (1) Two sites initially store the same contents. (2) Alice does "mv /foo /alice/foo". (2') Bob concurrently does "mv /foo /bob/foo" on another node. Because Bob's update has a newer timestamp (ts=9) than Alice's (ts=8), we want Bob's to win over Alice's. (3) When Alice's node receives the update from Bob's, the replica of file 51 will notice that its backpointer has changed from [52, foo] to [53, foo]. This change triggers the replica to delete the entry from `/alice` and add the entry to `/bob`.*

This option gives the user confirmation that her updates have been delivered to remote nodes and allows her to take actions contingent upon stable delivery, such as emailing her colleagues about the new contents. The red button, however, still does not guarantee a single-copy serializability, as it cannot prevent two users from changing the same file simultaneously.

6 Failure recovery

Failure recovery in Pangaea is simplified due to three properties: 1) the randomized nature of replica graphs that tolerate operation disruptions; 2) the idempotency of update operations; including NFS requests; and 3) the use of a unified logging module that allows any operation to be re-started.

We distinguish two types of failures: temporary failures and permanent failures. They are currently distinguished simply by their duration—a crash becomes permanent when a node is suspected to have failed continuously for more than two weeks. Given that the vast majority of failures are temporary [11, 3], we set two different goals. For temporary failures, we try to reduce the recovery cost. For permanent failures, we try to clean all data structures associated with the failed node so that the system runs as if the node had never existed in the first place.

6.1 Recovering from temporary failures

Temporary failures are handled by retrying. A node persistently logs any outstanding remote-operation requests, such as contents update, random walk, or edge addition. A node retries logged updates upon reboot or after it detects another node's recovery. This recovery logic may sometimes create uni-directional edges or more edges than desired, but it maintains the most important invariant, that the graphs are *m*-connected and that all replicas are reachable in the hierarchical name space.

Pangaea reduces the logging overhead during contents-update flooding, by logging only the ID of the modified file and keeping deltas only in memory. To reduce the memory footprint further, when a node finds out that deltas to an unresponsive node are piling up, the sender discards the deltas and falls back on full-state transfer.

6.2 Recovering from permanent failures

Permanent failures are handled by a garbage collection (GC) module. The GC module periodically scans local disks and discovers replicas that have edges to permanently failed nodes. When the GC module finds an edge to a failed bronze replica, it replaces the edge by performing a random walk starting from a gold replica (Section 4.4).

Recovering from a permanent loss of a gold replica is more complex. When a gold replica, say *P*, detects a permanent loss of another gold replica, *P* creates a new gold replica on a live node chosen using the criteria described in Section 4.1. Because gold replicas form a clique (Section 3.3), *P* can always detect such a loss. This choice is flooded to *all* the replicas of the file, using the protocol described in Section 5, to let them update their uni-directional links to the gold replicas. Simultaneously, *P* updates the local replica of the parent directory(ies), found in its backpointer(s), to reflect *P*'s new gold-replica set. This change is flooded to other replicas of the directories. Rarely, when the system is in transient state, multiple gold replicas may initiate this protocol simultaneously. Such a situation is resolved using the last-writer-wins policy, as described in Section 5.2.

Recovering from a permanent node loss is an inherently expensive procedure, because data stored on the failed node must eventually be re-created somewhere else. The problem is exacerbated in Pangaea, because it does not have a central authority to manage the locations of replicas—

all surviving nodes must scan their own disks to discover replicas that require recovery. To lessen the impact, the GC module tries to discover as many replicas that needs recovery as possible with a single disk scan. We set the default GC interval to be every three nights, which reduces the scanning overhead dramatically while still offering the expected file availability in the order of six-nines, assuming three gold replicas per file and a mean server lifetime of 290 days [3].

7 System evaluation

This section evaluates the design and implementation of Pangaea. First, we investigate the baseline performance and overheads of Pangaea and show that it performs competitively with other distributed file systems, even in a LAN. Further, we measure the latency, network economy, and availability of Pangaea in a wide-area networking environment in the following ways:

- We study the latency of Pangaea using two workloads: a personal workload (Andrew benchmark) and a BBS-like workload involving extensive data sharing. For the personal workload, we show that the user sees only local access latency on a node connected to a slow network and that roaming users can benefit by fetching their personal data from nearby sources. Using the second workload, we show that as a file is shared by more users, Pangaea progressively lowers the access latency by transferring data between nearby clients.
- We demonstrate network economy by studying how updates are propagated for widely shared files. We show that Pangaea transfers data predominantly over fast links.
- To demonstrate the effect of pervasive replication on the availability of the system, we analyze traces from a file server and show that Pangaea disturbs users far less than traditional replication policies.

