Lecture 18: April 6

Lecturer: Prashant Shenoy *Scribe:* ***Disha***

18.1 Byzantine Fault Tolerance

The two main type of faults include Crash failure and Byzantine faults. Fault tolerance during crash failure tells how to deal with server which crashes silently and detecting failures can be achieved by sending Heartbeat messages.

In Byzantine faults, server may produce arbitrary responses at arbitrary times. It needs higher degree of replication to deal with the fault tolerance. To detect k byzantine faults, we need 2k+1 processes. This was Byzantine two Generals Problem where each general sees that the other two generals are not agreeing with the answer and not sure which general is right. So we need 2k+1 process to detect k faults. To reach agreement we need 3k+1 processes. It is expensive to deal with Byzantine faults.

18.2 Reaching Agreement

If message delivery is unbounded, no agreement can be reached even if one process fails and slow process are indistinguishable from a faulty one. If the process are faulty, then appropriate fault models can be used such as BAR Fault Tolerance where nodes can be Byzantine, Altruistic and Rational.

18.3 Reliable One-One Communication

One-one communication involves communication between a client process and a server process whose semantics we have already discussed during RPC. Possibilities of failures in one-one communication:

a. Client unable to locate server

b. Lost request messages

c. Server crashes after receiving request

d. Lost reply messages

e. Client crashes after sending request

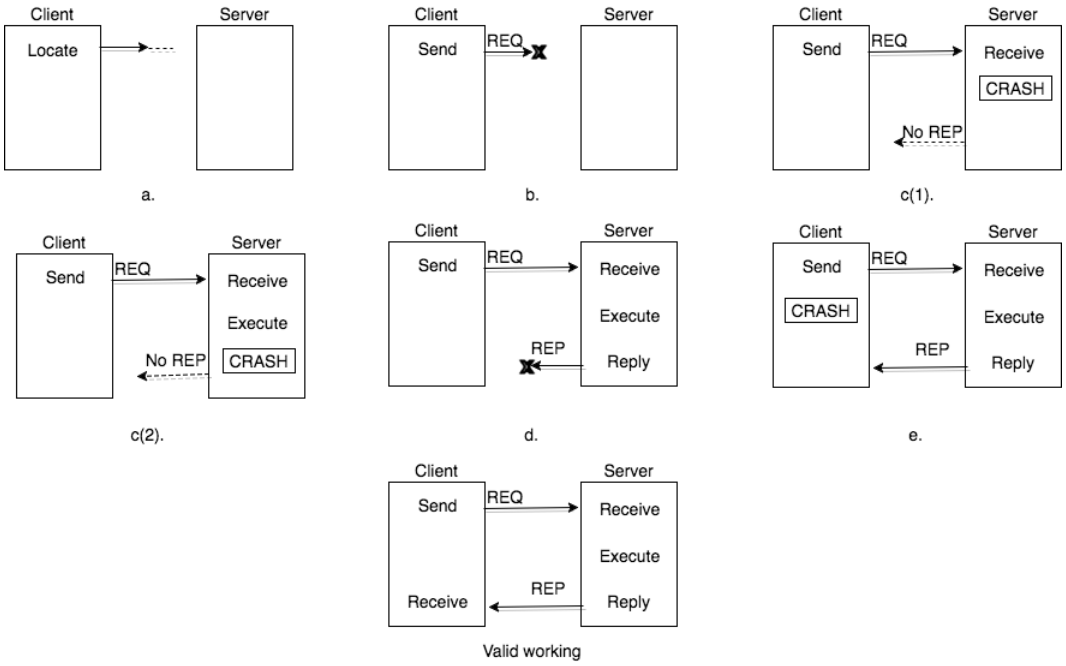


Figure 18.1: Each of the failures enumerated in this section is depicted here. The figure ids correspond to the failure type they are depicting.

