Implementing the 3 Phase Commit

Md. Shariar Kabir(0419052047)

June 18, 2021

1 Problem Statement

The goal of this project is to implement a consistent distributed music "playlist" using three phase commit. A distributed playlist is an un-ordered list of pairs of the form ⟨song_name, URL⟩. It can consist of two or many devices. The system ensures *consistency*, by satisfying the *ACID* property.
Commands to modify the playlist should obey the four interfaces: **Add**, **Get**, **Edit**, and **Delete** (see 1).

2 System Architecture

The system consists of a set of servers, one of which is referred as the **coordinator**. A client (we refer it as "**the master**") is used for sending 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.
Commands used by Client nodes are shown in table 2. These commands are used to generate different test case scenario and for checking the state and transaction history of the corresponding client node. 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 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 to any server, and that server should respond. The master sends add/delete commands to the coordinator, and the coordinator acknowledges 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.

Command	Purpose	Usage
halt	stops the client. Additionally, sends a halt message to each of the servers. Use this to terminate the entire simulation.	halt
start	start a client process	<pid> start <total> <port>
add	Used to add a song to the playlist. If the song already exists then it will have no effect on the playlist.	-1 add <song> <URL>
edit	Used to edit a song in the playlist. If the song does not exist then it will have no effect on the playlist.	-1 edit song1 URL1

delete	Used to delete a song from the playlist. If the song does not exist then it will have no effect on the playlist.	-1 delete song1
get	Used to get a song's url from the playlist. If the song does not exist then it will return NONE.	-1 get song1
crash	Immediately stops the client	<pid> crash

Table 1: Commands used by master node.

Command	Purpose	Usage
<ctrl> c	use this to instantly stop a node. (We don't catch this signal, the node can stop at any point in execution.)	<ctrl> c
crash	a more kind way of stopping a process. Type this into the terminal and the process will finish doing whatever it was doing and will terminate.	crash
vetonext	Cause the node to vote NO on the next votereq.	vetonext
playlist	Cause the node to print its current playlist.	playlist
actions	Cause the node to print all transactions it has performed on the playlist.	actions
crashBeforeVote	Process will crash before the next vote.	crashBeforeVote
crashAfterVote	Process will crash after the next vote.	crashAfterVote
crashAfterAck	Process will crash after the next ack.	crashAfterAck
crashVoteReq	Cordinator will crash before the next vote-req.	crashVoteReq
crashPartialPreCommit	Cordinator will crash before the next precommit.	crashPartialPreCommit
crashPartialCommit	Cordinator will crash before the next commit	crashPartialCommit

Table 2: Commands used by Client nodes

3 Actions

The possible actions performed by the system is illustrated below

3.1 Start

As shown in figure 1

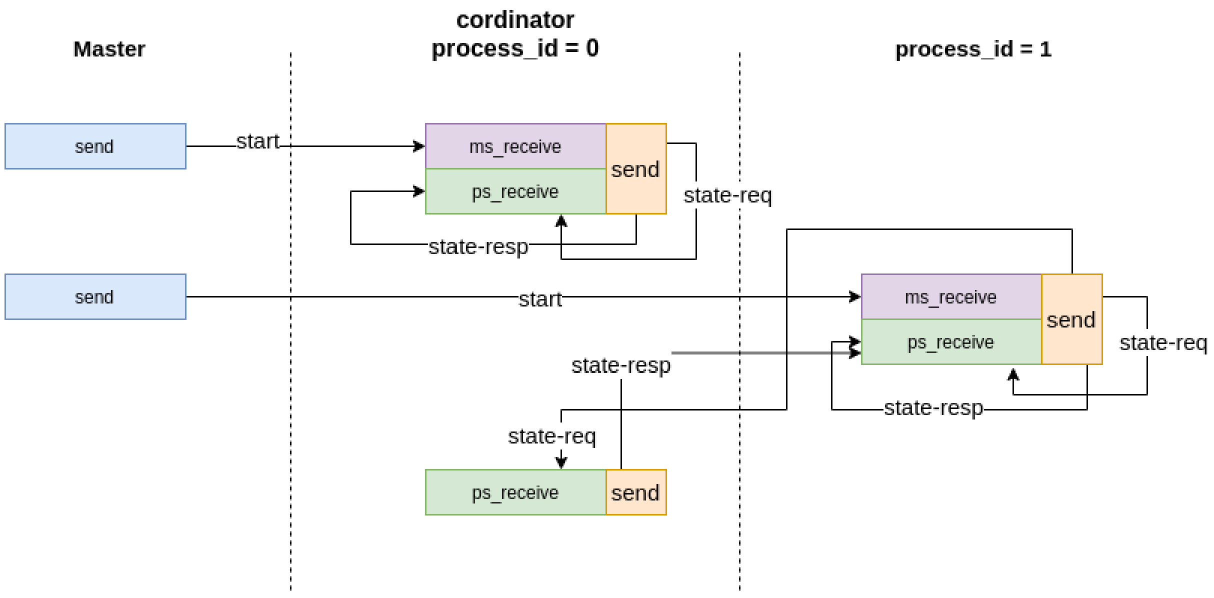

Figure 1: Start Operation

3.2 Add/Edit/Delete

The actions *add*, *edit* and *delete* each performs the similar way. For convenience only add operation is shown in the figure 2.

