A Survey of Secure, Fault-tolerant Distributed File Systems

Piyush Agarwal
Harry C. Li
UT Austin 2003

Abstract: We survey four secure fault-tolerance distributed file systems: Farsite, OceanStore, Ivy, and Frangipani. We analyze each with respect to fault-tolerance, scalability, usability, maintenance overhead, and consistency. Finally, we present a taxonomy for such file systems based upon their failure models, update mechanisms, and data location schemes.

1. Introduction:

One of the fundamental abstractions that a computer user expects is a file system. A file system provides an interface to the user where he can read, write, copy, and remove files. The most common file systems are the ones provided locally on a machine. The disadvantage of this approach is that if the machine fails, then all users of that file system will have lost all their data. The common solution to this problem is for a system administrator to make backups of the system in case of failure. This technique, however, is slow, cumbersome, and does not scale once hundreds of individual computers are involved.

An attractive solution to guarding against failures is a distributed file system, one in which the designers have spread the actual implementation of the file system across multiple computers connected through a network. Simply distributing a file system is often not enough to yield desirable properties that outweigh the work involved in building it. Designers have to make careful choices to yield guarantees of fault-tolerance, usability, scalability, and consistency.

This work surveys secure, fault-tolerant, distributed file systems. In particular, we aim to compare Farsite [1], OceanStore [6], Ivy [11], and Frangipani [16]. We will discuss each system with respect to our metrics of fault-tolerance, usability, scalability, and consistency. The closest work to ours is a survey by Satyanarayanan [17]. However, he conducted his survey several years ago, at a time when fault-tolerance and scalability were not the mainstream concerns. Guan et al. [4] discuss more recent file systems, but neither present structured metrics for comparison nor suggest a taxonomy. Finally, we will introduce a taxonomy that highlights critical design choices for any secure, fault-tolerant, distributed file system.

We organize the rest of the paper as follows. Section 2 presents our approach to comparing distributed file systems, detailing our four metrics. Section 3 discusses each file system with respect to these metrics. Section 4 introduces our taxonomy for categorizing secure fault-tolerant distributed file systems. Section 5 comments on related work and concludes our paper.

2. Research Approach:

We begin by discussing desirable properties of any distributed file system. After deciding upon these, we describe four metrics that encapsulate our properties and allow us to qualitatively measure each system.

By distributing a file system across multiple computers, we increase the likelihood of a single component failure. However, users of a distributed file system actually expect the system to be tolerant of such local failures. If it were not, then one of the major motivations for building distributed file systems evaporates. Moreover, as users hear more about hacker attacks, they wish for their distributed file system to be secure even under drastic conditions, such as when a hacker is actually successful in taking over a computer. In short, users expect some level of *fault-tolerance* from a distributed file system.

Metric 1: Fault-tolerance – The ability of a system to continue normal operation despite failure of one or more of its components. In the context of distributed file systems, a fault could be a server crash resulting in loss of data, or a server acting in an arbitrary manner (Byzantine), as would occur due to a bug or malicious operation. The difference in approaches lies in the techniques each file system employs to handle failures.

Distributed file systems are prevalent in universities, corporations, and government offices. Each of these is a dynamic entity, expecting to grow and change with time. A corporation with five

thousand employees has little need for a static file system that can only handle up to six thousand users. If the company grows at only five percent every year, then the file system will be obsolete in just four years. Further, system performance should degrade gracefully as usage steadily increases. An important measure of a file system is its *scalability*.

Metric 2: Scalability -- A system's ability to easily adapt to changes in size and use while maintaining acceptable performance. With considerable effort put in designing such systems, it becomes imperative that these systems be viable for an extended period of time.

Though a file system may be able to grow with the addition of more computers, this advantage could be overshadowed if it requires a system administrator to power down all current machines at night, manually reconfigure each one, and power up the machines again. From the system administrator's perspective, such a system would not be practical. Similarly, a distributed file system should not require users to learn an obtuse interface. From the user's perspective, the system should export semantics and interfaces similar to systems they have already used, such as Windows® and Unix.