These failures can be dealt by:

- Using reliable transport protocols such as TCP. (b and d can be dealt with in this manner)
- Handling at application layer. (a, c and e can be dealt with in this manner)

18.4 Reliable One-Many Communication

18.4.1 Reliable multicast

If there are lost messages due to network inconsistencies, we would need to retransmit messages after timeout. There are two ways to do this:

- ACK-based schemes
- NACK-based schemes

ACK-based schemes :

- Send acknowledgement(ACK) for each of the message received. If the sender does not receive the ACK from a receiver, after timeout it retransmit the message.
- Sender becomes a bottleneck: ACK based scheme does not scale well. As number of receivers in the multicast group grows (say 1000 - 10,000) then the number of ACK messages that needs to be processed also grows.
- ACK based retransmission works well for one-one communication but doesnot scale for one-many communication. Large bandwidth gets used in acknowledgment process which results in **ACK explosion**.

Figure 18.2: Here, all the receivers have their last packet received as #24 except receiver 3 which missed packet #24. Hence, it's last packet is #23. As soon as it receives packet #25, it knows it missed the packet #24.

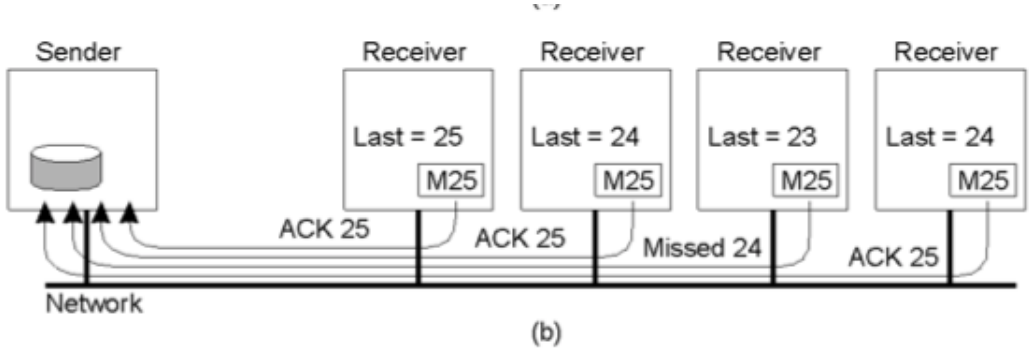


Figure 18.3: Each receiver now sends an acknowledgment ACK either in form of received packet #25 or missed packet #24. As we can see for a single packet, sender receives 'n' ACKs

Question How to reduce the overhead of ACK in one-many communication?
Ans. Instead of sending acknowledgements send negative acknowledgement (NACK).

NACK-based schemes :

- NACK-based schemes deals with sender becoming a bottleneck and ACK-explosion issue.
- ACK indicates a packet was received. NACK indicates a packet was missed.
- Scheme explanation: Send packet to multicast group, if receivers receives a packet, they don't do anything. If receiver sees a missing packet, it sends a NACK to nearby receiver as well as the sender. Sender or neighbouring receivers would re-transmit the missed packet. This optimization works only if the neighboring receivers have the received packets stored in a buffer.
- Sender receive only complaint about the missed packets and this scheme scales well for multicast as the #NACKs received is far less than the #ACKs, unless a massive amount of packet loss.

