CMPSCI 677 Operating Systems **Spring 2025**

Lecture 19: April 14

Lecturer: Prashant Shenoy Scribe: Manan Parikh(2025), Yiming Huang(2024), Aishwarya Bhave(2023)

Lecture Overview

- Part 1: Consensus
- Part 2: PAXOS
- Part 3 : RAFT

19.1 Consensus

Definition: get a group of processes to agree on something such as database replication etc. Consensus also means getting the set of processes to agree when some of the processes fail via atomic. More formally, we want to achieve reliability in presence of faulty processes:

- Requires processes to agree on data value needed for computation.
- **Examples**: The operation could be anything - whether to commit a transaction, agree on identity of a leader, atomic multicasts, atomic broadcasts, distributed locks.

The failures are in context of crash faults or Fail-stop failures i.e., a process produces correct output while it is running, but the process can hang/go-down and hence will not produce any results. **Note**: When there are no failures, there are protocols like 2 Phase commits that we discussed, to come to an agreement. For instance, committing a transaction.

Byzantine Consensus vs Consensus Byzantine consensus is when we need to come to an agreement in case of processes that are byzantine-faulty i.e., faulty processes continue to run and produce malicious outputs and prevent agreement. Whereas consensus is a benign scenario where some processes fail to respond.

Q: How to decide what consensus protocol to use? Depends on what we're trying to achieve. PAXOS, RAFT can be used for crash faults, for implementing a basic fault tolerance mechanism. Byzantine is more elaborate and used in case where we do not want malicious actors to cause confusion as in case of crypto-currency.

19.1.1 Properties of a Consensus Protocol

- **Agreement**: Every correct process *agrees* on the same value. (fundamental to consensus)
- **Termination**: Every correct process *decides* on some value.
- **Validity**: If all processes *propose* a value(v), all correct processes must *decide* on that value, v.

- **Integrity**:
  - Every correct process *decides* at most one value.
  - If a correct process *decides* on a value, a process must have *proposed* that value.

Q: What does 'all' in validity mean? We will have failures, if any process or say the co-ordinator crashes during agreement, we will not have consensus. But the protocols define that, if we have a majority(and not all) of the nodes up and running and they agree on a value, we have consensus.

Q: Difference between agreement and termination A safety and liveness question, Agreement can agree on null value, it's a safety property, and termination means we make agreement on actual value, so it's a liveness property, we need both for the system to go on.

19.1.2 2PC/3PC Problems

Both two phase commits and three phase commits experience problems in the presence of different types of failures. While the **safety** property can be ensured, the **liveness** properties cannot always be guaranteed due to node failures and network failures: the system will never perform an operation that leads to an inconsistent state (satisfying the safety property) but can still be deadlocked (violating the liveness property). We describe a few caveats associated with each type of commit below.

Two Phase Commit

- It must wait for the coordinator and the subordinates to be running.
- It requires all nodes to vote.
- It requires the coordinator to always be running.
- Either all commit or no one commit.

Three Phase Commit

- It can handle coordinator failures.
- But network failures are still a problem.
- Add an additional stage of pre-commit compared to 2PC when the coordinator receive the commit, and if the coordinator says commit again, all the nodes move to commit stage.

3PC never get confused for abort and commit because of the existence of pre-commit stage. There has been an implicit assumption that there could only be node failures instead of network failures during a two or three phase commit. While a node could crash in the network, the network would never experience any issues. Suppose, however, that the network was partitioned into two partitions due to some problem. Although both partitions will continue to function correctly, each partition cannot communicate with each other. **By definition, if the network is partitioned due to some problem, a two or three phase commit cannot work because every node is required to vote on the answer.**

In order to eliminate such an assumption, we have to revisit the definition of *agreement*. Rather than requiring the vote of every node, we can just require the vote of the majority of nodes. Therefore, if the network were to be separated into two partitions, the partition with the majority of nodes can still continue

to function properly. This idea forms the basis of **Paxos**, a consensus protocol. **Instead of requiring every node to vote, Paxos only requires the majority of nodes to vote.**

Replication for Fault Tolerance Technique 1: split all incoming requests among replicas. (If one replica fails, other replicas take over its load) Technique 2: send each request to all replicas. use 2PC, 3PC, Paxos, a replica can produce wrong results