Metric 3: Usability/Manageability -- The ease and effectiveness with which a system can be used and maintained in the manner intended by the designers. Interfaces presented to users should be intuitive and familiar. Maintenance operations should be straightforward and require minimal manual reconfiguration by the administrator.

Distributed file systems must solve some issues that a local file system does not, e.g., concurrency. Consider a situation where two users concurrently write to the same file. When they later read the file, what should they see? Perhaps they would see only one user's version of the file, discarding the other user's work. Or perhaps they would see an interleaved file. Or perhaps each user would somehow still see the version of the file he wrote. Designers of distributed file systems must provide *consistency*.

Metric 4: Consistency – The formal semantics that a file system guarantees all users even under concurrent operations and failures. Reading from a file after a successful write should return the updated data, provided no intermediate writes committed in the meantime.

---

3. Case studies:

In this section, we outline four distributed file systems and analyze each along the four metrics we proposed in section 2.

3.1 Farsite

Farsite is a serverless distributed file system that is implemented on a set of incompletely trusted desktop machines [1]. A subset of desktop machines collectively acts as a centralized server. The main design goal of Farsite is to provide secure, reliable file service by using unreliable, insecure desktop workstations. The system administrator designates a set of machines as the *directory group* (or logical server). Farsite achieves security through public key cryptography. A certificate authority grants each user and machine a digital certificate. Farsite validates users, machines and directory groups through their respective certificates.

Each member of a directory group runs a Byzantine agreement protocol with the other members to ensure they all perform the same operations in exactly the same order. Thus, meta-data is replicated across all members of a directory group. Unlike metadata, Farsite does not replicate file data across each member of the directory group. Instead, it stores file data on separate machines (file hosts) and keeps the secure hash of the contents with the directory group.

3.1.1 Fault Tolerance

Farsite assumes a Byzantine failure model in which any machine (directory group member, file host, or client) can crash or begin performing malicious actions. We describe what mechanisms are in place for each of these scenarios.

Since operation of the directory group can tolerate Byzantine failures, we need only worry about a machine that leaks information to the outside world. Users can guard their privacy by encrypting the contents of their files with their private keys. Thus, a malicious directory group member can only corrupt data. However, this is not even allowed since all members execute a Byzantine agreement protocol.

Farsite tolerates failures of file hosts by replicating file data across multiple hosts. If a file host crashes, then the replicas are still available. If a file host corrupts file data, the tampering can be detected because directory group members store a secure hash of the file contents.

A user cannot affect another user's files because all file operations are mediated by the directory group. Further, a user must sign his file operations. If your signature does not match the list of authorized users for a particular file, then the directory group rejects your request.

Farsite achieves increased availability by migrating the operations of failed machines to operational ones. Farsite performs the above migrations based on the replicated information available on other machines.

3.1.2 Scalability

The ideal model of decentralization is one in which users perceive the system as monolithic even when their accesses span many machines. Farsite achieves this by exporting a logical file server while implementing the server on a group of machines. The novelty of Farsite is that it provides reliability and availability from the collective performance of a group of unreliable and untrusted machines.

Designating a group of machines to act as a logical server rather than employing a centralized server, however, does not remove the bottleneck. For example, a directory group of five machines would be a potential bottleneck when handling thousands of clients. Farsite therefore employs *delegation of namespace*. A directory group can issue *delegation certificates* designating a chosen set of machines to act as a different directory group, managing a part of the namespace. The original directory group redirects all future accesses from clients for files in the delegated namespace to the new directory group.

To improve performance, clients cache mappings of pathnames to directory groups. Farsite employs hint-based pathname translation. When requesting a certain file, a client first finds the directory group with the longest matching prefix. If that directory group no longer manages the namespace, the client is either redirected to the directory group that actually manages that namespace or it finds the next longest matching prefix and repeats the process.

The disadvantage in executing a Byzantine agreement protocol for fault-tolerance is performance – a serious concern with respect to scalability. Farsite employs a lease mechanism to manage concurrent accesses to files. A user can only write to a file provided he has the corresponding lease. Farsite also places a hard limit on the number of files that can be opened concurrently for

writing and a soft limit on the files opened for reading to prevent excessive consistency management.