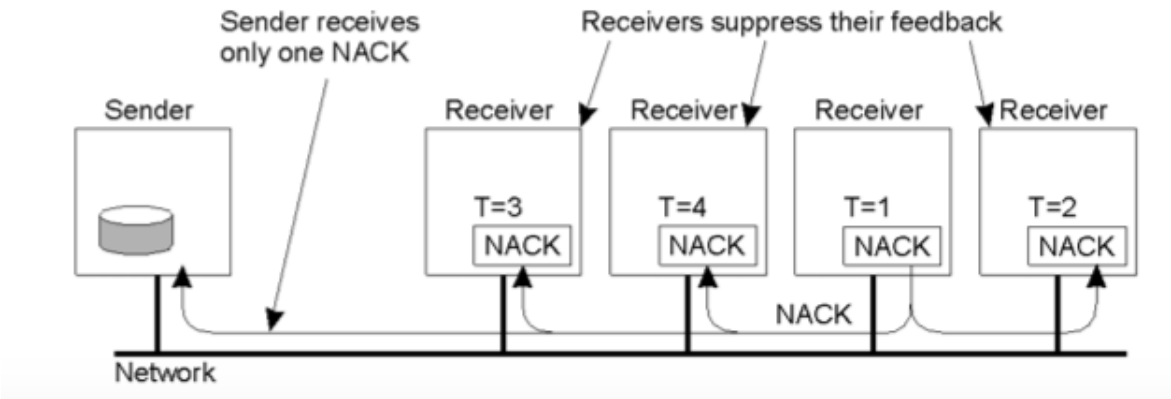


Figure 18.4: Each receiver now suppresses their ACK feedback. Only receiver 3 sends a NACK to other receivers and the sender.

Question How does the receiver know that it missed a packet?
Ans. Assuming the packets are received in sequence and each packet have a packet number. If receiver sees a gap in the sequence, it knows a packet was missed and sends a NACK.

Question Is their a possibility that receive can move from one IP address to another ?
Ans. This possibility exists and is true if its one-one or one-many communication. Socket connection breaks if the IP address changes and connection needs to be re-established. The above mentioned schemes does not handle node mobility.

Question How to deal with last packet or if sender sends only one packet as receiver may never know if it missed the packet?
Ans. Send Dummy packet at the end of transmission and to make sure that dummy packet is acknowledged.

18.4.2 Atomic multicast

Atomic multicast guarantees **all or none**. It guarantees that either all processes in a group receive a packet or no process receives a packet.

Example: Replicated databases We can't have a scenario where M out of N DB replicas have executed some DB update an the rest haven't. It needs to be ensured that every update to the database is made by all or none.

Problem How to handle process crashes in a multicast?

Solution Group view: Each message is uniquely associated with a group of processes.

If there is a crash:

- Either every process blocks because 'all' constraint will not be satisfied.
- Or all remaining members need to agree to a group change. The process that crashed is ejected from the group.
- If the process rejoins, it has to run techniques to re-synchronize with the group such that it is in a consistent state.

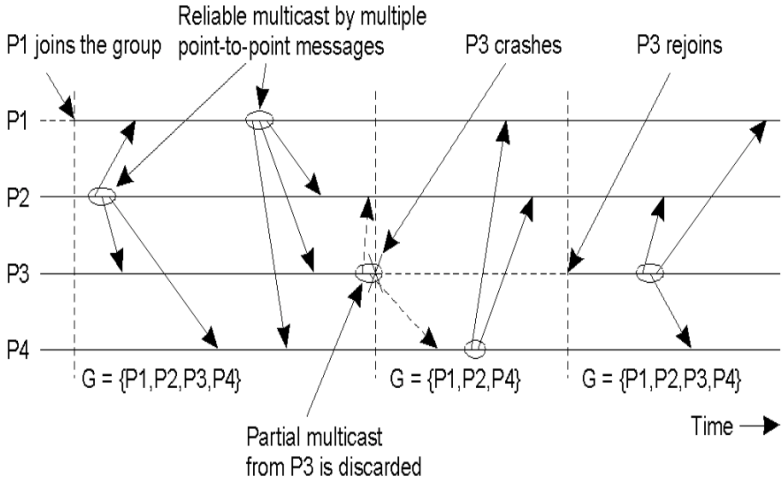


Figure 18.5: Initially all process are up and are part of a group {P1,P2,P3,P4}. All the messages are being reliable multicasted to each of the processes. At dotted line2, P3 crashes while sending a message. From this point onwards, the group {P1,P2,P3,P4} will not maintain the 'all' property of atomic multicast. Hence, P1, P2 and P4 agree on a group change and then start atomic multicast amongst themselves (the new group). At a later point P3 recovers and rejoins. At this point, it run synchronization algorithms to bring itself up-to-date with other members of the group it wants to rejoin.