Question: What if the coordinator fails after the precommit state? Answer: In the precommit state, the coordinator has sent the precommit message (indicating a decision to commit) to all participants. If at least one participant has received this message, they will have transitioned to the precommit state. Others that have not yet received it remain in the ready state. If a participant in the ready state discovers that another participant is in the precommit state, it can safely assume that the coordinator sent out the precommit message, which is why at least one participant is in precommit state. Therefore, all participants can safely transition to the precommit state and eventually commit the transaction, even without the coordinator. However, it's also possible that the coordinator entered the precommit state but failed before sending the message to any participant. In that case, all participants remain in the ready state, waiting for the coordinator's decision. After a timeout, if none of them has received the precommit message, they can safely abort the transaction. When the coordinator eventually recovers, it sees that the participants have aborted, and it can safely transition to the abort state as well.

Question: Does the approach discussed in above answer helps if cooridinator crashes? Answer: This approach does not help if process crashes. Because, if process crashes, then coordinator does not receive a vote from a participant, and coordinator will be stuck, because in order to send a decision, it needs to receive votes from all the participants, it cannot go on majority, everyone must vote. So the approach discussed in above answer helps when coordinator/participant crashes, but not when process crashes.

Question: What happens after you reach precommit? Answer: First coordinator sends a message to all the participants to go to precommit state, once it has received acknowledgement from all the participants, it will send another message for global commit.

19.2 Paxos: Fault-tolerant agreement

Paxos lets nodes agree on the same value despite node failures, network failures and network delays. but cannot deal with Byzantine Fault **Use-cases include:**

- Nodes agree X is primary (or leader)
- Nodes agree Y is last operation (order operations)

The protocol is widely used in real systems such as Zookeeper, Chubby and Spanner. ***Leader*** is a process that tries to get other processes to agree on a value. For instance, a process says, I propose that the value after computation is X and gets other process to agree that the output after computation is X. Therefore, leader is essentially a proposer. If majority of the processes agree then, the value is agreed upon. If not, then either the leader tries again or some other process becomes a leader and attempts consensus. **Note:** There can be multiple leaders and can attempt to get others to agree on a value.

Question: what method would you want to use this as opposed to other approaches (2PC/3PC) ? *Answer*: This can be used in case of failures in the node as well as multiple leader failures. Since it is a quoram based protocol, it allows leader election even if some processes fail as long as majority of them are up.

19.2.1 Paxos Requirements

Paxos satisfies the following properties:

- Safety (*Correctness*)
  - All nodes must agree on the same value.
  - The agreed upon value must be computed by some node.
  - **Note:** We do not want just trivial consistency i.e.; everyone agrees value is zero or null. Therefore, the value that is agreed upon must be computed by some node.
- Liveness (*Fault Tolerance*)
  - If less than $\frac{n}{2}$ nodes fail, the remaining nodes will eventually reach agreement. This allows the system to make progress in the presence of failures.
  - Note that that liveness is not guaranteed if there is a steady stream of failures as the protocol determines what to do. If a node fails in the middle of the protocol, it must be restarted.
- **Why is agreement hard?** Because even in the face of failures, we still need to reach agreement.
  - The network might be partitioned.
  - The leader may crash during solicitation or before announcing the outcome of voting. While the current round will not produce any results, a new leader will be elected through leader election. All nodes will then vote again.
  - A new leader may propose different values from the value that had been agreed upon originally.
  - Several nodes may become a leader at the same time. This is possible when the network is partitioned due to a network failure. The left half will elect a new leader while the right half will have the old leader, and they will still continue to function properly. Both sides of the partition may agree on different things unfortunately.

19.2.2 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
  - *Learners* do not play an active role, but agree on proposed value.
- Proposals are ordered by unique proposal numbers.
  - Node can choose any high number to try and get proposal accepted
  - An acceptor can accept multiple proposals.
    - If a proposal with value v is chosen, all higher proposals have value v.
- Each node maintains:
  - **n_a, v_a:** The highest proposal number and accepted value during that proposal.
  - **n_h:** The highest proposal number seen so far
  - **my_n:** the current proposal number that is in progress.

19.2.3 Paxos Operation : 3 Phase protocol

Phase 1: Prepare Phase Leader understands what other processes have seen or accepted before.

