Name: Chen Chen, E-mail: chen1647@purdue.edu, Chao Tang, E-mail: tang191@purdue.edu

# CS542 Project Report

## 1 Statement of the Problem

This project is an implementation of the three phase commit protocol and the termination protocol by Chen Chen and Chao Tang. We implemented a reliable centralized three phase commit protocol by UDP and message re-transmission. The system can be easily scaled to a large amount of machine. After the co-ordinator is crashed, the termination protocol can re-elect a co-ordinator from the remain sites and continue processing. Basically it's a non-blocking protocol. The program can run of linux machines (we tested on xinu machines).

## 2 Methods or Procedures

We use C++ and linux socket API to implement the three phase commit protocol. For the purpose of simplicity, we didn't implement clients to submit transactions to the co-ordinator. Instead, the co-ordinator will automatically generate a transaction (use a transaction number to differantiate the transactions) and send to the cohorts. We specify the votes of each cohort in a file (strategy.txt), such that I can modify the file to change the vote of each cohort.

We have two classes. Base class MSG and sub-class ThreePC. Base class is used for socket setting up, ip address getting, message sending and receiving and system configurations. Sub-class ThreePC is the detailed implementation of three phase commit and the termination protocol.

After runing make, the progam will be compiled to threepc. The command to run the program is:

./threepc -t strategy.txt -h ip2.conf -m 1 -p 10086

-t is the strategy that each cohort take. -h is the ip address of each site and the corresponding site number. -m specifies which site to be the master site. (The site here is 1). -p specifies the port number used to send the message.

Detailed description of each file.

- **msg.cpp** is the implemetation of class MSG. The constructor will set up the socket, getting the ip address of the site and getting the ip of other sites in the system from file **ip.conf**. Besides, I have the function sendMessage(), recvMessag() and sendAllMsg(). SendMessage sends the message (specified by message type) to a specific site (specified by the site number), recvMessage receives the message from other site. sendAllMsg() will send the message to all the sites in the system except itself and sites which are denoted down by the session vector.

- **msg.h** is the header file of **msg.cpp**. The class Tag in **msg.h** is used to indicate that a site has sended back the replied message. If it times out for TRY times, the site will be considered to be down and the session vector will be set.

- **threepc.cpp** implements the three phase commit protocol and termination protocol.

  - Three phase commit protocol: We implemented the algorithm according to the wiki pedia page of three phase commit (https://en.wikipedia.org/wiki/Three-phase_commit_protocol)

with some minor modification. The modification is in the pre-commit state, if all cohorts vote yes, then even though some cohort may crash, master should commit after time out. All other are the same as what is described in this wiki page.

  * Coordinator:
    1. The coordinator receives a transaction request. If there is a failure at this point, the coordinator aborts the transaction (i.e. upon recovery, it will consider the transaction aborted). Otherwise, the coordinator sends a canCommit? message to the cohorts and moves to the waiting state.
    2. If there is a failure, timeout, or if the coordinator receives a No message in the waiting state, the coordinator aborts the transaction and sends an abort message to all cohorts. Otherwise the coordinator will receive Yes messages from all cohorts within the time window, so it sends preCommit messages to all cohorts and moves to the prepared state.
    3. If the coordinator succeeds in the prepared state, it will move to the commit state. However if the coordinator times out while waiting for an acknowledgement from a cohort, it will abort the transaction. In the case where all acknowledgements are received, the coordinator moves to the commit state as well.
  * Cohort:
    1. The cohort receives a canCommit? message from the coordinator. If the cohort agrees it sends a Yes message to the coordinator and moves to the prepared state. Otherwise it sends a No message and aborts. If there is a failure, it moves to the abort state.
    2. In the prepared state, if the cohort receives an abort message from the coordinator, fails, or times out waiting for a commit, it aborts. If the cohort receives a preCommit message, it sends an ACK message back and awaits a final commit or abort.
    3. If, after a cohort member receives a preCommit message, the coordinator fails or times out, the cohort member goes forward with the commit.

