MoData - A Distributed Hash Table Based Filestore for Everyone

Merritt Boyd, Russell Cohen, Joseph Lynch

May 10, 2013

1 Problem

Persistent, reliable, online storage solutions such as Dropbox, Box, and Google Drive are all for-pay services controlled by large multinational companies. Many users do not wish to pay for these services and do not want their data controlled by large companies. At the same time, most computer users have excess disk storage and network bandwidth. It seems that a system where people share their bandwidth or storage in exchange for a free, peer-hosted cloud storage alternative is very viable. We strive to create a system to allow access to your files wherever you go, without relying on a central party required to host files.

The storage solution should prevent the users files from being compromised either by malicious corruption and make best possible efforts to prevent loss.

2 Solution

We present a decentralized peer-to-peer key-value storage system targeted towards nodes with low availability. The software is intended to be run on personal computers, especially laptops, which may frequently disconnected from the network and possibly powered off. By heavily replicating user data across the cluster, we aim to maintain availability of user data from any device even in the absence of much of the peak size of the network.

Our system stores files indexed by a key which is simply a hash of the file's contents. As we are not optimizing for peer-to-peer file sharing, we provide no in-system mechanism for listing a user's files – this metadata must be maintained and transferred by the user out-of-band, for instance on a USB flash drive. This provides extra security by requiring a token not stored in the cloud to access files.

3 Implementation

The project is split into two main components, the server and the client. The server is a straightforward RESTful API that implements a Kademlia based DHT, and the client is a collection of python scripts that provide an easy to use file interface as well as additional redundancy.

3.1 Server - Kademlia DHT

The server implements the standard Kademlia Distributed Hash Table algorithm [1]. Kademlia can automatically handle nodes joining, leaving and publishing key/value pairs to the network. The algorithm is fundamentally peer to peer, and nodes join by bootstrapping in on old nodes. Each node is given an identifier, and then consistent hashing based on the XOR metric is used to evenly distribute keys among available nodes. All keys and node-ids are 160 bit identifiers. The "closeness" metric of XOR is used to define that a key K's closeness to a node N is defined as $K \oplus N$. Because XOR satisfies the triangle identity, this guarantees that there is a globally correct set of closest nodes for any given key.

All distributed operations must first do a distributed find node, which collects nodes nearby to the key by calling primitive find-node on hosts that a particular peer knows about. By chaining these requests together, we can build a list of nearest nodes in O(log(n)) time, where n is the size of the network. After finding the list of nearest nodes, stores then send primitive store operations to those nodes, and find-values send primitive find-value calls. All changes are persisted to disk.

Our server exposes the following REST API to meet the Kademlia specification.

Endpoint	**Type**	**Action**
/find-node/{node-id}	GET	Returns locally known nodes near the node-id
/find-value/{key}	GET	Returns the locally known value for a key
/store?key={key}&data={data}	POST	Locally stores a value for a key
/ping	GET	Health check, also updates contacts
/distributed/find-node/{node-id}	GET	Returns globally known nodes near the node-id
/distributed/find-value/{key}	GET	Returns the globally known value for a key
/distributed/store?key={key}&data={data}	POST	Stores the value for the key on nearby nodes

3.2 Client - Erasure Coding File Transfer

To help cope with low availability, files are erasure encoded. This gives us a knob which we can turn to set the percentage of chunks that the system can loose, but still recover the file. Our client uses using Reed-Solomon codes, an optimal erasure encoding scheme. It produces a number of chunks. From those chunks it creates a metadata file containing the hash for each chunk. A hash is enough information to retrieve the data from the DHT. The client then attempts to get sufficient chunks to decode the file, decoding the resulting chunks client-side.

4 Challenges and Analysis

The most challenging part of this project was getting a functional kademlia based DHT and building additional availability guarantees into the system.

Because we are assuming that the target consumer will be a laptop, we chose usage analysis of wifi networks as the starting point for availability analysis. In particular, we assume that most wifi users behave in similar fashions to that of the Mountain View wide wifi provided by Google [2]. Using these assumptions, we can do a quick back of the envelope calculation for data loss. Assuming that we republish keys every hour, we only have to care about losing data in a given hourly period, since at the end of that period we will re-replicate the key. We can choose the replication factor of Kademlia to trade off bandwidth with availability, and a commonly chosen value is 20. Due to consistent hashing, we can say that node failures are independent, and that the probability of a given node failure is equal to the probability of failure conditioned on the node type. Node type is split into long lived, which are on all day, medium lived which are on for between 100 and 1000 minutes, and short lived which are on for less than an hour.

$$\begin{gathered} Pr(lose\_key) = Pr(k\_nodes\_fail) = Pr(single\_failure)^{20} \\ Pr(single\_failure) = Pr(failure|long\_lived) * Pr(long\_lived) + \\ Pr(failure|medium\_lived) * Pr(medium\_lived) + \\ Pr(failure|short\_lived) * Pr(short\_lived) \end{gathered} \tag{1}$$

We can estimate the duration probabilities from the Wifi paper Figure 8 [2] and to be as conservative as possible we say that $Pr(failure) = 1$ for all types except long lived, for which $Pr(failure) = 0$, this yields $Pr(single\_failure) = .8$, and $Pr(lose\_key) = 0.01$. Due to erasure coding, the probability of losing a file with N chunks is $Pr(lose\_key)^{\frac{N}{2}}$. This means that to lose a 1 Megabyte file, one must lose 125 keys (assuming a 4kb chunk size). Losing access to a file therefore has probability $Pr(data\_loss) \approx 5 * 10^{-243} \approx 0$. Even if access is lost due to massive correlated failure, it will be restored once nodes return to the network.

5 Bibliography

References

[1] P. Maymounkov and D. Mazieres, “Kademlia: A peer-to-peer information system based on the xor metric,” in *Peer-to-Peer Systems.* Springer, 2002, pp. 53–65.

[2] M. Afanasyev, T. Chen, G. M. Voelker, and A. C. Snoeren, “Usage patterns in an urban wifi network,” *Networking, IEEE/ACM Transactions on*, vol. 18, no. 5, pp. 1359–1372, 2010.