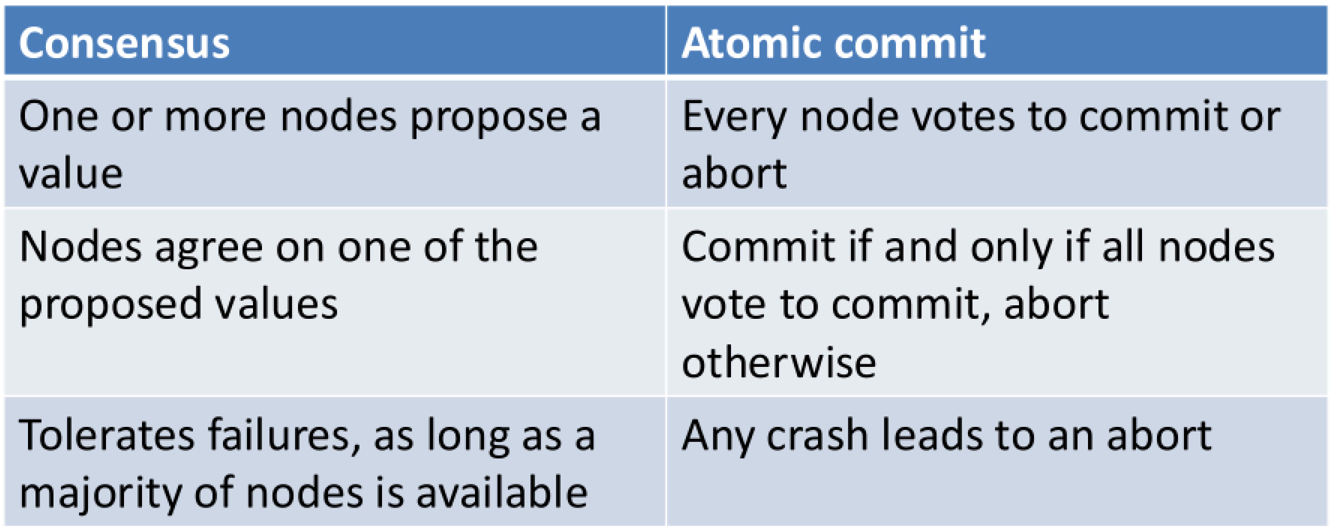
07.Distributed Agreement in Practice

If you have a majority of nodes up and working correctly you need to have a protocol to agree on a given topic(numbers).

COMMIT PROTOCOLS

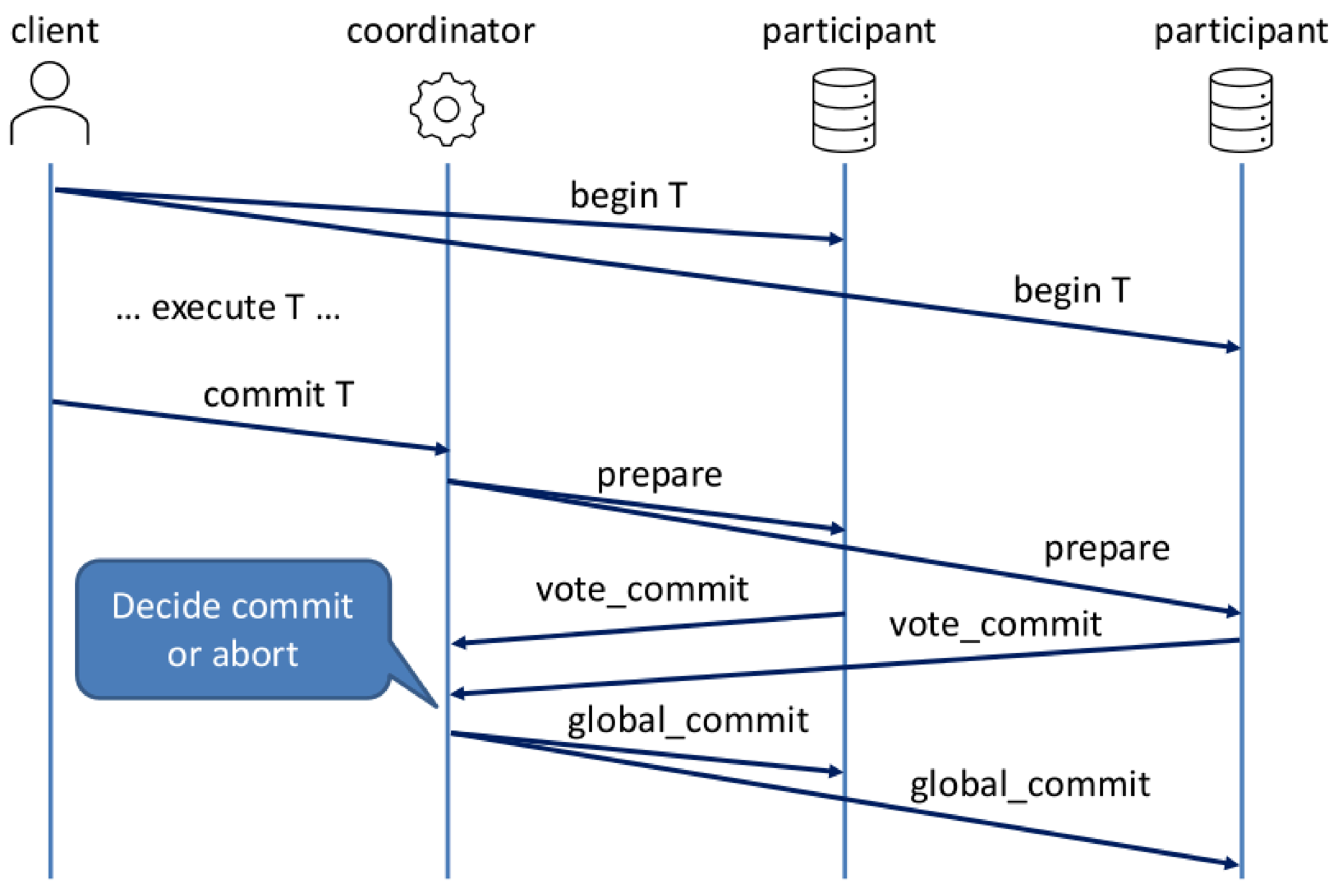
You have a database partitioned on multiple nodes. People can send requests for something and you want to reach an agreement for the transaction to be successful that is to have the information on multiple nodes to be reachable. If one node abort you abort the transaction on all nodes. This ensure the Atomicity property.
Atomic commit and consensus