- A node decides to be leader and proposes a value
- Leader chooses my_n >n_h
- Leader sends **<prepare, my_n>** to all nodes. **Note that,** during this, the value proposed is not sent, it's just the prepare message with proposal number.
- Upon receiving **<prepare, n>** at acceptor:
  - If n <n_h: Reply with <**prepare-reject**. (Since, already seen a higher # proposal.)
  - Else:
    - n_h = n (Protocol will not accept proposal lower than n)
    - Reply **<prepare-ok, n_a, v_a >.** (Send back the most recently accepted proposal # and value)
    - Reply can be null, if you haven't seen any proposals yet and this is the first proposal.

Phase 2: Accept Phase

- If leader gets **<prepare-ok>** from majority *(Actions taken by leader)*
  - V = non empty value from the highest n_a received from prepare phase.
  - 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>*(Actions taken by acceptor)*:
  - 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.

Question: Are we assuming the number of nodes is fixed? Can new nodes join? *Answer*: It is not sure new nodes can join since we are assuming because as mentioned previously here that if you have a steady stream of failures and recovery liveness is not guaranteed so we can have failures but then if new nodes are joining and they suddenly start saying something that they're in participant participate that's a problem then that round will fail the nodes can join but that round will failure to return. It may also cause problems in majority voting.

Q: Can Proposals go on indefinitely? At the beginning, no one has agreed to anything, leader gets null and chooses a value V. Another proposer suggests a value and it gets accepted and so on. Essentially the value will not change and this is similar to electing the same leader over and over again. While anyone can start a proposal at any time, the agreed value will not get affected. However, the phase 3 or decide phase cannot happen if a new proposal with higher proposal number has started making rounds. Nodes may decide to reject the proposal and accept a new one. And this is possible since we can have multiple leaders. Therefore, there must be a gap between decide phase and new proposals for decide phase to happen. To re-iterate, this doesn't change the value however.

Q: What if you have same proposal numbers? Proposal numbers are unique, Paxos will not work if two proposals have same number. We can append PID (process id) to make it unique. This is similar to Lamport's clock ordering to convert partially ordered to fully-ordered events where we append process id.

Properties

- Property 1: any proposal number is unique.
- Property 2: two sets of acceptors have at least one node in common
- Property 3: value sent in phase 2 is value of the highest numbered proposal received in responses in phase 1.

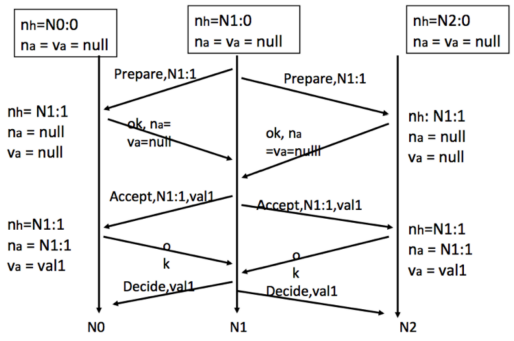

Figure 19.1: Example of Paxos with 3 servers

An example with three nodes namely N0, N1, N2 where N1 is the proposer:

- Prepare Phase:
  - N1 sends prepare messages to N0 and N2 i.e. <prepare, N1:1>where 1 is the proposal number and not the value we are trying to get consensus on.
  - N0 and N2 haven't seen any proposals before so send <prepareok, n_a = null, v_a = null>to N1.
- Accept Phase:
  - Values received at N1 after prepare phase are null, so N1 decides on val1 as accepted value and sends accept messages to N0 and N2 as <accept, N1:1, val1>
  - N0 and N2 send <accept-ok>to N1.

  - Decide Phase:
    * N1 sends <decide, val1>to N0, N2.

When we have one leader, the protocol converges easily. But say N0 decides to become a leader while N1 is trying to get consensus as the proposer, the proposal from N0 will get discarded as a new proposal with higher proposal number is now available. Learners/Acceptors can choose to agree to the new proposal.

