# Paxos Writeup

Nicholas Marton, Ian O'Boyle

December 6, 2015

## 1 Leader Election

For leader election, we use the bully algorithm; the implementation of which is within the Bully.py file. Leader election is done by a Node in the Node.py file. A Node begins participating in leader election when it is first executed. The Node's internal method *elect_leader* creates a thread that listens for TCP connections from other nodes and which initiates a new election every 6 seconds (this is represented with the *poll_time* parameter; this parameter can be changed, but from empirical testing, 6 seconds is the earliest time for which all Node's receive all exchanged TCP messages).

## 2 Node

The Node class acts as a single instance of a machine running the both the Bully algorithm and Paxos algorithm.

The Node objects distributes the listening involved in Paxos across the following five threads:

1. A UDP server to receive all UDP messages' the sole purpose of this server is to pass UDP messages to the Proposer or Acceptor respectively. After parsing the UDP message, the contents are given to the corresponding subroutine's queue.
2. A Proposer object, which acts as a server itself, handling all Proposer related messages defined in the Synod algorithm.
3. An Acceptor object, which acts as a server itself, handling all Acceptor related messages defined in the Synod algorithm.
4. A *learner* server, which polls a queue located in the Acceptor object; this queue contains the commit messages that the Node receives. The server dequeues any commit messages and writes them to the Node's log along with updating the calendar to the calendar in the most recent log position whenever such a message is received.
5. Additionally, the main thread acts as a listener and UI to the user.

Additionally, the Node class provides methods to parse input into Appointment objects, save and load Node objects according to the specifications provided in the project description and a method that looks for a file named "IP_translations.txt" which it uses to build an IP-Port table with an entry corresponding to each Node in the network.

Node's also contain a terminate method to shut down all servers and their corresponding threads before terminating the main thread. Termination is initiated by entering "quit" by the user.

### 2.1 Acceptor

The Acceptor class is implemented in Acceptor.py. A Node object contains an Acceptor object and "starts" the Acceptor when the Paxos algorithm is initiated via the *start* method. This method acts as a server and responds to messages meant for an Acceptor in the Synod algorithm; namely "prepare", "accept" and "commit". A Node passes these messages to the Acceptor by placing them on the Acceptor's *command_queue* attribute. The

Acceptor server will then dequeue and respond to any message as it should according to the Synod algorithm in FIFO order. Additionally, Acceptor objects contain two dictionaries *_ackNums* and *_ackVals* containing ackNum's and ackVal's respectively, which are indexed by integer keys corresponding to the log slot (i.e. the instance of the Synod algorithm to which the ackNum's and ackVal's belong).

### 2.2 Proposer

The Proposer class is implemented in Proposer.py. A Node object contains a Proposer object and "starts" the Proposer when the Paxos algorithm is initiated via the *start* method. As in the Acceptor class, this method acts as a server and responds to messages meant for a Proposer in the Synod algorithm; namely "propose", "promise" and "ack" and the Node passes these messages to the Proposer by placing them on the Proposer's *command_queue* attribute. The Proposer server will then dequeue and respond to any message as it should according to the Synod algorithm in FIFO order. Additionally, Proposer objects create two threads:

1. The first thread it creates acts as a server (found in the subroutine of the "start" method under the name "_listen_to_promises"), which polls a dictionary of lists and waits to respond with accept messages for a specific log slot until a majority of promises are found for that log slot. It chooses the calendar for accept messages according to the selection process of the value "v" defined in the Synod algorithm.

2. The second thread acts as an "ack" server, (found in the subroutine of the "start" method under the name "_listen_to_promises") which polls a dictionary of lists and waits to respond with commit messages for a specific log slot until a majority of acks are found for that log slot. Once a majority is found, the Proposer distributes the commit message and marks that a commit for that log slot was sent out to avoid sending duplicate commit messages in the next iteration of the server's listening.

Each Proposer object also has a dictionary attribute *my_proposals*, which holds any proposals either created at or sent to that Proposer so that "v" can be chosen correctly when being selected for accept messages.

## 3 Calendar and Appointments

The implementation of the Appointment class can be found in Appointment.py. The Appointment class acts as a specific entry in the calendar. The Calendar classes' implementation can be found in Calendar.py. The Calendar class acts as a container for a set of Appointment objects, and has the necessary methods for comparing Calendar objects, self serialization for sending, and output representation. The Calendar class also prevents conflicting Appointments from being entered. Additionally, both Calendar.py and Appointment.py have a suite of unit tests to ensure basic functionality is correct.

## 4 Paxos

The Paxos algorithm is initiated by the Node object upon execution. The initiating method named "paxos" calls an internal method "_do_paxos", which first "starts" the Proposer and Acceptor objects and starts the learner thread (these correspond to 2. 3. and 4. listed in the Node section above). It then proceeds by creating the UDP server 1. listed in the Node section above. It then parses any UDP packets received and passes them on.