Autonomous and Decentralized Replication in the Pangaea Planetary-Scale File Service

Yasushi Saito, Christos Karamanolis
Internet and Systems Storage Laboratory
HP Laboratories Palo Alto
HPL-2001-323
December 10th , 2001*

E-mail: {ysaito, christos} @hpl.hp.com

Pangaea,
replication,
file system,
consistency,
membership

Pangaea is a planetary-scale file service that supports sharing of frequently written files as well as read-only files. Pangaea uses massive replication to achieve low access latency and high availability. This paper focuses on a key feature of Pangaea—randomized, optimistic protocols for managing many replicas efficiently for billions of files replicated on hundreds of servers. Replica membership is maintained as a sparse but strongly connected graph, per file. Updates to the membership and contents are propagated along the graph edges. The feasibility of Pangaea is evaluated using a prototype and a simulation.

* Internal Accession Date Only
Approved for External Publication
© Copyright Hewlett-Packard Company 2001

Autonomous and decentralized replication in the Pangaea planetary-scale file service

Yasushi Saito and Christos Karamanolis
HP Labs, Storage and Content Distribution Dept.
{ysaito,christos}@hpl.hp.com

Abstract

Pangaea is a planetary-scale file service that supports sharing of frequently written files as well as read-only files. Pangaea uses *massive replication* to achieve low access latency and high availability. This paper focuses on a key feature of Pangaea—randomized, optimistic protocols for managing many replicas efficiently for billions of files replicated on hundreds of servers. Replica membership is maintained as a sparse but strongly connected graph, per file. Updates to the membership and contents are propagated along the graph edges. The feasibility of Pangaea is evaluated using a prototype and a simulation.

1 Introduction

Global, seamless data exchange is the holy grail of distributed systems research. Partially, this is already happening in the forms of WWW and P2P systems. The Pangaea project aims to build an autonomous and decentralized file service that can be used not only to distribute read-only data, but also to serve people's daily storage needs—e.g., for document sharing, data crunching, and archiving. Its primary targets are large multinational corporations or groups spread over the world that want to share data transparently over a file system. Thus, Pangaea must hide the effects of wide-area networking and data distribution, to the extent possible. Our goals follow:

Performance: Pangaea should look and feel like a local file service by hiding wide-area networking latencies.

Availability: Any large distributed system experiences frequent changes. Pangaea should *heal* automatically after server failure or replacement without disturbing users.

Autonomy: With servers spread over the globe, they will belong to different, although cooperating, administrative domains. Thus, each server should be able to set its own resource management policy, e.g., the amount of disk space to offer and when to reclaim it.

Pangaea achieves these goals by *massive, decentralized, and optimistic replication.* It replicates popular files massively on all servers that desire access to the file. There is no single server that permanently manages data or protocol. Finally, it requires no synchronous coordination among servers to modify file contents or to add or remove replicas; all changes to the system are propagated in the background epidemically. This paper focuses on a key challenge massive replication faces: the management of the replica membership and consistency. We describe randomized algorithms for low-cost, fault-tolerant replica membership management and optimistic strategies for data consistency.

1.1 Overview of Pangaea

Pangaea builds a unified file system name space over hundreds of servers spread over the world. We assume that servers are trusted; relaxing the trust relationship is future work (Section 5). It is a peer-to-peer system in a broad sense, in that its structure is totally decentralized and symmetric, and a user can theoretically use any server to access any file. To ensure performance, we assume that each user usually mounts (or logs into) a specific server for a period of time, e.g., a day, although she can roam over different servers over time. We call such server her "local server" for the period.

In Pangaea, every file[1] a user accesses is replicated at her local server. Thus, files shared by many people (e.g., the root directory) are replicated on virtually every server. On the other hand, the user's personal files are created on her local server and a few remote servers for availability, and they stay that way. An update to a file can be issued at any replica at any time, and it is "flooded" to all others in the background. This *massive replication* hides network latency and contains the user's working set to let her work even when the local server is disconnected. Its totally decentralized nature also allows for better site autonomy by letting any replica (or server) be removed or replaced at any time transparently to the user.