Issues :

- Network Partitions: For a network that has an odd-partition, if there is majority on one side, nodes can come to an agreement whereas they cannot if network is evenly partitioned.
- Timeout:
  - A node has max timeout for each message
  - Upon timeout, it declares itself as leader and restart Paxos
- Two Leaders:
  - Either a leader was not able to execute decide phase (due to lack of majority accept-oks as nodes encountered a higher proposal from other leader) OR,
  - One leader causes the other leader to use its value.
- Leader Failures: This case is same as two leaders or a timeout where a node will decide to become the leader and restart Paxos.

Q: How can there be two source of truth on network partition: In Network partition, the majority can make decision, but the minority cannot make a decision since it cannot get majority of nodes to agree with.

Q: What if a node fails, and by the time it gets rebooted, the proposal has already been accepted, and decision has been made?: That node can catch up with any other node, to get all the updates.

Q: Let's say there are 3 followers, If leader gets one response value 2, and other 2 null, why leader would accept value 2 and not null?: Paxos does not make decision on what majorly has been reported to the leader, majority should be from participating nodes, so as long as there is only one live participant, leader can accept that value instead of accepting null values from failed nodes from which it got null values.

Q: Are there any rules on how many replicas you need?: At least 2 in Paxos. If you have n replicas, you need strictly greater than half of them to be up.

Q: What happens if the decision fails in the decision phase?: At any point, if the leader does not receive acknowledgment, it will have to restart.

Q: Is it possible both partitions in PAXOS does not have majority?: Yes, you might have some nodes that just crashed. Consider example of 10 nodes, 5 of them crashed, 2 of them ended on one side, and another 3 ended on another partition. In this case, there is no majority and progress.

19.3 RAFT Consensus Protocol : understandable consensus protocol

The RAFT protocol is based on how a part-time parliament functions. A parliament is able to pass laws despite some members being out of attendance, or members showing up to the parliament at different times. It reaches consensus despite attendance (read failures, in case of processes).

Raft uses replicated logs or State Machine Replication (SMR) to implement the protocol. Assume we have n servers and each server stores a replica of log of commands and executes them in that order.

How do we replicate logs in multiple places while keeping the order consistent? Raft implements a leader election protocol. All incoming requests then go to the leader and it decides the order of execution and informs everyone, as opposed to sending each request to everyone and then deciding on an order. Therefore, we need to elect a responsible leader. And if leader fails, we elect a new one and clean the logs to ensure consistency. We must note that if we have majority i.e. N/2 +1 nodes, consensus can be reached, otherwise it cannot. Also, if an entry is committed, all entries preceding it are committed.

Note : All the metadata such as who was the leader node, term number etc along with the log vaule needs to match for a logs between process and majority to be considered equivalent.

Log Replication Example: In case of three servers, the request z = 6 goes to the presumed leader. Leader writes it in log file and sends prompt to other nodes to append it to their logs. The consensus module ensures that the order is maintained. Every committed request is executed. The value needs to first be appended and then committed to the logs.

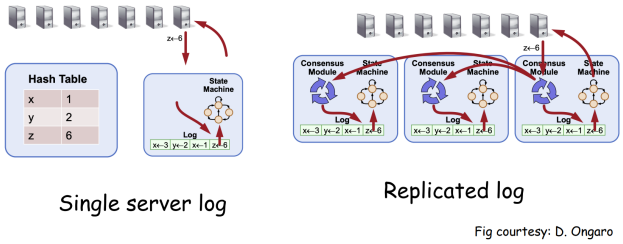

Figure 19.2: Example of Log replication

Consensus approaches:

- Leaderless/Symmetric: Client can send the request to any server and that server decides the order of execution.
- Leader-based/ Asymmetric: One server becomes leader and tells followers what to do. Raft is a leader-based consensus protocol

Overview of RAFT operations

- Leader election: Nodes must select one server to serve as RAFT Leader. There must be provision to detect leader crash and provision to elect a new leader in case of a crash.

- Normal operation: This involves performing log replication, leader receiving client commands, appending incoming requests to log. Leader then replicates log to followers. We must ensure safety i.e., committed logs must not get impacted by leader crash and there must be at most one leader at a time.

