Programming Assignment 4: Competing Proposals in Paxos

Due Wednesday Nov. 6, 9:00 pm,

1 Lab Task

In this lab, you will implement a normal operation of Paxos, but without a leader, where two proposers are trying to get their proposals to succeed. This is similar to the example from slide 46 from the Paxos slides. We will refer to all servers as peers.

Each peer starts by knowing a list representing the set of peers that are part of this Paxos configuration. Peer ids will be assigned based on the line in the file, starting at 1. To make it easier, you will specify in the configuration file, what role a peer has: proposer, acceptor, learner.

First you will implement normal Paxos operation where the process with the lowest id will try to propose a value and nothing goes wrong. Follow the description of the variables from slide 45. The second case you will implement is when both peer1 and peer2 submit a proposal.

At the end, all peers should print the chosen value. You can assume that the number of peers is 5. You should test your code with 5 nodes and provide the testcases described below.

You can use `C/C++`, `Java`, or `Go`. Communication can be implemented with TCP. Communication should be reliable. All messages must be printed *verbatim*—they should all be valid JSON strings.

1.1 PART 1: Normal operation of Paxos with 1 proposer (50 points)

After all peers have started, the peer with the lowest id will initiate a proposal and perform the Paxos normal operation as described in slide 45. Each peer will print on the screen what message they sent and what message they received. Note that from the configuration file, a peer will know if it is an acceptor or not. This will help with minimizing the number of messages that are sent. The proposer will communicate only with its acceptors as defined in the configuration file. For example, you can configure that peer2 and peer3 are acceptors for proposer peer1. You don't need to worry about the learners learning the value.

Sent and received messages should be of the following format, where <TEXT> should be replaced with your program-specific values. Message type can be one of: "prepare", "prepare_ack", "accept", "accept_ack". Action should either be "sent" or "received". Message value should be "X" or "Y".

```
{"peer_id":<SENDER_ID>, "action": "<ACTION>",
"message_type":"<TYPE>",
"message_value":"<VALUE>",
 "proposal_num":<PROPOSAL_NUMBER>}
```

Example:

```
{"peer_id": 1, "action": "sent",
"message_type":"prepare_ack",
"message_value":"X",
"proposal_num":5}
```

When a value is chosen by a peer, it should print the following:

```
{"peer_id":<SENDER_ID>, "action": "chose",
"message_type":"chose",
"message_value":"<VALUE>",
"proposal_num":<PROPOSAL_NUMBER>}
```

Example:

```
{"peer_id":3, "action": "chose",
"message_type":"chose",
"message_value":"Y",
"proposal_num":7}
```

TESTCASE 1: All peers start, peer with the lowest id initiates a proposal with a value X given through the command line. All peers print the messages they receive and send. (Peers that are not involved in the protocol print nothing). All peers that participate in the protocol should agree on value X.

1.2 PART 2: Two competing proposals with different values (40 points)

In this part, there will be two proposers that will try to push their own value independently of each other. In this case, you will need another configuration file where you specify who the acceptors are for proposer 1 (the peer with the lowest id, peer1) and for proposer 2 (the peer with the highest id, peer5), similar the example in the slides, the acceptors will work with multiple proposers.

The key here is that peer5 should initiate the proposal *after* peer3 sends the accept to the proposal from peer1—as in the example in slide 46. All peers should end up with value X, even if peer5 was trying to push value Y.

To control when peer5 starts its proposal, you can use a timeout. From testcase 1 you will have some idea how long it takes for proposal 1 to complete.

TESTCASE 2: Peer1 starts with value X, after peer3 answers with accept to peer1's proposal, peer5 starts a proposal with value Y. All peers should agree on value X. The same printed messages should be used as in part 1.

2 Implementation

You need to implement this algorithm in `C/C++`, `Java`, or `Go`, and your implementation must allow the user to configure the execution of the process. You will again be using Docker to package your program. Additionally, you must use Docker Compose as a container orchestrator. Instructions on how to use Docker Compose are provided in the Docker Tutorial. You will be provided with a Docker Compose file for each of the two testcases. You must be able to run each of the Compose configurations and see the expected outputs printed to the screen. The expected interface for your containers is provided below. If you adhere to this interface, then Docker Compose will work seamlessly, without any changes.

You will also receive a `hostsfile.txt` specifying the hosts and roles of the five peers for each testcase. You may test on your own hostsfiles, but to receive full credit, your code must be compatible with the one provided. The hostsfile has the following format. Each line starts with the hostname of the peer followed by a colon. To the right of the colon is a comma-separated list of the roles that peer fills. If the roles are specific to a peer, then the role is appended with a numeric identifier. For example, the acceptors of `proposer1` have the role `acceptor1`. An example hostsfile may look like the following

```
peer1:proposer1
peer2:acceptor1,acceptor2
peer3:acceptor1,acceptor2
peer4:acceptor1,learner1
peer5:proposer2
```

Note: you must implement all CLI arguments *exactly* as written below to receive full credit.

```
Building:
    docker build . -t prj4

    Running this in your project's directory should
    build your project's Docker image using your
    Dockerfile. It should copy over the code and
    hostsfile to the image and compile your project.

Single Container Usage:
    docker run --name <hostname> --network <network> --hostname <hostname> \
           prj4 -h <hostfile> [-v <value>] [-t delay]

    --name <hostname>
      This specifies the name of the Docker container,
      this name should match the one in your hostsfile

    --network <network>
      This is the user-defined Docker network your project
      will run in, refer to tutorial in additional instructions
      to learn how to create one and why you need one.

    --hostname <hostname>
      This specifies the hostname of the Docker container,
      this name should match the one in your hostsfile. It can be used by other
      containers in the same network.

    -h hostsfile
      The hostsfile is the path to a file that contains
      the list of hostnames that the processes are
      running on. It assumes that each host is running
      only one instance of the process.
      ...

      All the processes will listen on the same port.
      The line number indicates the identifier of the process
      which starts at 1.

   -v value
      This is the value used if the peer is a proposer. It is a character.

   -t delay
      This is the time in seconds proposer 2 will wait before starting its proposal with
      its value v. This will be needed for the second testcase scenario.

Orchestration:
   docker compose -f [PATH TO COMPOSE FILE] up
```

3 Submission Instructions

Your submission must include the following files (10 points):

1. The **SOURCE** and **HEADER** files (no object files or binary)
2. A **MAKEFILE** to compile and to clean your project
3. A **README** file containing your name, instructions to run your code and anything you would like us to know about your program (like errors, special conditions, etc.)
4. A **REPORT** describing the system architectures, state diagrams, design decisions, and implementation issues

Your submission should also include scripts, code, explanations on how to run the 2 TEST-CASES described above. To receive full credit for each part, you need to provide the required testcases. You must also be able to view the correct testcase output for each Docker Compose setup.

Submission is through gradescope.

4 Additional resources

You may find the following resources helpful

- Socket programming: `http://beej.us/guide/bgnet/`
- Unix programming links: `http://www.cse.buffalo.edu/~milun/unix.programming.html`
- C/C++ programming link: `http://www.cplusplus.com/`
- Docker tutorial: `https://github.com/iowaguy/docker-tutorial`