18.4.3 Implementing virtual synchrony

Reliable multicast and atomic multicast are only two ways of implementing virtual synchrony. There are many variants of these techniques as well as other virtual synchrony techniques which may be used in different application based on the requirements of the application.

- Reliable multicast: Deals only with network issues like lost packets or messages. There is no message ordering. NACK based.
- FIFO multicast: Variant of reliable multicast where each sender's message are sent in order. But, there is not guarantee that messages across senders would be ordered as well.
- Causal multicast: Variant of reliable multicast. Causal dependence across messages which are sent in order.
- Atomic multicast: Totally ordered, all or nothing delivery. Deals with process crashes
- FIFO atomic multicast: Variant of atomic multicast.
- Causal atomice multicast: Variant of atomic multicast.

Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Figure 18.6

18.5 Distributed commit

Atomic multicast is an example of a more general problem where all processes in a group perform an operation or not at all. Examples:

- Reliable multicast: Operation = Delivery of a message
- Atomic multicast: Operation = Delivery of a message
- Distributed transaction: Operation = Commit transaction

Possible approaches

- Two phase commit (2PC)
- Three phase commit (3PC)

18.5.1 Two phase commit

Two phase commit is a distributed commit approach used in database systems which takes into account the agreement of all the processes in a group which have replicated database copies. This approach uses a coordinator and has two phases:

- Voting phase: Processes vote on whether to commit
- Decision phase: Actually commit or abort based on the previous voting phase

The algorithm for this approach can be explained using Fig 18.7.

- The coordinator first prepares or asks all the processes to vote if they want to abort or commit a transaction.
- All the processes vote. If they vote commit, they are ready to listen to the voting results.
- The coordinator collects all replies.
- If all the votes are to commit the transaction, the coordinator asks all processes to commit.
- All processes acknowledge the commit
- In case of even a single abort transaction vote including coordinator process's own abort vote, the coordinator asks all processes to abort.

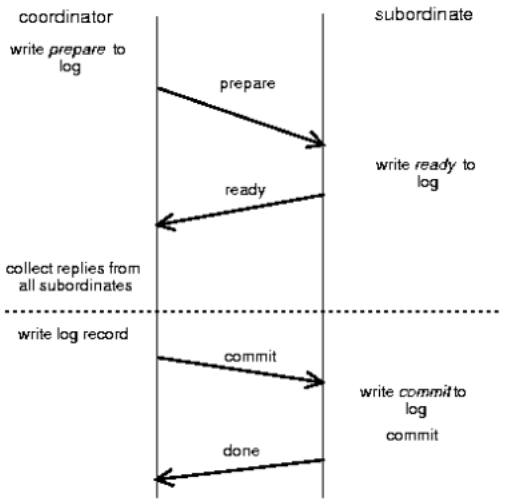
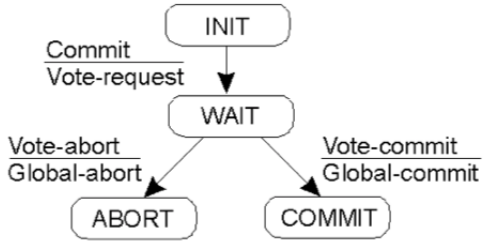
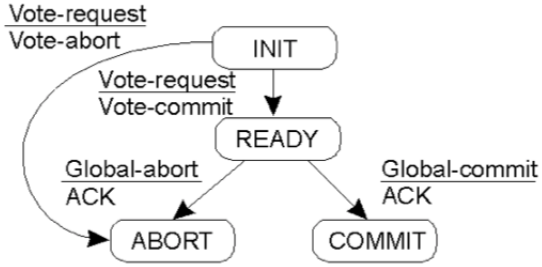


Figure 18.7: Steps showing a successful global commit using 2PC approach



(a) **2PC: Coordinator's state transition.**

- From INIT state, the coordinator asks all processes to vote and goes into WAIT state
- If any one process votes abort, the coordinator goes to ABORT state and issues global-abort.
- If all processes vote commit, coordinator goes in COMMIT and issues a global-commit.



