CS 5414: Principles of Distributed Computing Fall 2017

Lab 0: Chat Room

In this project you are asked to create the networking foundation for the rest of your projects. You will implement a simple distributed chatroom where participants have the capability to broadcast messages and detect failures. It is in your best interest to create an easily reusable implementation.

You must work on this project alone.

This document is intended to help you in your implementation: you will be well-served by reading it carefully.

1 Overview

Your goal is to implement a simple, FIFO consistent chat room using TCP sockets. Your system will consist of a set of servers, all connected to each other. Each server will keep track of all messages it has received from other servers (including itself) in FIFO order. Additionally, each server should keep track of the other servers it believes are alive at the moment.

Each server receives commands from a master and responds to that master. A list of these commands is shown in Table 1 and the responses are shown in Table 2.

To convince yourself and us that you have built a successful project, your code will have to pass a number of test cases. The test cases typically require the system to process some command and to behave correctly in the presence of failures. The master connects to all the not-crashed servers and will play a central role in your ability to run the test cases.

The master is free to send `get`, `alive`, and `broadcast` commands to any server, and that server should respond in a timely manner if appropriate.

We have developed a master client and we are providing its source code to you. See `https://github.com/Cornell-CS5414/chatroom`. Given this is the first time we use this code, it may contain bugs. If you find any problems in the master, please feel free to let us know. You can either issue a new pull request on github, or you can report on Piazza.

2 Failures

Your implementation must be able to detect and tolerate servers failing. When a server fails, the other servers' message logs should not be impacted. Furthermore, correct servers should be able to detect which server has failed and remove it from their alive set.

Servers fail by crashing. It should be possible to either cause the crash of a process non-deterministically (as in an operator issuing a ctrl-C) or deterministically (for instance, by having processes crash

according to testing scripts). Currently the master kills servers by issuing a `SIGKILL` to the process. Your implementation must be compatible with this.

3 Performance Requirements

Your implementation must be able to satisfy certain performance requirements.

- Your server must be fully operational within 3 seconds of being started. Specifically it must be prepared to handle commands from the master accurately and all other servers must be able to recognize that it is alive.
- Your servers must be able to detect a server has failed within 1 second.
- Your system must be able to handle at least 10 broadcasts from the same server within 1 second accurately. This also implies that if the master were to sleep for a second after sending 10 broadcast commands, all alive servers must have received the broadcast and be prepared to respond accurately to a `get` command.
- An additional implication is that if the master were to sleep for a second after sending a broadcast command, the associated message should come before all other subsequent broadcasts in all message logs.

4 Implementation Hints

- You can use any programming language to implement the system—still, as a courtesy, if the programming language you plan to use is particularly esoteric, please talk to Matt or Cong.
- Servers should not retain their message logs after a crash. Each server begins with an empty log and should not attempt to catch up on logs previously sent logs.
- When using some programming languages (e.g., Java), you may find that when a server crashes, its ports are in use and can't be reused after a restart. This is due to the fact that Java sockets will reserve port numbers for a certain amount of time even after a crash. Look for a way to resolve this issue.
- In this project, we let all logical servers run on the same physical computer. However, each server must be a separate process that uses TCP sockets to communicate with the other servers and the master.
- The correctness of your implementation should not be based on any "artificial" delays imposed by you (e.g. thread.sleep(5000)).

5 How Master and Servers Communicate

You can test the protocol between master and servers using the provided master program. When sending a sequence of commands on a TCP channel, the protocols uses "\n" as a separator. Details of the protocol are listed in Table 1

In the table, we use $\langle id \rangle$ to denote the process id. The list of process ids will be $0, 1, \ldots, n-1$,

where n is the total number of processes.

The table shows in the left column the commands that the master can receive as input, and in the mid column the corresponding commands that it issues to the server with id $\langle id \rangle$.

In the `start` command, n is the total possible number of servers (processes), *port* is the master-facing port. The master 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.

For simplicity, we assume messages contain only English letters and numbers.

The scripts used in the table are described in the next section.