- Termination protocol:
  * Coordinator Failure:
    1. Re-election part: I used bully algorithm for implementing the re-election of coordinator. I implement this part according to the bully algorithm wikipedia page. ("http://en.wikipedia.org/wiki/Bully_algorithm") When the coordinator fails at WAIT state or PREP state (ready state or precommit state), the remaining sites first starts doing re-election for a new coordinator. All sites that are alive sends INQUIRY to other sites to announce election. The site which receives INQUIRY will compare the value (sender sites id) with its id. If it has bigger id, it sends an ALIVE message back to the sender which basically tells that site to quit election. The site which receives ALIVE message will quit election and wait for more messages coming from the site that has higher id. If a site does not receive any ALIVE message back which means no site has high id than itself, it will announce victory by sending VICTORY message to other sites. The site which receives VICTORY messages will set the coordinator id to be the id of the sender of this message.

2 After re-election: I implemented centralized termination protocol. The newly elected coordinator will first send TRANSIT message to set state on the other sites to be its state. For example, if the newly elected coordinator is in PREP state, it will set the state of other sites to be PREP. There are only two exceptions. If the newly elected coordinator is in ABORT or COMMIT state, we do not need to send the TRANSIT message because it is the three phase commit protocols job to do so. I implement this part according to the paper written by D. Skeen, Nonblocking commit protocols, D. Skeen, ACM SIGMOD, 1981..

* Cohort failure: If coordinator timeouts in WAIT state, the transaction is aborted. If coordinator timeouts in PREP state, the transaction is commited. If coordinator timeouts in in COMMIT or ABORT state, the coordinator does not receive acknowledge whether the transaction is committed or aborted. But it does not need to take any special action in this case because we already know whether the transaction is commitable or not.

- **threepc.h** is the header file of **threepc.cpp**. It has the definition for the threePC object. It contains site state definition (INIT, WAIT, ABRT, CMT, PREP). It also contains test functions as well as the test function controller which is used to control which function to run. There is also a function called **redundentReply**. This function is used to handle reciving duplicate messages due to message re-transmission.

# 3 Data Collected

We used a file called strategy.txt to keep a record of each sites decision (yes or no) for each transaction.

Below is an example of the layout of the file:

7
1 1 1 1
0 1 1 1
1 0 1 1
1 1 0 1
1 1 1 0
1 1 0 0
0 1 0 0

The "7" means there are 7 transactions. We have 4 sites. Each row stands for the decision of the four sites for the transaction. Each column stands for each site. For example, the first row "1 1 1 1" means that for transaction 1, all sites will answer yes vote.

Our test cases:

We have five test functions to simulate failures on coordinator or cohorts under different states.

- **testElection1()** This function simulates the situation that the coordinator fails at WAIT state before it receives all yes, no votes from cohorts. To call this function, simple set the varibale TESTELECTION1 to true in the **setTest()** function. In this case, the correct way is to abort the ongoing transaction because we do not know what other sites vote for this transaction when the coordinator is at WAIT state. This transaction turns out to be aborted in our test case, so our test case passes.

- **testElection2()** This function simulates the situation that the coordinator fails at PREP state (prepare state). In this state, it already receives all yes-no votes from cohorts. To call this function, simple set the varibale TESTELECTION2 to true in the **setTest()** function. In this case, the correct way is to commit the ongoing transaction because we already know that all cohort sites have voted yes for this transaction when the coordinator is at PREP state. This transaction turns out to be commited in our test case, so our test case passes.
- **testSubsequentFail()** This function simulates the situation that the coordinator fails at PREP state (prepare state). Then the newly selected coordinator fails right after it is selected as the coordinator (Subsequent failure). To call this function, simple set the varibale TESTSUB to true in the **setTest()** function. In this case, the correct way is to commit the ongoing transaction because we already know that all cohort sites have voted yes for this transaction when the coordinator is at PREP state. This transaction turns out to be commited in our test case, so our test case passes. In addition to the above test case, this test case has shown that our system is robust to subsequent failures under the situation that the newly elected coordinator fails.
- **testSlaveDown1()** This function simulates the situation that the cohorts fails right after it receives CANCMT msg from coordinator (initially asking for yes or no vote). To call this function, simple set the varibale TESTSLAVEDOWN1 to true in the **setTest()** function. In this case, the correct way is to abort the ongoing transaction because the failed cohort has not answered yes or no for the transaction. Our implementation will make the coordinator resend the CANCMT message four times to the non-responding cohorts. If after four times resending, the cohort is still not responding, this transaction will be aborted. Also the session vector for the failed cohort is set to be 0 (site down). So our test case passes.
- **testSlaveDown2()** This function simulates the situation that the cohorts fails right after it receives DOCMT msg from coordinator (global commit). To call this function, simple set the varibale TESTSLAVEDOWN2 to true in the **setTest()** function. In this case, the correct way is to commit the ongoing transaction because the failed cohort has answered yes for the transaction. It fails when it is about to commit locally. Our implementation will make the coordinator resend the DOCMT message four times to the non-responding cohorts. If after four times resending, the cohort is still not responding, this transaction will be committed globally. Also the session vector for the failed cohort is set to be 0 (site down). So our test case passes.

