Toward Eidetic Distributed File Systems

Xianzheng Dou, Jason Flinn, and Peter M. Chen

Abstract

We propose a new point in the design space of versioning and provenance-aware file systems in which the entire operating system, not just the file system, supports such functionality. We leverage deterministic record-and-replay to substitute computation for data. This leads to a new file system design where the log of non-deterministic inputs, not file data, is the fundamental unit of persistent storage. We outline a distributed storage system design based on these principles and describe the challenges we foresee for achieving our vision.

1 Introduction

Two important goals of recent file system research and commercial product development are retention of past state (e.g., versioning file systems) and interconnecting file data (e.g., by storing provenance information).

Past state is typically retained by efficiently storing checkpoints of file information; systems that provide this capability include Elephant[21], WAFL[10], Wayback[24], and Ori[12]. Versioning file systems typically retain checkpoints of file state at specific intervals (e.g., on operations such as file close, at specific time intervals, or as designated by an administrator). There is a fundamental tradeoff in choosing the checkpoint frequency: more frequent checkpoints increase storage costs, but less frequent checkpoints limit the data that can be recalled.

Connections between file data may include temporal relationships [22] or provenance [14]. The fundamental tradeoff in such systems is the detail of connection information: more detail increases storage costs, but less detail limits what information can be recalled about connections and provenance during future queries.

These systems have explored the design space where state retention and provenance are solely or mostly the responsibility of the file system. Recently, we proposed and prototyped the concept of an *eidetic system* [6], in which the entire operating system is responsible for providing similar functionality. An eidetic system can recall any past user-level state of the computer system, including file data, system inputs and outputs, and transient process state (address space and register values). Further, it can reason about the provenance of that state at byte granularity using dynamic information flow tracking to understand what past data and computation affected that state and what future data and computation was in turn influenced by the state in question. These properties are a superset of those provided by versioning and provenance-aware storage systems.

Whereas our prototype considered only local storage and grafted eidetic support onto an existing file system, this paper explores a clean-sheet design for a distributed eidetic file system. We specifically consider the case of a small number of personal devices and cloud servers cooperating to provide the eidetic property within the context of a distributed file system. In one sense, we are clearly choosing extreme endpoints for the tradeoffs in the first two paragraphs. Choosing those endpoints frees us to explore radically new design choices for structuring distributed file systems. For instance, one surprising result is that the fundamental unit of data retention becomes the log of non-deterministic inputs rather than the actual file data.

An eidetic system uses deterministic record-and-replay [8] to regenerate process state. This is a DVR-like capability in which the execution of one or more processes can be recorded and precisely replayed at any time in the future. Deterministic record-and-replay is based on the observation that most of a computer system's execution is deterministic; thus, if the system records the value of non-deterministic events such as the results of system calls and thread scheduling decisions and supplies the same values during re-execution, it can guarantee that the replay executes the same sequence of instructions and generates the same outputs.

Thus, an eidetic system can substitute computation for data. Rather than store a checkpoint for a given application or file system state, the system can regenerate the state by re-executing the computation that produced that state. On the other hand, it is not practical to substitute data for computation; having two checkpoints of file system or application state reveals little about the computation that transformed the first state into the second.

The logical outcome of this observation is that the logs of non-deterministic operations, not file data, are the fundamental unit of persistent storage in an eidetic file system. File system data can be regenerated by replaying some subset of the logs, while it is not usually possible to infer the logs of non-determinism or resulting com-

putation from file system snapshots of any granularity. Of course, storing file system state is necessary for acceptable performance; one would not want to reproduce all past computations simply to read the current value from a file. The difference is that we can regard all file data, directory information, and metadata as cached values which can be discarded and regenerated on demand.

Moving from a checkpoint-based design to a log-based one presents several opportunities for optimization. A distributed file system can choose to transmit or store a log of non-deterministic operations in lieu of the data produced by those operations. It can reduce resource usage by avoiding regenerating file system state until data is actually read. It can deduplicate the logs of non-determinism to reduce storage costs for application executions with similar actions. In this rest of the paper, we outline one possible design for an eidetic file system and describe an agenda for exploring such optimizations. It is our hypothesis that eidetic systems will prove more effective at realizing the goals of state and provenance retention than prior file-system-only solutions, while achieving equal or reduced resource costs than some systems that provide only a subset of these properties.

