Master's Thesis Design Document (Terse Version)

Michael Murphy

October 26, 2016

1 Idea

A distributed data store that makes location and availability explicit, rather than trying to hide or abstract them away.

The system will be designed for high-latency or intermittent links (minutes to weeks), by assuming that network partitions are the norm. It will make availability explicit by providing location and latency hints and estimates that client applications can use to schedule processing. It will allow local writes at all times, and will handle inconsistency by storing competing versions of data until a client that is familiar with the data model can merge them.

The system should work with a wide range of file sizes (kilobytes to gigabytes), focusing on media storage and synchronization (binary blobs, 1 MiB to 4 GiB in size).

2 Architecture

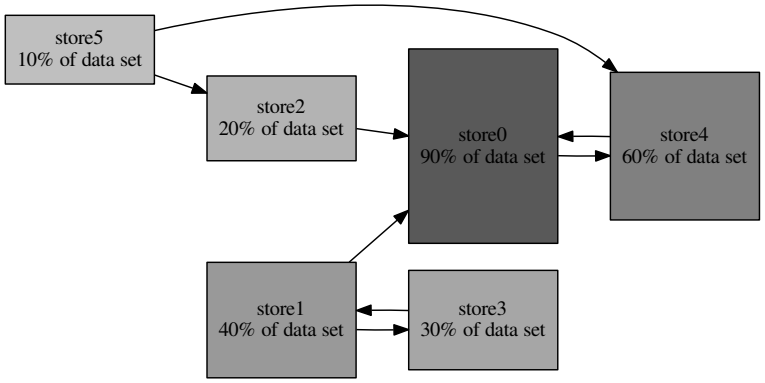
Data will be distributed across several *stores*, which will each hold a portion of the data set. Stores can be diverse, from mobile devices to storage clusters, and each will choose which data to hold according to its capacity and processing needs.

The data store network will be decentralized and unstructured (Figure 1). Stores can be connected to neighbors in an ad-hoc manner, with a user- or application-defined topography. The number of neighbors will be limited by the capabilities of the store's device. We are aiming for one neighbor to tens, or perhaps hundreds of neighbors.

(Possible extension: a gossip protocol to relay information about stores that are not direct neighbors.)

Each store will be autonomous and able to perform processing on any data that it has a local copy of. Data not currently available locally can be listed and requested. The listing will give hints about the estimated time of availability,

Figure 1: Stores in an ad-hoc network



calculated based on latency, bandwidth, and file size. This metadata will naturally fall out of date between syncs with neighbors. The age of that data can be part of the hint.

We are envisioning a filesystem with an `ls` command that lists these hints as part of its output:

```
-rw-r--r-- 1 user user  121306 Oct 21 18:28 local   filex
-rw-r--r-- 1 user user   25475 Oct 21 17:52 100ms   filey
-rw-r--r-- 1 user user   32031 Oct 21 17:52 20min   filez
-rw-r--r-- 1 user user   74968 Oct 18 17:12 missing filexx
-rw-r--r-- 1 user user   83977 Sep 22 21:23 unknown fileyy
```

The system should detect storage errors and never lose data inadvertently. Local data stores should detect corruption, and replication between stores should provide data safety. However, data can be explicitly removed from the data set, or it can be removed from the local store, relying on remote stores to keep replicas.

3 Design

The data set will be a normal hierarchical filesystem, and it will be presented as a normal filesystem. We do not want to reinvent the filesystem, and we do not to have to modify local applications to work with the data.

The storage will be modeled on version control systems, specifically Git and it's directed acyclic graph data structure. There will be a content-addressable blob store to de-duplicate and store data, and a working tree where the files can be read and written by normal applications.

(Possible optimization: the working directory can be a virtual filesystem that is a view into the object database.)

Files over a certain threshold will be broken into smaller chunks in the blob store.

Like a version control system, history of the data set will be kept, and the history can diverge into branches that can be merged later. Indeed, like Git, each individual store is also a branch. But unlike Git, not all blobs are required to be present on all systems. Some blobs may not be available on a given store, and history may be deliberately pruned to save space. Algorithms and client applications must work with the blobs they have locally and the hints about the availability of remote blobs.

4 Implementation