Diamond: Many Devices, One File System

Guy Hugot-Derville

May 8, 2009

Abstract

Diamond is a read/write highly-available and partition-tolerant file system. It allows a user to edit the file system at the same time, from many devices.

Diamond consists of a set of logs, one for each device, which is stored on a trusted online backend, accessible through a simple put/get interface. This allows end-user to easily switch to another storage provider. Each device appends to its own log to modify the file system and reads from all the logs to find data.

Diamond presents data through a conventional file system interface. End-users can mount either a read-write version of the device current file system version or read-only mount a previous version viewed by any device. This allows the end-user to access all the file versions, may they be from the past or another device, and to modify its current version.

1 Introduction

Although end-users own an increasingly number of devices, no synchronization tool has emerged as a standard. The lack of interoperability, confidentiality and simplicity prevents the average user from jumping the gap. Our answer is a readable and writeable shared file system. The solution is easy to use since the end user is exposed to a well-known interface: the file system. In addition we provide command line tools to change the default view on the file, be it an older version or a conflicting version from another device. It is interoperable: like Cumulus[5] we implement our file system on top of a simple put/get interface; hence it can be easily adapted to different online storage services which allows the end user to switch between providers. Although we do not implement it, we believe confidentiality properties can be easily added.

Our system heavily borrows from the Ivy[1] file system. Accordingly, each device stores its update to the file system in a log which is readable by other devices. We achieve this by storing the log on the local disk and synchronizing it with a storage backend, like the Amazon Simple Storage Service (S3). Then each device scans periodically for updates in all the logs. This allows them to expose an up-to-date version of the file system.

We provide availability and partition tolerance to the end user: he should always be able to write to the file system, even though he is not connected to the central backup: although the world is more connected every day, we still cannot rely on a highly available and high capacity Internet connection. Under this assumption, the CAP theorem says we cannot provide consistency.

Inconsistencies are detected with version vectors: each update is stored along with the most recent version from other mount point the current mount point knows about. Hence if two version vectors are not comparable, this means conflicting updates occurred. But we are still able to reconstruct the history of the conflicting updates. This provides us with a weaker consistency property: each device will eventually have the same view on a file. Even though the updates are inconsistent, the mount point is able to reconstruct all the version of the file viewed at any moment on any device and is able to expose this to the end user. This property relies on our storage backend being itself eventually consistent, like Ama-

zon S3: for any update appended by a mount point to its log, all the mount points will eventually see the update.

2 Design

The Diamond file system is represented by a set of logs. Each device writes its changes to the file system to one log. Although a device writes only to one log, it reads from all the logs to learn about the changes from other devices. The logs are stored on an online storage service, accessible from any device connected to the Internet. Each device stores a unique mutable record under a fixed key, which always points to the most recent log record uploaded by a device. This record - called the *log-head* - acts as an entry point to the log. Each log record contains a version vector. This allows Diamond to detect conflicting log records and to order the other ones. Each device maintains a local cache of the filesystem history on the local file system. This is similar to the stackable file system paradigm introduced in NFS[6]. This history is maintained by updating the file system with new log records either created by itself or uploaded to the online storage service by other devices. This allows a device to navigate the current file system without contacting the distant server. This also allows a device to navigate a past version of the file system without rebuilding it from all the previous log records.

2.1 Backend

A simple put/get interface is exposed, provided by the S3 backend.However, this could be easily extended to use other storage providers, DHash/Chord, or a custom protocol.

The log records form the minimum data necessary to rebuild the file system history. These records are stored on a storage backend. In order to save storage space on the backend we could compact multiple successive log records relative to one file into one log record. This would lead to a trade-off between storage space and history granularity. This is left as future work: in the current implementation log records are never deleted.

