Lecture 18: April 12

Lecturer: Prashant Shenoy   *Scribe:* **Spoorthi Siri Malladi (2024), Anthony Micciche**

18.1   Overview

In previous lecture, we have started Fault tolerance, in this lecture we will continue it and cover following topics

- Agreement in presence of faults

- Reliable Communication

- Distributed Commit

18.2   Agreement in Faulty Systems

The two main type of faults are crash failures and Byzantine faults. Fault tolerance during crash failures allows us to deal with servers which crash silently. Detecting failures can be achieved by sending "heartbeat" messages. In a system where we only have silent faults, if the system has k faults simultaneously then we need $k+1$ nodes in total to reach agreement. In Byzantine faults, the server may produce arbitrary responses at arbitrary times. It needs higher degrees of replication to deal with these faults. To detect k byzantine faults, we need $2k+1$ processes. Byzantine faults are much more difficult to deal with.
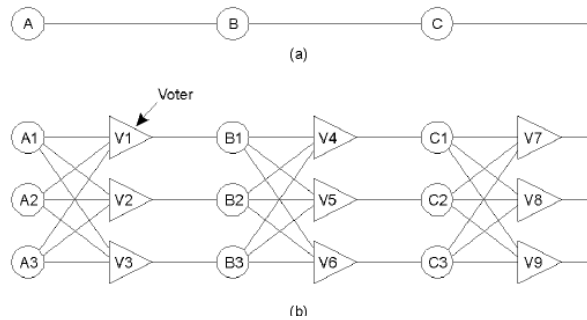

Figure 18.1: Failure masking by redundancy

Question: Is Figure 18.1 for Byzantine or not?
Answer: You can do Byzantine with it. Crashes are easy. Crashes can tolerate two faults, but you can do Byzantine as well.

18.2.1   Byzantine Faults

Let's explore two situations where the aim is to reach to an agreement on a one-bit message, Scenario-1: Two perfect process with faulty network (Two Army Problem) Scenario-2: faulty processes with perfect network (Byzantine generals problem)

18.2.1.1   Two Army Problem

In this problem, two armies in separate camps must agree on whether to attack a fort for the attack to succeed. They communicate via messengers, each sending a vote message ("attack" or "retreat") and waiting for an acknowledgment ("ack") from the other. However, enemy interference leads to unreliable communication, leaving the army uncertain if their messages or acknowledgments were received.

For example, suppose the first general sends an "attack" message. The second general receives the message and sends an "ack," but the messenger carrying the "ack" is killed. The first general never receives the ack, so they do not attack. The second general attacks, but the attack fails because the first general does not attack. Suppose instead the second general waits to receive an additional "ack" from the first general before attacking. But suppose the third messenger carrying the second "ack" is killed—now the first general will attack and the second general will not. So instead the first general waits for the second general to send a third "ack"... and so on, which leads to an infinite loop.

Summary : Two perfect processes can never reach an agreement in the presence of an unreliable network.
Question: If the network is faulty, can we not use cryptography?
Answer: We can use cryptography but we still cannot deal with an unreliable network that completely drops messages.
Question:In TCP, 2 ACKs are adequate, in two army problem why two acks are not sufficient ?
Answer: Firstly, TCP uses 1 ACK, secondly in TCP we are not trying to reach an agreement to coordinate something. To know if a message is delivered or not an ACK is sufficient, which is not the same case when we are trying to reach an agreement.
Question: Can you reach probabilistic agreement in the Byzantine general problem?
Answer: The network of communication in this problem is faulty, thus there is no such thing as probabilistic agreement. This problem cannot be solved if the underlying communication network is faulty.
Question: What does it mean to see if the network is reliable?
Answer: Messages are delivered and delivered correctly. For example, TCP does that for us. But, we don't know if the generals can be trusted or not.

18.2.1.2   Byzantine Generals Problem

In this problem N generals are trying to reach to an agreement with a perfect channel but M out of N generals are traitors.
Problem 1: Here $M=1$ and $N=4$. A recursive solution to the problem is provided in 18.2 In this, each node collects information from all other nodes and sends it back to all others so that each node now can see the view of the world from the perspective of other nodes. By simple voting, each node can decide on one correct value or spot if something's wrong, like a Byzantine failure/traitor.