[1] Pangaea treats a directory as a file with special contents (Section 2.1). Thus, we abuse the term "file" to refer to both regular files and a directories when there is no fear of confusion.

1.2 Challenges in massive replication

Massive replication certainly has several potential downsides. The first is the computational and storage overhead of metadata management. While this problem is genuine, servers in cooperative-work environments, which we target initially, are known to waste much of their resources idle [1]. The second is the overhead of propagating updates to files with many replicas. We believe this to be a non-issue, because popular files are updated rather infrequently. The third problem is the algorithmic challenge of managing a large number of replicas in a decentralized setting. Section 2 presents simple randomized algorithms for maintaining the replica membership of a file. The final issue is the lack of a strong consistency guarantee because of our decentralized and optimistic approach. We discuss three symptoms of this problem—the freshness of contents, update conflicts, and lack of atomic multi-object updates—and our solutions in Section 3. Section 5 presents other issues we plan to investigate in the future.

1.3 Related work

Wide-area data sharing is not a new idea. Many systems have been designed with differing focuses in mind. None of them, however, solved all our goals.

Massive replication is similar to persistent caching (e.g., AFS and Coda [4]) in terms of performance. Pangaea, however, is more available because of its decentralized and optimistic nature. For example, it provides read-any, update-any replication, and allows any replica be removed from the system and yet ensures the consistency of other replicas. LBFS [5] saves network bandwidth by exchanging fingerprints of file fragments instead of file contents. Its idea complements ours; while LBFS reduces bandwidth consumption, we reduce access latency. Locus [9] provided a file system with optimistic, fine-grain replication. Pangaea extends Locus by supporting frequent replica addition and removal.

Mobile data sharing services, such as Ficus, Roam [7], RefDBMS [2], and Bayou [6], allow mobile users to replicate data massively and exchange updates epidemically among them. They, however, lack a replica location service, and humans are responsible for synchronizing devices to keep replicas consistent. In contrast, Pangaea keeps track of the location of replicas and distribute updates proactively and transparently to the users.

Recent peer-to-peer systems (e.g., CFS, CAN, Pastry, and Oceanstore) provide flat distributed hash tables using randomization techniques. While Pangaea shares many of the goals with them—e.g., decentralization, availability and site autonomy—it must solve different problems because of the difference in application requirements. In particular, Pangaea explicitly stores a graph of replicas and supports distributed protocols for its maintenance, because replica locations are chosen by user activity, not by randomization. Pangaea also supports hierarchical name spaces, ensuring that links between files are kept consistent. Pangaea does use hash tables for directory entries, but it simply replicates the entire table on each directory replica.

2 Managing replica membership

Pangaea experiences very frequent replica additions and removals, because it manages billions of files, each of which is replicated independently on many servers. Thus, it calls for inexpensive and scalable mechanisms for efficiently managing the replica membership for each file and reliably distributing updates among replicas.

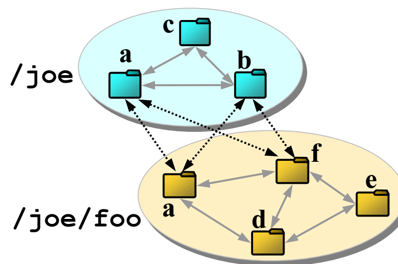

Figure 1: *A replica graph. Each file (an oval) is replicated on its own set of replicas. Alphabet labels show the names of the servers that store replicas. An edge between replicas of a file shows that the replicas know of each other. Updates to the file are propagated through these edges. Edges are also formed between some of the file's replicas and those of its parent directory. These links enable path traversal.*

Pangaea decentralizes both replica membership management and update distribution to achieve this goal. There is no entity that centrally manages replicas for a file. Instead, Pangaea lets each replica maintain links to a few other random replicas of the same file. Links to some of these replicas are also recorded in the entry of the parent directory, and they act as starting points for accessing the file. Figure 1 shows an example. As discussed in the following sections, Pangaea ensures, with very high probability, that the replica "graph" of a file exhibits certain desirable properties. In particular, it keeps the graph's diameter small and maintains the graph k-connected (i.e., at least k disjoint paths between any pair of replicas), where k is a configuration parameter usually chosen to be three or four. This design addresses our goals as follows:

Highly available and scalable update distribution:
Pangaea lets any replica issue a new update and *floods* the update along the graph edges. Thus, the system can tolerate at least k simultaneous replica failures. With

timely graph repairing (Section 2.2), it can essentially tolerate arbitrary number of failures over time. The graph structure also helps distributing the update propagation overhead among servers.