The storage backend is accessed through a simple put/get interface. put(k, v) stores the log record v under the key k. get(k) retrieves the log record stored under the key k. Log heads are stored under the name log-head-i, where i is the view id. This allows the client to download directly the latest log head for any view. This allows the clients to learn the sequence number of each view, since it is in the log-head. A client can then download all the records, which are store under the name view-v-seq-s, where v is the view id and s the sequence number of the record in the view v.

We currently support only one backend: the Simple Storage Service from Amazon.com, but the simple interface allows additional backends to easily be added. One could think of other online storage services, a local service, or even a distributed service like DHash.

2.2 Log Records

A log record contains the minimum data needed to replay a single modification to the file system. For instance, a Write log record contains the newly written data, but not the new file length. Instead it is computed on the fly when the logs are traversed.

Each log record contains two additional fields: timestamp and version. timestamp, which are only used to determine the atime, mtime and ctime according to the UNIX file system semantics. It cannot be used to designate a past version of the file system: unsynchronized clocks from different devices would lead to inconsistencies. Instead, when a new log record is downloaded from the backend, it is associated with the version vector of each device's highest sequence number. Since the log records are downloaded sequentially, a total order exists on these version vectors. This allows navigation of the file system history with consistency. The version field stores the version vector of the update. This allows detection of conflicting updates and the order the remaining updates.

2.3 Combining Logs

A set of logs is called a view. A view record, in the "root directory" of the storage backend, contains the device number of all the devices participating in the view. This allows access to the *log-head* record of each device and subsequently to all the log records. If the storage backend supports only one view at a time, the view record is stored under the key *view*. Otherwise we expect the device to know which view it should mount.

We use version vectors to detect conflicting log records and order the remaining ones. A version vector is a tuple containing the sequence number of each log in the view at the time the update was made. For instance the version vector $(1, 3)$ indicates the log 1 had sequence number 1, and the log 2 had sequence number 3. We says that two log vectors u and v are comparable if $u < v$ or $v < u$ or $u = v$, where $u < v \Leftrightarrow \nabla i, u_i \leq v_i \wedge \exists i, u_i < v_i$ and $u = v \Leftrightarrow \nabla i, u_i = v_i$. We notice that all the version vectors are different: for each new update, a given device increase its own sequence number, hence even if he has not learned about a new update from another device, its new version vector is strictly superior to the preceding one.

The version vector is an easy way to express the following: when I created this update I knew about the versions referenced in my version vector, thus I do not conflict with these versions. Hence $u < v$ means u and v do not conflict. If u and v are not comparable, the corresponding updates are conflicting. Note that conflicting updates do not necessarily means there is a conflict: the updates may affect different parts of a file system. This case is handled transparently by the local cache.

When a conflict arise, a branch appears in the version history of a file. To remember subsequent updates to a conflicting version still conflicts, we stop updating the sequence number of the other devices in the version vector. This allows us to grow the conflicting branch in the version history. In addition, a user can manually merge two branches by explicitly issuing a merge record, whose version vector is greater than the version vector of all the record in the branches. This is done with a special merge program we provide, which communicates with the file system. When conflicting versions of a file exist, the local branch is mounted. A user can read-only mount the file system version of another user. At the same time, he can specify a previous version of the file system.

2.4 Log semantics

Files and directories are identified by 160-bit i-numbers. Since we allow updates to be performed asynchronously, different devices can assign the same i-number to different files. In order to reduce the risk, we choose i-numbers randomly.

Although file mode is supported, file owner and group is left as future work. Because the backend is assumed to be accessible by everyone, some sort of cryptography would have to be used to enforce security properties. Instead the files belong to the user who runs the file system and to a predefined group *diamond*.

The log is being used in the following way: if Diamond receives a non-updating request it scans the logs backwards either from the beginning or from a point in the past, if specified. If two log records are conflicting, priority is given to the log record that belongs to the current device, or a default one is chosen. This leads to select a version branch, in order to present only one version of the file to the end-user. Diamond stops scanning the log when it has gathered enough information or it reaches the local cache, where it finds the missing information. If Diamond receives an updating request it also appends a new log record to its log.