Also, when re-election happens, we can transfer a cohort to a co-ordinator smoothly.

# 4 Experiences

- Experience from Chen Chen:
  - Because this is a team project and we use git to maintain our code. Because we didn't design a good interface at the beginning, during the implementation, there are some conficts when we do git merge operations.
  - When I build up the underlay UDP interface, I didn't write the creation of socket in the class constructor, it always cause some strange problems. After putting the creation in a constructor, the problem is solved.

- **recvfrom** will get blocked when there is no messages, I first use **ioctl** to make the socket non-blocking and use a polling approach to handle in coming messages. This approach is inefficient and error prone. Then I changed it to **select** and with it's built in time out mechenism I even don't need to implement a timer class myself to control time out. Besides it's more efficient.

- Experience from Chao Tang:

  - When I was working on termination protocol, the problem I encountered is that the termination protocol needs to leave the state of sites consistent with three phase commit protocol. Specifically, when a site is elected as the new coordinator, we need to be very careful about modifying the state for each sites so that the termination protocol will lead the transaction to an abort or commit state. Also, making re-election robust for subsequent failure is another hard problem I met. I need to be careful when dealing with timeouts, because timeout may happen at different situation. I need to set up state information correctly to deal with those situations separatly to make it work.

# 5 Observations

The project we implemented is basically a high quality simulation of the three phase non-blocking protocol. If we add clients in our protocol, we can claim our system to be a good reliable commitment protocol.

If the client is added to the system, there might be situations where several clients wants to send a transaction to the co-ordinator at the same time. This may lead to disaster because in our simulation if there is an outstanding (on going) transaction, the co-ordinator will not issue a new transaction to other cohorts. Only when the current transaction gets aorted or commited, the co-ordinator will re-issue a new transaction.

So in the case when several client submit a transaction, I think we have at least two ways to handle this. First, we build a buffer queue for the transactions submitted by the client. If there is an outstanding (ongoing) transaction, co-ordinator will push the new transaction into the buffer. Only after the current transaction is done, co-ordinator will re-issue a new transaction from the head of the queue. This approach is easy to implement, but it has at least two flaws. First, the buffer may run out of space and the transactions may get lost. Second, it's not efficient since new transactions may wait for the completion of the outstanding transaction.

Another way is we can keep a state for each transaction. Why do we need to wait for the ongoing transaction to finish in the above method? Because we should mentain the state of the co-ordinators and cohorts. If we keep a state for each transaction, the co-ordinator may send the transactions to the cohorts as soon as it gets a new one. But it needs more synchronization in the three phase commit protocol.

Another thing which needs to clarify when the clients is added to the system is clients needs to be informed of the newly selected co-ordinator, if the previous co-ordinator crashes and the termination protocol selected a new co-ordinator. To solve this, we may add another server to the system called guard. Every server (co-ordinator and cohorts) knows guard and can communicate with it directly. Every time a new co-ordinator is selected, it will inform guard himself. Clients can only communicate to guard and guard will forward the transaction to the current co-ordinator.

We think of a project called Paxos. It is similar with Three Phase Commit which is also a non-blocking protocol. Paxos does not block if a majority of processes are correct. It can be used to solve more general concensus problems. Compared to Three Phase Commit and termination protocol, it also has a re-election phase which will guarantee non-blocking. Besides, Paxos renders a simpler, more efficient algorithm (minimal message delay), and has been proved to be correct.

# 6 Work Distribution

This is a team project by Chen Chen and Chao Tang, and we think our contribution to it is 50% to 50%.