CS7610-MilesBenjamin-Hwk3

Question 1 - Part A
In this scenario we have 1 crashed server (a, 1) and 4 remaining servers (b, 2), (c, 3), (d, 4), (e, 5). When server b detects that server 1 has crashed by the expiration of the Progress Timer, they start the leader election process.

Server b then clears it's view change and last enqueued lists along with its prepare and prepare_oks messages. It then increments the last attempted view id, and constructs a new View Change message using this new id. Server b then sends the vc message to all servers and saves off the message.

View Change { server_id: 2, attempted: last view +1 }

Upon receiving the view change message from server b, each other server goes through the same process as b but without incrementing the view index. Each peer waits until it's received N/2 +1 of these messages. At which point they set a new progress timer and server b starts the next phase of the process.

View Change { server_id: 3, attempted: new view }

Server b marks that it is installing the currently proposed view. It constructs a prepare message using the new view and the leader's local aru value. It saves off this message, then creates a data list using the local aru, which lists any proposal or global order update messages. Server b then creates a prepare ok message using this list, saves it to the Prepare_OK data struct and syncs all of this info with the disk in case of failure. Finally server b sends the prepare message it constructed earlier to all servers.

Prepare { server_id: 2, view: new view, local_aru: 2 }
Prepare_OK { server_id: 2, view: new view, data_list: [Proposals] }

When servers receive the prepare message, since they are in the midst of the leader election, they will install the new view. They each save off the prepare message, create their own data list and prepare ok message. Then they shift their state to non-leader and send the prepare ok back to server b. Once server b has received N/2 +1 such prepare ok messages the process is over and regular functionality resumes.

Prepare_OK { server_id: 3, view: new view, data_list:[Proposals] }

Question 1 - Part B

In this scenario we have a server who crashed (1, a) and 4 remaining servers (b, 2), (c, 3), (d, 4), (e, 5). Servers b and c both detect the crash and trigger the new leader election sequence. They each begin by clearing out their view change data, incrementing their last attempted and sending out a view change message.

View Change { server_id: 2, attempted: attempted +1 }
View Change { server_id: 3, attempted: attempted +1 }

As all the servers receive these two view change messages, they prepare and send their own as above. Once N/2 +1 of these messages have been received servers b and c both move on to the prepare stage. Servers b and c both pre-install the new view and create prepare and prepare OK messages as above.

Prepare { server_id: 2, view: new view, local_aru: 2 }
Prepare_OK { server_id: 2, view: new view, data_list: [Proposals] }
Prepare { server_id: 3, view: new view, local_aru: 2 }
Prepare_OK { server_id: 3, view: new view, data_list: [Proposals] }

As they send out their prepare messages one of two things will happen. As servers receive the prepare messages, either they will have not seen this new view, at which point they will install the view and reply with a Prepare_OK message. Or they will have seen the view at which point they will resend the Prepare_OK message to the leader. Depending on the order of which these messages are received either server b or c will get N/2 +1 Prepare_OK messages, but not both. We'll say server c becomes the new leader. Eventually server c will send out a VC_proof message to all the servers.

VC_Proof { server_id: 3, installed: new view }

When server b receives this message, it will realize it's been beaten and resume normal functionality as a non-leader.

Question 2
The RAFT algorithm stores logs with an entry id and no view id. This is unlike Paxos where everything requires a view id to put things in context. The reason RAFT can dispense with the view id and rely solely on entry id is that RAFT relies on an authoritarian leader server to make most of its decisions. Because the leader is entirely authoritative, it can decide, without checking with the other servers, what order to log information and when it is safe to replicate the log. In the case of a leader failure, the process with the longest / most complete log ultimately wins.

Question 3
The proof of work component in bitcoin serves two purposes. First it provides the timestamp hash for the block, thus finalizing the block. Second it solves an issue for determining majority

decision when there are conflicting chains. Using the proof of work model, nodes vote using CPU power instead of, say IP addresses. Since it's impossible to fake the proof of work, majority CPU power wins in the case of a conflict in chains.

In a bitcoin transaction, the transaction can considered to be committed (aka finalized) once the block that the transaction is listed in is finalized and added to the chain. Up until that point the transaction broadcast could be lost or could be part of a branch which ultimately fails. Once the transaction is in a block and a new block has started to continue the chain it can be considered finalized.

Bitcoin gives a coin to a node each time a new block is made to incentivize network members to continue to support the network as well as to distribute coins. They also use transaction fees, which build up as a block is being developed when transactions have slight incongruities, to pay nodes for their work. At this point they have stopped giving out whole coins and simply pay nodes using transaction fees.