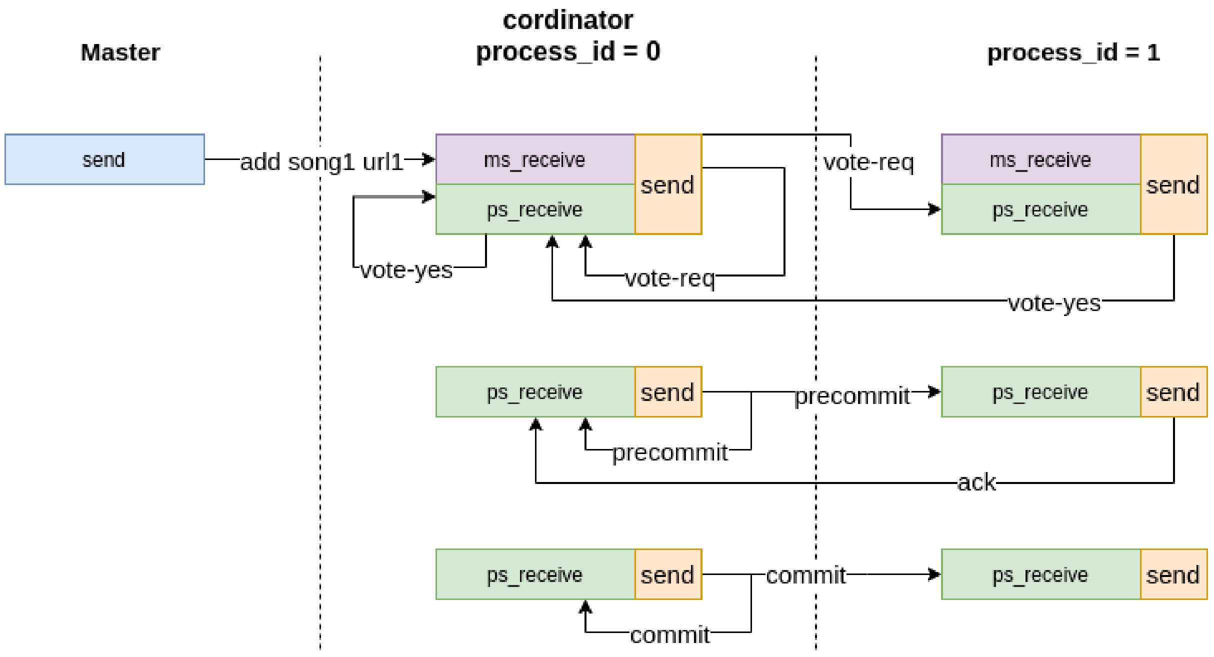

Figure 2: Add Operation

3.3 Get

As shown in figure 3

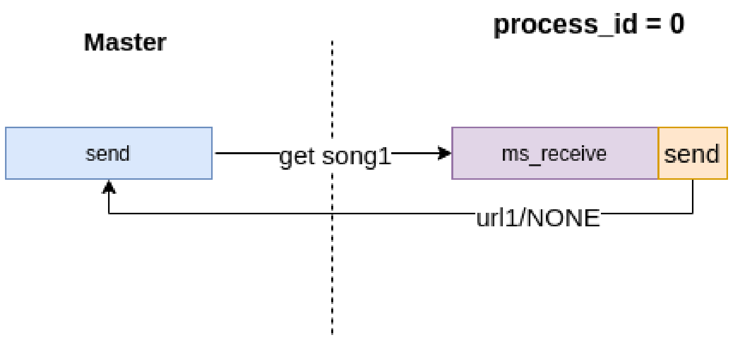

Figure 3: Get Operation

4 Failure Model

Below we describe the failure models(crashes) and corresponding timeouts:

- No failure, but a participant votes NO
  - The transaction will be aborted.
- Process failures
  - **crashBeforeVote**: process_timeout_vote will occur action will be aborted
  - **crashAfterVote**: process_timeout_ack will occur action will be committed
  - **crashAfterAck**: no timeout will occur action will be committed
- Cordinatior failures
  - **crashVoteReq**: coordinator_timeout_vote_req will occur re-election and termination_protocol will decide abort
  - **crashPartialPreCommit**: coordinator_timeout_precommit will occur re-election and termination_protocol will decide abort as processes are uncertain (TR3)
  - **crashPartialCommit**: coordinator_timeout_commit will occur re-election and termination_protocol will decide commit as processes are committable (TR4)

5 Recovery Model

The system provides failure models for the following scenarios:

- Node fails while no transaction is running and gets back online
  - Participant will have the latest playlist
  - Actions/Transactions performed by that client will be reset
- A participant P fails during a transaction X and gets back online
  - If P fails before casting a vote, X will be aborted
  - If P fails before giving ack, X will be committed

  - If P fails after giving ack, X will be committed

6 Implementation Detail

I will implement three phase commit using Python 3. There will be a collection of processes, with 'internal ids' ranging from 0 to N-1, where N is the total number of *backend* processes. One of the backend processes is designated as coordinator; it receives commands from a master and directs a run of 3PC. Each backend process listens on port 20000+i, where i is the process's 'internal id'. The process has one thread listening on 20000+i, one thread communicating with the master, and N threads maintaining sockets to each of the open ports. In addition, there are five timer threads, responsible for detecting various timeouts.

The file *trans_conroller.py* contains the code which handles the actual connections between backend processes. Each instance of *trans_conroller.py* has a *trans_manager.py* object *TM*, which will govern the logic of three-phase commit. The two (controller and manager) communicate via a minimalist TCP API. Namely, each *TM* object is given a send(pids[], msg) function which sends msg to each of the given processes, then returns the array of processes which successfully received msg. Moreover, each *TM* has functions receive(msg) and receive_master(msg), which are called when *trans_conroller.py* gets a message from another backend / the master.