Highly available and scalable membership changes: This mechanism can add a replica when it can find any k live replicas, no matter how many other replicas are unavailable. Moreover, adding or removing a replica only involves adding or removing a small number of edges between replicas, regardless of the total number of replicas.

High available and scalable replica location: Each directory entry records multiple replica locations for the file, and the directory itself is replicated just like a regular file. Thus, this mechanism allows for path name traversal even when some servers are unavailable.

The following sections introduce random-walk-based protocols for dynamically reconstructing a graph in response to replica addition or removal.

2.1 Adding a replica

When a file is first created, the user's local server selects a few random servers in addition to itself (a gossip-based membership service is consulted during this phase [8]), creates replicas on these servers, spans graph edges among them, and registers the locations of the replicas in the parent directory's entry.

A replica is added when a user tries to look up a file missing from her local server. The goal is to create a new replica on the server and span at least k distinct edges to existing replicas. As there is no central replica-management authority, nodes are chosen using *random walks*, starting from a replica found in the parent directory's entry and performing a series of remote procedure calls (RPCs) along the graph edges. We try to pick replicas with fewer outgoing edges, based on the presumption that (1) adding an edge to such replicas will likely improve the total graph connectivity, and (2) balancing the number of edges of replicas will help distribute the storage and computational overhead evenly among servers. Section 4 shows that this heuristic keeps the graph well and evenly connected, for modest walking distances. After picking the k "best" nodes, the server issues a multi-party RPC request to the k replicas to span edges between them and the new replica. The replica contents are retrieved from one of these replicas as a side effect of the RPC. This protocol ensures that the resulting graph is k-connected (assuming that it was k-connected before).

Directories in Pangaea are files with special contents. They are replicated using the same random-walk-based protocol. Thus, to look up a file, the parent directory must also be replicated on the server where the file is accessed from. This recursive step potentially continues all the way up to the root directory. Replicas of the root directory are discovered through an external service, e.g., DNS or LDAP.

2.2 Removing a replica

A replica is removed either involuntarily or voluntarily. It is removed involuntarily when its host server remains down for a long period or when its disk crashes. When a replica detects the death of a neighbor in the graph, it autonomously initiates a random walk exactly as in the case of replica addition and re-spans an edge with another live replica. This protocol is probabilistic, in that it may disconnect the graph and create orphan replicas in the worst case. The randomization incorporated into the protocol, however, makes it extremely unlikely. We confirm this claim in Section 4.

Pangaea also supports voluntary replica removal, which happens when a server runs out of disk space or decides that the replica is not worth keeping. The protocol is the same as above, except that the retiring replica proactively sends notices to all its graph neighbors so that they can start adding edges immediately and minimize the window of vulnerability of graph disconnection.

2.3 Maintaining minimum replication factor

A downside of the decentralized graph approach is the difficulty of enumerating all replicas for a single file. In particular, it is not trivial to ensure a minimum number of replicas for a file. We solve this problem by marking some replicas "golden". Servers storing golden replicas must give them a high priority to stay on disk. We currently mark replicas golden when a file is first created and keep these initial replicas on the disk as long as possible. Graph edges to golden replicas are also marked golden. When a golden replica is suspected dead by a graph neighbor, the neighbor initiates a whole-graph sweep, similar to contents-update flooding, to locate live golden replicas and "engolden" another replica if needed.

3 Dealing with optimism

Wide-area data sharing demands read-any / update-any replication to ensure availability and performance. Thus, Pangaea manages both the membership and contents of replicas *optimistically* without any synchronous coordination. Optimism causes three potential problems: the lack of consistency guarantees for file contents, update conflicts, and the inability to update multiple files atomically, which in particular affects directory operations like `mkdir` and `rename`. The following sections discuss Pangaea's approach for handling these problems.

