CS 5414: Principles of Distributed Computing Fall 2019

Lab 1: Three-Phase Commit

Due: Oct 20, 11:59:59pm

In this project, you are asked to experience, for the first time in this course, the joys and tribulations of going from Theory to Practice. We are going to start by looking at Three-Phase Commit and, to connect the bare protocol to at least a semblance of an application, we are going to consider the protocol in the context of a simple playlist editing utility, outlined below.

You can work at the project in groups of size no larger than three.

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 consistent distributed music "playlist" using three-phase commit. This is an unordered list of pairs of the form ⟨songName, URL⟩. Commands to modify the playlist should obey the three interfaces: Add, Get, and Delete (see Table 1).

The system consists of a set of servers, one of which serves as the coordinator. A client (we refer to it as "the master") sends commands to the coordinator and receives responses from the coordinator. A list of these commands is shown in Table 1. The master may also send commands to any server (including a participant) to ask that server to crash.

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 maintains the ID of the current coordinator, and it will play a central role in your ability to run the test cases.

If the test-case calls for a server to crash, it will be up to the master to send the appropriate command to the relevant server. If it calls for issuing get command, the master is free to send the command to any server, and that server should respond. The master should send add/delete commands to the coordinator, and the coordinator should acknowledge the master to inform it whether the command was accepted or aborted. If the coordinator crashes, the master must wait until one of the servers informs master that it has become the new coordinator.

We have developed a master client and we are providing its source code to you on `https://github.coecis.cornell.edu/Cornell-CS5414/2019lab1`. While we've tried to test the code thoroughly, it may contain bugs. If you find any problems in the master, please feel free to let us know and report it on Piazza.

2 Failure cases

Your implementation should remain correct under all failure scenarios that are supported by the Three-Phase Commit protocol.

Nodes 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 once they send certain messages to other processes).

The possible failure cases include, but are not limited to:

- Participant failure and recovery.
- Coordinator failure and recovery. (Note: after the coordinator recovers, it no longer acts as the coordinator for the current transaction).
- Cascading coordinator failure: the coordinator fails, another one is elected, then it fails, etc.
- Future coordinator failure: before the coordinator crashes, the would-be coordinator must already be dead. Your protocol must handle this case by moving on to the next available coordinator. In order to be able to demonstrate this easily, this lab requires that the coordinator election is a round-robin among the participating processes.
- Partial Precommit: the coordinator will crash after having sent a Precommit message to only a subset of the participants.
- Partial Commit: the coordinator will crash after having sent a Commit message to only a subset of the participants.
- Total failure: all processes fail and after some time some/all of them recover. You should be able to demonstrate that the participants will only make progress after the "Last process to fail" group has recovered. For any given set of recovered processes, the coordinator should be the one among them with the smallest id. Once a set of recovered processes determines that it does not contain the last process to fail, it allows any newly recovered processes to formally join the set of recovered processes–and the new set tries again to satisfy the condition.

Some or all of these failure cases (and more) will be covered in our test cases. We provide you a subset of those cases to get you started. You should make sure to write your own!

3 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 us.
- Each participant has a local playlist with a set of songs. The coordinator (who also has a local playlist) can instruct the participants to perform Add, Get, or Delete operations on their local playlists. Participants then vote on whether they accept the outcome or not. If the size of the URL in bytes is larger than the value of a process pId+5, then the process votes NO, otherwise it votes YES. de
- You may find useful to keep the playlist as a Hashtable⟨String,String⟩. See java.util.Hashtable for more information if you are coding in Java.
- 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 or UDP sockets to communicate with the

other servers and the master.
- You can implement the stable storage required by the DT Log used during recovery by simply writing to disk.
- The correctness of your implementation should not be based on any "artificial" delays imposed by you (e.g. thread.sleep(5000)).

4 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 the following table.