7.1 Prototype implementation

We have implemented Pangaea as a user-space NFS (version 3) server using the SFS toolkit [19]. Our prototype implements all the features described in the paper, except that support for recovery from permanent failures (Section 6) is still fragmentary. Pangaea currently consists of 30,000 lines of C++ code.

A Pangaea server maintains three types of files on the local file system: *data* files, the *metadata* file, and the *intention-log* file. A data file is created for each replica of a file or directory. The node-wide metadata file keeps the extended attributes of all replicas stored on the server, including graph edges and version vectors. Data files for directories and the metadata file are both implemented using the Berkeley DB library [30] that maintains a hash table in a file. The intention-log file is also implemented using the Berkeley DB to record update operations that must survive a node crash. All the Berkeley DB files are managed using its "environments" feature that supports transactions through low-level logging. This architecture allows metadata changes to multiple files to be flushed with a sequential write to the low-level log.

Type	#	CPU	Disk	Mem
A	2	730MHz	Quantum Atlas 9WLS	256MB
B	3	1.8GHz	Quantum Atlas TW367L	512MB
C	4	400MHz	Seagate Cheetah 39236LW	256MB

Table 1: *The type and number of PCs used in the experiments. All the CPUs are versions of Pentiums.*

7.2 Experimental settings

We compare Pangaea to Linux's in-kernel NFS version 3 server and Coda, all running on Linux-2.4.18, with ext3 as the native file system.

We let each Pangaea server serve only clients on the same node. Both Pangaea and NFS flush buffers synchronously to disk before replying to a client, as required by the NFS specifications [4]. Coda supports two main modes of operation: strongly connected mode (denoted *coda-s* hereafter) that provides open-close semantics, and weakly connected mode (denoted *coda-w* hereafter) that improves the response-time of write operations by asynchronously trickling updates to the server. We mainly evaluate coda-w, since its semantics are closer to Pangaea's.

Table 1 shows the machines we used for the evaluation. All the machines are physically connected by a 100Mb/s Ethernet. Disks on all the machines are large enough that replicas never had to be purged in either Pangaea or Coda. For NFS and Coda, we configured a single server on a type-A machine. Other machines are used as clients. For Pangaea, all machines are used as servers and applications access files from their local servers. For CPU-intensive workloads (i.e., Andrew), we used a type-A machine for all the experiments. The other experiments are completely network-bound, and thus they are insensitive to CPU speeds.

For our wide-area experiments, we built a simulated WAN to evaluate Pangaea reliably in a variety of networking conditions. We routed packets to a type-B FreeBSD node (not included in the table) running Dummynet [26] to add artificial delays and bandwidth restrictions. This router node was fast enough never to become a bottleneck in any of our experiments.

7.3 Baseline performance in a LAN

This section evaluates Pangaea's performance in a LAN using a sequential workload without data sharing. While such an environment is not Pangaea's main target, we conducted this study to test Pangaea's ability to serve people's daily storage needs and to understand the system's behavior in an idealized situation.

We created a variation of the Andrew benchmark[6] that simulates a single-person, engineering-oriented workload. It has the same mix of operations as the original Andrew benchmark [13], but the volume of the data is expanded twenty-fold to allow for accurate measurements on modern hardware. This benchmark, denoted *Andrew-Tcl* hereafter, consists of five stages: (1) *mkdir*: creating 200 directories, (2) *copy*: copying the Tcl-8.4 source files from one directory to another, (3) *stat*: doing "ls -l" on the source files, (4) *grep*: doing "du" and "grep" on the source files, and (5) *compile*: compiling the source code. We averaged results from four runs per system, with 95% confidence interval below 3% for all the numbers presented.

