## Ravi Kiran Rao Bukka

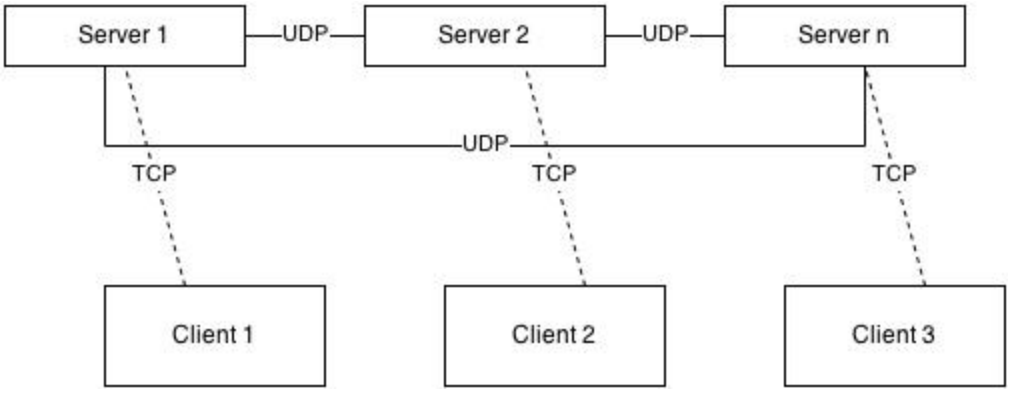
# CS505- Project 3 Report - Paxos protocol

**System Architecture:**

The system has two independent modules that interact with each other:

1. Server - Waits for requests from Client to execute.
2. Client - Sends update to any local server for execution.

The Server module is run in multiple machines and each of these instances interact with each other using UDP protocol. The Server also a TCP socket listening for connections to be accepted from a client.Below is the overview of the architecture of the whole system. The Client module can also run in multiple machines (either same or even different executable) to send an update to the local server.
Each server can communicate with any other server in the system. In essence they form a clique with direct connections. Below is the overview of the system architecture:




Each server has to meet the requirements of ordered delivery which is guaranteed by paxos. Paxos is a robust protocol in which global ordering is coordinated by an elected leader..

**State Diagram:**

Each server will be in one of the following three states during the protocol : (It starts with a blank or Leader Election by default.)

1. Leader Election.
2. Regular Non Leader.
3. Regular Leader.

The servers use the following messages for communication:

1. Client Update - Used to store an update from client. It is also forwarded to the leader of the view to begin ordering.
2. Prepare and Prepare OK - These messages are used during the leader election phase. Until a server gets a majority of Prepare OK messages for its prepare message,, it would not be a leader. This message also has a history of proposals and globally ordered updates.
3. Proposal - Used to send a client update as a request for execution to the servers. The update is bound to sequence number here.
4. View Change - Once a server goes to a starts the leader election process , it sends a message to all the other servers notifying its last attempted view.
5. View Change Proof - Each server sends a View Change Proof or VCProof message periodically to get a server which is lagging behind to front.
6. Accept - An accept message acknowledges a proposal saying it is willing to execute it. If a server get s a majority of accepts it stores the update in its globally ordered message queue.
7. Globally Ordered Update - It is the update with majority accepts. It is stored in the global history.

Each server maintains the following set of timers:

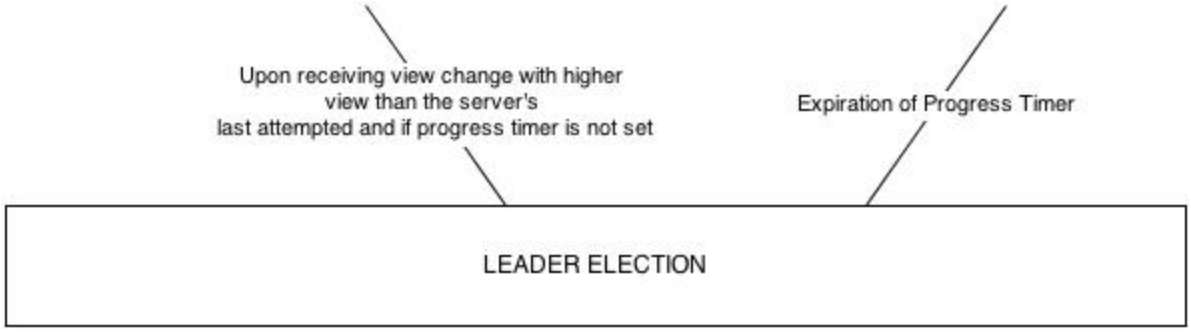
1. Progress Timer.
2. Client Update Timer.
3. View Change Proof Timer.

Each server moves forward in the following way:

1. The server's come to a consensus on one of them being elected as the leader.. In this process, each server uses a progress timer and a set of message exchanges. A server uses the Prepare message during the leader election process. If any of the server's gets majority of Prepare_OK messages , it changes its state to a REG_LEADER. The other servers will go to the REG_NON_LEADER state.

As this is asynchronous communication, the state changes may oscillate between REG_LEADER and REG_NON_LEADER till all the servers come to a consensus in the same view. At any given point the server id of a server should be Last installed view % Total number of servers in the system.