In the table, we use $\langle id \rangle$ to denote the process id. An id value of -1 in an input command indicates the coordinator (the master will translate -1 to coordinator 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 are free to use any port between 20000 and 29999. `crashXXXX` commands can specify any valid subset of process ids as its target (the table uses ids 2 and 3 just as an example). In particular, the set of ids could include all processes, or be empty.

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

The scripts used in the table are described in the next section. Samples of the scripts are provided with this document.

5 Submission guidelines

- You must provides two text files and three scripts. You can find samples for them in the assignment package.
  - `README.txt` This text file should contain adequate documentation for a programmer with knowledge of Three-Phase Commit to run test cases and/or the protocol (and, if appropriate, contain the answer to the extra-credit question (see Section 7).
  - `requirements.txt` This text 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.

Input→Master	Master→Server	Comments
⟨*id*⟩ add songName URL	add songName URL	add/edit a song if the receiver is coordinator
⟨*id*⟩ delete songName	delete songName	delete a song if the receiver is coordinator
⟨*id*⟩ get songName	get songName	get a song URL from any server
⟨*id*⟩ start ⟨*n*⟩ ⟨*port*⟩	—	master starts a process with `./process id n port`
exit	—	master calls `./stopall` then exit
⟨*id*⟩ crash	crash	the receiver process crashes itself immediately
⟨*id*⟩ crashAfterVote	crashAfterVote	the receiver process crashes itself after sending vote response if it is participant
⟨*id*⟩ crashBeforeVote	crashBeforeVote	the receiver process crashes itself after receiving vote request but before sending response
⟨*id*⟩ crashAfterAck	crashAfterAck	the receiver process crashes itself after sending ack response if it is participant
⟨*id*⟩ crashVoteREQ 2 3	crashVoteREQ 2 3	the receiver process crashes itself after sending vote request to processes 2 3 if it is coordinator
⟨*id*⟩ crashPartialPreCommit 2 3	crashPretialPreCommit 2 3	the receiver process crashes itself after sending precommit to processes 2 3 if it is coordinator
⟨*id*⟩ crashPartialCommit 2 3	crashPartialCommit 2 3	the receiver process crashes itself after sending commit to processes 2 3 if it is coordinator

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
msgLength-coordinator ⟨*id*⟩	when a process is elected as coordinator, it sends its id to master
msgLength-resp ⟨URL⟩	when a get request received, the server replies with the URL without running 3PC protocol. If the songName is not in the playlist, replies with "NONE".
msgLength-ack commit/abort	when an add/delete request received, the coordinator runs 3PC protocol and acknowledges the result (commit or abort)

Table 2: Messages from the servers to the master.

– `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 deletes all the log files you have created so that another test can be run.
  - Please make sure `build`, `process`, and `stopall` scripts have executable permission and can be run directly with command `./build`, `./process` ⟨`id`⟩ ⟨`port`⟩, and `./stopall`.
- You are also expected to include your test cases in your submission, following the sample test cases we provide.

You will be forming groups and submitting your code via CMS. Please make a tar file named NedId.tar (for example `la13.tar`) with the following structure (replace `la13` with your NetId.

```
la13
├── 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 la13.tar la13`, where la13.tar is the tar file name and la13 is the directory name. Please run command `tar -tf la13.tar` to make sure it outputs something similar to the following.

```
la13/
la13/build
la13/stopall
la13/process
la13/README.txt
la13/src/
la13/src/three_phase_commit.c
la13/tests/
la13/tests/test1.input
la13/tests/test1.output
la13/requirements.txt
```

An example tar file is provided with this document.

If your group includes more than one person, you should concatenate their NetIds using underscore "_" as the connector, as in `la13_yp348`).

Submissions not satisfying these guidelines will be returned without being graded.

6 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 test cases to test your program thoroughly.
- 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 the code of academic integrity. For example, it is not allowed to write a program that provides the expected output to master but does not implement three-phase commit protocol.

7 Extracredit

The textbook (and slides) contains one mistake! Find it for extra credit! To find it, try thinking through the failure scenarios you need to address. Once you do find it, describe it in README.txt