2 Design goals

We first list the design goals for an eidetic distributed file system. Most importantly, it should maintain the eidetic properties of recording all state and provenance across servers and mobile devices. This means that it must persistently store the logs of non-determinism for all processes executed on those devices. To recover provenance, it must also store the linkage between bytes written by one process and bytes read by another; e.g., which process wrote the bytes read from a file or socket.

Next, it should support an eventual consistency model appropriate for mobile devices that are intermittently connected and potentially unreliable. We have targeted the Coda weakly-connected consistency model [13] in which cloud server(s) host the first-class replica and clients host second-class replicas. Client updates are aggressively pushed to the server, which reintegrate changes with the complete first-class replica of file system state. Once the server accepts an update, other clients can retrieve the data even if the originating client is lost or unavailable.

Our final goal is to minimize resource usage while providing acceptable performance. Resources consumed by our system include storage, network bandwidth, computation cycles, and energy for both clients and servers. Since eidetic systems allow some resources to be traded for others (e.g., we can sometimes reduce bandwidth by replaying a computation to regenerate data rather than transmitting files over the network), we will need to develop methods to compare usage of different resources.

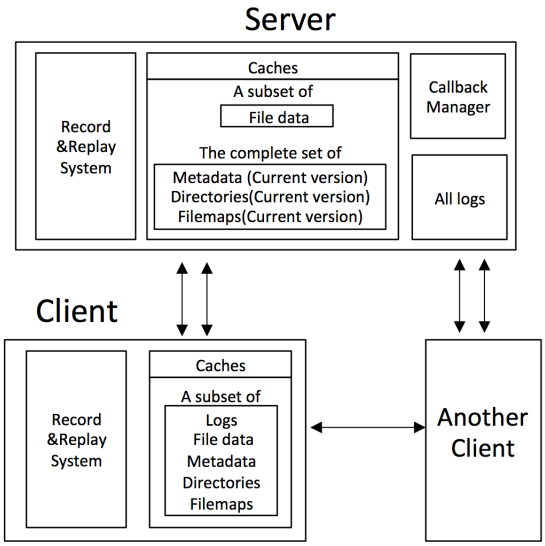

Figure 1: System architecture

3 Designing a distributed eidetic file system

3.1 Architecture

Figure 1 shows the types of data that a distributed eidetic file system should store. The fundamental storage type is a log of non-deterministic operations. Each log contains the results of system calls, timing and value of signals, and order of synchronization operations (e.g., lock acquisition and atomic instructions). This is sufficient to guarantee deterministic replay of race-free processes or groups of processes; we discuss handling data races in Section 4. The storage utilization of such logs is reasonable. In an approximately one-month deployment of eidetic systems on our workstations [6], we found that, after considerable optimization, the logs for all user-level processes consumed less that 2.6 GB of storage per workstation per day, so a 4 TB hard drive could store over four years of data. The distributed file system stores the logs of all user-level processes, assigning each a globally-unique log identifier. In order to support data provenance, each log operation that reads data produced by another process (e.g., via a file, socket, or other form of IPC) provides the log id(s) and operation(s) within those logs that produced the data. This information provides a causal ordering across log operations.

Replaying a log deterministically regenerates the outputs of the recorded process(es). This leads to the following invariant:

File Safety Invariant*: File system data is persisted as soon as all logged non-deterministic inputs that causally precede the operations that output that data have been made durable.*

This observation is similar to the durability property of committing operations to a file system journal. However, reading data using only logs of non-determinism would be horribly inefficient since each read operation would require reproducing the computation that wrote the data. Thus, the file system should also store the values of data and metadata for current files and directories (in much the same way that a journaling file system stores the current file system state in addition to the journal). However, the collection of such objects can be treated as a cache; we can delete values that we expect to not read in the future and for which the regeneration cost is trivial.