3.1.3 Usability / Manageability

Farsite provides the benefits of a local desktop file system (low cost and privacy) along with the benefits of a central file server (location-transparent access and a shared namespace). Multiple roots each acting as a virtual file server can exist in the system.

It requires no central administration beyond that needed to bootstrap the initial system and authenticate new users and machines. While running, a Farsite system will autonomously replicate files, migrate operations, and delegate namespaces without human intervention. The authors claim that overhead after configuration is minimal.

Other possible tasks for local administration like hardware upgrades and backup of private data also have appropriate support. Hardware upgrades are treated as just another case of machine failures, in which case the system responds by migrating responsibilities to operational machines. However, a system administrator could of course notify the system that a particular machine will go down soon, so that Farsite can take proactive measures.

3.1.4 Consistency

Farsite maintains directories and files in consistent states through two techniques: Byzantine agreement protocols and lease mechanisms.

Each directory group member stores a current replica of the metadata that the group manages. Upon receiving a client request, each member runs a Byzantine agreement protocol to ensure consistency in the face of replication. The Byzantine protocol guarantees agreement as long as fewer than a third of the machines misbehave.

Farsite controls accesses to files for reads and writes through a lease mechanism. For example, to write to a file, a client must obtain a write lease, whose semantics guarantee that no other client has an unexpired write lease for the same file. Similarly, clients obtain read leases to read from files. If a client requests a write lease while other clients hold unexpired read leases, all outstanding read leases are revoked and the corresponding clients are notified of the concurrent access.

Different types of leases exist in a Farsite system. A read-only 'content' lease guarantees that the client always sees fresh data, a read-write 'content' lease is effectively a lock that enables the client to make consistent modifications to the file locally. The lease is recalled when another client makes a request for read-write content lease on the same file. This mechanism, however, is not suitable for large-scale write sharing that would result in frequent lease recalls. A maximum lease expiration time is provided to safeguard against the failure of a client while it holds the lease.

A 'namespace' lease is provided for exclusivity in renaming and/or creating files in a namespace region. With the exclusivity provided by the lease mechanism there is no need in Farsite for conflict detection and resolution as employed in other distributed file systems. The disadvantage, however, is that this mechanism incurs overhead even for non-concurrent accesses.

3.2 OceanStore

OceanStore [6] is a global storage utility that guarantees a number of desirable properties: fault-tolerance, availability, consistency, and durability. It is unique in that the designers of OceanStore targeted it for global scale storage. It presents many new concepts like *deep archival storage* and *nomadic data*. Deploying a global-scale storage utility immediately opens the door for malicious attacks, and thus, OceanStore assumes a Byzantine failure model.

Objects - the fundamental units in OceanStore are assigned globally unique identifiers (GUIDs) and can only be modified through *updates*.

3.2.1 Fault-tolerance

OceanStore addresses an issue of fault-tolerance that many other file systems do not consider: regional disasters. Even in the presence of such disasters, OceanStore aims for availability, consistency, and reliability. It does so through *nomadic data*, a concept that objects (the fundamental units in OceanStore) are not bound to a particular machine. Rather, they can float from machine to machine, dispersing and replicating themselves around the world and thus tolerating local power outages and natural disasters. OceanStore also tolerates Byzantine failures by placing a set of replicas for a file in charge of all updates to that file. These primary replicas run a Byzantine agreement protocol and disseminate the results of each update to all other (secondary) replicas.

OceanStore provides a third fault-tolerant mechanism unseen in our other file systems: *deep archival storage.* The goal of deep archival storage is for a user to be able to store a file and be assured with high probability that the file will remain in the system years later in the presence of numerous faults. OceanStore achieves this through Merkle trees (for integrity) [10] and erasure codes (for data loss) [8, 13]. The erasure code mechanism treats an input file as n fragments, and transforms them into m (m > n) fragments such that the original input file can be reconstructed using any n of the m fragments. OceanStore then distributes these fragments across multiple servers, preventing regional disasters from making a file unavailable.

Similar to Farsite, OceanStore users can prevent leaking of sensitive information by encrypting their files prior to storing them on servers. Servers guard against unauthorized writes by running a Byzantine agreement protocol.

