Fluidbackup: A Distributed Peer-to-Peer Backup System

May 8, 2015
6.824 Final Project

Favyen Bastani (fbastani@mit.edu), Benjamin Chan (benchan@mit.edu)

Introduction

As more of the world moves online, peer to peer solutions for a diversity of computational tasks become more relevant. Information is increasingly distributed across devices and shared across machines, individuals, and organizations. A large number of clients are increasingly relying on a small number of centralized service providers for technical solutions. This centralization inherently introduces several risks, as service providers are attacked, go offline, or even out of business, and does not scale intuitively with the scale of the network.

Ideal systems would take advantage of the diversity of nodes in a network, in an attempt to distribute computational responsibility in a robust way. The backup of user data, in particular, is especially important as documents move onto hard drives and across devices. Hence, we present Fluidbackup, a distributed peer-to-peer backup system that seeks to take advantage of network peers to create a low-cost, robust backup. This report describes the high-level design of Fluidbackup, as implemented in Golang by the authors.

Problem Statement and Context

Two observations make Fluidbackup especially relevant:

1) Users often have large amounts of bandwidth and disk space free on personal devices such as desktops, laptops, tablets, and smartphones. At the very least, users have space that they would be willing to trade for a secure, reliable backup.
2) Users spend increasingly large amounts of time connected to the internet. Transferral of large amounts of data over an extended period of time becomes feasible as bandwidth becomes cheaper.

Fluidbackup takes advantage of these observations to provide a backup system at low cost, with existing space. Ideally, a client would be able to simply run the software, and immediately have a functional, trusted, backup system.

Design and Implementation

Overview

Fluidbackup comprises multiple peers working in unison to backup a set of data. Each peer is responsible for storing a portion of this data, as established through a set of peer agreements. Each peer also acts independently and selfishly, to maintain a level of fairness and equality in the network.

Each file is broken into blocks, which are then distributed independently. The block is encrypted using a symmetric key (which is then encrypted by the private key and stored with the metadata), and divided into chunks with client-side erasure coding. Fault-tolerance is achieved because Fluidbackup only needs to recover a portion of the chunks to reconstruct the original block. Each chunk is distributed onto a separate trusted peer that has free space under the active storage agreement.

The client also maintains metadata for each replicated file, with a list of peers, and a set of keys to encrypt and decrypt data, adding a level of security. In addition, the client creates storage agreements with peers, and runs an algorithm to distribute its backup based on those agreements. Finally, the client stores blocks on disk from other peers performing their own respective backups.

Peer Structure

The implementation of Fluidbackup involves 6 components: a parent *fluidbackup* module, a *filestore* module responsible for abstracting away the storage of files, a *blockstore* module for storing blocks; a *PeerList* tracking all known peers, *peer handlers* for simulating remote peers, and a *protocol* module for handling network communication and agreements.

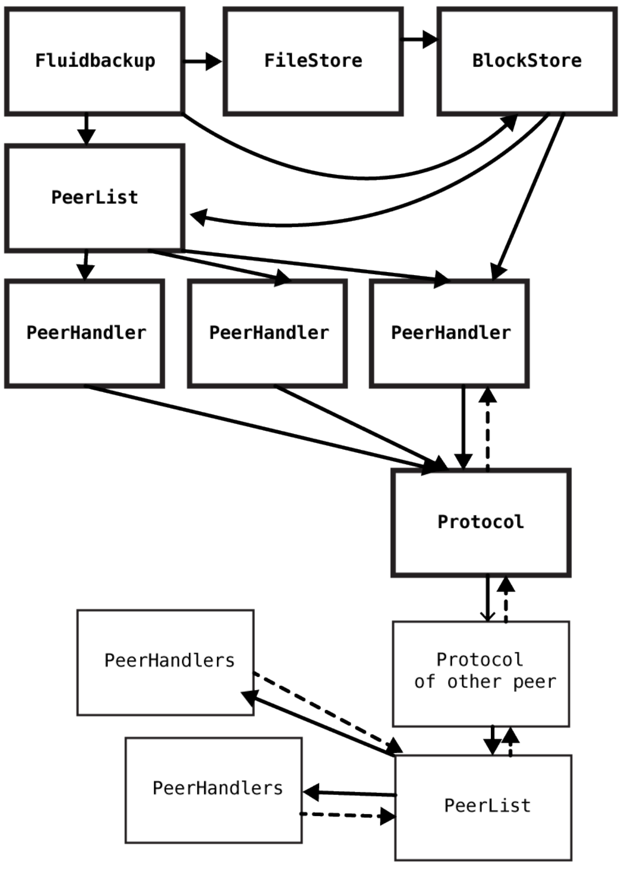

Figure 1. A diagram of the modules comprising Fluidbackup

The diagram above shows the flow of a backup. Each module is discussed in detail:

