CS-505 Project 3

Short Overview:

The project is about implementation of Paxos algorithm using the pseudo code provided in the paper *http://www.cnds.jhu.edu/pub/papers/cnds-2008-2.pdf*. This algorithm is used to maintain the replicas of same data upon multiple servers using the state-machine paradigm. The protocol helps them to globally order the actions to be performed upon their local data. Since they all start in the same initial state so executing the actions in same global order also keep the final state of their data consistent. It is a robust protocol where the global ordering in each view is controlled by the leader. The leader server of each view is known to be the server with id as (i%N) + 1 (if server id starts from 1), where i is the current view number. However there are steps in algorithm like reconciliation, recovery phase which we are not implementing as a part of this project. Still we will make sure that all the servers will execute the updates sent from client over their local data in same order.

System Architecture:

The system consists of multiple servers which communicate among themselves over the UDP socket. The communication network is unreliable and asynchronous i.e. the messages over the network can get delayed, dropped and duplicated but we assume that there is no Byzantine behaviour by any of the server. If a packet is reached from one server to another then the packet is not corrupted. There can be multiple clients associated with each server which will communicate with servers using the TCP socket. Each client will send the updates request to its local server who is responsible for executing the client updates and sending reply to him in turn. There might be scenarios where server has not send replied to the client and has closed its connection in which case client should just print the proper message and continue with the next update.There is a file named ***"hostfile.txt"*** which is shared among all the servers to know about each other. The hostfile contains the hostname of all the machines participating in the algorithm. The communication is duplex for each server and they have their own sender side and receiver side. We have the hostfile for client as well which is used to populate the server id in the client update message. The architecture is same as discussed in the paper[1]. The important phases of protocol are:

Leader Election:

The leader election starts after expiration of the progress timer. In the leader election phase each of the server sends a View Change message to others with the requested view number. If the majority of the View Change messages are received by a server then it means that server has pre-installed the view and it doubles it's progress timer to give the leader more time to execute the update. Upon receving the majority of View Change messages they all wait for the Prepare messages. When the leader gets the majority it shifts to the prepare phase and transmits Prepare messages for the agreed view to all the other server.

Prepare Phase:

In prepare phase the leader sends the Prepare message to all the servers with the current installed view. After sending the prepare message the leader waits for the prepare ok messages from other servers. While the non-leader servers upon receiving the prepare messages checks is the view is already installed or they need to install it and sends the prepare ok message storing it in its data structure too (to compensate for the sending to itself). After receiving the majority of prepare ok message for current view the leader changes its state to REG_LEADER and manages its update and pending queue updates. Whereas the non-leader process upon sending the prepare ok message moves to the REG_NONLEADER state and updates its last installed and clear the update queue. Now the non-leader server waits for any proposal or for

progress timer to expire to suspect the unstable leader. Whereas leader process sends the proposal to other servers for the unordered proposal in the global history or updates in the update queue (if any).

