Distributed Systems and Algorithms
CSCI - 4510/6510, Fall 2017
Project 2 - Twitter Implementation with Paxos

Sida Chen, Yitong Wu

Abstract

For this project, we have implemented a distributed system similar to Twitter. Each site runs the same copy of our software, and are able to send tweets to other sites in a distributed systems solution.

Technology

- Python 2.7

Installation

1. Clone the repository.
2. Rename config-sample.json to config.json
3. In config.json, under "sites", update each site's "name" as a unique site name, "addr" as site's ip address, and "port" as site's port.

Running

1. Use the following command (site id starts with 0):
   ```
   python main.py <site id (node number)>
   ```
2. Use tweet/block/unblock/view for operations. In case of an incorrect command, the program will prompt the user the correct usage.

Included Files

- **client.py** - controller of a single server. The client acts as a controller that interacts with its own server, and provide data output for the user as a viewer.
- **server.py** - API implementation of sending and receiving messages between TCP sockets.
- **message.py** - API implementation of message wrapper.
- **tweeter.py** - API implementation of server-to-server communication manager.
- **paxos.py, model.py** - paxos algorithm implementation of tweet sending.
- **reliablestorage.py** - handles the reliable storage file creation and reading.
- **config.py** - helper files to for configuration parsing.
- **main.py** - The main class that starts both the server and the client.

Implementation

Upon running main.py, the program will separate itself to server.py and client.py. The server is the module, while the client is the controller and viewer. The node id is determined by its first argument, and the server will initialize according to node id provided.
For client-to-server communications, the client can control the server by sending a TCP request, example format as follows:

```
{"head": "tweet", "body": "hello"}
```

Server will respond with a status code indicating whether the operation is successful, along with additional data such as tweet list, if necessary:

```
{"stat": 200, "body": "<extra data>"}
```

For server-to-server communications, the server will respond only if necessary. For “propose” and “accept” operations, other servers will respond with “promise” or “ack”. For “commit” operations, other servers will not respond. Example server-to-server data format as follows:

```
{"head": "recv", "body": "<extra data>"}
```

When a site i sends a tweet/block/unblock operation

- i “propose” with other servers. Other servers will send back “promise” message. Majority of successful nodes required to proceed. Timed out servers treated as fail.
- i “accept” with other servers. Other servers will send back “ack” message. Majority of successful nodes required to proceed. Timed out servers treated as fail.
- i “commit” to other servers. Servers receives the commit message will store message’s value in their file.

When a site i recovers from shutdown

- i performs a synod algorithm for each log entry i has been missing.
- i will be able to continue to operate once all log entries are restored.

Data storage scheme

- Data is stored in the folder “<site id>”. Within each site’s designated folder, for each log entry, the entry is stored in the file “_.<block id>”.
- The last log entry will be a dummy value. The value will be replaced when an actual value is present. This value is used to indicate end of the input.
- The data stored on file can survive crash failures.

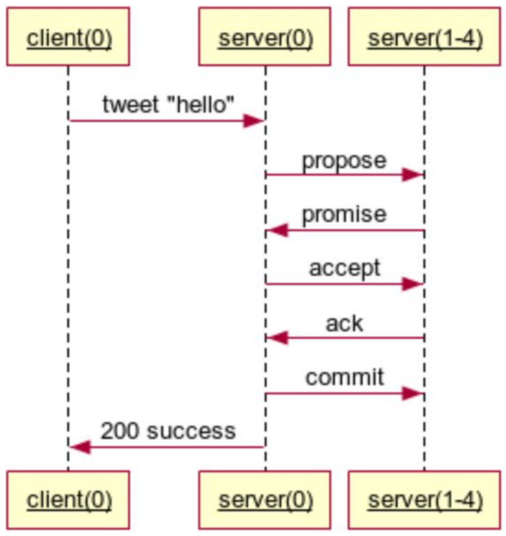
Known issues

- This implementation does not run correctly on Windows. It works fine on MacOS or Linux with Python 2.7.12.

Appendix

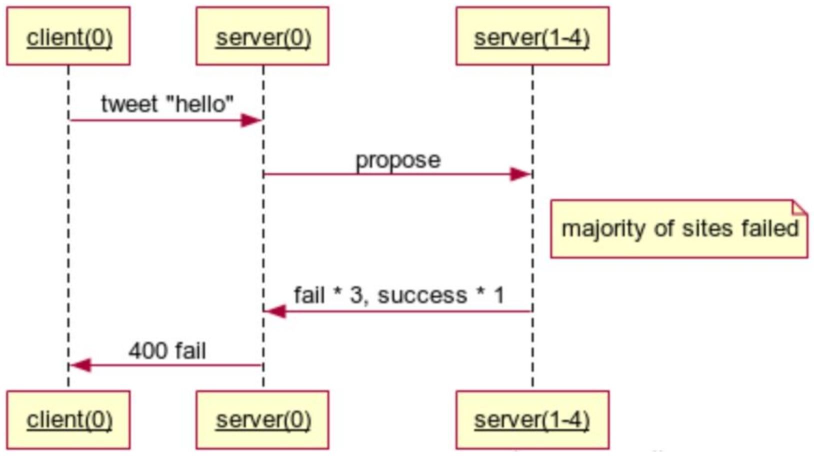
Example UML diagram for a successful “tweet” operation:

- `Site 0 > tweet hello`



Example UML diagram for an unsuccessful “tweet” operation:

- `Site 0 > tweet hello`


client(0)
server(0)
server(1-4)
tweet "hello"
propose
majority of sites failed
fail * 3, success * 1
400 fail
client(0)
server(0)
server(1-4)