Cody Littley
Rezwana Rimpi

Project 1 Write-up

Contents

Included in the tarball submitted you will find the following files:

- 3pc.jar -- The executable solution
- config -- Contains configuration files for five nodes
- logs -- An empty directory, the location used by 3pc.jar to store log files
- playlists -- An empty directory, the location used by 3pc.jar to store playlist files
- portCleaner.py -- A small python script that frees ports from rogue processes
- reset.sh -- Run this shell script to reset the simulation
- source.tar.gz -- Contains the source files (in an Eclipse project)
- README.pdf -- This document

Running

This project was built and tested exclusively on Linux. Expect the unexpected on other operating systems.

The configuration files provided allow for the simulation to be run with 5 nodes: 1 client and 4 servers. Nodes 0-3 run as servers, node 4 operates as a client.

The configuration files are set up to support operation on a single machine (localhost). By default it uses ports 55550 through 55554. In order to free these ports, run the following command: **python portCleaner.py**. If you manually change the ports, you will also need to update the list of ports at the top of portCleaner.py.

In order to launch a node, first navigate to the directory containing 3pc.jar. Run the following command: **./3pc.jar <name of the configuration file>**

Note: the config file should be stored in the **config** directory. Do not type the full path to the config file, rather type the path relative to the **config** directory.

Examples:

./3pc.jar config0.txt -- this launches node 0 (a server)
./3pc.jar config4.txt -- this launches node 4 (the client)

Interacting With The Simulation

The ***client*** accepts the following commands. Simply type them into the terminal that is running the client.

halt: stops the client. Additionally, sends a halt message to each of the servers. Use this to terminate the entire simulation.
add: Used to add a song to the playlist. The client will prompt the user for a song title and a URL. If the operation is successful the client will receive an acknowledgement. Even if the servers decide to commit an add operation, if the song already exists then it will have no effect on the playlist.
edit: Used to edit a song in the playlist. The client will prompt the user for a song title and a new URL. If the operation is successful the client will receive an acknowledgement. Even if the servers decide to commit an edit operation, if the song does not exist then it will have no effect on the playlist.
delete: Used to delete a song from the playlist. The client will prompt the user for a song title. If the operation is successful the client will receive an acknowledgement. Even if the servers decide to commit a delete operation, if the song does not exist then it will have no effect on the playlist.

A ***server*** node accepts the following commands. Simply type them into the terminal that is running the client.

<ctrl>-c: use this to instantly stop a node. (We don't catch this signal, the node can stop at any point in execution.)
halt: 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.
timeout T: Set the timeout of a node in seconds. It is strongly recommended that if you set this manually that the primary be given a shorter timeout than the participants.
speed S: Cause the node to run more slowly. Setting this will cause a node to only check for incoming messages every S seconds. If you introduce large values be sure to adjust timeout accordingly.
vetonext: Cause the node to vote NO on the next votereq.
playlist: Cause the node to print its current playlist.
actions: Cause the node to print all actions it has performed on the playlist.
deathafter M: The node will halt after receiving M messages. Setting M to zero will cause the node to halt immediately after receiving its next message. Setting M to -1 will return the node to normal behavior (i.e. not halting after a certain number of messages).

deathaftern M N: The node will halt after receiving M messages from node N. Setting M to zero will cause the node to halt immediately after receiving a message from N. Setting M to -1 will return the node to normal behavior with respect to node M.
partialpre M: This will cause the node to halt after sending M PRECOMMIT messages. This can be performed on nodes that are not currently the primary (but may become primary in the future). Setting M to -1 will return a node to normal behavior.
partialcomm M: This will cause the node to halt after sending M COMMIT messages. This can be performed on nodes that are not currently the primary (but may become primary in the future). Setting M to -1 will return a node to normal behavior.

At any point in time, the user may examine the logs and playlists of each node by examining the contents of the **playlists** and **logs** files. DO NOT EDIT THESE FILES MANUALLY!

