Consensus and Replication: An Exploration in Distributed Systems

Puddlestore is a distributed, networked filesystem involving use of the Raft consensus protocol[1] and the Tapestry distributed object location and routing (DOLR) layer[2] to replicate objects and come to agreement about the state of the network. The idea was first proposed and implemented at UC Berkeley as "Oceanstore"[3] but my implementation in Go (undertaken with Evan Fuller as part of CS138) implements a subset of Oceanstore, while adding hash salting, publishing path caching, and file-level snapshots, generated automatically on writes. Users can interact with the network with a custom shell after starting the requisite servers and connecting them to each other.

At a high level, the complete program involves 1) a membership server, 2) individual Raft nodes for storing current version data, 3) individual Tapestry nodes for storing actual data and file blocks, and 4) a client CLI for interacting with the system through a shell. At a minimum, the system needs one membership server, one Raft node, and one Tapestry node to be ready for use. To interact with the filesystem, you also need a client. As many Raft and Tapestry nodes as wanted can be connected, with the membership server handling multiplexing of requests to different nodes and replication across the system. Files and directories are represented as an `inode` struct, identified by unique ids (AGUIDs). Directories have references to other files, and files have references to byte arrays of data called data blocks (`dblock`). The current version of a dblock or inode is identified by a random version id called a VGUID. Raft stores mappings from AGUIDs to VGUIDs, and Tapestry stores mappings from VGUIDs to the associated dblock or inode.

1) The membership server starts an RPC server to handle adding raft and tapestry nodes. It keeps a master list of all nodes that have attempted to connect to the system through the server, and is the master source for connections. All nodes must contact membership in order to get the address of another Raft or Tapestry node to connect to when first starting up or, as a client, if their current node is unreachable. Membership also starts up one Raft node and one Tapestry node when it first initializes. The server also functions as a light-weight CLI to query connected nodes.
2) Raft nodes store mappings from AGUIDs to VGUIDs, mapping the absolute identifier for a file to its current version. Raft functions on consensus, so all nodes necessarily agree

[1] In Search of an Understandable Consensus Algorithm (Extended Version)

[2] Tapestry: A Resilient Global-scale Overlay for Service Deployment

[3] The Oceanstore Project

about the current version of a given file. Exactly `N` Raft nodes must be connected, no more and no less, where `N` is a configurable constant to be changed at build time.

3) Tapestry nodes store the actual inodes and data blocks for file data. Based on the hash of the VGUID, the requester is routed to a root node that stores the location of the publisher, which in turn stores the actual VGUID to datum mapping. Hash salting ensures that the publisher of a particular datum is accessible from more than one node, while publishing path caching allows short circuiting (hence quicker lookups) on finding who's publishing a given mapping. Together, those two additions are effective at speeding up requests for failed nodes and simplifying long lookups chains.
4) The client implements our API as a CLI (shell) that a user can interact with using standard commands (ls, cd, rm, mkdir, touch, stat, diff, and the like). Multiple clients can interact with the filesystem at once, though concurrent changes may be lost. The underlying RPC requests determine reliability and speed, so all commands are synchronous.

Generally speaking, the system is fault tolerant if requests time out or are lost along the network. At worst, a client may have to retry a command. However, Puddlestore is fault intolerant when nodes go down. If a Tapestry node goes down, all data published by that node is irrevocably lost. If a Raft node goes down and a majority of the other nodes are still connected to each other, no mappings are lost and the majority can still function. However, if no majority of nodes can connect to each other, the system cannot make progress.

Additional Features

One major addition that deviates from Oceanstore is hash salting. Instead of storing which Tapestry node is publishing a given VGUID to datum mapping on one root node, the VGUID is hashed to a configurable number of different hashes and stored on that number of roots. If any of those roots go down, a different Tapestry node can be contacted to get the publisher, assuming the downed node is not also the publisher. This helps stability when nodes go down. Additionally, Puddlestore implements publishing path caching. Instead of storing the publisher of a hash at only one node, it's stored at every single node along a chain of nodes routing to the root from the publisher itself. If you happen to contact any of those nodes on your way to the root, you can short circuit the rest of the chain and jump immediately to the publisher. This moderately reduces average lookup times by about 20% in our default system configuration.