Below is the diagram showing when a server shifts to the state of 'LEADER ELECTION''.

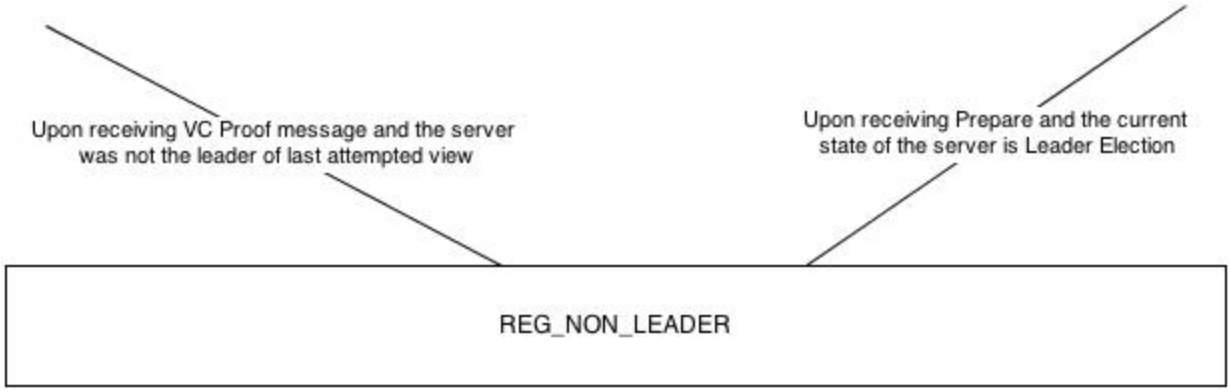


In the LEADER ELECTION phase we take the following actions:

If we get a VC_Proof, we come up to speed.

Below is the diagram showing when a server shifts to ‘REG_NON_LEADER’ state. When in this state on expiration of progress timer again shifts to LEADER ELECTION.

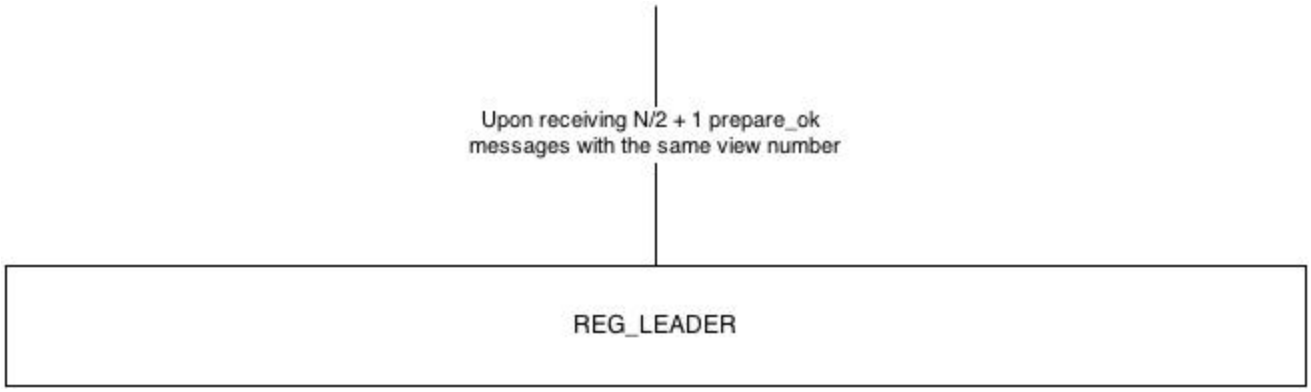
1. View_Change: If got majority of them then Preinstall view & Send Prepare if I am the leader,
4. VC_Proof: Come upto speed if lagging behind,
5. Client_Update: If it is from local client then buffer it.




In REG_NON_LEADER phase:

1. Reject View Change messages.
2. Reject VC_Proof messages.
3. Client_Update: Update data structures & forward to leader,
4. Proposal: Update structures & send Accept to all,
5. Accept: Check if the update for this Accept can be executed,
6. Restart update timer if it expires for a local client and resend the update to the leader.

Below is the diagram for ‘REG_LEADER’ state. If the progress time expires it again shifts to ‘LEADER ELECTION’.



1. View_Change: Reject,
2. VC_Proof: Reject,
3. Client_Update: Update data structures & send Proposal,
4. Accept: Check if the update for this Accept can be executed,
5. Restart update timer if it expires for a local client.

**Design Decisions:**

1. The best possible way to implements timers is to create threads. But due to inherent complications of using threads, debugging issues, locking issues, could not use threads. (given the time limit).

2. Instead used 'select' to poll on the different UDP , TCP file descriptors with a timeout.. The transmission of VC Proof and the expiration of update timer is not as granular as it can possibly be . Also value of these timers is to be tested experimentally and then decided given a network. (Possibly learning a preferred timeout).

3. The value of these timers is in "Server.h" file and they can be changed to experiment with. Whenever progress timer expires we are sure most of the time that update timer and vcproof timer have expired so we make the appropriate function calls to send these messages. Experimented for a lot of time on turret to adjust these timers.

**Implementation Issues:**

1. Using 'select' for timeout and reading sockets did cause some issues and needed major adjustment in calling different timer expiries like update timer and vcproof timer.