3.1 Consistency of replica contents

The flooding algorithm described in Section 2 currently provides no hard guarantees about the degree of replica divergence; imposing stronger guarantees, such as an open-close consistency (cf. Sprite and AFS) or `flock(2)`, is a subject for future investigations. We argue below, however, that our "best-effort" approach can manage the window of inconsistency within about a second in most cases, and that it is practical enough for most applications. First, a new update is pushed proactively through the graph edges, which minimizes the propagation delay. Second, since the random-walk protocol keeps the graph's diameter fairly small (e.g., 4 or 5 for graphs with hundreds of replicas; see Section 4), the flooding terminates in a small number of network hops.

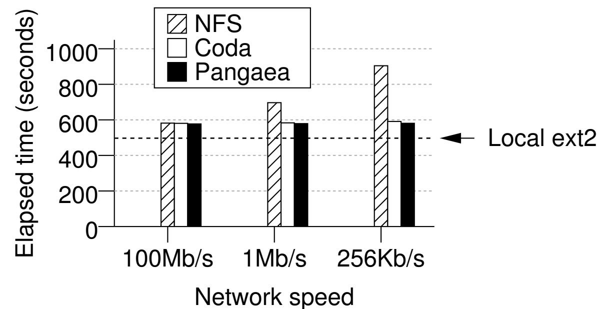

Figure 2: *Performance of Pangaea under various network conditions. "Local ext2" shows the elapsed time on a local ext2 file system.*

3.2 Conflict resolution

Conflicting updates to the replica membership are detected using version vectors and resolved by merging. Suppose that one membership update voluntarily removes replica a, and another adds replica b and spans edges between b and replicas $\{a, c\}$. Then, replica a or b (or both, in which case another, cascaded round of resolution happens afterward) will perform its own random walk, pick a replica, say d, and issue the following update as the merge result: remove a, add b, and span edges between b and $\{c, d\}$. The merge result is pushed to the participants of the three updates, i.e., to $\{a, b, c, d\}$ and the neighbors of a. Because the third update causally follows the original two (i.e., its version vector dominates the others'), its recipients will override the original updates. This membership resolution algorithm provenly ensures the consistency of the resulting graph under any combination of concurrent updates: a graph edge is spanned only between two live replicas, and no "orphan" replicas, disconnected from others, are created because of merging.

Concurrent updates to file contents are also detected using version vectors and merged semantically. We currently use a protocol similar to that of Locus [9].

3.3 Maintaining the consistency of the hierarchical name space

The previous sections described Pangaea's decentralized approach for maintaining links among replicas of a single file. In fact, because Pangaea offers a hierarchical name space, it must also manage links between a file and its parent directory to enable path traversal. Pangaea uses the same graph-based techniques to maintain such inter-file links by treating parent directory replicas as a part of the file's replica membership. Figure 1 shows an example. Directory "/joe/" is replicated on servers a, b and c, while file "/joe/foo" is replicated on servers a, d, f and e. The resulting replica graph for "/joe/foo" includes all seven nodes. The directory entries run the full random-walk protocol to maintain links to the file consistently, but they do not participate in contents-update flooding. This design allows for directory links to be renewed properly after replica removal. Directory replicas that participate in the file's replica graph should not be removed, because doing so would result in cascading membership changes in the graphs of all the directory's children. Thus, we mark these replicas "golden" (Section 2.3).

4 Evaluation

This section evaluates the feasibility of Pangaea. We study the performance of our early prototype and demonstrate that massive replication can effectively hide slow networking conditions. We also use simulation to show that Pangaea's randomized graph protocols maintain good graph properties.