Our logs of non-determinism record the results of all system calls. This means that reads of file data can constitute a substantial portion of our log. However, many operations read the same data. In our prototype, we reduced log size with a read-after-write cache that holds past file versions. Read operations in the logs refer to the file version in the cache by reference, reducing storage utilization when the same data is read multiple times. If the desired file version is not in the cache, then the processes that wrote the desired data can be replayed to reproduce the needed values. In our prototype [6], we found that the RAW cache reduced log sizes from 4–97% (average 46%) across a wide range of applications. We could extend the RAW cache to also include file system metadata (e.g., directories and attributes) but the potential gains appear to be minimal. Analyzing approximately one month worth of logs from that prototype (30 GB of data), we found that such a cache would only reduce log sizes by an additional 2.7%.

An eidetic system requires one additional storage type for efficiency: a *filemap* that relates which log operations wrote different byte ranges of a file. Write operations update the filemap and read operations copy the relevant information into the log of the reading process for the set of bytes returned. This supports provenance queries and also allows the eidetic file system to determine which logs to replay in the event that needed file data is not cached. Filemaps can always be regenerated from logs and are cached on the server and clients in the same way as directories and file metadata.

Our distributed file system uses a Coda-like consistency model in which only the server hosts a complete first-class replica of file system data. This means that the server is responsible for storing all logs generated by any file system client. Clients store a subset of those logs. As a client executes applications, it asynchronously pushes the logs of non-determinism to the server in a manner consistent with the causal order of log operations. Thus, when the server acknowledges that it has received and stored a log range, all data written by operations in that range are durable. Since clients may be slow or temporarily disconnected, they should persist log data to local storage until the server acknowledges receipt; after that point, they are free to delete the logs.

We plan to use callbacks to maintain cache consistency. Clients will register callbacks when they cache logs, files, metadata, and filemaps; the server will send callback breaks when it persists a log that modifies any of these objects. Upon receipt of a callback break, the client will flush the specified object from its cache and obtain a new version on demand. Since we use optimistic concurrency, two clients may make conflicting updates to file system data. We will use version vectors to uniquely name each such version; including the vectors in the log of non-determinism will ensure that the data read can be precisely identified for replay. Version vectors will also be used to support manual conflict resolution.

Since the server is expected to have substantial storage resources, we envision that it will store the current state of directories, file system metadata, and filemaps; the storage requirements for these types are expected to be relatively small. In contrast, the server may choose to store only a subset of file data and regenerate content on demand via replay.

3.2 Write path

As an application executes, the client records a compressed log of nondeterministic inputs. This log is asynchronously persisted to disk, unless it is forced by an application sync. In addition, the locally cached copies of files, filemaps, and metadata are updated asynchronously, so that future reads will be efficient. Logs are pushed to the server asynchronously, with the order of log transmission consistent with the causal order of operations within the log. The server acknowledges log data as it is received; the client must pin a log region in its cache until the acknowledgment is received.

After storing a log region, the server scans it to determine the set of objects updated. It issues callbacks to any clients that cache updated objects. The server also uses the version vectors in the log to detect conflicting updates from different clients and flag these for manual resolution (if an object is in conflict, there will be multiple current versions until the conflict is resolved). The server updates the current version(s) of all directories, attributes, and filemaps modified by the log region. Most information needed for such updates is already included in the log of non-determinism; we add a small amount of data such as file names and timestamps so that such updates can be made without replaying the logged computation.

Next, the server decides whether it will cache the updated version of each file modified by the log region. Since it has persisted the log region and logged operations that causally precede that region, it already has the ability to regenerate any version. It may choose to defer

caching this version because it believes that it is unlikely to be read by another client and/or because the cost of regenerating the version is expected to be small. If the file is, in fact, never requested by another client, the server saves both network bandwidth and storage.

The server may choose to immediately cache the updated version of one or more files if sharing is likely or the cost of regenerating the file later is expected to be high. In this case, it can fetch the modified version from the client or replay computation to regenerate the version. Note that replaying the log may require updating other file versions read by the recorded application, so a recursive replay could be required.