Table 2 shows the time to complete the benchmark. Throughout the evaluation, label pang-*N* stands for a Pangaea system with *N* (gold) replicas per file. Pangaea's performance is comparable to NFS. This is as expected, because both systems perform about the same amount of buffer flushing, which is the main source of overhead. Pangaea is substantially slower only in `mkdir`. This is because Pangaea must create a Berkeley DB file for each new directory, which is a relatively expensive operation. Pangaea's performance is mostly independent of a file's replication factor, thanks to optimistic replication, where most of the replication processing happens in the background.

Coda's weakly connected mode (coda-w) is very fast. This is due to implementation differences: whereas Pangaea and NFS flush buffers to disk after every update operation, Coda avoids that by intercepting low-level file-access (VFS) requests using a small in-kernel module.

Figure 7 shows the network bandwidth used during the benchmark. "Overhead" is defined to be harbingers and update messages that turn out to be duplicates. Pang-1 does not involve any network activity since it stores files only on the local server. Numbers for pang-3 and -4 show the effect of Pangaea's harbinger algorithm in conserving network-bandwidth usage. In this benchmark, because all replicas are gold and they form a clique, Pangaea would have consumed 4 to 9 times the bandwidth of pang-2 were it not for harbingers. Instead, its network usage is near-optimal, with less than 2% of the bandwidth wasted.

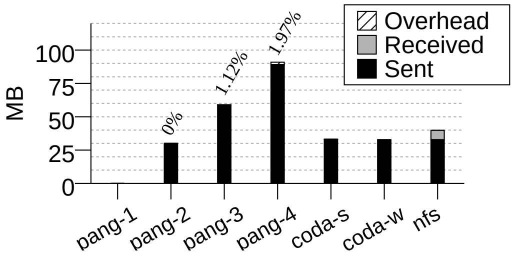

Figure 7: *Network bandwidth consumed during the Andrew benchmark. The "overhead" bars show bytes consumed by harbingers and duplicate updates. The numbers above the bars show the percentage of overhead.*

Table 3 shows network bandwidth consumption for common file-system update operations. Operations such as creating a file or writing one byte show a high percentage of overhead, since they are sent directly without harbingers, but they have only a minor impact on the overall wasted bandwidth since their size is small. On the other hand, bulk writes, which make up the majority of the overall traffic, incur almost no overhead.

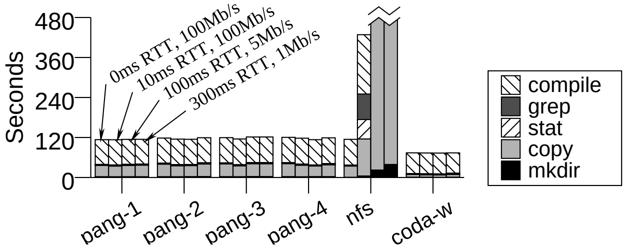

Figure 8: *Andrew-Tcl benchmark results on a node with a slow network link. The labels next to the bars indicate the link speeds. For Pangaea, these are the links between any two servers; for NFS and Coda, they are the links between clients and server. NFS took 1939 seconds in a 5Mb/s network, and it did not finish after two hours in a 1Mb/s network.*

7.4 Performance of personal workload in WANs

We ran the Andrew-Tcl benchmark to study the performance of the systems in WANs for a personal workload. Since this workload involves no data sharing, the elapsed time depends (if at all) only on the latency and capacity of the link between the client and the server. Figure 8 shows the time needed to complete the benchmark. Pangaea and

[6] This benchmark is available from `http://www.hpl.hp.com/personal/ysaito`.

	pang-1	pang-2	pang-3	pang-4	NFS	Coda-s	Coda-w	ext3
mkdir	2.04	2.04	2.18	2.28	0.316	2.25	0.047	0.021
copy	3.40	3.79	3.85	3.90	3.50	201.0	0.85	0.264
stat	0.91	0.90	0.90	0.91	0.87	0.86	0.86	0.162
grep	2.09	2.11	2.13	2.13	2.20	1.22	1.20	0.925
compile	74.4	75.3	75.8	75.9	77.2	90.2	62.1	61.5
Total	82.84	84.14	84.86	85.12	84.08	295.5	65.05	62.87

