Distributed Systems and Algorithms

Project 2

Theo Browne and Paul Revere

General Overview

To demonstrate our understanding of Zookeeper and similar distributed systems, we have built a distributed system in Python in the style of Zookeeper. A given number of servers are running at any time with identical file systems. Changes to the filesystem are requested by a client, forwarded from the server to the "leader", and the "leader" approves the changes and broadcasts them to every server. The servers are "followers", and the lead server has a separate "leader" thread. This "leader" thread runs alongside the follower thread on the lead server, so changes approved by "leader" are sent to every follower, even itself.

When a "transaction_request" is made by a follower to the leader, the leader sends out a "transaction_proposal" to all followers. If the majority of the servers reply with a "transaction_acknowledge", the transaction is sent to every follower as a "transaction_commit", completing a two-phase commit.

When a "commit_change" message is received by a "follower", the change is committed to the file system and written to the hisory (also worth noting that the history is logged as a local file "server_(server_num).history").

Upon coming online, a given server immediately launches an election. The server will attempt to connect to every address in the config, storing the sockets that are successfully connected. For each successful connection, an "election" message is sent. Once a reply

is received from every "election" message (if the server fails after receiving the message, it will reply with an empty string), it knows the election is over. Any server with an id higher than the server that initialized the election will start an election of it's own. The first server to not receive a "higher_id" in it's election_responses becomes leader.

When a server goes down, anything connected to it will receive a blank string upon an attempt to read. When a disconnection occurs, the socket it occured on is removed from the list of active sockets. If the disconnection was the leader, a new election is held.

If a new server comes online when a file system is already established, the election winner will likely be the server that was already leading before the new server came online. When the election is completed, every non-leader node wipes it's history and file system. When a new leader is initialized, it first sends a 'leader' message to let clients know that it is a leader, and follows it up with all of the 'commit' messages in it's history, bringing all followers up to speed with the latest state of the file system.

Network configuration

To run the server, the following command is run:

```
python server.py (server_number) (path_to_config) [Optional: path_
```

The config file is a list of server numbers, IPs, and ports to connect to. The connection information corresponding to the argument given in server_number is assigned to the sockets in that server's listeners.

There are 3 ports for each IP. One for the client, one for server-to-server connections, and one for Leader-to-Follower connections. This allows us to reliably know where new messages are coming from.

To detect the status of a given socket, we use the C library Select. Select takes a list of sockets and returns which ones are ready to be read, written to, or have errored out. The "errored out" functionality of Select is unreliable at best as any socket that disconnects

without a proper "shutdown" is readable (but only sends a blank string).

If the path_to_history argument is provided, the server assumes it is rebuilding from a logged history before reconnecting to the network. It builds it's filesystem up based on the provided history before initiating an election.

The client connects to any given server through it's client_listener port. It only connects temporarily while sending a message, and only maintains the connection until a reply is received. This allows for a theoretical "unlimited" amount of clients connected.