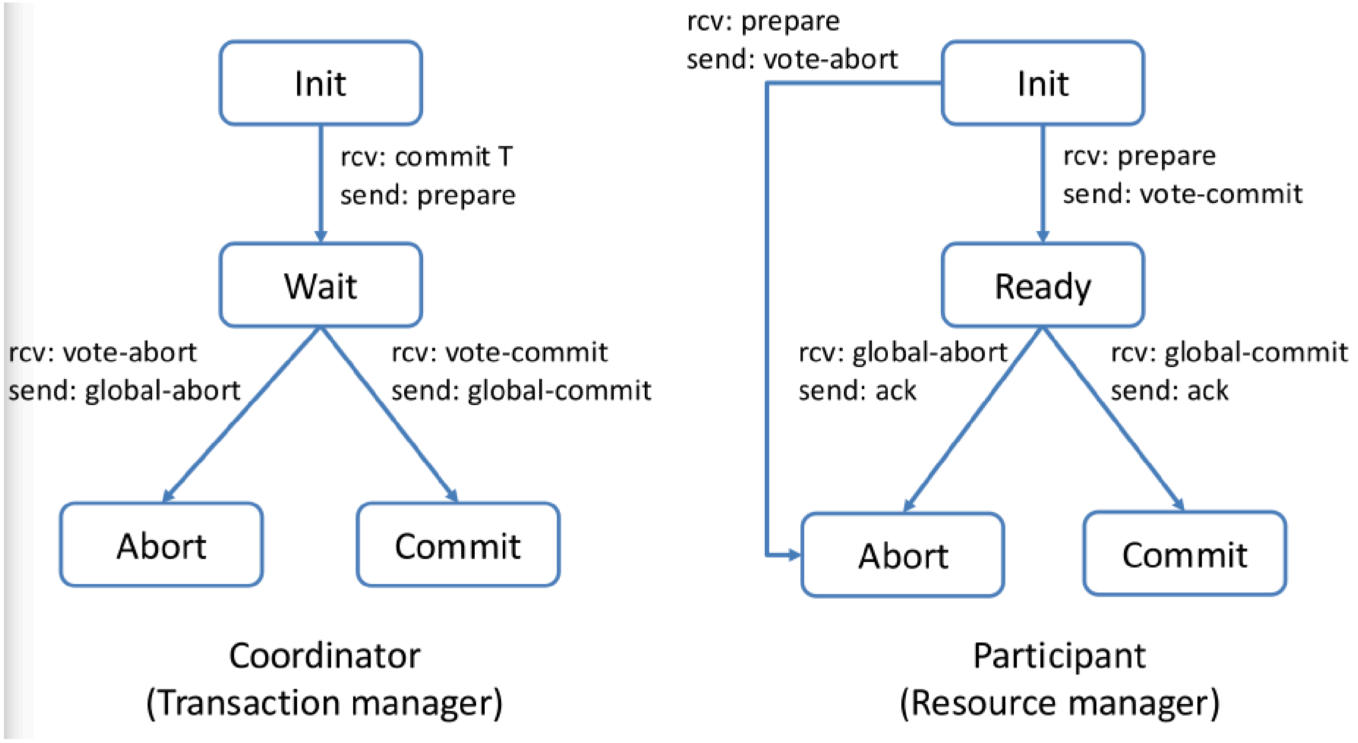


Consensus	Atomic commit
One or more nodes propose a value	Every node votes to commit or abort
Nodes agree on one of the proposed values	Commit if and only if all nodes vote to commit, abort otherwise
Tolerates failures, as long as a majority of nodes is available	Any crash leads to an abort

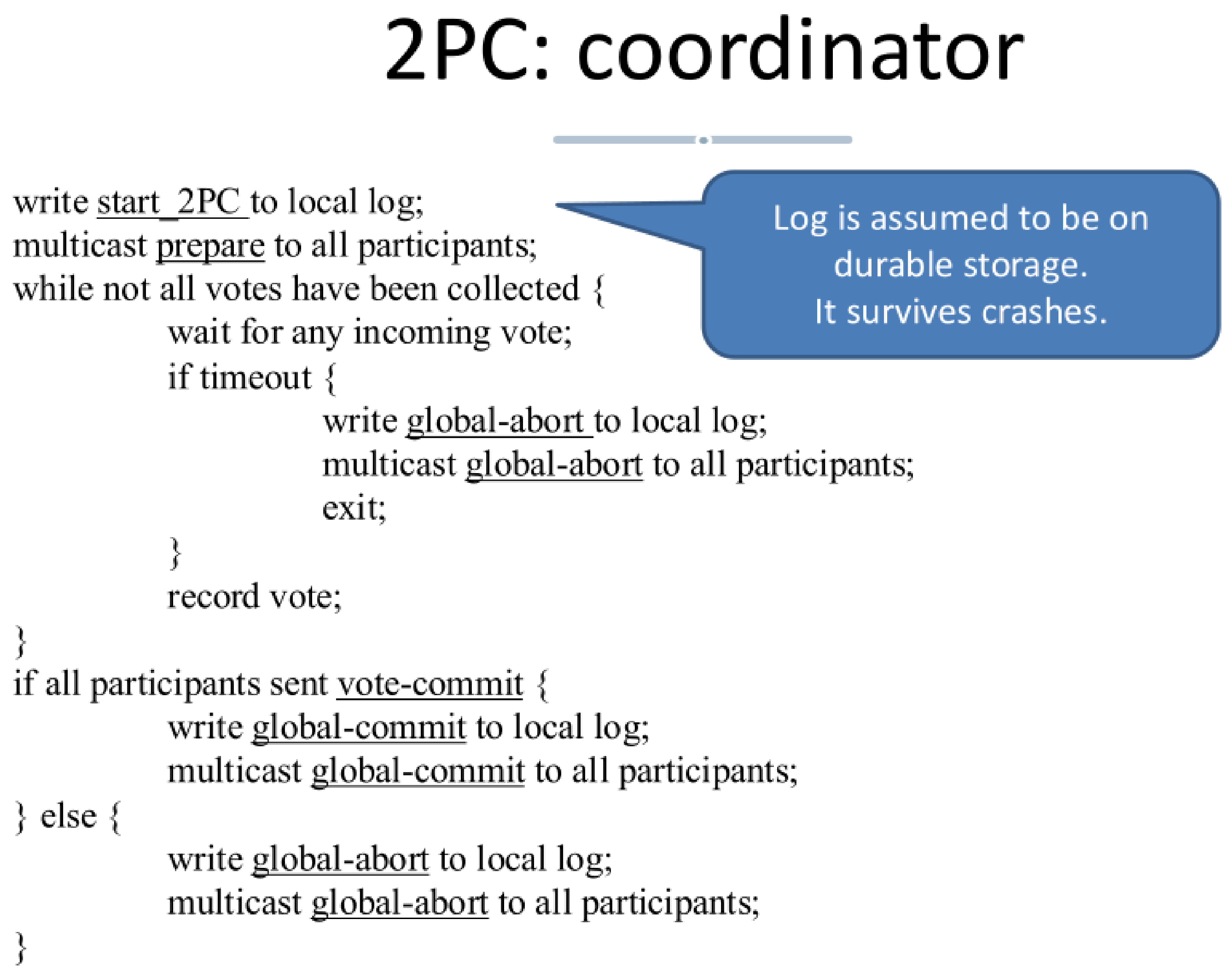
The problem is we have multiple partitions of the databases and need to react to the single user transaction.