For instance, if Diamond receives a file creation request, it choses a random new i-number and appends an Inode record with this i-number. It then appends a Link record with the new i-number, the file name and the directory i-number. A summary of Diamond log records is provided in table 1.

Type	Fileds	Meaning
Inode	type (file, directory, symlink), i-number, mode	create new inode
Write	i-number, offset, data	write data into file
Link	i-number, i-number of directory, name	create a directory entry
Unlink	i-number of directory, name	remove a file or directory
SetAttrs	i-number, changed attributes	change file attributes
Merge	none	merge branches

Table 1: Summary of Diamond log records

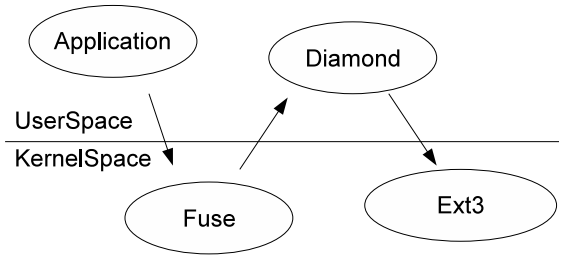

Figure 1: Overview of the Diamond file system: an application emits a system call to the kernel, which is forwarded by Fuse to Diamond, which in turns acces the underlying file system to answer to the request.

3 Implementation

3.1 Overview

Currently, Diamond consist of three programs: a user-level file system server, a synchronization tool and an administrative tool: merge.

The user-level file system server is implemented using fuse as shown on Figure 1. Although this impacts the performance of our system, we believe this is affordable: our value proposition is not about performance, but functionality. Performance si still acceptable for an average-user workload. The server stores the file system data in a unique directory, as log records. As described above, the log records allows for the reconstruction of the data requested by the user.

The synchronization tool uploads newly written log records to the backend storage and downloads the log records uploaded by other clients. In order to limit the network consumption, and minimize the Amazon per request cost, synchronization is only performed every second.

The merge administrative tool allows the end-user to merge the branches of a file. It is simply invoked with the file name as argument, and merge communicates with Diamond by setting a special extended attribute.

No tool for listing the different version of a file, or mounting one of these versions has been developed yet. Now that the foundation is complete, this should be straightforward.

3.2 Underlying File System Structure

Diamond relies on an existing file system to store the log records. This allows the end-user to mount a diamond file system without repartitioning its hard disk, and allowed faster development of Diamond itself.

Each inode of the Diamond file system has its own directory in the underlying file system. We rely on the implementation of the underlying file system to provide a quick lookup access to these directories, since there is one directory per node. In practice, most of the file systems implements lookup in log time.

Log records are stored in the corresponding directory: a WRITE record will be stored iunder the inode of the file, a LINK will be stored iunder the inode of the parent directory. We incorporate the version vector and the view id of each record in its file name. This allows us to discard conflicting records and sort the remaining without reading the actual record.

In addition, there is a special directory named out-

Type	Diamond	Underlying File System
1 MB Read	322ms	192ms
1 MB Write	1,362ms	343ms
Untar	4.0s	1.1
Make	7.2s	5.5s

Table 2: Micro and Macro Benchmarks

going. It contains hard links to all the log records. This directory is used by the synchronization tool to quickly access log records that have not yet been uploaded. Otherwise it would have to scan regularly all the inode directories to find newly added log records.

4 Evaluation

4.1 Benchmarks

The very interest of Diamond is to have performance comparable to a local file system, while still having the advantages of a network file system. Thus, we first compare Diamond to a local file system. To be fair, we compare it to its underlying file system, in this case the linux ext3 file system.

Most of the numbers in table 2 are averages over 10 runs. An exception is the Read on the ext3 file system because it would have been a hit in the cache. Instead, after creating the test file, we copyied a large file to empty the read cache, and proceeded to the test only once. Diamond has currently no read cache, thus the trick is unecessary.