- *filestore*: A new file is registered with the filestore, which handles all file operations, writing files to disk, and so forth. The filestore then divides the file into *B* blocks, and passes these blocks to *blockstore* for backup in the distributed system.
- *blockstore*: Each registered block is divided into redundant shards, with an erasure coding scheme. In the authors' implementation, the go-erasure library[1] is used to accomplish this. Two parameters are set for each block: *N*, the number of shards, and *K*, the minimum number of shards needed to recover a block. The default values are 12 and 8 respectively. Each shard is then assigned a peer, according to *peerList*, and propogated to the peer, through its *peer handler*.
- *peerList:* The peer list module is responsible for keeping track of all known peers, and their trust ratings. In order to boost the reliability of the system, Fluidbackup keeps a trust score for each peer, which is elaborated on below. The PeerList helps the block store select available peers according to a trustworthy distribution, and also forwards messages to peers back from the protocol.
- *peerHandler*: The peer handler represents a remote peer, and keeps track of the local peer's expectation of the remote peer state. It also includes a set of handlers that are called in response to remote events, forwarded by the protocol and peerlist.
- *Protocol*: This module is the interface between a peer and the rest of the network. It implements a set of RPC calls that are integral to the system. They are:
  - *ProposeAgreement*: Establish an agreement between the local peer and the remote peer to store each other's data for a day. See the "Peer storage agreements" section.
  - *StoreShard:* Store a shard on another peer. Blockstore uses this to directly attempt storage.
  - *DeleteShard*: Delete a stored shard from another peer.
  - *RetrieveShard:* Retrieve a stored shard.
  - *ShareNewPeer*: Ask another peer to share its list of peers. See "Peer Discovery."

Special consideration is given to Fluidbackup peer agreements, peer discovery, and the stored metadata.

Peer storage agreements

To store data on a peer, the client, of course, must reciprocate. Each client will make agreements with specific peers to store each other's blocks. First, a client will allocate an amount of data it is willing to store from the other peer, then send the peer a request for storage of some other amount of data. Upon agreement, both peers will have promises for data storage. Agreements and promises transcend shutdowns and outages, and are verified occasionally in the background.

Each peer, upon receiving a block with a corresponding storage promise, will store it for 1 days. Each client, upon uploading a block, will perform verification. Verification is done by comparing hashes with stored information in metadata. If this verification fails, the client is allowed to renege its "space" promise subject to tolerance for outages.

[1] github.com/somethingnew2-0/go-erasure

A major issue with peer-to-peer backup is that peers may frequently go offline. We assume that most peers will be active for at least one interval each day (different time intervals can also be used). Then, peers will only terminate agreements if verification fails repeatedly for an extended duration greater than the one-day threshold. Additionally, *N* and *K* parameters can be tuned to get guarantees on failure probability based on observations and assumptions on peer reliability (e.g. how long we have observed the peer and how frequently the peer is offline). The current default parameters are chosen to be compatible with a sandboxed environment.

Peers have an incentive to store data for an entire day, even when the related peer has gone offline. Were the related peer to come back online (which is likely), the storage peer would have wasted valuable network bandwidth replicated its data elsewhere. Additionally, sticking with the agreement preserves their trust score, which is important in being able to obtain data space for itself on other peers.

Trustworthiness of Peers

Peers also maintain a trust score for each known remote peer. With each successful transaction (a shard storage, retrieval, or verification), the trust score is increased geometrically, and with each failure, it is reduced. Shards (across blocks) are then distributed to match the distribution of trust, to create a more reliable system.

This also gives an incentive for peers to maintain uptime: namely, it gets as much out of the network as it puts in.

The trust map is maintained by the *Peer List*.