Our prototype is implemented as a user-space loopback NFS server [3] on Linux. At this early stage, it is premature and lacks some of the proposed protocols. In particular, it substitutes random walks with spanning edges to whatever replicas found in the directory entry. We compare Pangaea to NFS and Coda [4] by compiling the Linux source code. We set up two machines, both equipped with 750MHz Pentium 3, 256 MB of memory, and 8GB SCSI disks. One machine runs either an NFS server, a Coda server, or a Pangaea server, and it stores the source files. Another machine, running either a NFS client, a Coda client, or a Pangaea server, reads the files from the first machine and compiles. For Coda, the files are cached persistently on the second node; for Pangaea, the files are replicated on the second node. We simulated slow networking conditions by restricting the network bandwidth between the two servers. Figure 2 shows the elapsed compilation time for the three file systems. Pangaea and Coda show stable performance under any network condition, although Pangaea's decentralized architecture will offer better availability and autonomy at a large scale. In contrast, NFS works well when the network is uncon-

	N	W=2		W=3		W=4	
		D=3	D=4	D=3	D=4	D=3	D=4
Average diameter	**100**	4	4	4	4	4	4
	400	5	5	5	4	5	4
Average connectivity	**100**	-	-	3	4	3	4
	400	-	-	3	4	3	4

Table 1: *The behavior of the randomized graph maintenance algorithm. N is the number of nodes in the graph, D is the number of edges spanned for a new replica, and W is the random-walk hopping distance. The entries marked "-" show that the graph disconnected before the simulation was completed. The graphs disconnected after 10,000 to 300,000 iterations, corresponding to a period of a week to half a year with one change/minute, or one to 35 years with one change/hour.*

strained, but its performance drops rapidly over slow networks, because of its large network traffic generated due to its stateless caching architecture.

The simulation studies the behavior of our graph maintenance protocols for a single file. We created a random graph for the file and stressed it through a series of random node (replica) addition and removal using our graph-maintenance protocols. We subjected the graph to up to two million additions/removals, which corresponds to a period of four years when the membership change for a particular file happens every minute, or 230 years when the change happens every hour. We changed three parameters: the number of nodes in the graph (N), the number of edges spanned for each new node (D), and the number of edge "hops" taken by each random walk (W). Table 1 shows the average properties of the graph. It demonstrates that the graph's diameter is fairly small for all configurations, and it grows only slowly at $O(\log N)$. The results also show an interesting trade-off between random-walk distance and connectivity maintenance. Picking too small a random-walk distance, while cheap, may disconnect the graph. The reason, intuitively, is that it can only select nodes in the neighborhood and skews the distribution of edges. The random-walk distance of at least the graph's half-diameter allows selecting nodes truly randomly and can maintain the connectivity for an arbitrarily long period.

5 Future work

Our research on Pangaea opens many questions that demand further investigation. Currently, Pangaea replicates a file on every server that accesses it. We plan to investigate more intelligent replica placement heuristics utilizing information such as storage capacity, inter-node network bandwidth, content type, and access demand. On a related issue, massive replication currently incurs high startup overhead; a roaming user cannot fully enjoy its benefits until her working set is replicated on the local server. We plan to investigate integrating mobile storage devices into our system.

We assume that all Pangaea servers are trustworthy. For now, this is a reasonable assumption, since we assume servers belong to a single administrative domain (e.g., a multi-national corporation). We plan to relax the level of trust for two types of information managed by the system: file contents and the graph structure. For contents, we plan to use end-to-end data encryption. The challenge is the treatment of meta-data; e.g., the directory entries should be encrypted to avoid information leak, but doing that would obstruct graph-shape maintenance. For the graph structure, we plan to investigate the prevention of Byzantine servers from destroying the graph and prohibit update propagation. For instance, we want to at least let a user keep all her files in the local server even when remote servers are Byzantine.

References

[1] 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 *ACM SIGMETRICS*, pages 34–43, Santa Clara, CA, USA, June 2000.

[2] Richard A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California Santa Cruz, December 1992.

[3] David Mazières. A toolkit for user-level file systems. In *USENIX Annual Technical Conference*, Boston, MA, USA, June 2001.

[4] 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.

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

[6] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *16th Symp. on Op. Sys. Principles (SOSP)*, pages 288–301, St. Malo, France, October 1997.

[7] David H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, UC Los Angeles, 1998.

[8] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *IFIP Int. Conf. on Dist. Sys. Platforms and Open Dist. Proc. (Middleware)*, 1998.

[9] 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.