For the two macro benchmark, Untar and Make, we used our own source tree as test data.

What we can notice is that the performances are similar. This validates our approach. The differences are minimal for the make process, because the job is mostly compute intensive. On the other hand, write is deceiving. This is due to our implementation: each time we add a record, we scan the corresponding inode directory to determine what the new version vector should be. Since a write generates a lot of small write records - their size is 4kB on our linux machine, 64kB on our Mac - the operation performs slowly. This could be easily fixed by sorting pointers directly into the log records to represent the branches. Then it would not be necessary to learn about all the relevant version vectors to determine the new one.

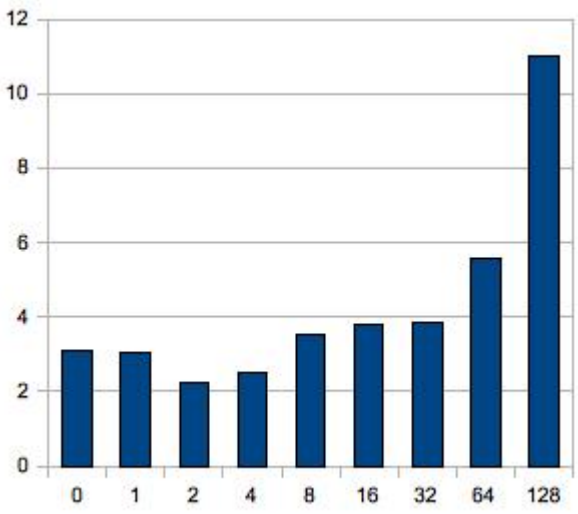

Figure 2: Synchronization time: time for a write record to propagate from one view to another. y is in seconds, x in kB.

4.2 Update propagation in WAN

We next want to know how efficient Diamond be at keeping the different views in sync. To illustrate this, we mount two views on the file system. We measure how much time a write on one view propagates to the other view. This is simply done through the standard file system interface, using a simple python script.

First, the minimum latency seems to be around 3 seconds. Since both of the synchronization tools sync every second, this seems fairly reasonable. Second, the transfer rate seems to be dominant over 32kB. This gives us a modest 13kB/s syncing rate. This is mostly explain by the fact that each log record correspond to a different S3 request. Thus the client identifies itself for each log record. Thus, performance could be largely improved by uploading log records in a group, whenever possible.

5 Related Work

5.1 Network File System

NFS introduced the notion of a network file system to the community. Hence it can be seen as an ancestor to diamond. But many aspects are different: first and foremost NFS allows many users to access a file system on a physical node, whereas Diamond allows one user to acces a file system from many nodes. This difference comes from different constraints: in the 1970's, storage space was sparse which lead to the idea of sharing one big hard drive. Nowadays storage space is cheap but end-users have an increasing numbers of devices. NFS is not partition tolerant, whereas Diamond is. Again, contraints are different: in the 1970's, computers were used almost exclusively in local networks which decrease the risk of partition. Nowadays, we want our laptop or iPhone to still be able to work when no Internet access is available. This leads to another difference: under the availability property, the CAP theorem implies that NFS can be consistent whereas Diamond can only provide an eventually-consistent view of the file system.

5.2 Thin or thick cloud

Cumulus is a backup utility which advocates for thin cloud: minimum functionality should be required from the cloud so that different backends can be implemented. This gives the end-user the freedom to choose his storage space provider.

Like Cumulus, we make the thin cloud assumption: we access the backend through a simple *put/get* interface. But we improve Cumulus in two ways: first, each version of a file can be accessed, not only the version of a file at a given snapshot date. Second, content can be added from multiple nodes at the same time. On the other hand we do not optimize Diamond for a backup use case, as Cumulus does: this is not the goal.