(b) **2PC: Subordinate process's state transition**

- A process may vote abort and go directly into ABORT state. This is because this single abort would lead to global-abort.
- A process may vote commit and go into READY state.
- On being READY and receiving an abort, the process goes into ABORT state.
- On being READY and receiving a commit, the process goes into COMMIT state.

Question How the coordinator is chosen?
Ans. Leader Election.

Question If the process is voting for aborting, is the process up/down?
Ans. If the process is voting the assumption is that the process is up. If the process is down then there will not be any response. This scheme provides safety property but not liveness property. Drawback of two phase commit process is blocking when the coordinator crashes. If the process crashes, eventually the transaction aborts when the coordinator does not hear back from the process.

Question What if it takes long for the process to vote commit?
Ans. Process can vote to commit and coordinator makes decision to abort or to commit.

Question What if the process is byzantine faulty?
Ans. Two phase commit scheme does not work if the process is byzantine faulty. we are assuming crash fault tolerance in both two phase and three phase commit.

Question Is the global abort message sent by coordinator?
Ans. The result of the vote is always sent by the coordinator in decision phase.

Question When the global abort message is sent by coordinator?
Ans. If any process vote abort the coordinator sends global abort to all processes.

Recovering from a crash : When a process recovers from a crash, it may be in one of the following states:

- INIT : If the process recovers and is in INIT state, then abort locally and inform coordinator. This is safe to do since this process had not voted yet and hence coordinator would be waiting for its vote anyway.
- ABORT: The process being in ABORT state means that coordinator would have issued a global-abort based on the abort vote of this process, hence the process can safely stay in the state it is or move to INIT state.
- COMMIT: The process being in COMMIT state means the coordinator already had issued global commit and this process now can safely stay in this state or move to INIT state.
- READY: The process in this state may be due to a variety of possibilities hence as soon as any process recovers and finds itself in a READY state, it checks other processes for their state to get hint of the group status.
  The table describes the actions of recovered process P on seeing the state of a process Q and the reason for such action.

State of Q	Action by P	Reason
COMMIT	Make transition to COMMIT	Any process can be in commit only if coordinator issued a global-commit
ABORT	Make transition to ABORT	2 scenarios: • If process Q has aborted itself. Then coordinator would issue a global-abort. Hence, P can abort. • If process Q aborted because of a global-abort. P can abort in this case too. .
INIT	Make transition to ABORT	If process Q is in INIT means it has not voted yet. Thus, voting phase is still going on. Process P can abort safely.
READY	Contact another participant	Since, based on process Q's READY state, process P can't infer much. Hence, P should ask another process.

If process Q is in READY : Process Q being in READY state requires a further analysis of action:

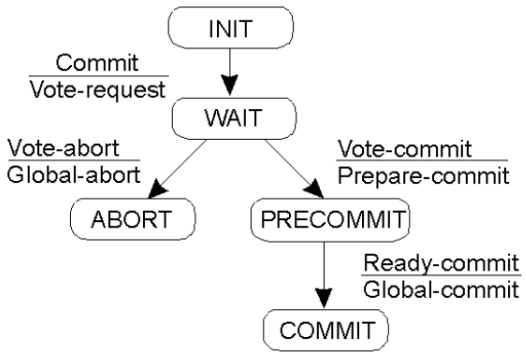
- Keep asking other processes about their state
- If at least one of them is not in the READY state then choose an appropriate action from the table above.

- If all of them are in the READY state and are waiting to hear from the coordinator, process P can't make a decision yet. All other processes can't make any decision either.
  The reason: Coordinator itself is a participant in the vote, hence, based on the action it takes after recovering, the option decided by the processes as a group may be wrong. That is:
  - All processes can't just commit because coordinator may recover and want to abort.
  - All processes can't just abort because coordinator may recover and see that every process had voted commit and want to commit and issue a global-commit. Other processes in abort state would lead to inconsistent state.