3.2.2 Scalability

The designers of OceanStore envisioned a global scale data utility. It is not surprising, therefore, that OceanStore scales up to 10^{10} users with approximately 10^{14} files. Operation on this scale necessitates mechanisms different from those of other distributed systems. For example, due to the sheer size of the system with servers spread across the globe, the need to provide transparency and reduce latency of access becomes even more important. Hence, OceanStore supports *nomadic data* and *promiscuous caching.*

Objects in OceanStore migrate from server to server. They do this for fault-tolerance, and to adapt to changing usage patterns. Replicas roam (like nomads) closer to sites of frequent accesses. This reduces network latency, and thus, improves performance for the end user. In addition, users can aggressively cache objects anytime and anywhere. These two design choices greatly improve performance. However, they complicate data location.

OceanStore locates objects in one of two ways. First, a server attempts to find an object using a fast probabilistic algorithm based on attenuated Bloom filters [2]. If the first technique fails, then a server uses a slower deterministic algorithm. A Plaxton-like distributed data structure [14] stores the location of each OceanStore object. Using a Plaxton-like algorithm, servers locate any object in O(log n) hops, where n is the number of servers in the system.

3.2.3 Usability / Manageability

The OceanStore API provides a clean abstraction divorcing the details of a global-scale file system from the properties desirable of a distributed file system. The API provides a user with full access to sessions, sessions guarantees, updates, and callbacks. For users who want to use OceanStore like a prototypical Unix file system, OceanStore also provides interfaces to the API, deemed facades. Different facades exist for different semantics. The designers of OceanStore have implemented a façade that emulates traditional Unix file system semantics.

An important property of OceanStore is *introspection*, the ability for the system to adapt its behavior to usage patterns and current conditions. Objects in OceanStore are not limited to files and directories; an object in OceanStore can monitor usage patterns and take proactive actions to enhance performance and load balance the system.

3.2.4 Consistency

OceanStore's ideas of nomadic data and promiscuous caching benefit scalability and performance. However, they make maintaining consistency and coherence more difficult. Earlier systems such as Coda [5] and Bayou [15] compromised consistency for high availability, while systems such as Farsite assumed only small scale write sharing and hence found the lease mechanism (effectively a lock) satisfactory.

OceanStore allows large-scale write sharing. Therefore, OceanStore employs an update model based on conflict resolution that can support strong consistency (ACID semantics) as well as weaker consistency semantics. An update in OceanStore is a list of predicates and corresponding actions. An update is applicable to a file if at least one predicate evaluates to true. In that event, the actions associated with the first true predicate are applied to the file.

Recall that each file in OceanStore has a primary set of replicas in charge of updating it. The primaries are usually machines located in high-connectivity, high-bandwidth regions of the network. To resolve conflicts, the primaries use a Byzantine agreement protocol to decide on a total ordering of conflicting updates. Once decided, they apply each update sequentially and notify other replicas in the system of the total ordering. Depending on an application's semantics, secondary replicas can sometimes propose a tentative total ordering that application's can use. Of course, the primaries' decision will overwrite any tentative ordering, but this provides better performance, provided an application is willing to tolerate weaker semantics.

3.3 Ivy

Ivy [11] is a distributed read/write peer-to-peer filesystem. The file system itself consists solely of a set of logs, one log per participant with permissions to update the filesystem. Each participant finds data by consulting the logs of all such participants, but appends only to its own log to record changes. Given the same set of logs, all participants establish identical total orderings on log records, and thus, form the same view of the file system, even in the presence of concurrent updates.

Participants store their logs in DHash [3], a secure distributed hash table. DHash performs much of the authentication and replication necessary for handling benign failures and limiting malicious activity. Each participant maintains a pointer to the end of his log in a publicly readable data block. Therefore, an Ivy file system is uniquely determined by a set of these pointers. A *view block* in DHash is an immutable data block that identifies where each participant stores the pointer to the end of his log. Ivy uniquely identifies a file system with the content-hash key of the view block, which is essentially a self-certifying pathname [9].