Table 2: *Andrew-Tcl benchmark results in a LAN environment. Numbers are in seconds. Label pang-N shows Pangaea's performance when it creates N replicas for each new file. Ext3 is Linux's native (local) file system.*

	pang-1	pang-2		pang-3		pang-4		NFS	coda-w	coda-s
	Bytes	Bytes	Overhead	Bytes	Overhead	Bytes	Overhead	Bytes	Bytes	Bytes
create	0	248	0%	1.29K	60%	2.61K	68%	503	1.46K	1.96K
write 1B	0	323	0%	854	61%	2.01K	68%	667	944	935
write 50KB	0	52.04K	0%	104.98K	1.49%	157.44K	1.52%	53.21K	55.56K	82.13K
write 25MB	0	26.22M	0%	52.44M	0.01%	78.67M	0.02%	26.76M	1.56M	38.75M

Table 3: *Network bandwidth consumption for common file-system operations. Shows the total number of bytes transmitted between all the nodes for each operation. "Overhead" shows the percentage of the bandwidth used by harbingers and duplicate updates.*

Coda totally hide the network latency, because the benchmark is designed so that it reads all the source data from the local disk, and the two systems can propagate updates to other nodes in the background. On the other hand, the performance of NFS degrades severely across slow links.

7.5 Roaming

Roaming, i.e., a single user moving between different nodes, is an important use of distributed file systems. We expect Pangaea to perform well in non-uniform networks in which nodes are connected with networks of different speeds. We simulated roaming using three nodes: *S*, which stores the files initially and is the server in the case of Coda, and two type-A nodes, C_1 and C_2. We first run the Andrew-Tcl benchmark to completion on node C_1, delete the `*.o` files, and then re-run only the compilation stage of the benchmark on node C_2. We vary two parameters: the link speed between C_1 and C_2, and the link speed between them and *S*. As seen from Figure 8, the performance depends, if at all, only on these two parameters.

Figure 9 shows the results. It shows that when the network is uniform, i.e., when the nodes are placed either all close by or all far apart, Pangaea and Coda perform comparably. However, in non-uniform networks, Pangaea achieves better performance than Coda by transferring data between nearby nodes. In contrast, Coda clients always fetch data from the server. (Pangaea actually performs slightly better in uniformly slow networks. We surmise that the reason is that Pangaea uses TCP for data transfer, whereas Coda uses its own UDP-based protocol.)

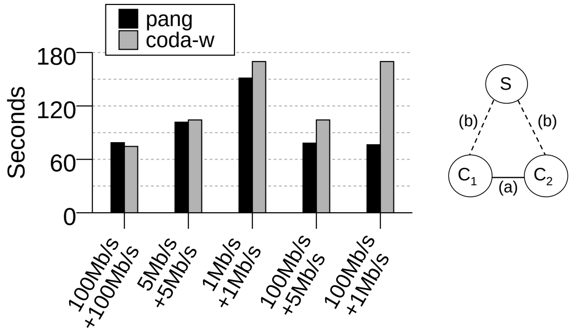

Figure 9: *The result of recompiling the Tcl source code. 100Mb/s + 1Mb/s, for example, means that the link between the two client nodes (link (a) in the right-side picture) is 100Mb/s, and the link between the benchmark client and the server (link (b)) is 1Mb/s. The speed of other links is irrelevant in this experiment.*

7.6 Data sharing in non-uniform environments

The workload characteristics of wide-area collaboration systems are not well known. We thus created a synthetic benchmark modeled after a bulletin-board system. In this benchmark, articles (files) are continuously posted or updated from nodes chosen uniformly at random; other randomly chosen nodes (i.e., users) fetch new articles not yet read. A file system's performance is measured by two metrics: the mean latency of reading a file never accessed before by the server, and the wide-area network bandwidth consumption for files that are updated. These two numbers depend, if at all, only on the file size, the number of existing replicas (since Pangaea can perform short-cut creation), and the order in which these replicas are created (since it affects the shape of the graph). We choose an article size

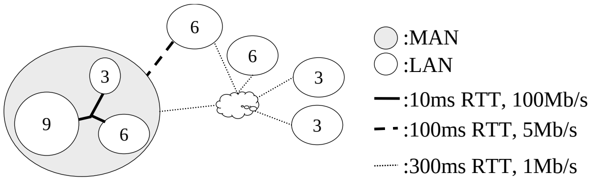

Figure 10: *Simulated network configurations modeled after our corporate network. The gray circle represents the SF bay area metropolitan-area network (MAN), the upper bubble represents Bristol (UK), and the other bubbles represent India, Israel, and Japan. The number in a circle shows the number of servers running in the LAN.*

of 50KB, a size typical in Usenet [29]. We try to average out the final parameter by creating and reading about 1000 random files for each sample point and computing the mean. We run both article posters and readers at a constant speed (≈5 articles posted or read/second), because our performance metrics are independent of request inter-arrival time.