Problem of 2PC If the coordinator crashes without delivering the results of a vote, all processes will be deadlocked. This is called **blocking property of 2 phase commit**.

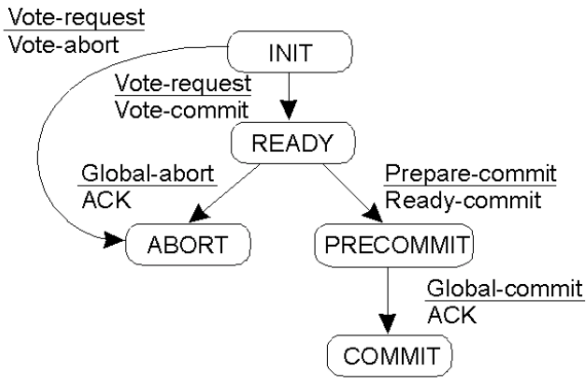
18.5.2 Three phase commit

Three phase commit is a variant of two phase commit which takes care of the coordinator crash.(blocking property of 2 phase commit)



(a) **3PC: Coordinator's state transition.**

- From INIT state, the coordinator asks all processes to vote and goes into WAIT
- If any process votes abort, the coordinator goes in ABORT and issues global-abort.
- If all processes vote commit, inform processes to prepare for commit. Go in the PRECOMMIT state.
- Once all the processes have moved to PRECOMMIT, then issue a global-commit and go into COMMIT state.



(b) **3PC: Subordinate process's state transition**

- A process may vote commit and go into READY state.
- It may vote abort and go directly into ABORT state. This is because this single abort would lead to global-abort.
- On being READY and receiving an abort, the process goes into ABORT state.
- On being READY and receiving a prepare-commit, the process goes into PRCOMMIT state.
- Once in PRECOMMIT, the processes move to COMMIT state on receiving global-commit.

How does the 3^{rd} phase PRECOMMIT help? :
Recollecting, blocking problem of 2-phase commit scenario. If a process recovers from a crash and finds itself to be in a READY state, it asks another process about its state. To this the reply is READY state. The processes are still to hear from the coordinator. If every process is in the ready state and the coordinator crashed and can't tell what the outcome of the vote is. A decision can't be made in case of 2PC. However, even in such a scenario a decision can be made safely in case of 3PC. Assuming the coordinator had gone into a PRECOMMIT state and crashed.
The processes can decide among themselves and ABORT.

- If the co-ordinator recovers and finds itself in the PRECOMMIT state, it could ABORT the transaction.
- In 2 phase this could not have been possible because the co-ordinator would have gone into COMMIT phase and rest of the processes would have ABORTed leading to an inconsistent state.

Question What happens if co-ordinator crashes after everyone is in pre-commit?
Ans. If every process is in PRECOMMIT and not in READY state, they can go ahead and COMMIT. This is because, once co-ordinator recovers, it may ask other processes for the state to which the reply would be COMMIT. This way co-ordinator can go in a COMMIT state as well.

18.6 Replication for Fault Tolerance

The basic idea is to use replication for the server and data among multiple nodes and is anyone fail we can still make progress.

- Technique 1: split incoming requests among replicas
  - If one replica fails, other replicas take over its load
  - Suitable for crash fault tolerance (each replica produces correct results when it is us).
- Technique 2: send each request to all replicas
  - Replicas vote on their results and take majority result
  - Suitable for BFT (a replica can produce wrong results)
- 2PC, 3PC, Paxos are techniques

18.7 Consensus, Agreement

Agreement vs Byzantine agreement vs Consensus :

- Agreement: Agreement of processes to do some task such as perform operation, or agreeing to commit or agreeing on a leader etc.
- Byzantine agreement: In the context of Byzantine General's problem - Coordinator or the source process takes a decision based on the vote by every process.
- Consensus: All processes act as a coordinator and ask all other process for a majority vote. This is a group decision and does not require a coordinating process. Also, go-ahead is based on majority votes and not all votes.