3.3.1 Fault-tolerance

Since Ivy stores all logs in DHash, it is resilient to crash failures. DHash replicates each record that a participant appends to his log. Further, DHash also replicates the pointers to the end of participants' logs. Therefore, even if a participant crashes, his log is still available, and remaining participants still see the changes he made to the file system through his log.

Against more malicious behavior, Ivy adopts a novel idea of trust. Participants cooperate to form a file system, and thus, each participant implicitly trusts every other participant. Ivy does not provide any level of access control. However, if a participant misbehaves, the remaining participants can essentially force him out of the file system. They accomplish this by establishing a new view block that no longer contains a pointer to that participant's log. Henceforth, they ignore that participant's log. Observe that they have essentially undone all his malicious actions because a participant cannot modify any other log but his own. However, by ignoring his log, the remaining users may need to fix some resulting inconsistencies. They do so through the ivycheck tool.

3.3.2 Scalability

If each participant iterated through every participant's log for every file operation, the system would not scale beyond a few dozen peers. To avoid this operation, Ivy provides a snapshot mechanism where each participant stores the most up-to-date state of the file system. Ivy uses the underlying DHash server to store the snapshots. Though each participant has his own private snapshot, the snapshots across many participants are largely identical, and hence, DHash automatically shares their storage.

Though snapshots obviate the need for a participant to iterate through entire logs, each participant must still contact everyone else for the newest updates before attempting any file operation. The communication bottleneck as the system scales up becomes acute.

A more severe bottleneck for scalability is the overhead in adding and removing participants from a file system. Any time Ivy adds or removes a participant from the file system, it must create a new file system (a view block) and, in the case of removal, must repair potential inconsistencies. Existing participants must also agree on the new membership.

3.3.3 Usability / Manageability

Ivy provides a kernel device driver so that users of an Ivy file system are (ideally) unaware of the underlying infrastructure. A system administrator simply needs to mount an Ivy file system by specifying a view block. In ideal conditions, users can treat the file system similarly to NFS. However, Ivy does not currently support mechanisms to inform the user in case of a write/write conflict. The user must explicitly run the provided lc tool to detect that a write has been missed.

An Ivy file system continues to function even in disconnected operation. Each participant in a partition can still append records to his log and access files that were in the system prior to the partition. However, when two partitions merge, unexpected and undesirable results may occur. When connected operation resumes, Ivy does *not* notify users even if there were conflicting updates. Since log records are totally ordered, merging logs that were updated in different partitions may yield undesirable interleavings. An Ivy user can only detect such a problem by running a separate tool that presents the user with the state of the file systems in different partitions prior to the merge. A user can then manually make the appropriate changes.

As discussed earlier, Ivy also makes removing a user from the system very cumbersome.

3.3.4 Consistency

Ivy achieves much of its consistency through immutable log records. Ivy attaches version vectors on each log record (similar to vector clocks). This establishes a partial order on all log records. Users also agree on a total ordering among participants, and thus, establish a *total* ordering on

log records. This common total ordering allows two participants to view the same file system even if concurrent reads and writes occurred in the past. However, at the time participants execute concurrent operations, Ivy's semantics allow unintuitive events to occur.

This occurs because an operation may come after another in the total ordering, but commit first in real time. For example, consider two concurrent operations "delete file1" and "rename file1 file2". Suppose the rename operation commits and then the delete commits. In the window of time between the two commits, a participant could see file2, and an instant later, find neither file1 nor file2, though only a "delete file1" operation occurred. In summary, at any time participants may only see a subset of concurrent operations that will be totally ordered after operations that will commit later in real time.

As mentioned earlier, Ivy does not notify users when connected operation resumes, which may result in file system inconsistencies.

3.4 Frangipani

Frangipani [14] is a cluster file system, where every client trusts every server. The structure of Frangipani is similar to Ivy in that it is a file system layer implemented on top of a storage abstraction layer. Petal [7] is a distributed storage service that provides a virtual disk abstraction with a distributed lock server, and Frangipani is the file system implemented on top of Petal. As a consequence, Frangipani inherits much of its scalability, fault tolerance and easy administration from this underlying storage system. The Petal storage system can tolerate one or more disk or server failures, provided a majority of servers are operational and each replicated data block remains somehow physically accessible. Petal and the lock service are also distributed for fault-tolerance, scalability and load balancing.