RECOVERY Mode:

We handle node recovery in the following scenarios:

- A primary/participant fails during a transaction X and gets back online
    - immediately while the transaction is still running.
    - after the transaction X has been finished.
    - after more than one transactions have been finished (including X)
    - after more than one transactions have been finished (including X) and another transaction Y is running.
- A node fails while no transaction is running and gets back online
    - after 0 or more transactions have been finished
    - after 0 or more transactions have been finished and a transaction Y is running.
- Recovery from total failure:
    - when all processes crash at the same time and get back online at a time/one after another.
    - when processes crash one after another and get back online at a time/one after another.

  Processes recover from the total failure only when the process that failed last gets back online

Extra Credit

There was a flaw in the original protocol. This flaw was the failure to add PRECOMMIT to the log.

Example problem: Lets say the primary was able to commit locally (after sending PRECOMMIT) and then the entire system crashes. Everybody but the primary comes back online. Everybody is uncertain (because they didn't log PRECOMMIT) so they decide to abort. However, the original primary has committed.

Our submission correctly handles this situation. You may invoke this scenario in the following manner:

Slow all of the nodes down. For example, use **timeout 20** and **speed 5**. This will give you the opportunity to kill nodes at the appropriate time.

Type **partialcomm 0** into the current primary. This will cause the primary to halt immediately after committing and before sending any COMMIT messages.

After you see that the primary has crashed, you must wait for the remaining nodes to time out. If the other nodes don't notice that the primary has crashed (before they halt themselves) then they will wait for the primary to come back alive before resuming the protocol.

As soon as you see the remaining nodes time out, quickly kill them all using **<ctrl>-c**.

Bring all nodes but the original primary back online. These nodes will elect a new primary and will properly decide to commit the pending transaction.

Test Cases:

Failure Testing

1. No failure, but a participant votes NO:
    a. To model this scenario first apply ***vetonext*** at any participant to sets its next vote NO, then issue a command (add/ edit/ delete) from user.
2. Participant failure:
    a. To model this scenario, first apply ***deathafter 1*** or ***deathaftern 2 0*** at any participant (assuming node 0 is primary), then issue a command (add/ edit/ delete) from user. In case of ***deathafter 1*** the participant will die immediately after receiving vote-req, so the user request will be aborted. In case of ***deathafter 2 0*** the participant will die after receiving vote-req and precommit (since all other participants voted YES), so the user request will be committed.
    b. Try (a) in more than one participants.
3. Coordinator failure:
    a. To model this scenario, first apply any of the commands: ***deathafter, deathaftern, partialpre, partialcomm*** at primary, then issue a command (add/ edit/ delete) from user.
4. Future Coordinator failure:
    a. Suppose node 0 is primary. Apply ***deathaftern 2 0*** at node 1. So node 1 will fail after receiving precommit from 0. Apply partialpre 2 at node 0. So node 0 will fail after sending precommit to node 1 and 2. So node 3 will timeout on

node 0 and call the election protocol that will elect node 1 as primary, but node 1 has been crashed already. Eventually node 2 and 3 will commit the transaction.

5. Cascading Coordinator failure:
    a. Suppose node 0 is primary. Apply ***partialpre 1*** at node 0, node 1 and ***partialpre 0*** node 2 . So node 0 will fail just after sending precommit to node 1. Nodes waiting on node 0 will call election protocol on timeout and node 1 will be elected new primary. Since node 1's state is committable, so it will send precommit to all. But node 1 will fail as soon as it sends precommit to node 2. Node 2 being the new primary is responsible to send to send precommit to node 3, but fails before that. Finally node 3 becomes the primary and being in the uncertain state, aborts the transaction.

Recovery Testing

6. Kill a participant as soon as it votes and bring back online immediately.
7. For each of the cases 1 - 5, bring back the failed node/nodes either immediately or after a couple of transactions.
8. Perform the extra credit scenario. This is also a scenario of recovery from total failure.
9. To model another recovery from total failure scenario, apply ***partialpre 1*** at node 0, node 1 and at node 2 and ***partialpre 0*** at node 3. So all nodes will fail one after another. Then bring them back in the order they failed. They should be able to commit the transaction when the last failed node gets alive.