Properties of a consensus protocol with fail-stop failures :

- Agreement: every correct process agrees on same value
- Termination: every correct process decides some value
- Validity: If all propose v, all correct processes decides v
- Integrity: Every correct process decided at most one value and if it decides v, someone must have proposed v.

18.7.1 2PC, 3PC problems

Both have problems in presence of failures. **Safety** is ensured but **liveness** is not

Safety : Nothing bad would happen. Worst case, every process is deadlocked. But no wrong values in the operations.

Liveness : No blocking. There should be some progress even in adverse scenarios.

2PC :

- must wait for all nodes and coordinator to be up
- all nodes must vote
- coordinator must be up

3PC :

- handles coordinator failure
- but network partitions are still an issue

18.8 Paxos

Paxos lets nodes agree on same value despite: node failures, network failures and delays Majority rather than all nodes participate in the voting.

- Use cases:
  - Nodes agree X is primary (or leader)
  - Nodes agree Y is last operation (order operations)
- General approach:
  - One (or more) nodes decides to be leader (aka proposer)
  - Leader proposes a value and solicits acceptance from others
  - Leader announces result or tries again
- Widely used in real systems in major companies and proposed independently by Lamport and Liskov

18.8.1 Paxos Requirements

- Safety (Correctness)
  - All nodes agree on the same value
  - Agreed value X was proposed by some node
- Liveness (fault-tolerance)
  - If less than N/2 nodes fail, remaining nodes will eventually reach agreement
  - Liveness not guaranteed if steady stream of failures

- Why is agreement hard?
  - Network partitions
  - Leader crashes during solicitation or after deciding but before announcing results,
  - New leader proposes different value from already decided value,
  - More than one node becomes leader simultaneously

18.8.2 Paxos Setup

Entities : Proposer (leader), acceptor, learner

- Leader proposes value, solicits acceptance from acceptors
- Acceptors are nodes that want to agree; announce chosen value to learners

Proposals are ordered by proposal number :

- node can choose any high number to try to get proposal accepted
- An acceptor can accept multiple proposals
  - If prop with value v chosen, all higher proposals have value v

Each node maintains :

- n_a, v_a: highest proposal number and accepted value
- n_h : highest proposal number seen so far
- my_n: my proposal number in current Paxos

Question How does a process decide to be a proposer?
Ans. It is asynchronous. Any process can decide to be a proposer just like leader election, where any process can start the election algorithm