Predicting which strategy to use is an interesting research challenge. The server must estimate the probability that a version will be requested by another client. It must also estimate the cost of replay; we expect that the user-level compute time of the recorded process is a good start for such an estimate. The server must balance storage costs, network costs, and compute costs. Finally, in the event that a non-cached file version is needed in the future, the server should estimate the probability that it will need to re-execute the logs vs. the probability that it will be able to fetch an appropriate version from the cache of the modifying client.

3.3 Read path

When a client application reads file system data or metadata, it first checks if it has the current version cached (as determined by it having not yet received a callback break for the specified object). We expect the vast majority of accesses to hit in caches. If the object is not cached, the client asks the server for a *recipe* for retrieving the version. The recipe may simply be the version of the requested object, or it may specify that the version should be fetched from another client (e.g., one located on the same sub-network as the requesting client). Alternatively, the recipe may specify that the client regenerate the version by replaying a log. If the files read by the log are not present in the client's cache, the recipe may include additional instructions on how to fetch or regenerate those files.

The server can choose a good recipe because it has knowledge of the state of all client caches (through the callbacks it maintains). It also can estimate the cost of replaying logs since it stores all logs and can therefore read recorded hints such as user-level compute time. Note that in the event that it does not cache a requested file version, the server itself can perform the re-execution to regenerate such content before shipping the version to the client. As with the write path, formulating the best strategy that minimizes network and compute cost is an interesting research problem.

4 Challenges

Since we envision a radical departure from current file system design, there are naturally a number of challenges we will need to overcome.

Data Races: Deterministic record and replay has low overhead for multithreaded processes only if it can assume the absence of unknown data races [6] (the outcome of known data races can be added to the log of nondeterminism). After a race, a replayed execution may therefore produce different results (e.g., file data) than a recorded execution. We plan to add hashes or checksums of file system modifications to the log to detect such occurrences. Once a divergence is detected, we can search through possible race outcomes until we find one consistent with the log. Previous research [1, 18] has shown search to be feasible if races are rare (which matches our experience with the eidetic system prototype) and logs are detailed (and our logs are indeed detailed). This converts the problem of races from a correctness issue to a performance one.

Different processor architectures: There are a variety of processor architectures across mobile devices and servers. However, deterministic record and replay assumes that the ISA is constant between recording and replaying. One way to address this issue is to employ several co-located servers with different architectures, each of which can replay logs from different types of clients. Another method is to run a replay within an emulator such as QEMU [20]. The Paranoid Android[19] project showed that the speedup gain from mobile phones to servers was more than sufficient to mitigate the slowdown caused by replaying in an emulated environment.

Privacy: Since replay logs reproduce any state, users must implicitly trust the servers that replay those logs (even if the logs are encrypted for storage, they must be unencrypted during replay). If cloud servers are not trusted, one potential solution is to leverage recent hardware support for private code execution in the cloud [3].

Log size: Our prototype eidetic OS [6] used a number of compression techniques to enable projected storage of four or more years of computation from a workstation on a single 4 TB hard drive. However, further reductions in log size would increase the benefit of eidetic distributed file systems by reducing both storage and network bandwidth consumption. One promising avenue we plan to explore is deduplication of log data across multiple executions of the same program. Our hypothesis is that there is a considerable amount of repetitiveness in application activities; for instance start-up sequences may be very similar from run to run. Potentially, we could encourage such similarities by making time values, thread scheduling, and other sources of non-determinism behave more deterministically.

5 Related Work

Many prior file systems [7, 10, 12, 16, 21, 24] have maintained past state in the form of versions or checkpoints. However, these systems face a tradeoff between checkpoint granularity and storage costs. Even if all file modifications are retained, they still lack the ability to reproduce past computations and intermediary process state. Provenance-aware storage systems [14, 15] preserve causal histories of file data and capture the relationship between processes and files. However, they do not capture the causal history within process execution as they cannot reproduce non-deterministic computation.