In this benchmark, we run multiple servers in a single (physical) node to build a configuration with a realistic size. To avoid overloading the CPU or the disk, we choose to run six virtual servers on a type-B machine (Table 1), and three virtual servers on each of other machines, with the total of 36 servers on 9 physical nodes. Figure 10 shows the simulated geographical distribution of nodes, modeled after HP's corporate network. For the same logistical reasons, instead of Coda, we compare three versions of Pangaea:

- **pang:** Pangaea with three gold replicas per new file.
- **hub:** This configuration centralizes replica management by creating, for each file, one gold replica on a server chosen from available servers uniformly at random. Bronze replicas connect only to the gold replica. Updates can still be issued at any replica, but they are all routed through the gold replica. This roughly corresponds to Coda.
- **random:** This configuration creates a graph by using simple random walks without considering either gold replicas or network proximity. It is chosen to test the effect of Pangaea's graph-construction policy.

We expect Pangaea's access latency to be reduced as more replicas are added, since that increases the chance of file contents being transferred to a new replica from a nearby existing replica. Figure 11 confirms this prediction. In contrast, the hub configuration shows no speedup no matter how many replicas of a file exist, because it always fetches data from the central replica.

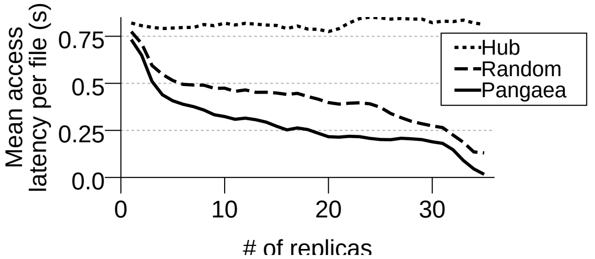

Figure 11: *The average time needed to read a new file in a collaborative environment. The X axis shows the number of existing replicas of a file. The Y axis shows the mean latency to access a file on a node that does not yet store a replica of the file.*

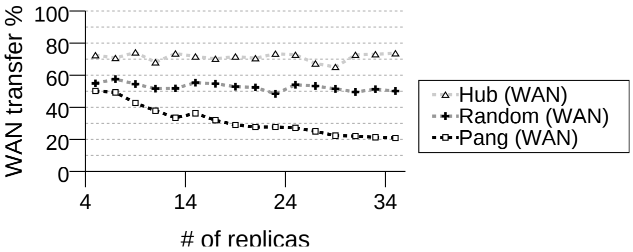

Figure 12: *Wide-area network bandwidth usage during file updates. The Y axis shows the percentage of traffic routed through the indicated networks. "WAN+MAN" shows the traffic that flowed through non-LAN (i.e., those with ≥10ms RTT), whereas "WAN" shows the traffic that flowed through networks with ≥180ms RTT (see also Figure 10).*

Figure 12 shows the network bandwidth consumption during file updates. Although all the systems consume the same total amount of traffic per update (i.e., $(\text{\#-of-replicas} - 1) * \text{filesize}$), Pangaea uses far less wide-area network traffic since it transfers data preferentially along fast links using dynamic spanning-tree construction (Section 5.1.3). This trend becomes accentuated as more replicas are created.

Figure 13 shows the time the pang configuration took to propagate updates to replicas of files during the same experiment. The "max" lines show large fluctuations, because updates must travel over 300ms RTT links multiple times using TCP. Both numbers are independent of the number of replicas, because (given a specific network configuration) the propagation delay depends only on the graph diameter, which is three, in this configuration. We believe that 4 seconds average/15 seconds maximum delay for propagating 50KB of contents over 300ms, 1Mb/s links is reasonable. In fact, most of the time is spent in waiting when constructing a spanning tree (Section 5.1.3); cutting the delay parameter would shrink the propagation latency, but potentially would worsen the network bandwidth usage.