6 Submission Guidelines

- You must provide two markdown files and three scripts. You can find Java samples (using maven) for the script in the assignment package.
  - `README.md` – This file should contain adequate documentation for course staff to run test cases.
  - `requirements.md` – This file includes packages (with version number) needed to compile and run your program, in a human readable manner. This information is crucial for us to compile and run your code, especially if you decide to use uncommon packages. And, if we can't run it, we can't give you credit for it.
  - `build` – This script compiles your code and generates binary. If you use C/C++, you may want to have a simple 'make' command in the script. If you use Python/Perl/Ruby, you may let this script do nothing. If you use Java, you may use 'javac' command or tools like Ant or Maven to compile your code.
  - `process` – This script accepts three arguments: process id, maximum number of processes, and client/master-facing port. If you write in C/C++/Golang, you can make your binary named `process`, or use another binary name and call it in `process`. If you write in Python/Perl/Ruby, you can simply call your script in `process`. If you write in Java, use 'java' command to run your code in `process` script.
  - `stopall` – This script kills all your processes. You can use either "`killall`" command or "`ps aux | grep XXX | awk '{print $2}' | xargs kill`". Please make sure it doesn't kill the master and auto-grading program. `killall python` is a good example of killing the master, which is NOT allowed to be put in to the `stopall` script.
  - Please make sure `build`, `process`, and `stopall` scripts have executable permission and can be run directly with command `./build`, `./process` ⟨`id`⟩ ⟨`n`⟩ ⟨`port`⟩, and `./stopall`, even when it is empty if you use Python.
- You are also expected to include your test cases in your submission, following the sample test cases we provide.
- Please make sure your code works on Linux. You can use any Linux distribution in recent 5 years or so. We will primarily use Ubuntu 16.04 LTS in a virtual machine above VirtualBox to test your code. If you use other distributions, please specify in the `README.md` file.

You will be submitting your code via CMS. Please make a tar file named NedId.tar (for example

`cd564.tar`) with the following structure (replace `cd564` with your NetId).

```
cd564
├── build
├── stopall
├── process
├── README.txt
├── src
│   └── xxx.java/.c/.py etc.
├── tests
│   ├── test1.input
│   ├── test1.output
│   ├── test2.input
│   ├── test2.output
│   └── ...
├── requirements.txt
└── you can put extra files here such as Makefile if you use C/C++
```

You can create the tar file by command `tar -cf cd564.tar cd564`, where cd564.tar is the tar file name and cd564 is the directory name. Please run command `tar -tf cd564.tar` to make sure it outputs something similar to the following.

cd564/
cd564/build
cd564/stopall
cd564/process
cd564/README.txt
cd564/src/
cd564/src/lab0.c
cd564/tests/
cd564/tests/test1.input
cd564/tests/test1.output
cd564/requirements.txt

Submissions not satisfying these guidelines will receive a grade penalty or be returned without being graded.

7 Evaluation

- We use the provided master program to test the correctness of your program. Each test case is an input to the master. We check the correctness of the values returned to the master as a result of a get command.
- Besides the sample test cases we provided, you are supposed to write your own test cases to test your program thoroughly.

- In order to grade this lab automatically, we will use a script to run test cases. The test cases include but not limited to the samples.
- Intentionally fooling the test script is a violation of code of academic integrity. For example, it is not allowed to write a program which provides the expected output to master but does not implement the described protocol.

8 Appendix

Input→Master	Master→Server	Comments
⟨*id*⟩ start ⟨*n*⟩ ⟨*port*⟩	—	master starts a process with `./process id n port`
exit	—	master calls `./stopall` then exits
sleep ⟨*n*⟩	—	master sleeps for n milliseconds
⟨*id*⟩ crash	—	The master crashes the given process using a kill signal. Your `process` must be compatible with this.
⟨*id*⟩ get	get	the receiver responds to the master with its message log
⟨*id*⟩ alive	alive	the receiver responds to the master with the ids all processes it thinks are alive
⟨*id*⟩ broadcast ⟨*message*⟩	broadcast ⟨*message*⟩	the receiver broadcasts the given message to everyone alive, including themselves

Table 1: Table of commands. The left column shows commands provided as input to the master; the center column the corresponding commands issued by the master to the servers.

Server→Master	Comments
alive ⟨id1⟩,⟨id2⟩,...	when a server is asked to return all alive severs, responds by giving a list of the sever ids in ascending order, including itself
messages ⟨message1⟩,⟨message2⟩	when a process is asked to get its messages, responds by giving a list of all messages it has received in FIFO order

Table 2: Messages from the servers to the master.