Two phase commit (2PC)



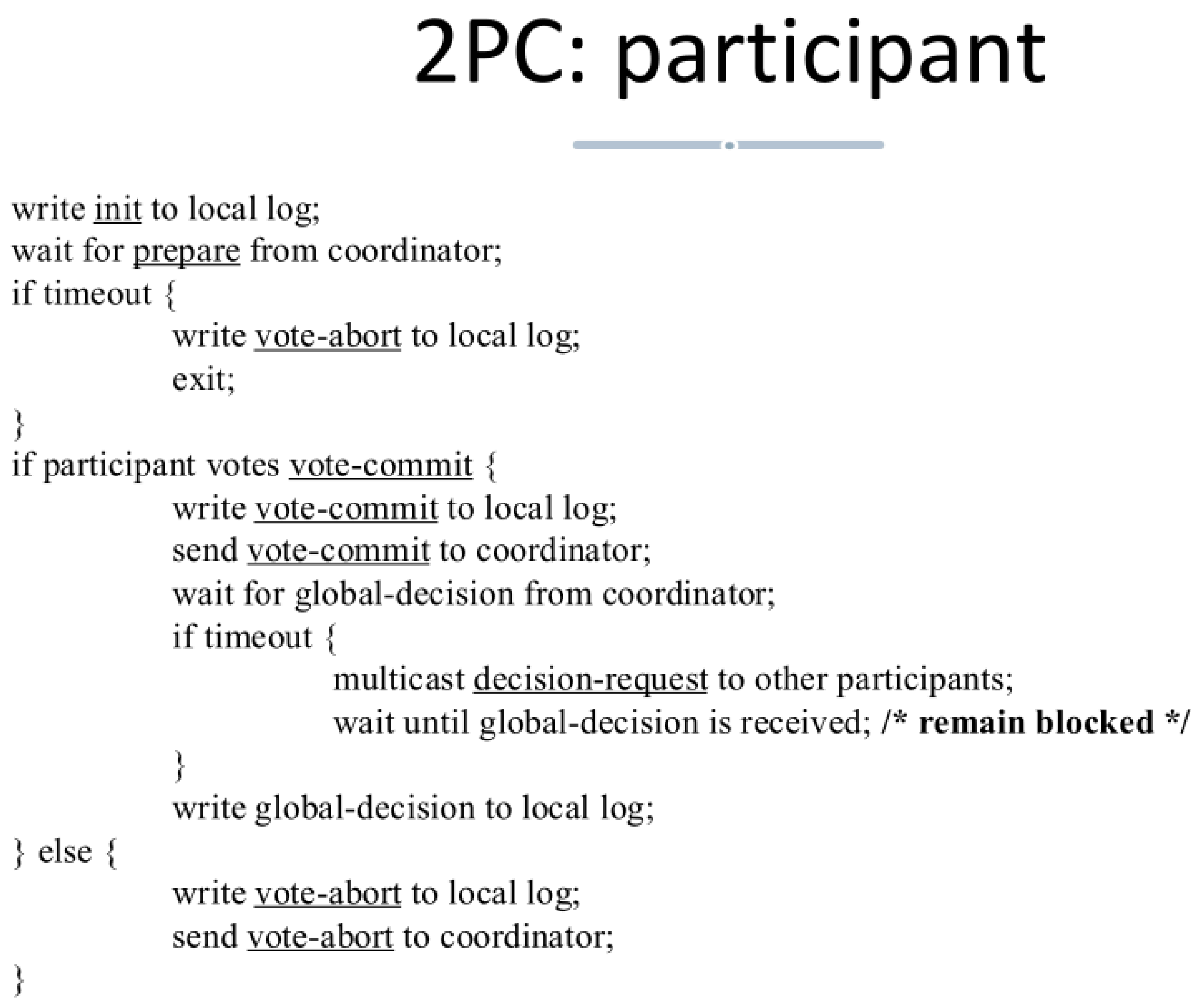


A client ask to execute a transaction to the database(that is composed of multiple participant). Then ask the coordinator to commit the transaction. The coordinator ask all the participant to prepare to commit the transaction and wait for the answers of the participants. If all the participant answers the coordinator send a global commit action. If one of the participant answer with abort the coordinator abort the transaction sending a global abort and not a global commit. This works if there are no failures. The failures that can happen are: a single participant that fails, if the coordinator don't receive an answer after a given timeout it assume that the answer is abort and abort the transaction. If the coordinator fails the participant can abort from the init state as it is impossible that a global commit occurs. If the participant is in a ready state and the coordinator crashes the participant cannot take a decision on its own, it can wait for the coordinator to recover or request the decision to other participant. If the other participants have information from the coordination it can be recover from them. Instead if there is a node in init state the participant can abort as that node didn't receive the vote commit message from the coordinator. If everybody is in the ready state we don't know what to do. This is a safe **BUT** blocking protocol.



2PC: coordinator

```
write start_2PC to local log;
multicast prepare to all participants;
while not all votes have been collected {
        wait for any incoming vote;
        if timeout {
                write global-abort to local log;
                multicast global-abort to all participants;
                exit;
        }
        record vote;
}
if all participants sent vote-commit {
        write global-commit to local log;
        multicast global-commit to all participants;
} else {
        write global-abort to local log;
        multicast global-abort to all participants;
}
```