Question: Does Figure 18.2 fail if the number of traitors outnumber the other generals? **Answer**: Yes.
Question: In problem 1, what exactly we do in round 2?
Answer: Generals broadcast all the information they have from every other general in the previous round.
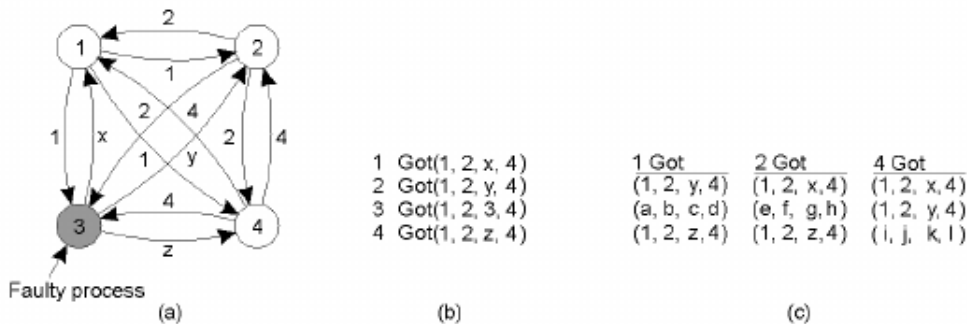**Question**: What happens if a fault process send same incorrect message in round 2?

Figure 18.2: Solution to the byzantine general's problem-1, 1 traitor out 4 generals over a reliable communication network. a)The generals announce their troop strengths b)The vectors after each general assembles after round 1 c)The vectors that each general receives after round 2.

Answer: If that is the case, it is acting like a normal process and then it won't be identified or isolated. However if, instead of broadcasting their strengths, they perform an addition task, such as computing and returning the value of 2+2, any incorrect answer would be detected in this scenario.

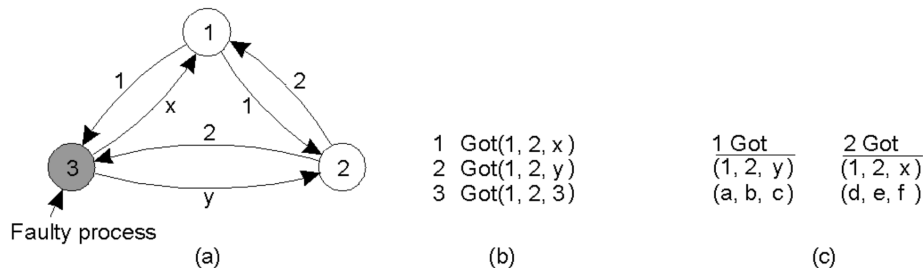problem 2 : Here $M = 1$ and $N = 3$. We do same steps as problem 1 in 18.3

Figure 18.3: byzantine general's problem-2, 1 traitor out 3 generals over a reliable communication network.

In this case we can detect the fault but cannot isolate it, in order to isolate the faulty process we need one more process which functions correctly.

In a system with k such faults, $2k+1$ total nodes are needed to only detect that fault is present, while $3k+1$ total nodes are needed to reach agreement, despite the faults. Therefore, agreement is possible only if $2k+1$ processes function correctly out of $3k+1$ total processes.

18.2.2 Reaching Agreement

If message delivery is unbounded, no agreement can be reached even if one process fails and slow processes are indistinguishable from a faulty ones. If the processes 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 Communication