5.3 Security

SUNDR[4] aims at being an improvement over NFS by allowing the server to be totally untrusted. This is done by signing each read or write request including an history summary and uploading it to the backend.

In our current implementation, we do not implement any security features. As future work we could easily implement a self-certifying file system, as is done in Ivy, and enforce confidentiality by encrypting the log records. But we would not try to prevent the backend from equivocating with a SUNDR scheme. Otherwise we would not be able to access the file system in case of a network partition, which is one of our primary goals.

5.4 Consistency

Dynamo[3] and Bayou[2] have two radically different approaches to dealing with inconsistencies. Bayou considers inconsistencies to be a bad thing and thus tries to solve them by merging the data. On the other hand Dynamo considers detecting and exposing inconsistencies to the end-user to be a feature: he can then do a semantic merge.

Our system draws from both approaches: on one hand, diamond merges all the data available to present an eventually-consistent view of the file system history to the end-user, like Bayou does. But unlike Bayou, our merge operation is lossless: all versions of the files are accessible to the end-user, even though only the default version is presented. On the other hand, diamond lets the end-user manually merge different versions of a file, like Dynamo does.

In addition, an optional external merge program could be specified, based on different criteria like the file extension. This is left as future work.

5.5 Ivy

Ivy is the closest system to Diamond. Ivy allows multiple users from multiple nodes to edit a distributed file system. This is done with a similar architecture: the system maintains a log for each participant. Logs records are stored on the DHash layer, which allows it to be distributed.

Unlike Diamond, Ivy does not allow the user to access the full history of the file system, including the different version branches of a file. Ivy only provides access to the most recent conflicting versions. This

makes the file system conflict with the real world: when inconsistencies appear, a full history may be needed by the end-user to reconcile the different versions.

Unlike Diamond, Ivy allows different users, but no security property is provided: implementations is just assumed to behave correctly in preventing a user from reading or modifying a file he has no rights to. In Diamond we only allow one user, which is safe. On the other hand, unlike Ivy, Diamond is not a self-certifying file system. This is left as future work.

Ivy is a distributed file system. Diamond can be one, when used with a backend such as DHash. But this prevents the file system from being eventually consistent: because DHash is not eventually consistent, the *log-head* block used may be different from one device to another, which can lead to an inconsistent view on the log structure, yielding an inconsistent view on the file system. Instead, when using an eventually-consistent backend, Diamond is eventually consistent. This is one of our primary goals.

Conclusion

Diamond introduces a new way to deal with file systems. One is not limited anymore to a specific node, and need not to worry about synchronizing data between nodes. This remarkable propterty is achieved by decentralizing conflicting detection with the use of version vector and allowing the user to resolve the conflicts. The system is fully working and already usable. Performances are even decent, altough they could be improved even more.

References

[1] Thomer M. Gil Athicha Muthitacharoen, Robert Morris and Benjie Chen. Ivy: A read/write peer-to-peer file system. *MIT Laboratory for Computer Science.*

[2] Karin Petersen Alan J. Demers Mike J. Spreitzer Douglas B. Terry, Marvin M. Theimer and Carl H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[3] Madan Jampani Gunavardhan Kakulapati Avinash Lakshman Alex Pilchin Swaminathan Sivasubramanian Peter Vosshall Giuseppe DeCandia, Deniz Hastorun and Werner Vogels. Dynamo: Amazon's highly available key-value store. *Proceedings of the 21st ACM Symposium on Operating Principles (SOSP)*, 2007.

[4] David Mazières Jinyuan Li, Maxwell Krohn and Dennis Shasha. Secure untrusted data repository (sundr). *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[5] Stefan Savage Michael Vrable and Geoffrey M. Voelker. Cumulus: Filesystem backup to the cloud. *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[6] Steve Kleiman Dan Walsh Russel Sandberg, David Goldberg and Bob Lyon. Design and implementation of the sun network file system. *Proceedings of the 7th USENIX Annual Technical Conference*, 1985.