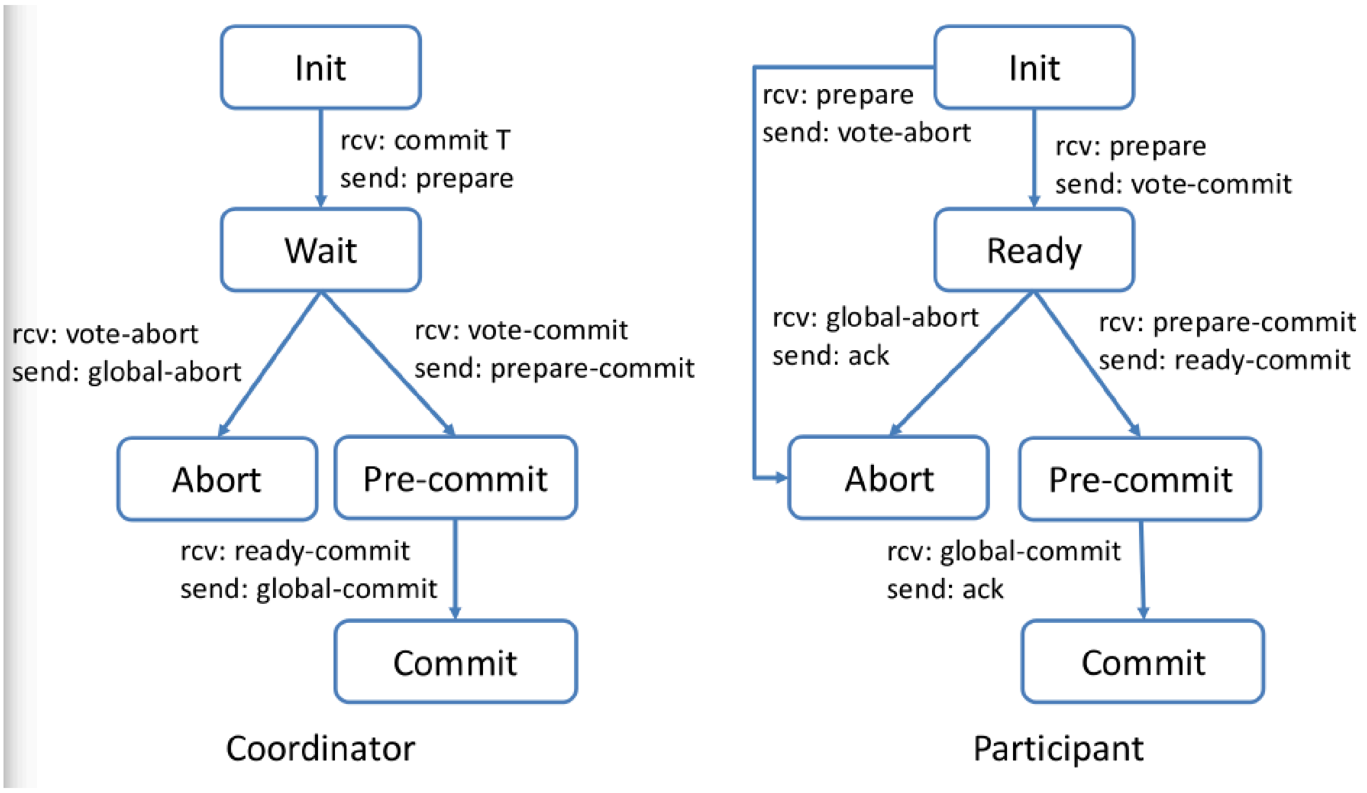
2PC: participant

```
write init to local log;
wait for prepare from coordinator;
if timeout {
        write vote-abort to local log;
        exit;
}
if participant votes vote-commit {
        write vote-commit to local log;
        send vote-commit to coordinator;
        wait for global-decision from coordinator;
        if timeout {
                multicast decision-request to other participants;
                wait until global-decision is received; /* remain blocked */
        }
        write global-decision to local log;
} else {
        write vote-abort to local log;
        send vote-abort to coordinator;
}
```

One of the assumption is that coordinator and participants write their state in a durable memory to be recovered from crashes. There are cases in which I'm blocking and not making progress(coordinator crashes and all participant in ready state). This also afflict the availability of the system as while blocked it cannot answer other transactions.

Three phase commit (3PC)

The main problem with 2PC is that we directly go from ready to commit or abort. So the idea is not to have a state that directly go to commit/abort. There is a pre-commit stage for the commit to know if everybody knows the final outcome.



If a single participant fails the coordinator blocked waiting for vote can assume abort as no participant can be in pre-commit. If the coordinator is blocked in pre-commit it can safely commit and tell the failed participant to commit when it recovers. If the coordinator fails if the participant is in init state the participant can go in abort state as before. If the participant is blocked waiting for a global decision(ready state) the participant can ask the others participants what to do. If the participant is in abort state I abort(same if it is in init state), if someone is in commit state I can go there too.If I'm in ready state I need to check if I'm in a majority of nodes that can talk to each other and if there is **at least a node in pre-commit state** I can safely commit. If they are all in ready state they can decide to abort.
3PC (quorum-based version presented above) guarantees safety: it never leads to an incorrect state. In a synchronous system, it also guarantees liveness: it never blocks if a majority of nodes are alive and can communicate with each other. In a synchronous system we can use a timeout to learn if a node is connected or not. In an asynchronous system, the protocol may not terminate. Intuitively, we cannot use finite timeouts to discriminate connected and disconnected nodes. More expensive than 2PC. Always requires three phases of communication
If a majority abort when it didn't have to there are protocols to recover from that state when the majority reconnect with all the system.

Commit protocols: summary

2PC sacrifices liveness (blocking protocol)
3PC more robust, but more expensive

- Not widely used in practice
- In theory, it may still sacrifice liveness in presence of network partitions
  General result (FLP theorem): you cannot have both liveness(transactions continue to happens) and safety(all the nodes arrive at the same final state) in presence of network partitions in an asynchronous system. The idea is that you work with timeout and assume that after them the nodes are not working.