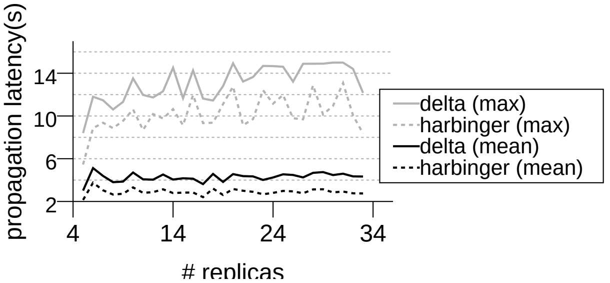

Figure 13: *The time needed to propagate updates to all replicas. The dashed lines show the time needed to distribute harbingers to replicas. They represent the window of inconsistency; i.e., time before which users may observe old contents. The solid lines represent the time needed to distribute actual updates. They represent the number of seconds users wait before seeing the new contents. The "mean" lines show the mean time needed for an update issued at one replica to arrive at all replicas, for a file with a specific number of replicas. The "max" lines show the maximum time observed for an update to arrive at all replicas of the file.*

7.7 Performance and network economy at a large scale

The previous section demonstrated Pangaea's ability to fetch data from a nearby source and distribute updates through fast links, yet only at a small scale. This section investigates whether these benefits still hold at a truly large scale, by using a discrete event simulator that runs Pangaea's graph-maintenance and update-distribution algorithms. We extracted performance parameters from the real testbed we used in the previous section, and ran essentially the same workload as before. We test two network configurations. The first configuration, called HP, is the same as Figure 10, but the number of nodes in each LAN is increased eighty-fold, to a total of 3000 nodes. The second configuration, called U, keeps the size of each LAN at six nodes, but it increases the number of regions to 500 and connects regions using 200ms RTT, 5Mb/s links.

Figures 14 and 15 show average file-read latency and network bandwidth usage in these configurations. These figures show the same trend as before, but the differences between the configurations are more pronounced. In particular, in the HP configuration, Pangaea propagates updates almost entirely using local-area network for popular files, since it crosses over wide-area links only a fixed number of times, regardless of the number of replicas. In the U configuration, Pangaea still saves bandwidth, more visibly when many replicas exist. The systems cannot improve read latency much in U, because most of the accesses are forced to go over wide area links, but Pangaea still shows improvement with many replicas.

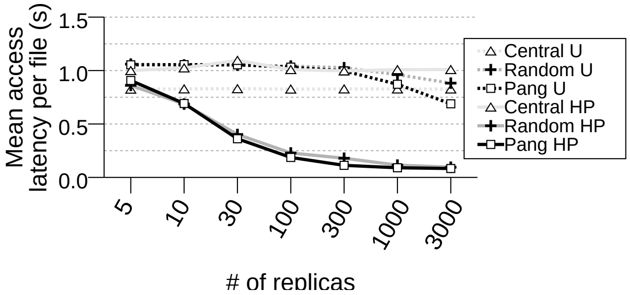

Figure 14: *File-reading latency in a simulated 3000-node system. The meaning of the numbers is the same as in Figure 11.*

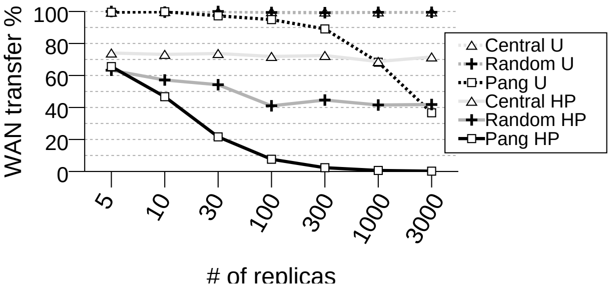

Figure 15: *Wide-area network bandwidth usage during file updates in simulated 3000-node systems. The meaning of the numbers is the same as in Figure 12.*

7.8 Availability analysis

This section studies the effects of pervasive replication, especially name-space containment, on the system's availability. A Pangaea server replicates not just replicas accessed directly by the users, but also all the intermediate directories needed to look up those replicas. Thus, we expect Pangaea to disrupt users less than traditional approaches that replicate files (or directories) on a fixed number of nodes.