3.4.1 Fault-tolerance

Unlike our other file systems, Frangipani assumes a crash failure model (as does Petal). Further, servers can communicate securely with each other (without cryptographic techniques). Frangipani relies on the underlying Petal replication scheme to provide availability in the face of failures. To provide further fault-tolerance, each Frangipani server maintains a log that all other servers can read, but to which only it can write. A server's log contains changes that this server was supposed to carry out on behalf of users. When a server crashes, other servers can run a recovery algorithm to replay all pending operations in the crashed server's log. The recovery algorithm ensures that the Frangipani file system stays in a consistent state.

Frangipani allows transparent server addition, deletion and failure recovery by utilizing persistent logs, locks, and the underlying Petal service. Frangipani uses write-ahead redo logging (i.e. for every update performed, a record describing the update is appended to the log) of metadata to simplify failure recovery. Failures may be detected either by a client, or when the lock service asks the failed server to return a lock it is holding and receives no reply. As long as the underlying Petal volume remains available, the system tolerates n-1 Frangipani server failures, where n is the number of servers.

3.4.2 Scalability

Since Frangipani is a cluster file system, its scalability properties do not compare well to systems such as Farsite and OceanStore. The most recent implementation only uses 256 servers. The designers of Frangipani intended it for an environment of engineers working cooperatively. However, for its given audience, it offers many valuable properties.

Frangipani provides improved load balancing over a centralized network file server by splitting up the file system load and shifting it to the machines that are actually using the files. This is a consequence of using a crash failure model instead of Byzantine. Frangipani allows transparent server addition, deletion and failure recovery by utilizing logs, locks and Petal.

Frangipani allows server addition and removal with little overhead. New servers need only know how to access the lock server and how to access its log for recording updates. Administrators remove servers by optionally forcing modified blocks to be written to disk or simply powering down the servers. Petal servers and lock servers can also be added and removed with equal ease.

3.4.3 Usability / Manageability

The implementers of Frangipani provide a kernel device driver for Unix systems. System administrators simply need to ensure that each computer appropriately mounts the Frangipani file system on start up. After this point, users see little difference between a Frangipani file system and the local file system.

Frangipani facilitates server addition and removal as mentioned above.

3.4.4 Consistency

Frangipani guarantees consistency through a distributed lock service. It employs multiple reader/single writer locks to coordinate access to the virtual disk and to keep the buffer caches coherent across the multiple servers. The granularity of locks is on files; in this case, Frangipani trades performance for simplicity.

Programs running on different machines have a coherent view of the file system, i.e., changes made to files or directories on one machine are immediately visible on all others. Frangipani provides the same semantic guarantees as on a local Unix file system through the distributed lock service. When the lock service detects conflicting lock requests, it asks the current holder of the lock to release or downgrade it to remove the conflict. The lock service handles server failures by letting locks expire.

4. Taxonomy:

This section describes a taxonomy that divides secure, fault-tolerant distributed file systems into categories. The difficulty in constructing taxonomies is that one design decision for a file system influences other design decisions significantly. The taxonomy should target the original design decision and not its effects. For example, an important design decision in OceanStore is to provide low latency access. This design choice manifests itself through nomadic data and promiscuous caching. This nomadic data concept in turn affects the fault-tolerance model. We aim to establish a taxonomy that cleanly classifies the original design decisions.

4.1 Update model

In a practical distributed file system, users should be able to read and write files without high overhead. However, any distributed file system must handle the scenario in which two users attempt to update a file concurrently. Designers of a file system have to make a decision to either allow conflicting updates or not. Choosing either one has its ramifications.

OceanStore and Ivy allow users to make conflicting updates. This decision allows a user to update a file at any time. However, the file system must then decide how to resolve the conflict when it is (and must be) detected. In OceanStore, users submit updates to any replica. Updates eventually propagate to a set of primary replicas, which execute a Byzantine agreement protocol and apply the updates to their copies of the file. The primaries then push the updates to all other replicas in the system. Ivy, on the other hand, decides on a total order based on the public keys of the participants responsible for concurrent updates. All participants a priori agree on this total order and get a consistent view of the file system.