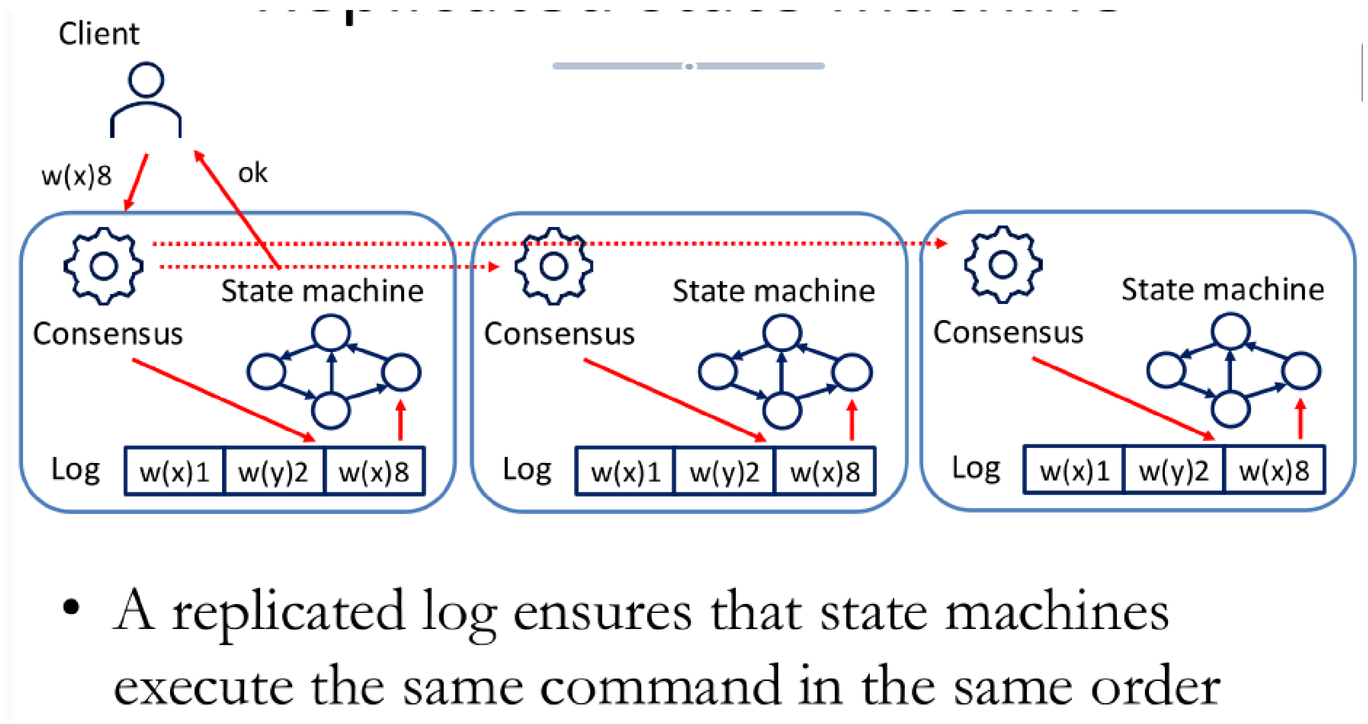
CAP theorem

Any distributed system where nodes share some (replicated) shared data can have at most two of these three desirable properties:

- C: consistency equivalent to have a single up-to-date copy of the data
- A: high availability of the data for updates (liveness)
- P: tolerance to network partitions
  In presence of network partitions, one cannot have perfect availability and consistency. Modern data systems provide suitable balance of availability and consistency for the application at hand (E.g., weaker definitions of consistency). We will discuss this under “replication and consistency”.

REPLICATED STATE MACHINES

The idea is instead of having partitions you replicate the same data across multiple nodes to improve fault tolerance. It is general purpose consent algorithm. It helps different machines to work as a group and give a continuous service even if some machines crashes. The machines are seen as one from the client(complex in practice). They operate on identical copies of the same state.
State machine: manages internal state, responds to external requests (commands)
Example: storage system: internal state: set of variables, external requests: read / write operations



- A replicated log ensures that state machines execute the same command in the same order

The client submits operations to the leader node that report the operations and the order to compute them to the other nodes of the system. There is a protocol to decide the order(Consensus protocol). Consensus Protocol: multiple nodes need to arrive to an agreement to the order of operations. If the leader crashes it is decided a new leader as it is this node that communicate to the client and need to communicate to the client the operations in the correct order they were decided to be committed from the starting leader. Assumptions are no Byzantine failures, majority of nodes need to be up and talking to each others and you can save your state to a durable storage.

Failure model

Unreliable, asynchronous communication

- Messages can take arbitrarily long to be delivered
- Messages can be duplicated
- Messages can be lost
  Processes
- May fail by stopping and may restart
- Must remember what they were doing
  - Record state to durable storage

Guarantees

Safety

- All non-failing machines execute the same command in the same order
  Liveness / availability
- The system makes progress if any majority of machines are up and can communicate with each other
- Not always guaranteed in theory
    - It is not possible, according to the FLP theorem
- Guaranteed in practice under typical operating conditions
- E.g., randomized approaches make blocking indefinitely highly improbable
  In theory we cannot have both safety and liveness but in practice we work on assumptions to guarantee liveness

Paxos

Paxos was the reference algorithm for consensus for about 30 years. Proposed in 1989 and published in 1998
Problems:

- Only agreement on a single decision, not on a sequence of requests (multi-Paxos solves the issue)
- Very difficult to understand
- Difficult to use in practice: no reference implementation and agreement on the details
  The original paxos resolve the agreement on a single values(extend to multiple-paxos, really difficult process)

Raft

Raft is equivalent to multi-Paxos in terms of assumptions, guarantees, performance
Design goal: understandability

- Easy to explain and understand
- Easy to use and adapt: several reference implementations
  Approach: problem decomposition
  The idea is starting from scratch and build a protocol with multiple consensus and each one is talked independently simplifying the decision on the consensus.
  Raft Decomposition
  There is a protocol for normal operations(how does the protocol work in absence of failures). The idea is the coordinator receives messages from the client and put them in a log and then I replicate the operations synchronously to

the other nodes. When I receive an acknowledgements from a majority I have done my job and tell the client the results. Leader election is when I am a participant and there is a failure and cannot hear from the leader and decide to start an election and use a protocol to maintain the log safe(no corruption or going back in time). Nodes can be in three states: follower, leader and candidate(node that started an election). If followers don't hear from the leader for a while they became candidates and start an election that if they win they become the leader.

Normal operation

Client sends command to leader
Leader appends command to its log
Leader sends AppendEntries to all followers
Once new entry committed

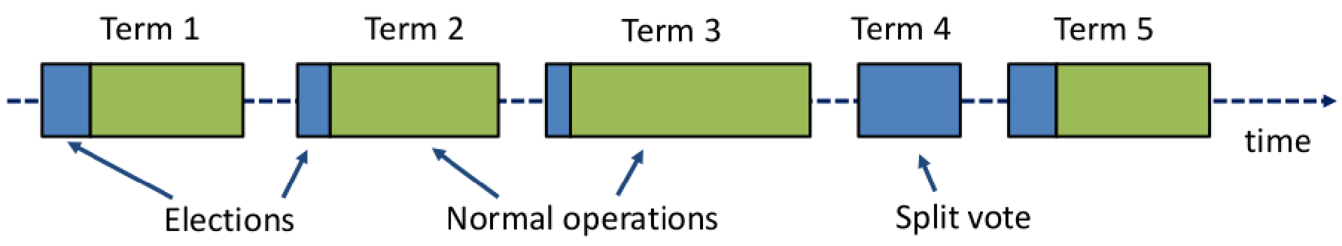
- Leader executes command in its state machine, returns result to client
- Leader notifies followers of committed entries in subsequent AppendEntries. Leader send out empty AppendEntries to the follower periodically to avoid the follower to think that the leader crashed.
- Followers execute committed commands in their state machines
  Crashed/slow followers?
- Leader retries AppendEntries messages until they succeed
  Optimal performance in common case
- One successful message to any majority of servers

Leader election

The leader periodically sends AppendEntries messages with its unacknowledged log entries