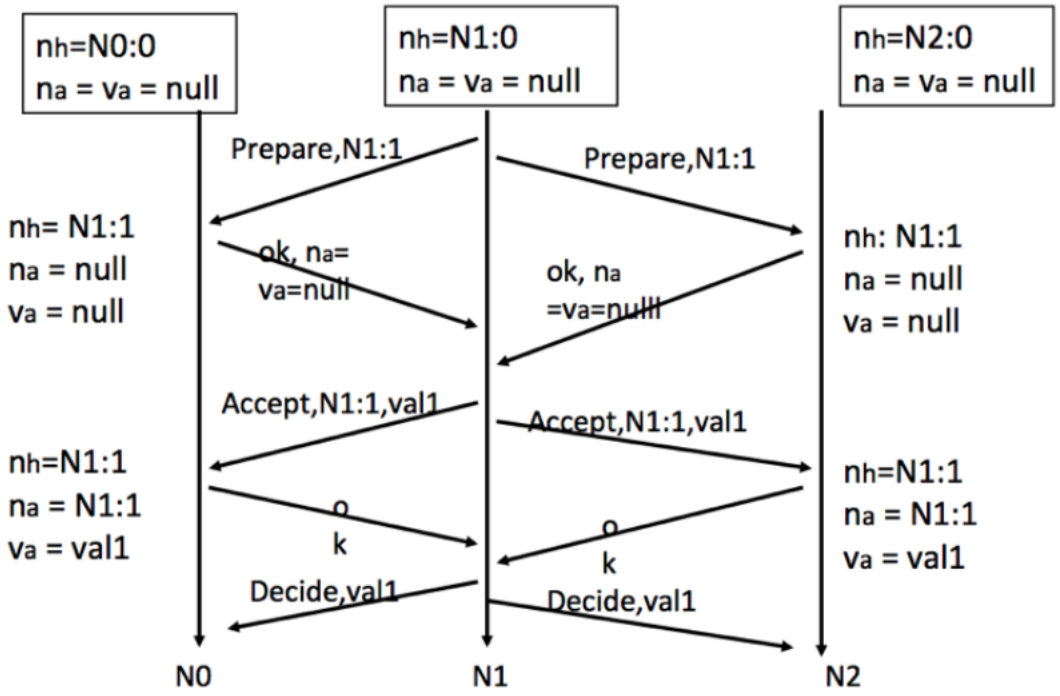
18.8.3 Paxos Operation

The 3 phases discussed in the class are given in the next page. Each variable's definition is given in the previous section.

- **Phase 1 (Prepare phase)**
  - A node decides to be a leader and propose
  - Leader chooses my_n > n_h
  - Leader sends <prepare, my_n> to all nodes
  - Upon receiving <prepare, n> at acceptor
    - If n < n_h
      - reply <prepare-reject> /* already seen higher # proposal */
    - Else
      - n_h = n /* will not accept prop lower than n */
      - reply <prepare-ok, n_a, v_a> /* send back previous prop, value/
      - /* can be null, if first */

- **Phase 2 (accept phase)**
  - If leader gets prepare-ok from **majority**
    - V = non-empty value from highest n_a received
    - If V = null, leader can pick any V
    - Send <accept, my_n, V> to all nodes
  - If leader fails to get majority prepare-ok
    - delay and restart Paxos
  - Upon receiving <accept, n, V>
    - If n < n_h
      - reply with <accept-reject>
    - else
      - n_a=n ; v_a = V; n_h = h; reply <accept-ok>

- **Phase 3 (decide)**
  - If leader gets accept-ok from majority
    - Send <decide, v_a> to all learners
  - If leader fails to get accept-ok from a majority
    - Delay and restart Paxos



(a) **This figure shows the Paxos operation.**

- The central line is a proposer and the other two are processes in the same group
- Prepare phase: Initially, it sends a <prepare, my_n> = <prepare, 1> to other processes.
- The acceptors upon receiving <prepare, 1> check if n >= n_h. Here they are, since acceptors have n = 1 and n_h = 0
- Prepare phase: They set their n_h = 1 and reply with <prepare-ok, n_a, v_a > = < ok, null, null >
- Accept phase: Leader or proposer got < prepare-ok > from majority. Here majority is 2.
- Accept phase: Leader sends < accept, my_n, V > = < accept, 1, val1 > to both the processes.
- Accept phase: Upon receiving < accept, 1, val1 > the processes first check if n ¿= n_h which is true, since n = 1, n_h = 1
- Accept phase: They set their values as n_a = 1, v_a = val1, n_h = 1 and reply with <accept-ok>
- Decide phase: Leader got <accept-ok> from majority and hence it sends < decide, v_a > = < decide, val1 > to all the processes

18.8.4 Issues

- Network partitions
  - With one partition, will have majority on one side and can come to agreement (if nobody fails)
- Timeouts
  - A node has max timeout for each message
  - Upon timeout, declare itself as leader and restart Paxos

- Two leaders
  - Either one leader is not able to decide (does not receive majority accept-oks since nodes see higher proposal from other leader) OR
  - one leader causes the other to use it value
- Leader failures: same as two leaders or timeout occurs