18.3.1   Reliable One-To-One Communication

One-one communication involves communication between a client process and a server process whose semantics we have already discussed during RPCs, RMIs, etc. In this we only discussed one-to-one communication, but here we are discussing replication. We need one-to-many communication (multicast or broadcast) in order to reach agreement. We need to extend the one-to-one scenario to the many-to-one scenario in order to solve the agreement problem. Figure 18.4 depicts several failure modes in the one-to-one scenario. These failures can be dealt by (1) Using reliable transport protocols such as TCP (b and d can be dealt with in this manner), or (2) handling failures at the application layer. (a, c and e can be dealt with in this manner)
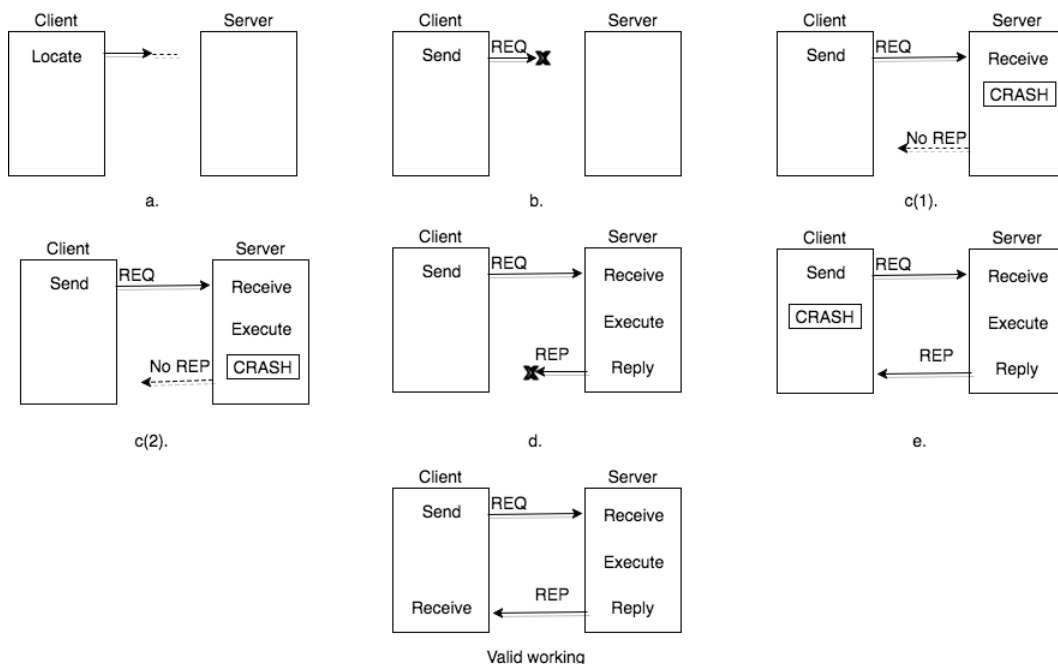

Figure 18.4: Types of failures in the one-to-one scenario. (a) Client unable to locate server. (b) Lost request messages. (c) Server crashes after receiving request. (d) Lost reply messages. (e) Client crashes after sending request.

18.3.2   Reliable One-To-Many Communication

Broadcast is sending a message to all nodes in a network. Multicast is sending to a subset of all nodes.

If there are lost messages due to network inconsistencies, we need to retransmit messages after a timeout. There are two ways to do this: ACK-based schemes and 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 retransmits 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 an **ACK explosion**.
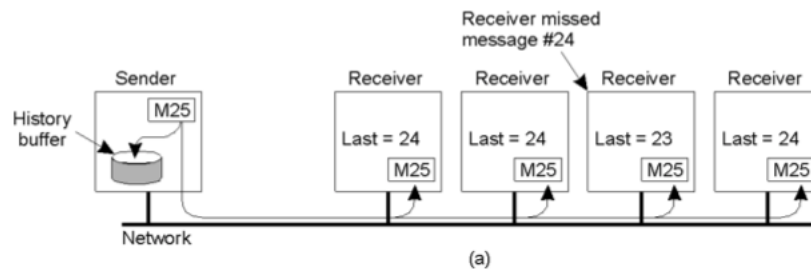

Figure 18.5: Here, all the receivers have their last packet received as #24 except receiver 3 which missed packet #24. Hence, it's last packet is #23. As soon as it receives packet #25, it knows it missed the packet #24.
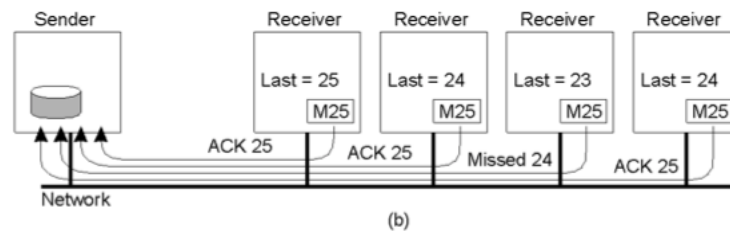

Figure 18.6: Each receiver now sends an acknowledgment ACK either in form of received packet #25 or missed packet #24. As we can see for a single packet, sender receives 'n' ACKs

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 the 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.