Like out proposed work, Nectar [9] associates data with its computation and substitutes one for the other. Data can be deleted if it is not accessed for a long time and recomputed on demand by rerunning the computation. However, Nectar only supports functional and deterministic LINQ programs, while our eidetic systems support general-purpose programs. Similarly, BadFS [4] and TREC [23] are capable of selective recomputation of deterministic operations by combining file systems with make-style batch processing.

One of our goals is to minimize network resource usage, possibly by increasing client and server computation. Systems such as LBFS [17] and Shark [2] explore a similar tradeoff by using deduplication to trade computation for a reduction in network traffic. The tradeoff in deterministic record-and-replay occupies a different spot on the compute/network tradeoff.

Like our work, operation shipping [5, 11] trades computation for communication by shipping a list of operations used for modifying a file to another machine. Without deterministic record-and-replay, this approach is very fragile as it is difficult to get the replayed operations to produce the same outputs as the recorded operations.

6 Conclusion

An eidetic distributed file system is a new point in the design space of versioning and provenance-aware file systems. Rather than the file system being mostly or solely responsible for providing these features, the rest of the operating system participates by supporting deterministic record-and-replay and tracking provenance via IPC. Our hypothesis is that an eidetic distributed file system can be more effective in providing state retention and provenance while achieving reasonable overheads. We are building the system proposed in this paper to prove or disprove that hypothesis.

References

[1] ALTEKAR, G., AND STOICA, I. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 193–206.

[2] ANNAPUREDDY, S., FREEDMAN, M. J., AND MAZIÈRES, D. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, May 2005), pp. 129–142.

[3] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (Broomfield, CO, October 2014).

[4] BENT, J., THAIN, D., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LIVNY, M. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation* (March 2004).

[5] CHANG, T.-Y., VELAYUTHAM, A., AND SIVAKUMAR, R. Mimic: Raw activity shipping for file synchronization in mobile file systems. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services* (Boston, MA, June 2004), pp. 165–176.

[6] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. M. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (Broomfield, CO, October 2014).

[7] Dropbox. `http://www.dropbox.com`.

[8] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 211–224.

[9] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[10] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In *Proceedings of the 1994 Winter USENIX Conference* (1994).

[11] LEE, Y.-W., LEUNG, K.-S., AND SATYANARAYANAN, M. Operation shipping for mobile file systems. *IEEE Transactions on Computers 51*, 12 (December 2002), 1410–1422.

[12] MASHTIZADEH, A. J., BITTAU, A., HUANG, Y. F., AND MAZIÈRES, D. Replication, history, and grafting in the ori file system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farminton, PA, October 2013), pp. 151–166.

[13] MUMMERT, L. B. *Exploiting Weak Connectivity in a Distributed File System*. PhD thesis, Department of Computer Science, Carnegie Mellon University, December 1996.

[14] MUNISWAMY-REDDY, K.-K., AND HOLLAND, D. A. Causality-based versioning. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (San Francisco, CA, February 2009).

[15] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (Boston, MA, May/June 2006), pp. 43–56.

[16] MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. A versatile and user-oriented versioning file system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (San Francisco, CA, March/April 2004), pp. 115–128.

[17] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Banff, Canada, October 2001), pp. 174–187.

[18] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (October 2009), pp. 177–191.

[19] PORTOKALIDIS, G., HOMBURG, P., ANAGNOSTAKIS, K., AND BOS, H. Paranoid android: Versatile protection for smartphones. In *Proceedings of the Annual Computer Security Applications Conference* (December 2010).

[20] QEMU. `http://www.qemu.org/`.

[21] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. *SIGOPS Operating Systems Review 33*, 5 (1999), 110–123.

[22] SOULES, C. A. N., AND GANGER, G. R. Connections: using context to enhance file search. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 119–132.

[23] VAHDAT, A., AND ANDERSON, T. Transparent result caching. In *Proceedings of the 1998 USENIX Annual Technical Conference* (June 1998).

[24] Wayback: User-level versioning file system for linux. `http://wayback.sourceforge.net/`.