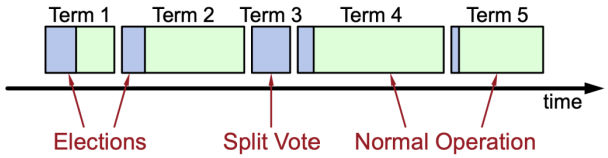

Figure 19.3: Terms

Terms:

- Time is divided into terms, a period when a certain node acts as the leader. Term does not change unless the leader crashes/fails.
- Each term has a blue followed by a green part. Blue parts represent leader election, green represents normal operation. If a term has only blue (a failed term), it represents a split vote or no majority to elect a node as the leader.
- All servers maintain the current term value.
- At any time, each server can be either of the three:
  - Leader: receives all client requests and does log replication
  - Follower: passively follows leader
  - Candidate: a node that participates in leader election

Q: What is a split: in network partition, multiple followers become candidates and no one of them can get elected, we need to restart the election.

RAFT Election :

- Election timeout: Communication is over RPCs and if no RPCs are received for a while from the leader, then increment current term and become a candidate.
- Elections are selfish. On an election timeout, candidate node votes for self to become a leader and sends an election message (RequestVote RPC) to followers.
  - If the node receives vote from majority, it becomes the leader and sends heartbeat message (AppendEntries RPC) to inform other nodes.
  - Failed election: If no majority votes are received within election timeout, the term gets incremented and a new election starts.
- Safety in election: In any election, at most one server wins since you can only cast your vote once per term. Also, there is random back-off in case of a failed election i.e. each node backs off for different amount of time. This ensures that some node starts the leader election and wins majority, while other candidates are in timeout.
- Liveness: One of the nodes will win the leader election.

Normal RAFT Operation

- Leader receives client commands and appends them to log.
- Each log entry has 3 things: Index (item no. in the log), term (current term value), command.
- Leader sends AppendEntry RPC to all followers.
- Once an entry is safely committed to log (i.e. leader got a majority vote for AppendEntry RPCs sent), the command is then executed and results are sent to the client.
- Committed entries are notified to followers in subsequent RPCs therefore the followers catch up in background. The followers apply the committed commands to their state machines.

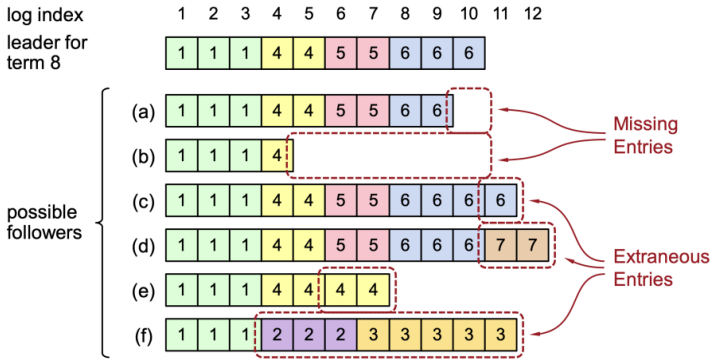

Figure 19.4: Inconsistencies in Logs Example

Log Consistency To verify if logs are consistent, leader informs the followers what the previous entry (index, term) in the log was. If the previous entry at the follower and the one sent by the leader do not match, then we know there is inconsistency. Log entries can become inconsistent due to leader failure.

- There can be missing entries as in the case of (a) and (b) followers. Possible causes can be a network partition or failure of those follower nodes when the entries came in.
- There can be extraneous entries as in the case of followers c, d, e and f. This can be because of leader partition, and some other nodes got new requests that haven't yet been committed.

The leader must synchronize the logs to ensure consistency by adding required entries to the missing ones and scrubbing extraneous entries by using pre-fix match. **Note:** These are all entries that have been appended to logs but not committed.

Log Repair The leader tracks nextIndex for each follower. It asks the follower if it has the entry at an index (index of last entry in leader's log) in its log. If the follower doesn't, the nextIndex decrements until a matching entry is found. All missing entries from this point onwards are sent to follower to catch up. In case of extraneous, the subsequent entries from index where we found the match at are deleted and leader replays the rest of the logs for follower to catch up on.