Another interesting addition to Puddlestore is file-level snapshots. When writing to a file, any modifications create a copy of only those particular data blocks that change, and make a new version of the file automatically. The underlying representation is as sparse as possible as a result, but we get the convenience of being able to access and read any version of a file at will. Of course, performance degrades as more writes occur, as none of the old versions are ever deleted. It offers convenience for the user, but does hurt amortized lookup times on heavily-trafficked systems. In our default configuration, we never created enough files to hurt lookup times. Through the API, the user can read from any version, write to the current version, and revert to any old version. Similar to the git branching model, we create a new version for a revert operation, ensuring no version data is ever lost.

Data Structures

As a unique identifier, AGUIDs appear in the form "`XXX-filename`", for a fixed-length random prepended string. A VGUID is simply the fixed-length random part of the string. AGUIDs don't change for the lifetime of a datum, whereas VGUIDs are newly generated for each new block/file. Data blocks (`dblocks`) are simply fixed-size slices of bytes (`[BLOCK_SIZE]byte`) with an associated VGUID. The struct for an `inode` is below. `Id` is an AGUID, `Name` is the current filename, `Ptype` is either `FILE` or `DIRECTORY`, `Size` is the current file size, `Blocks` are a fixed-size slice of data block VGUIDs and associated filename pairs, and `Version` is the current version of the file. The size of blocks and the number of blocks per file can be configured at build time.

```
type Inode struct {
   Id      Guid
   Name    string
   Ptype   PuddleType
   Size    uint64
   Blocks  [NUM_BLOCKS]namedGuid
   Version uint64
}
```

API and Client Interaction

The API for Puddlestore assumes many connected clients, manipulating files and directories on the fly. As such, every API call except `Open()` and `Create()` take in an AGUID to work on, versus the filename. Once created or obtained via a filename, a file can be safely referred to by its AGUID throughout a chain of calls. Of course, another client can delete the file as it's being used by a particular client, but the first client's call will return a clear error in that case. Additionally, `Open()` and `Create()` take absolute filepaths, relative to the root directory. These are all the API calls.

- `Open(name string, version uint64) (Guid, error)`
  Opens the file/directory `name` of version `version`, returning its AGUID and any error.
- `Create(name string, ptype PuddleType, parent Guid) (Guid, error)`
  Creates an inode of name `name` in directory `parent` of type `ptype`, returning the new AGUID and any error.
- `Revert(file Guid, version uint64) error`
  If it exists, reverts file with AGUID `file` to version `version`, otherwise erroring.
- `Rename(file Guid, parent Guid, newName string) error`
  Renames the file/directory identified by the AGUID `file` in directory `parent` to `newName`, erroring if anything fails.
- `Read(file Guid, pos uint64, toRead uint64) ([]byte, error)`
  Reads `toRead` bytes from file `file` at offset from start `pos`, erroring on bad file or offset beyond bounds of the file.
- `Write(file Guid, pos uint64, toWrite []byte) (uint64, error)`
  Writes `toWrite` range of bytes into file `file` at offset `pos`, erroring on bad file or offset beyond bounds of file.
- `List(dir Guid) ([]string, error)`
  Lists the contents of directory `dir`, returning the data as a slice of strings where each string contains some datum.
- `Stat(file Guid) ([]string, error)`
  Stat shows information about file `file`, based on the current inode contents like filesize, name, etc. Called on a directory, it shows the number of files in it currently.
- `Remove(file Guid, dir Guid) error`
  Deletes file or directory `file` from parent directory `dir`, erroring on absence of the file or other problem modifying the parent.