All updates to a file in OceanStore pass through a primary tier of replicas. This primary tier is responsible for establishing a total order over the conflicting updates and then sequentially applying them. However, an application with weak consistency semantics may retrieve a copy of the file from a secondary tier before the primaries push the final update to all replicas.

Farsite and Frangipani do not allow concurrent updates to a file. Both employ a lease/lock mechanism, where a user has exclusive write access to a file for a finite time interval. When that time expires, the user pushes his updates to the servers. The advantage of this approach is that the file system does not have to provide any conflict resolution schemes. In addition, all requests for a particular file will always return the most recent version (as opposed to OceanStore). On the other hand, if a user crashes while in possession of a lease, then the file system must either wait for the lease to expire or revoke it before letting another user access that file.

There is a distinct tradeoff between allowing conflicts and not. If a system allows conflicting updates, users can update the file at any time, but may have to accept stale files and unexpected results when another user's update immediately overwrites his. By not allowing conflicts, the system provides stronger consistency guarantees, but at the cost of sacrificing availability since users must obtain a lease before proceeding.

4.2 Failure model

The ability to deal with failures is crucial to any distributed file system. As a file system scales up, the probability of faults increases. When such situations arise, the system must still provide the same guarantees (consistency, availability, security) as normal operation.

Frangipani is a cluster file system, where a set of trusted servers communicate securely and only fail by crashing. Users submit updates to a server. That server then logs that update onto stable storage and proceeds to actually update the corresponding file. In the presence of crash failures, the surviving members can consult the logs of failed servers and replay the appropriate updates. As a consequence of a crash failure model, data does not have to be encrypted when stored on servers. Further, the degree of replication can be much less since a server can only fail by crashing.

Farsite and OceanStore assume a Byzantine failure model. Each system maintains a set of servers responsible for a set of directories and files. Since users do not implicitly trust any single server, every request and update triggers the execution of a Byzantine agreement protocol. In addition, files must be encrypted to prevent servers from leaking the contents of the file to unauthorized individuals. Further, users should have access to secure hashes of each file to validate the retrieved contents. The immediate consequences of designing a Byzantine fault-tolerant file system include higher overhead and replication.

Ivy is unique in its approach to fault-tolerance. By storing the logs on the DHash server, it ensures their availability even when the participant has crashed or is disconnected. Hence, it handles crash failures effectively. However, it implicitly trusts every peer that is participating in the file system. When a peer fails or acts maliciously, the other peers in the system form a new file system, excluding the ill-behaved peer from the group. The disadvantage of this approach is that a malicious peer could wreak havoc on the system until he was detected. Ivy does not address the problem of detecting such behavior. Once detected, however, correct participants can form a

new file system, excluding the malicious peer. Unfortunately, excluding a previously trusted user may leave the resulting file system inconsistent.

4.3 Locating files

As a distributed file system scales up, the contents of the file system will be spread across multiple computers to support low latency and highly available access. Such a system must provide a means to efficiently locate files and return errors only if the file does not actually exist.

In Farsite, clients contact a directory group. The directory group then responds with a list of nodes that store the file. A client can then contact any one of those nodes to retrieve the desired file. The basic scheme remains the same when a directory group delegates a section of its namespace to another directory group, except that only the second directory group responds to the client's request.

OceanStore uses two mechanisms to efficiently locate files. The first is a fast probabilistic algorithm using Bloom filters and the second is a deterministic solution using a Plaxton-like data structure. Each server maintains an attenuated Bloom filter that reflects which files it holds and which files its neighbors hold. By consulting these Bloom filters, a request can be satisified if the file is 'close' to the requestor. Otherwise, the servers resort to a deterministic mechanism that uses O(log n) hops.

Ivy and Frangipani are file system abstractions that reside on top of a storage layer. Both derive their locating mechanisms from the underlying storage layer, DHash for Ivy and Petal for Frangipani.