- Much more scalable than ACK-based schemes

- Effective only in networks with occasional drop-offs and is not suitable for highly lossy networks.
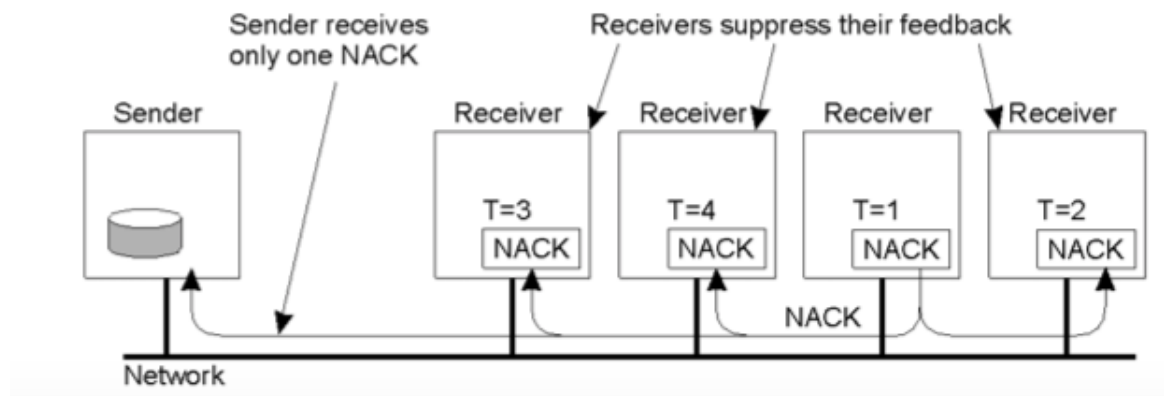
Figure 18.7: Each receiver now suppresses their ACK feedback. Only receiver 3 sends a NACK to other receivers and the sender.

Question Are messages not queued at each receiver and delivered in the same sequence? If a message is missed, can't we request it from one of the receivers?
Ans. Yes, we maintain a buffer to order and deliver messages sequentially. In case of a missed message, we send a NACK to both the sender and a subset of nearby receivers.
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.

Question Is a NACK-based part of TCP protocol or part of the higher level application?
Ans. You can ask that question of multicast itself. There are two versions of multicast. IP multicast where an IP address is a group address, and sending to that group IP send to the entire group. That is at the network level. You can also implement this at the application level. The NACK based scheme is mostly done at an application level.

Question Can you implement it on top of UDP
Ans. Yes. Application level multicast is implemented internally effectively as n unicast messages. Network level is just one socket, and it goes to n receivers.

Question Is it like a pub sub architecture?
Ans. Not exactly. This is more level than publish and subscribe. This is an abstraction one-to-one communication. You can build publish subscribe on top of this.

This scheme, only addresses how to send a message to all members of the group, it does not discuss other properties of multicast like:

- FIFO order: Messages will be delivered in the same order that they are sent.

- Total order: All processes receive messages in the same order. Total order does not require FIFO.

- Causal order: It is based on the happens before relationship. If send(m1) happens before send(m2), then the receive(m1) should also happen before receive(m2) between processes.

18.3.3 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.

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.
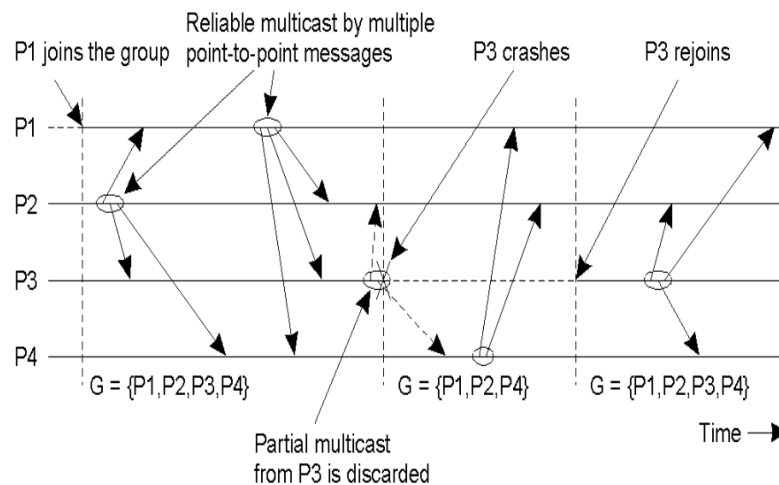

Figure 18.8: Initially all process are up and are part of a group {P1,P2,P3,P4}. All the messages are being reliable multicasted to each of the processes. At dotted line2, P3 crashes while sending a message. From this point onwards, the group {P1,P2,P3,P4} will not maintain the 'all' property of atomic 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.

