PuddleStore Design Document

Client API Structure

We plan on an object-oriented API structure centered around an Inode superclass with two subclasses, File and Directory. These three will have the following core API structure:

- Inode
  - GetLocation() string
  - GetParent() Directory
  - IsDirectory() bool
  - IsFile() bool
  - Remove() error
- Directory
  - List() []Inode, error
  - CreateFile(name string) File, error
  - CreateDirectory(name string) Directory, error
- File
  - Open() error
  - Close() error
  - Write(position uint64, data []byte) error
  - Read(position uint64) []byte, error

Client API Testing Application

We plan on starting with a basic CLI client application that uses our PuddleStore client library (the interface is described above) to navigate and modify the filesystem. We plan to use the ishell library the TA's used previously.

In particular, we think the best user interface is a basic bash-like shell with the following commands (mapped approximately to the API interface):

- `ls` - list the contents of the current directory
- `cd <dir>` - change the current directory to the subdirectory of the current directory specified by `<dir>`
- `mkdir <dir>` - create a directory in the current directory with the name `<dir>`
- `touch <file>` - create a file in the current directory with the name `<file>`
- `rm <inode>` - removes the given file or directory in the current directory with the name `<inode>`

- `cat <file>` - print the contents of the file in the current directory specified by `<file>`
- `edit <file>` - open the file in the current directory specified by `<file>` in a local text editor for editing, and then write that file to PuddleStore when the editor is closed.

If we have extra time after both our initial implementation and our A-level features, we may modify an existing file system web interface to work with our client application, possibly by creating a client library in another (web-enabled) language such as Python or Javascript.

A-Level Features

We plan to implement:

- Zookeeper as our membership and configuration server, storing information to help the Raft cluster, Tapestry cluster, and any configuration information for both. ZooKeeper stores information as a set of nodes, and we would need to create such a node for each server in our application.
  This would use the Go language binding. ZooKeeper does a number of useful things, such as providing us with information on which servers can be connected to and which are down (using ephemeral nodes).
- Tapestry publishing path caching; cache pointers to the object replica at all nodes between the replica and the root node, i.e. on all paths that the publishing heartbeat travels.
- Combination of erasure codes (which give excellent reliability) and republishing, involving host server's participation. Blocks are erasure-coded and the resulting fragments are distributed uniformly throughout the system using Tapestry. To reconstruct a block at some future time, a
  host simply uses Tapestry to discover a sufficient number of fragments and then performs the decoding process. Since this is time-consuming, when a host discovers that a file has been recovered via fragments, it republishes it.

In-Depth Testing

We plan to test our system beyond the client API by combining our previous testing frameworks for Tapestry and Raft such that we can spawn a complete PuddleStore cluster on a single machine, and then utilize testing helpers to facilitate specific system tests as well as oracle tests. We may also wish to create an additional testing framework for interacting with our client interface and with the membership server.

This will involve using our RaftCluster and Tapestry objects that store the state of our Raft and Tapestry clusters and provide methods to test them. We'll likely need to create a new PuddleStoreCluster that comprises both and any other testing methods, and then use that to write through system tests, particularly at the intersection of failures in both systems. We can re-use and augment many of our Raft and Tapestry tests to test those components individually.