We perform trace-based analysis to verify this prediction. Two types of configurations are compared: Pangaea with one to three gold replicas per file, and a system that replicates the entire file system contents on one to four nodes. Our trace was collected on our departmental file server, and it contains 24 users and 116M total accesses to 566K files [31]. To simulate a wide-area workload from this single-node trace, we assume that each user is on a different node; thus, all the simulated configurations contain 24 nodes.

For each configuration, we start from an empty file system and feed the first half of the trace to warm the system up. We then artificially introduce remote node crashes or wide-area link failures. To simulate the former situation,

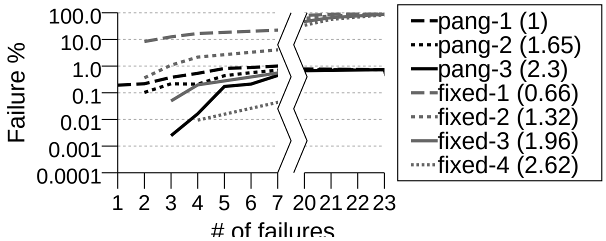

Figure 16: *Availability analysis using a file-system trace; the users of a failed node move to a functioning node. The numbers in parentheses show the overall storage consumption, normalized to pang-1.*

we crash 1 to 7 random nodes and redirect accesses by the user on a failed node to another random node. To simulate link failures, in which one to four nodes are isolated from the rest, we crash 20 to 23 random nodes and throw away future activities by the users on the crashed nodes. We then run the second half of the trace and observe how many of the users' sessions[7] can still complete successfully. We run simulation 2000 times for each configuration with different random seeds and average the results.

Figure 16 shows the results. For network partitioning, Pangaea wins by a huge margin; it shows near-100% availability thanks to pervasive replication, whereas the other configurations must rely on remote servers for much of the file operations. For node failures, the differences are smaller. However, we can still observe that for the same storage overhead, Pangaea offers better availability.

8 Conclusions

Pangaea is a wide-area file system that targets the needs for data access and sharing of distributed communities of users. It federates commodity computers provided by users. Pangaea is built on three design principles: 1) pervasive replication to provide low-access latency and high availability, 2) randomized graph-based replica management that adapts to changes in the system and conserves WAN bandwidth, and 3) optimistic consistency that allows users to access data at any time, from anywhere.

The evaluation of Pangaea shows that Pangaea is as fast and as efficient as other distributed file systems, even in a LAN. The benefits of pervasive replication and the adaptive graph-based protocols become clear in heterogeneous environments that are typical of the Internet and large intranets. In these environments, Pangaea outperforms existing systems in three aspects: access latency, efficient usage of WAN bandwidth, and file availability.

Acknowledgements

We thank our shepherd Peter Druschel, the anonymous reviewers, and the members of our group in HP Labs—especially, Eric Anderson, Mahesh Kallahalla, Kim Keeton, Susan Spence, Ram Swaminathan, and John Wilkes—for offering invaluable feedback that improved the quality of this work.

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] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless Network File Systems. In *15th Symp. on Op. Sys. Principles (SOSP)*, pages 109–126, Copper Mountain, CO, USA, December 1995.

[3] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Conf. on Measurement and Modeling of Comp. Sys. (SIGMETRICS)*, pages 34–43, Santa Clara, CA, USA, June 2000.

[4] B. Callaghan, B. Pawlowski, and P. Staubach. RFC1813: NFS version 3 protocol specification. http://www.faqs.org-/rfcs/rfc1813.html, June 1995.

[5] Pei Cao and Sandy Irani. Cost-Aware WWW proxy caching algorithms. In *1st USENIX Symp. on Internet Tech. and Sys. (USITS)*, Monterey, CA, USA, December 1997.

[6] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *18th Symp. on Op. Sys. Principles (SOSP)*, pages 202–215, Lake Louise, AB, Canada, October 2001.

[7] 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.

[8] 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.

[9] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global Internet host distance estimation service. *IEEE/ACM Trans. on Networking (TON)*, 9(5):525–540, October 2001.