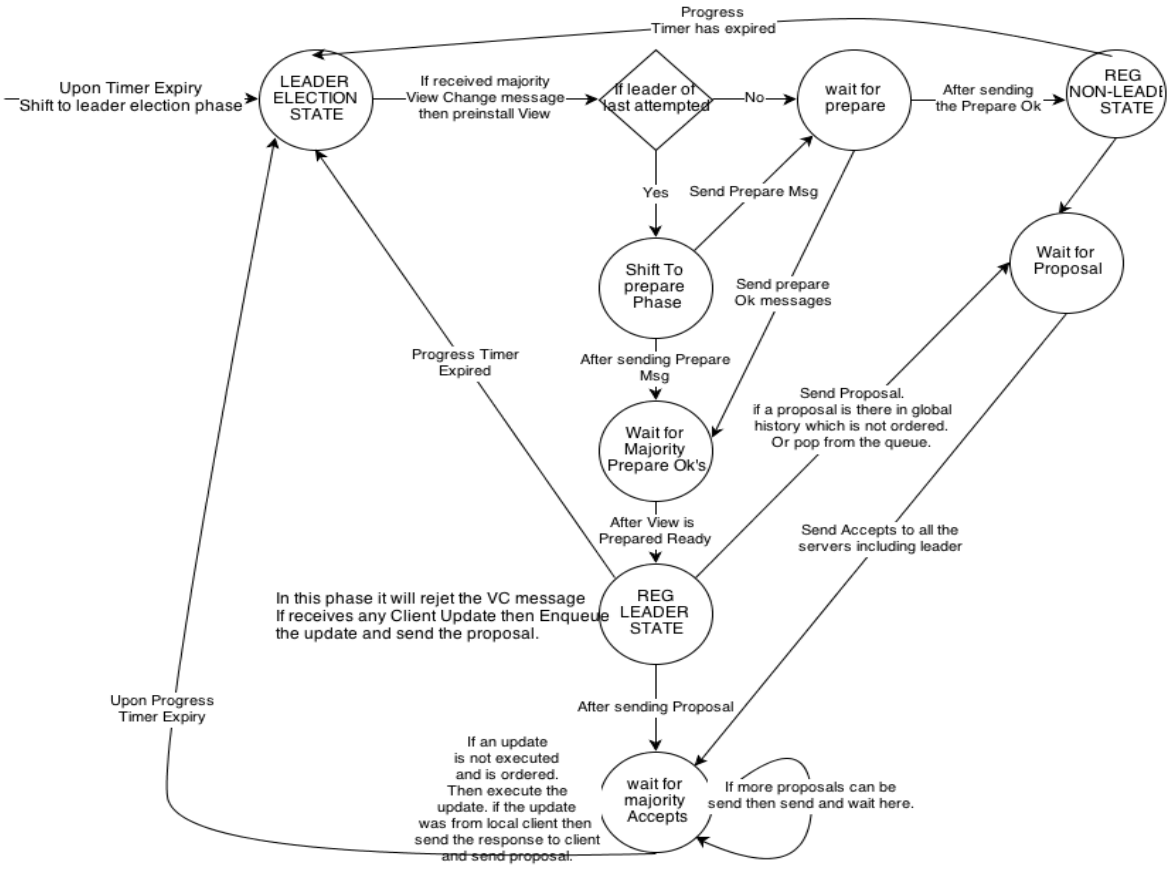
Global Ordering Phase:

After sending the proposal the leader waits for the majority accepts for this proposal from the non-leader servers. After receiving the majority accepts its says that it has globally ordered that update with a particular sequence number and its ready to execute when its local aru reaches this sequence number. The same logic hold for all the non-leader servers too. After receiving the required number of updates they advances their aru and see's if there is any ordered update to execute or not.

Client Handling:

The client update are handled in the different ways depending upon the state they are received in. If the local client of a server sends an update to a server, the server stores its in its pending updates and send it to the leader provided it is in REG_NONLEADER state. If a leader directly receives an update from its local client and it is in REG_LEADER state then it add it to its update queue and sends the proposal. Whereas if a server is in LEADER_ELECTION state and the client update is not from local client then it rejects the update.

State Diagram:



Design Implementation Details :

There are two separate files for executing the server (server.cpp) behaviour and the client (client.cpp) behaviour. The program server.cpp accepts command line arguments such as paxos port specified by “-p” option, “-h” to specify the location of the hostfile and “-s” to specify the server port. The client.cpp file accepts the mandatory parameters such as -s for server port, -h for host file, -i for client id and -c for command file.

The servers opens a tcp socket for communicating with the client and UDP sockets for communication between other servers. I have duly followed the pseudo code mentioned in paper[1] for the implementation.

In server.h file I have placed the signature of all the functions and data structures as mentioned in the paper.

In message.h I have placed the structures of the messages that is exchanged between the servers and client. I have changed the structure of Proposal, Globally Ordered Update and Prepare Ok messages to include the Client Update message information in full instead of just the uint32_t update value. The new structures are mentioned in the README submitted with this project.

There have been decisions that have been taken while fixing the timer specific values. Since we know that the update timer is smaller than the progress timer as former is used to send the updates message to leader and later is used for progress in the view to remove the unstable leader. I have kept the progress timer expiry value to be 2 seconds, update timer expiry value as 1 seconds and vc proof timer as 3 seconds. The vc proof message is used to catch up the lagged behind server immediately.

Issues:

While testing I came across few scenarios which requires some modification in the algorithm. For example:

a) I am checking for progress timer in shiftToRegLeader and shiftToNonRegLeader functions. Since there can be case for some servers that progress timer is not doubled after receiving view change messages as they have not received it from majority of serves. Now leader has send them the Prepare message and they have replied back with the Prepare_Ok message. So we need to restart the Progress Timer for this installed view which will be done by checking in ShiftToRegLeader and ShiftToRegNonLeader.

b) Advancing Aru while in shift to prepare phase to avoid execution of duplicate updates.