18.3.4   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 18.9:

Question: How do we know which processes are up and which are crashed?
Answer: To detect which nodes are up and which are crashed, we can implement several procedures like heartbeat messages to know the status of nodes. These are crash faults not byzantine, so they are easy to track.

Question: Why does atomic multicast have total ordering ?

Answer: Atomicity is a stronger property than total order, it is more expensive. Thus the system may as well have total order if it has atomicity.

Question: What happens in the scenario where some processes receive the message and other do not receive the message ?

Answer: There is a difference between receiving or delivering a message and applying/comitting the message. The commit of a message should take place only after consensus in order to ensure safety, this is discussed later on in the lecture.

Question: Is causal ordering stricter than FIFO?

Answer: Yes, FIFO only ensures ordering within a process whereas, causal ordering ensures ordering across processes.

18.4 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)

18.4.1 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 18.10

- 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.

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.
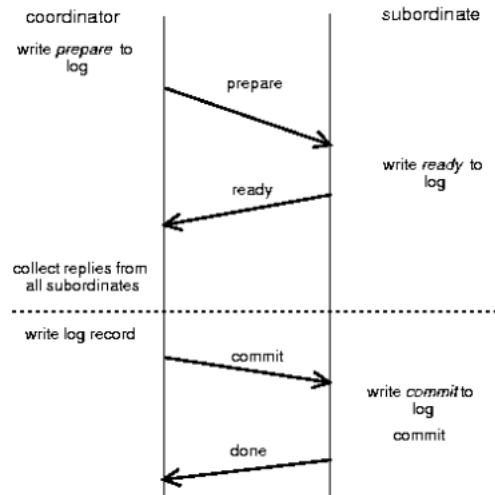
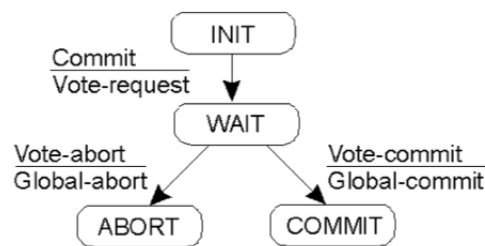Figure 18.10: Steps showing a successful global commit using 2PC approach

Figure 18.11: **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.

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.
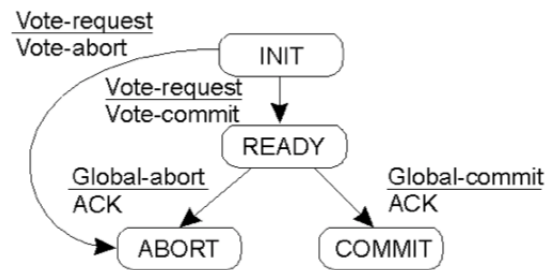
Figure 18.12: **2PC: Subordinate process's state transition.** A process may vote abort and go directly into ABORT state. This is because this single abort would lead to global-abort. A process may vote commit and go into READY state. On being READY and receiving an abort, the process goes into ABORT state. On being READY and receiving a commit, the process goes into COMMIT state.

- 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**.

Question: Is it feasible to discard coordinator and elect a new coordinator in case of deadlock ?
Answer: we can do that in subsequent process but it is still a deadlock for current process.

Question: If the co ordinator has send messages to some processes and not all and then it crashes then what happens ?
Answer: Two properties need to be discussed to answer this, safety and livenss. Safety: Nothing bad happens, the protocol does not reach an incorrect decision. Liveness: There is progress, the protocol reaches a decision. The 2pc guarantees safety and not liveness.

Question: what happens if a node crashes after it works to commit?
Answer: Process upon restarting/re-initialisation needs to check it's last state and make a decision.E.g: If it's in init state, that means that it did not vote for the operation otherwise it would have been in the ready state, in this case, it's best to abort and inform coordinator about the decision so it can make progress.

If the process was re-initialised with a "ready" last state, then it means that it voted before crashing but doesn't know the result, so it needs to check other process queues for that operation and figure out the decision.