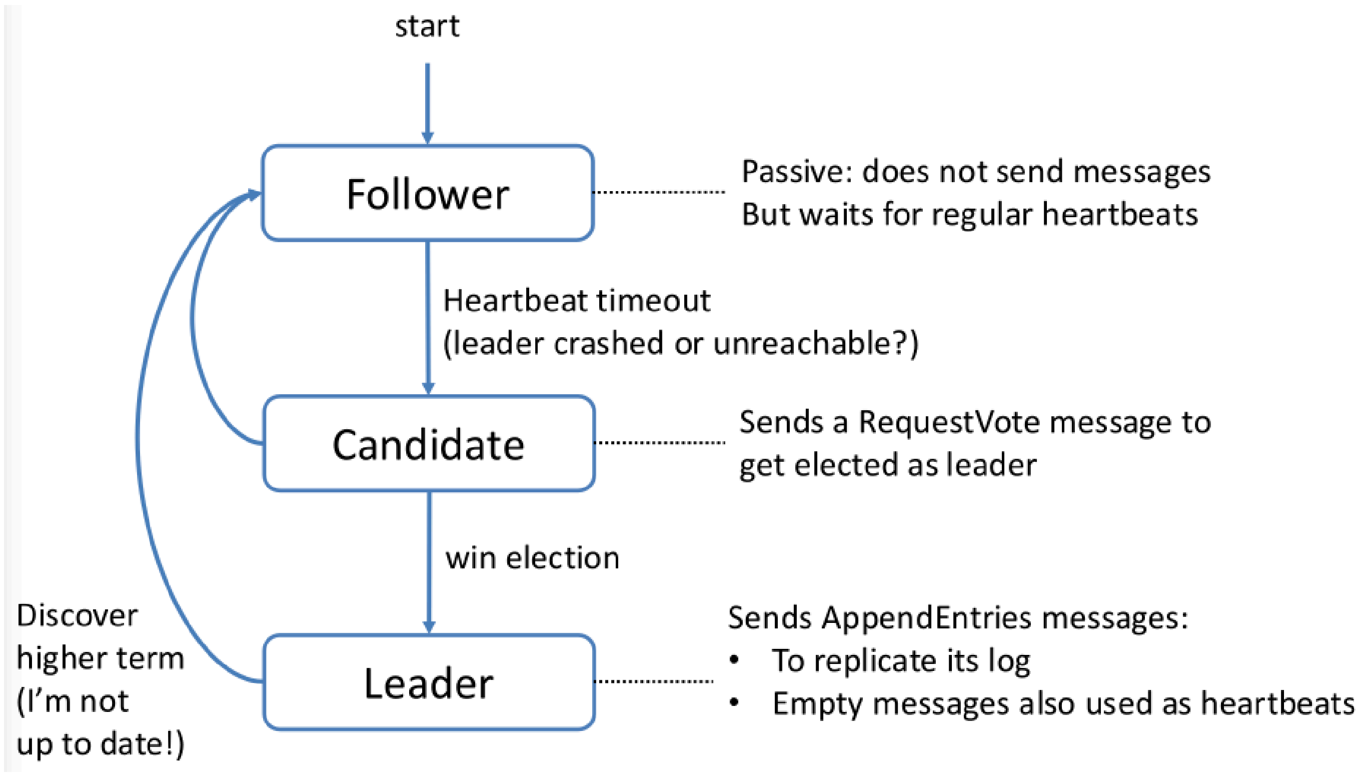
- Possibly empty
  Followers have a timeout: if they don’t hear from the leader, they start an election
- Timeout randomized to avoid many parallel elections. We want only one election at a time(single candidate for the role of leader)
- Range: 150 – 300 ms

Terms

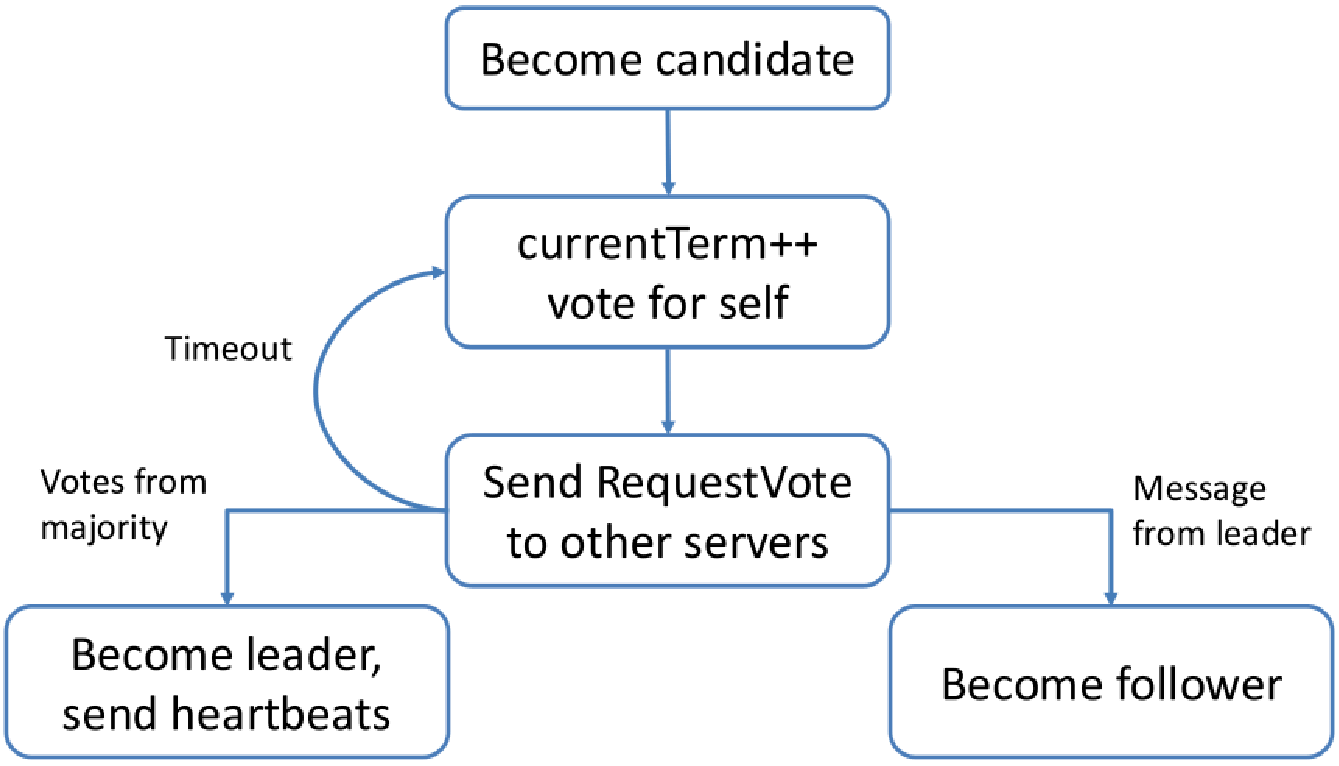


- Raft divides time into terms of arbitrary length
  - Terms are numbered with consecutive integers
  - Each server maintains a current term value
    Exchanged in every communication
  - Terms identify obsolete information
  - Each term begins with an election, in which one or more candidate try to become leader
  - There is at most one leader per term
    If there is a split vote (no majority for any candidate), followers try again when the next timeout expires. This needed to avoid the fact that a node thinking it is a leader crashes and recover and still thinks it is a leader. Each term start with an election. Since to became a leader I need the majority of votes it is almost guarantee that there is a single leader per term(There may be a split vote and then there isn't a normal operation part in the term, problem as there could be an infinite sequence of split vote(not in practice as we are starting elections at random time))

