CS 5450: Networked and Distributed Computing
Lab 4
Raft
Spring 2024
Instructor: Professor Vitaly Shmatikov
TA: Tingwei Zhang, Aditya Manthri

Due: 11:59PM, May 7 2024

1 Introduction

In this project we ask you to develop a distributed chat room that uses the Raft consensus protocol (`https://raft.github.io/raft.pdf`). In the previous lab, you implemented a P2P chat that didn't guarantee that users see messages in the same order. Now, consider heated online debates, where it's important to know who *started* the argument or suggested the correct solution *first*. We want all users to see messages in the same order and tolerate possible peer and communication failures. We ask you to develop a variation of the Raft protocol that would support such functionality and provide the report on your implementation specifics and limitations.

Recently proposed Raft consensus protocol provides similar fault-tolerance guarantees to Paxos, but has simpler communication algorithm and nicer online visualizations. Please check the original Raft paper and check the visualizations here: `http://thesecretlivesofdata.com/raft/` and here: `https://raft.github.io/` . It's important for you to understand the mechanics of leader election, message passing, and failure recovery.

Please work in a group of two and use Slack and Office Hours to have discussions on the assignment. There is one automated grading script and we will perform test of your applications according to defined steps in Section 3, your code and report quality.

2 Problem

You need to develop a chat messenger that uses Raft to communicate updates between users. The log of the Raft protocol is the history of chat messages submitted by users. Our goal is to always maintain and present the same sequence of messages that users type to their applications. Note, that we don't solve race conditions, but rather focus on a consensus where all users will see messages in exactly the same order.

Think why your implementation of Lab 2 cannot guarantee this and put this reasoning into your report.

The Raft protocol considers only the *leader* that can accept user's requests, however in your implementation you need to enable any node to accept new user messages, so it will look like a real chat.

The natural implementation would be for this node to just pass the message to the leader, and wait to receive a new communication from the leader with this message before adding it to its log. One of the downsides of this approach is that if the node loses communication with the leader, then it can't receive new messages. Make sure that messages are not lost when there is no elected leader or the leader is unavailable, and that the application re-sends them after connectivity is restored.

To better debug and test your application, we offer a similar proxy script as in hw2 which serves as the middleware between client and server. In this project, you still have to use C/C++ as your server's programming language.

3 Grading

To grade, we will run at most 5 nodes and check that the order of messages is the same for all nodes and the nodes that are not stopped have the same number of messages after consensus has been reached. To test this and help you debug your program, we will release a python grader which automatically runs through all test cases and gives a score. We will also release some sample test cases as well.

In your report tell us about your implementation, current limitations, and what are other features of Raft or other protocols can help improve the distributed chat application.

Intentionally fooling the script is a violation of the academic integrity code. For example, it is not allowed to write a program that provides the output expected by the test script, but does not implement Raft.

4 Proxy API Specification

Here we specify the API that both proxy and server should obey. **Note that it is important to follow these rules strictly otherwise the proxy program we provide will not work in a correct way.**

You can test the protocol between proxy and servers using the provided proxy program. When sending a sequence of commands on a TCP channel, the protocol uses "\n" as a separator. Details of the API are listed in the following tables.

In Table 1, we use <id> to denote the process id **which is a self-defined value and thus different from pid defined by OS kernel**. Process ids range from 0 to n-1, where n is the total number of processes. Table 1 shows in the left column the commands that the proxy can receive as input, and in the mid column the corresponding commands that the proxy issues to the server with id <id>.

In the start command, n is the total possible number of servers (processes), port is the port number used by the server process to accept sockets from others. **Note that the port here is an arbitrary number defined by the test case and cannot be fixed or presumed.** The proxy will connect to port and send requests to the server through it. For server-server communication, you can feel free to use any port between 20000 and 29999. Moreover, each server could only talk

to its "neighbour servers" so we suggest that you use a self-defined root-id(e.g., 20000) + pid as the port for server-server communication as in this way it will be easier to identify neighbours' ports. For example, a server with port number 20005(20000+5) could only talk to server with port number 20004 or 20006(if they exist). Your implementation should remain correct in the event of failures. For crash commands, the id number is the valid pid that you want to fail on.

Table 1: Client Commands

client→proxy	proxy→server	details
<id> msg <messagID> <message>	msg <messagID> <message>	client sends the message to proxy and then proxy transfers it to servers
<id> start <n> <port>		proxy starts a process with ./process id n port
<id> get chatLog	get chatLog	get the local chat log of that process, each message is separated by comma ','
<id> waitForAck <messagID>		proxy wait and block until the acknowledgement of message indexed by <messagID>, and resend this message to <id> if there is a timeout. Note that <id> here may refer to any arbitrary process. No resend will happen if <id> is set to -1. All later commands will be blocked except start and waitForACK.
exit		call ./stopall to kill all and exit
<id> crash	crash	the receiver process crashes itself immediately

Table 2 specifies what rules should server follow when transferring the message to proxy/client. Whenever the proxy sends a get chatlog message to a server, the server will respond to proxy with all local messages stored in its chatroom.

Table 2: Server Messages

server→ proxy	details
ack <messageID> <sequenceID>	server acknowledges client that the message indexed by <messageID> is <sequenceID>-th message in the chat log
chatLog <message>,<message>,...	server replies the local chat log separated by comma ",”

5 Submission guidelines

You must have following files/directories in your code submission:

- **proxy.py** file which is the proxy program we provide.
- ***.c/cpp** file which is the source code for server.
- **Makefile** which is the make file for the source code you provide.
- **process** This script accepts three arguments: process id, maximum number of processes and client/proxy-facing port. You can make your binary named process, or use another binary name and call it in process.
- **stopall** file which is the script used to kill all processes you evoke.
- **tests** directory which contains all test cases we provide(and feel free to add more cases made up on your own).

Please make sure Makefile, process, and stopall scripts have executable permission and can be run directly with command make, ./process <id> n <port>, and ./stopall.

The lab is due: 11:59PM Tue, May 7th 2024