[7] We define a session to be either a directory operation (i.e., unlink), or a series of system calls to a file between and including open and close. If any one of the system calls fails, we consider the session to fail.

[10] Richard A. Golding, Darrell D. E. Long, and John Wilkes. The refdbms distributed bibliographic database system. In *USENIX Winter Tech. Conf.*, San Francisco, CA, USA, January 1994.

[11] Jim Gray. A census of Tandem system availability between 1985 and 1990. *IEEE Trans. on Reliability*, 39(4):409–418, October 1990.

[12] Håkan Grahn and Per Stenström and Michel Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 11(3), June 1995.

[13] John Howard, Michael Kazar, Sherri Menees, David Nichols, M. Satyanarayanan, Robert Sidebotham, and Micahel West. Scale and performance in a distributed file system. *ACM Trans. on Comp. Sys. (TOCS)*, 6(1), 1988.

[14] 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.

[15] Leonard Kawell Jr., Steven Beckhart, Timoty Halvorsen, Raymond Ozzie, and Irene Greif. Replicated document management in a group communication system. In *Conf. on Comp.-Supported Coop. Work (CSCW)*, Chapel Hill, NC, USA, October 1988.

[16] Minkyong Kim, Landon P. Cox, and Brian D. Noble. Safety, visibility, and performance in a wide-area file system. In *USENIX Conf. on File and Storage Sys. (FAST)*, Monterey, CA, January 2002. Usenix.

[17] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *9th Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS-IX)*, pages 190–201, Cambridge, MA, USA, November 2000.

[18] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *USENIX Winter Tech. Conf.*, pages 95–106, New Orleans, LA, USA, January 1995.

[19] David Mazières. A toolkit for user-level file systems. In *USENIX Annual Tech. Conf.*, Boston, MA, USA, June 2001.

[20] Lily B. Mummert, Maria R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *15th Symp. on Op. Sys. Principles (SOSP)*, pages 143–155, Copper Mountain, CO, USA, December 1995.

[21] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *18th Symp. on Op. Sys. Principles (SOSP)*, pages 174–187, Lake Louise, AB, Canada, October 2001.

[22] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *5th Symp. on Op. Sys. Design and Impl. (OSDI)*, Boston, MA, USA, December 2002.

[23] 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.

[24] Konstantinos Psounis and Balaji Prabhakar. A randomized web-cache replacement scheme. In *Infocom*, Anchorage, AL, USA, April 2001.

[25] 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.

[26] Luigi Rizzo. Dummynet, `http://info.iet.unipi.it/˜luigi/ip_dummynet/`, 2001.

[27] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th Symp. on Op. Sys. Principles (SOSP)*, pages 188–201, Lake Louise, AB, Canada, October 2001.

[28] Yasushi Saito and Christos Karamanolis. Replica consistency management in the pangaea wide-area file system. Technical report, HP Labs, 2002. To be published.

[29] Yasushi Saito, Jeffrey Mogul, and Ben Verghese. A Usenet performance study, September 1998. `http://www.research.digital.com/wrl/projects/newsbench/`.

[30] Sleepycat Software. The Berkeley database, 2002. `http://sleepycat.com`.

[31] Susan Spence, Erik Riedel, and Magnus Karlsson. Adaptive consistency—patterns of sharing in a networked world. Technical Report HPL-SSP-2002-10, HP Labs, February 2002.

[32] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th Symp. on Op. Sys. Principles (SOSP)*, pages 172–183, Copper Mountain, CO, USA, December 1995.

[33] Chandramohan Thekkath, Timothy Mann, and Edward Lee. Frangipani: a scalable distributed file system. In *16th Symp. on Op. Sys. Principles (SOSP)*, pages 224–237, St. Malo, France, October 1997.

[34] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *IFIP Int. Conf. on Dist. Sys. Platforms and Open Dist. (Middleware)*, 1998. `http://www.cs.cornell.edu/Info/People/rvr/papers/pfd/pfd.ps`.

[35] Werner Vogels. File system usage in Windows NT 4.0. In *17th Symp. on Op. Sys. Principles (SOSP)*, pages 93–109, Kiawah Island, SC, USA, December 1999.

[36] 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.

[37] 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.