Server states and messages


start
Follower
Passive: does not send messages
But waits for regular heartbeats
Heartbeat timeout
(leader crashed or unreachable?)
Candidate
Sends a RequestVote message to
get elected as leader
win election
Discover
higher term
(I'm not
up to date!)
Leader
Sends AppendEntries messages:
• To replicate its log
• Empty messages also used as heartbeats

Leader election



Election correctness

Safety: allow at most one winner per term

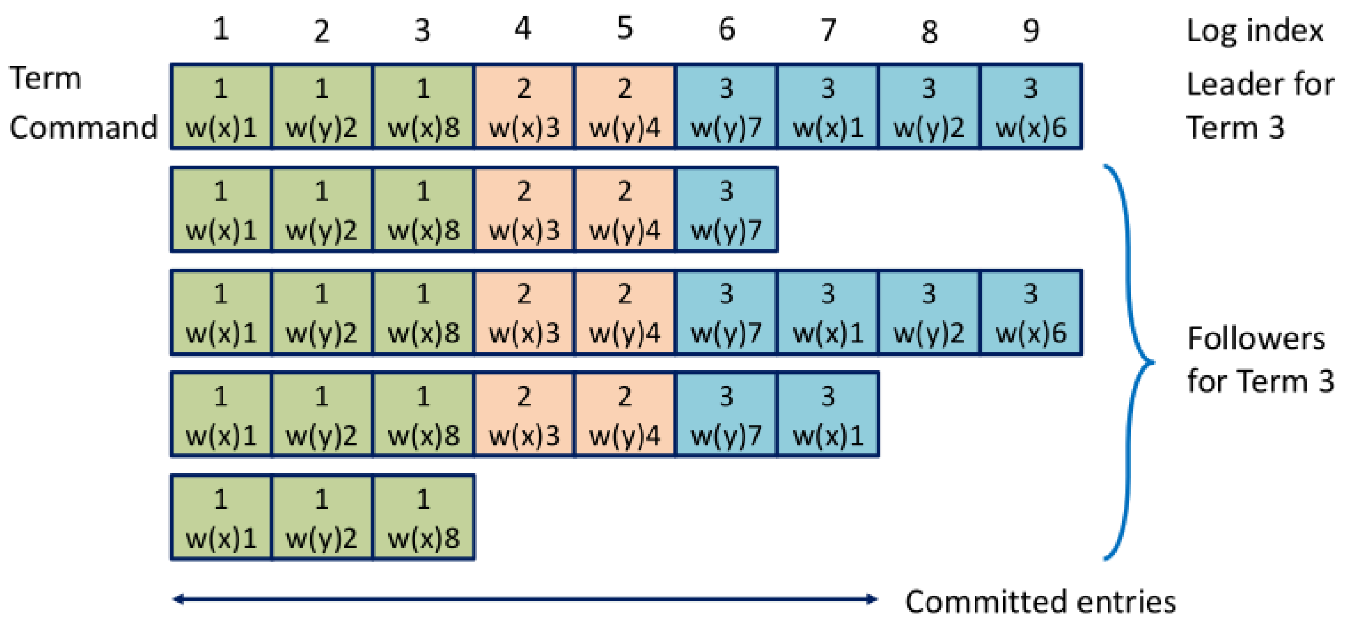
- Each server gives only one vote per term (persist on disk)
- Majority required to win election
  Liveness: some candidate must eventually win

- Choose election timeouts randomly
    - E.g., between 150ms and 300ms
- One server usually times out and wins election before others time out
  - Not guaranteed, but highly probable
  - Guaranteed liveness is impossible due to the FLP theorem
  Works well if timeout >> communication time

Network partitions

In the case of network partitions the system works on majority: it can commit to the clients only if the leader can communicate with a majority. If it cannot do that it cannot commit. The partition with a majority elect a new leader and continue working as if the system is not partitioned. When the partitions reunite the old leader/the other nodes recognise the fact that their term is not updated and update their log to uniform to the current term, the old leader also become a simple participant.

Log structure



Log index
1 2 3 4 5 6 7 8 9
Term
Command
Leader for Term 3
1 w(x)1 | 1 w(y)2 | 1 w(x)8 | 2 w(x)3 | 2 w(y)4 | 3 w(y)7 | 3 w(x)1 | 3 w(y)2 | 3 w(x)6
1 w(x)1 | 1 w(y)2 | 1 w(x)8 | 2 w(x)3 | 2 w(y)4 | 3 w(y)7
1 w(x)1 | 1 w(y)2 | 1 w(x)8 | 2 w(x)3 | 2 w(y)4 | 3 w(y)7 | 3 w(x)1 | 3 w(y)2 | 3 w(x)6
1 w(x)1 | 1 w(y)2 | 1 w(x)8 | 2 w(x)3 | 2 w(y)4 | 3 w(y)7 | 3 w(x)1
1 w(x)1 | 1 w(y)2 | 1 w(x)8
Followers for Term 3
Committed entries