5. Conclusions:

Distributed file systems have evolved from experiments confined to research labs into practical systems that thousands of users in universities and corporations expect. Researchers have proposed numerous techniques to address issues such as fault-tolerance, scalability, availability, and security, all properties that were weaknesses of distributed file systems only a decade ago.

We have examined four secure, fault-tolerant, distributed file systems with respect to four metrics. We have seen a profound interdependence among these metrics, namely meeting one metric may sacrifice another. File system designers have usually focused on a specific issue, taking it to a logical extreme (e.g. Coda's disconnected operation), or they have tried to strike a balance among a number of design issues. The taxonomy presented tries to underscore some of the fundamental choices in designing distributed file systems.

From our analysis of distributed file systems, we see two areas for future work: 1) the integration of group key management with these systems to facilitate sharing, and 2) support for data access from devices with limited resources. We believe that the recent trend towards mobile and ubiquitous computing introduces additional design issues for file systems. File systems may be expected to support efficient access from devices limited in connectivity as well as resources, e.g., PDAs, cellular phones, and palmtops.

Given the continued interest in distributed file systems as a data sharing utility, we foresee these systems integrated into everyday life. We anticipate distributed file systems integrated into every household computer, allowing seamless sharing of files between any two home users.

References:

[1] Atul Adya, Willaim J. Bolosky, Miguel Castro, Gerald Cermark, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, Roger P. Wattenhofer, “FARSITE : Federated, Available, and Reliable Storage for an Incompletely Trusted Environment”, Fifth Symposium on Operating System Design and Implementation(OSDI), December 2002.

[2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, volume 13(7), pages 442-426, July 1970.

[3] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. SOSP '01, October 2001.

[4] Ping Guan, Martin Kuhl, Zhimin Li, Xiaoyu Liu, “A Survey of Distributed File Systems”, University of California, San Diego, 2000.

[5] James J. Kistler, M. Satyanarayanan, “Disconnected Operation in the Coda File System”, ACM Transactions on Computer Systems, February 1992.

[6] 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”, Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000.

[7] Edward K. Lee, Chandramohan A. Thekkath, “Petal : Distributed Virtual Disks”, Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, 1996.

[8] M. Luby, M. Mitzenmacher, A. Shokrollahi. and D. Spielman, "Analysis of low-density codes and improved designs using irregular graphs," in Proc. 30th Ann. ACM Syrup. Theory Computing, pp. 249- 258, 1998.

[9] D. Mazieres, M. Kaminsky, M.F. Kaashoek, and E. Witchel, "Separating Key Management from File System Security," Proc. ACM Symp. Operating Systems Principles, pp. 124-139, Dec. 1999.

[10] R. Merkle, “Protocols For Public Key Cryptosystems”, IEEE Symposium on Security and Privacy, 1980.

[11] Athicha Muthitacharoen, Robert Morris, Thomer M. Gill, and Benjie Chen, “Ivy : A Read/Write Peer-to-Peer File System”, Fifth Symposium on Operating System Design and Implementation(OSDI), December 2002.

[12] Karin Peterson, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, Alan J. Demers, “Flexible Update Propagation for Weakly Consistent Replication”, Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP), October 1997.

[13] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Software | Practice and Experience, 27(9):995-1012, Sept. 1997.

[14] C. Plaxton, R. Rajaraman, A. Richa, “Accessing Nearby Copies Of Replicated Objects In A Distributed Environment”, Proceedings of ACM SPAA, pages 311-320, Newport, Rhode Island, June 1997.

[15] Douglas B. Terry, Marvin M. Theimer, Karin Peterson, Alan J. Demers, Mike J. Spreitzer, Carl H. Hauser, “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System”, Proceedings of the 15th ACM Symposium on Operating Systems Principles, December, 1995.

[16] Chandramohan A. Thekkath, Timothy Mann, Edward K. Lee, “Frangipani : A Scalable Distributed File System”, Systems Research Center, Digital Equipment Corporation.

[17] Mahadev Satyanarayanan, “A Survey of Distributed File Systems”, Carnegie Mellon University, February, 1989.