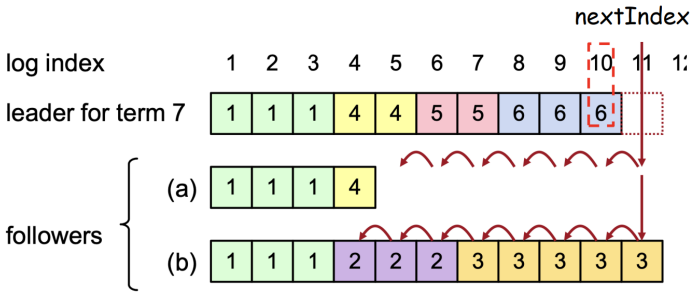

Figure 19.5: Log Repair Example

Leader crashed and some other node becomes the leader, how do we ensure consistency in this scenario? We check the committed entries until the time of crash and use that to ensure ordering.

When is consensus achieved in RAFT? Consensus is achieved when majority of the nodes have appended and committed the entries. To have consensus means we have agreed to commit to an order.

When does the commit actually happen? In Normal Operation, if majority of followers agree to append, then you commit the log.

During election, who ensures log is collected? The clients cannot send requests during election since there is no leader. Requests can be sent only after a leader is elected.

If the leader crashes while committing logs, what happens? RAFT has a way to handle this, TBA on piazza.

Q: Is there there any situation when we have match entry, but un-match entry before: Because the prefix always matches, and we always repair the log to make sure we have the prefix always matches, so there is no case when there will be entries that are not match before the latest matching entry

Q: Why using RPC: No specific reason. Can use different protocol

Question: Why do we have these extraneous entries at all ?

Answer: Assume there was a green leader at some point in time it produced three entries and then maybe there was a network partition , so that green leader and some nodes got disconnected from the majority and before we could realize that maybe that green leader sent an extra entry to this follower but it can't send it to the remaining majority but they are already in an unreachable so they will then what they will do is they will elect their own new leader which in this case was yellow and blue and so on so in the network rejoins you will see that some bad things may happen to the minority because before they realize that something had gone wrong they had already written some entries to their law okay and you've got to repair them.

Question: What if the step that is extraneous was a really important operation and we deleted them how is whoever made that request is going to figure this out?

Answer: In this case, because the leader got disconnected, it couldn't gather the majority vote and hence it never replied to client with a success response. So the client so the request will simply timeout or fail.

Question: Can you ask for logs after particular timestamp?

Answer: Yes.

Question: If a follower has bad entries in their log, why are we even discarding it?

Answer: The goal is to take leader's entry and replicate it in follower's. That's why unnecessary entries should be discarded.

Question: Is there a possibility of having an extraneous entry and then a correct entry?

Answer: Technically, it shouldn't be the case in which middle of the log is wrong. So prefix of the entries will always be the correct. Problem can be missing entries in the end or having some extra entries in the end, never something wrong in the middle. It is assumed that log has been repaired before making any changes (adding new entries or removing extra ones).

Question: How do follower knows, at what point it has to traverse back?

Answer: Leader can go to the follower and ask for an index. So leader can check for that index whether it is matching or not, if not, it will keep going back until leader finds the matching entry.

Question: Is repairing logs supposed to be expensive operation?

Answer: It's a linear operation.

Question: Is there an optimization possible, in case if log size is small, so that instead of sending a lot of messages, just send the whole log?

Answer: That's not what RAFT does, but it is optimization for sure that can be implemented.

19.4 Recovery :

We have discussed techniques thus far that allow for failure handling, but how recovery dictates how those failed nodes come back up and recover to the correct state. The techniques include periodic checkpointing of states and roll-back to a previous checkpoint with a consistent state in case of a crash.

- Independent Checkpointing
  - Each process periodically checkpoints independently of other processes.
  - Upon failure, work backwards to locate a consistent cut, last checkpoint.
- Logging
  - Is a common approach to handle failures in databases, file-systems.
  - Done by logging and re-playing logs.

Trade-offs between checkpointing and logging: *Checkpointing* doesn't need logs, it saves system state that can be used as last consistent state. This is expensive since we are writing entire system state to disk. But recovery is quick in case of checkpointing, since we are loading the system values from a file essentially. Whereas in *logging*, the logs have to be replayed/executed again from the point of failure. Adding logs to a file is cheap, but it is expensive in terms of recovery as in the case of processes being behind by a lot and all the missed logs have to be executed again. We can combine the two as well.

- Take infrequent checkpoints
- Log all messages between checkpoints to local stable storage.
- To recover: replay messages from previous checkpoint. This avoids re-computations from previous checkpoint.