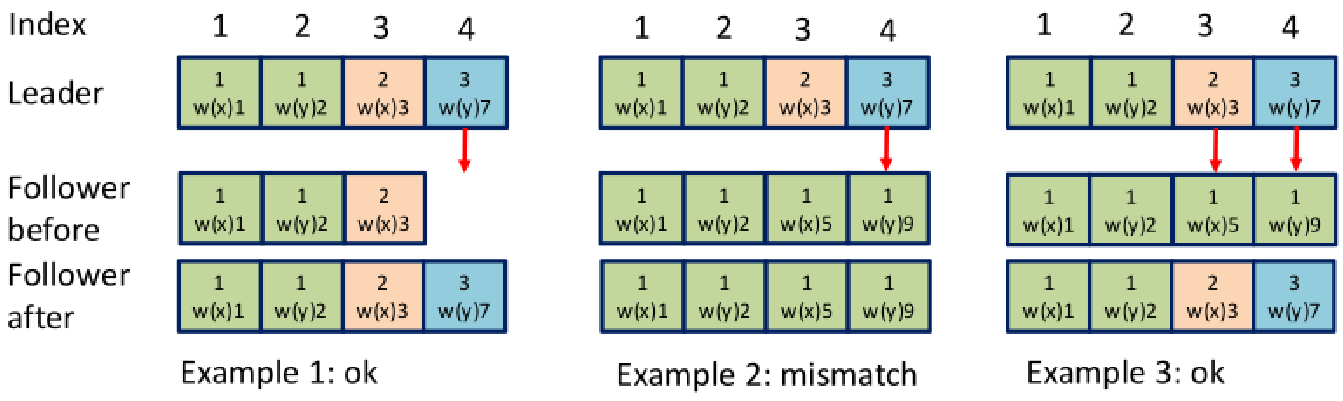
- Stored on disk to survive process failures
- Entry committed (by leader of its term) if replicated on majority of servers

This data structure including the terms, the commands and the nodes are stored on disk and the only one that can distribute this information is the leader and it can distribute the information only on the term it is the leader for with the AppendLog entry.
Crashes can result in log inconsistencies. Raft minimizes special code for repairing inconsistencies:

- Leader assumes its log is correct
- Normal operation will repair all inconsistencies
  Raft guarantees the log matching property. If log entries on different servers have the same index and term:
- They store the same command
- The logs are identical in all preceding entries
  If a given entry is committed, all preceding entries are also committed

Consistency check

AppendEntries messages include <index, term> of the entry preceding the new one(s)
Follower must contain matching entries, otherwise it rejects the request and the leader retries with lower log index. This implements an induction step that ensures the log matching property



Need a recursive protocol to accept a message if and only if the prefix of the log is the same. This also helps to reconstruct inconsistencies in the log. A leader is a leader if it has the higher value for the term.

Safety: leader completeness

Once log entry committed, all future leaders must store that entry. Servers with incomplete logs must not get elected:

- Candidates include index and term of last log entry in RequestVote messages
- Voting server denies vote if its log is more up-to-date
  This is important as the only one that send out messages and can reconstruct the log of nodes is the leader so it needs the correct log.

Communication with clients

In Raft, clients always interact with the leader

- When a client starts, it connects to a random server, which communicates to the client the leader for the current term
- If a leader crashes, the communication with the client times out
  Raft guarantees linearizable semantics
- All operations behave as if they were executed once and only once on a single copy
- More on this under “replication and consistency”
- In the case of a leader failure, a client may need to retry
  - The client attaches a sequential identifier of the request
  - Servers store the last identifier for each client as part of the log
  - Servers can discard duplicates
  Clients need to communicate in a good way. Their message can be written on the log and can be applied before the leader/network fails and the message is not communicated to the client. So even if the client retries to communicate the same message it need to attach something that permit to distinguish a message that is duplicated from a new one so the server can discard duplicated messages. Clients also have the illusion to communicate with a single server even if there are multiple servers behind.
  Additional information
  You can refer to the Raft (extended) paper for additional information on
- Log compaction: when it is safe to delete old entries and reduce space occupation?
- Change membership: how to consistently add/remove servers from the group?
- Performance

Use in distributed DBMS

Some modern distributed database management systems integrate replicated state machines and
commit protocols. The database is partitioned: inter-partition transactions must guarantee atomic commitment. Each partition is replicated: replicated state machines guarantee that all replicas process the same sequence of operations(either the partition or the log of operations is replicated on multiple nodes). This are characteristic of robust and safe system that don't need the 3PC protocol and can use 2PC as there is a replication layer that assure no crash on the coordinator.
You have two layer one to partition the database and one to duplicate the nodes using also RAFT to retrieve correct nodes.

BYZANTINE CONDITIONS

BFT Consensus

State machine replication can be extended to consider byzantine processes

- Known as Byzantine Fault Tolerant (BFT) replication
- Same assumption as Paxos / Raft
- Requires 3f+1 participants to tolerate f failing nodes
  There are extensions of the previous algorithms to recover from Byzantine failures.

Blockchains

Even if there are malicious users the idea is that we want to have a distributed consensus of some data such that we can exchange them and agree on these transactions. This is a Byzantine environments as people can try to double-spent, they try to use the same data twice/do as if the transaction never happened. The transaction on which we agree is the set of operation we have done from the start of the operation. The log contains all the transactions and we need to agree on the order of operation to avoid double-spending. The environment is open as everybody can join the blockchain even if they aren't members of it. You have a huge amount of people.
Assumptions

- Very large number of nodes
- The set of participating nodes is unknown upfront
- No single entity owns the majority of compute resources
  Guarantees
- No provable safety guarantees: only with high probability
  In these slides, we will refer to permissionless systems based on proof of work (e.g., Bitcoin)
- Different approaches may differ in terms of assumptions and guarantees

Bitcoin blockchain

The blockchain is the public ledger that records transactions:

- It contains all transactions from the beginning of the blockchain

- Each block of the chain includes multiple transactions
  It is a distributed ledger (replicated log)
- Transactions are published to the bitcoin network
- Nodes add them to their copy and periodically broadcast it
  Transactions are digitally signed
- If the private key is lost, bitcoins are lost too
  Adding a block to the chain requires solving a mathematical problem (proof of work)
- Takes in input the existing chain and the new block
  This links the block to the chain
- Solution difficult to find
  - Brute force
  - On average, one solution every 10 minutes
  - Complexity adapted to computational power
  Solution easy to verify
- Enables rapid validity check
  It is difficult to double spend as you need to resolve a complicated problem twice before someone else has seen and confuted the first solution you published.
  Proof of work computed by special nodes (miners) that collect new (pending) transactions into a block:
- Incentive: they earn Bitcoins if successful
  When a miner finds the proof, it broadcasts the new block. This defines the next block of valid transactions. The other miners receive it and try to create the next block in the chain. Global agreement on the order of blocks!
  What if two miners find a proof concurrently?
- The proof is very complex to compute
- Very unlikely that two computers will find a solution at the same time
- Very difficult for someone to force their desired order of transactions, since it would require a lot of compute power
  If two concurrent versions are created, the one that grows faster (includes more blocks) survive
- If same length → deterministic choice
  Still, there may always exist a longer chain I’m not aware of
- No one can be 100% sure of a given sequence
  Double spending

Someone with enough computational power and 1 Bitcoin can create two concurrent chains

- Chain 1: the Bitcoin is used to pay A and the chain is propagated to A
- Chain 2: the Bitcoin is used to pay B and the chain is propagated to B
- A and B can never be 100% sure that a conflicting chain does not exist!