Peer Discovery

Fluidbackup also implements a peer discovery system, such that, upon joining, a peer needs to know only a handful of peers in order to build a useful network of peers. It does this by allowing peers to ask each other for more peers, through the *ShareNewPeers* RPC.

Peers have an incentive to share valuable peer information because the act of sharing also increases its own trust score. *PeerList* maintains a map of peers related to each other (namely, which peers referred which other peers). If the trust score of a peer changes, and goes up, for instance, the referring peer's own trust score will also increase by a factor.

Metadata

The metadata for each file consists of a symmetric key (stored encrypted by the global private key), a peer identity describing who is storing each of the *N* erasure coding chunks, and *N* hashes of the chunk data to verify that the peer is actually storing our data. This data is currently stored locally; further work could be done to distribute this data as well.

Analysis

The authors implemented Fluidbackup in Golang, as a proof of concept for a peer to peer distributed backup system. While this version of Fluidbackup does not provide significant improvements in terms of backup speed, and face-value reliability, compared to a replicated, centralized service, Fluidbackup is valuable in the fact that the backup is *distributed*.

Initial tests reveal that Fluidbackup is practical, and able to store and retrieve files despite the outage of a local peer, with good reliability. The erasure coding scheme is able to tolerate most faults, trading off the useful-space to free-space ratio. This, combined with the practical elegance of a distributed peer-to-peer system in the context of todays systems, suggests that more work on peer to peer backup systems should be done, with the goal of a useful product.

Despite that, certain (inherent) design decisions call to question the practicality of such a design. The authors attempt to address those here:

Fluidbackup requires peers to have high uptime in order to effectively participate in the network. This is problematic for many home users who may have desktop and laptop computers that are kept offline when not in use (e.g. to decrease the chance of intrusion by the NSA); when peers detect that blocks are unavailable for such long durations, they will terminate storage agreements.

Instead of having users fully participate in the network, we propose a federated system where users interact with the network in combination with storage providers. So, a user might purchase a storage plan from any provider, and have that provider serve shard retrieval requests from peers when the user turns off his or her computer. Of course, we do not want users to be fully dependent on a storage provider or we would have the same issues as existing cloud backup services. Our system reduces this dependency by:

a) Using open protocols and free software so that users do not have to change their system setup when migrating storage providers, and can have stronger guarantees about the encryption algorithms used. This also allows anyone to set up their own storage provider -- if a user has a computer with high uptime, then they can participate directly in the network rather than interacting with a third party.

b) Implementing a trust system that ensures user data remains available for a short duration after a storage provider goes offline.

In the federated design, the user still runs a Fluidbackup node. However, when making storage agreements with peers, the user directs the peer to store blocks on the user's storage provider rather than directly on the user: before proposing a storage agreement, the user will contact the provider and reserve an amount of space for the peer; then the user sends the peer the storage agreement request, with an extra field indicating where the peer should send shard storage messages. If the proposal is rejected, then the user notifies the storage provider to remove the reservation. The storage provider will only accept shard storage messages that pertain to reservations. The user can terminate agreements by similarly forwarding the termination event as a reservation removal message to the storage provider.

In this way, the shards that a user should be storing are still being served when the user is offline, as long as the storage provider has reasonable uptime. If the storage provider disappears, then the user still has some time to retrieve data.

One potential problem is that we are still placing the responsibility of re-replicating shards when peers go offline on the user. This is acceptable as long as the user is online once every few days: peers are expected to have high uptime over long durations, so it is unlikely that (N-K) peers will fail in the same interval. Nevertheless, a scheme where the storage provider handles re-replication is always possible; the storage provider would need to retrieve K shards for each block that is under the erasure coding goal, decode to obtain the encrypted block contents, and then send shard storage requests as appropriate; the user would need to be notified about these modifications when the user comes back online.

Conclusion

Fluidbackup shows that a peer-to-peer distributed backup system is feasible and practical, and is a worthy expenditure of engineering time. Using erasure coding, combined with a peer trust map provides good fault tolerance and system reliability, which is especially important given the nature and trust given to a backup system. In addition, a simple peer discovery protocol makes joining this network easy for clients and users alike. It is the authors hope that this work is useful, and provides insight into the design of distributed, peer-to-peer reliable storage systems.