POLITECNICO
MILANO 1863

Distributed agreement in practice

Alessandro Margara
alessandro.margara@polimi.it
https://margara.faculty.polimi.it

Outline

- Commit protocols
  - 2PC
  - 3PC
- Consensus for state machine replication
  - Raft
- Consensus under byzantine conditions
  - Blockchain

COMMIT PROTOCOLS

Atomic commit

- Atomic commit is a form of agreement widely used in database management systems

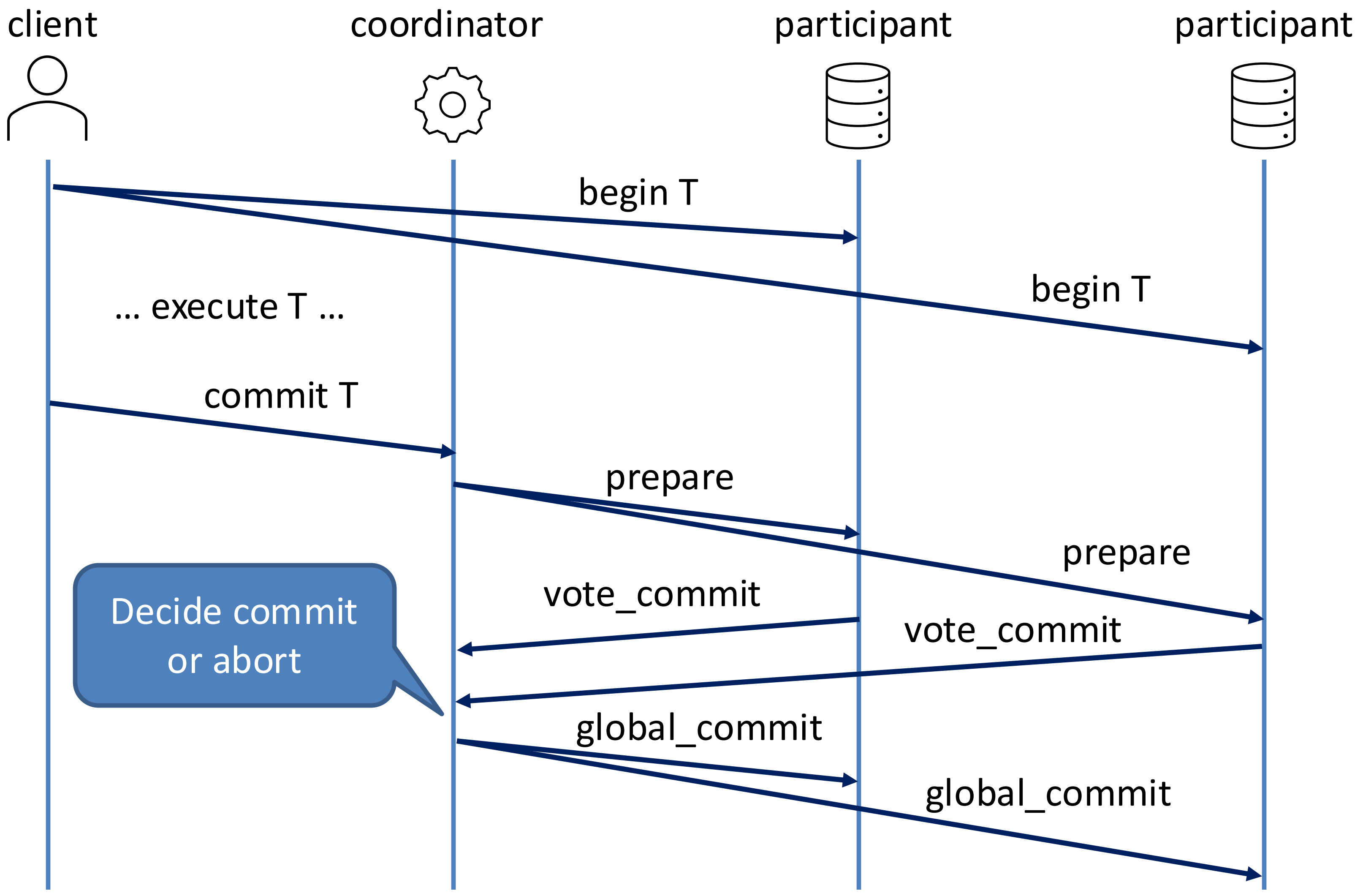
- Ensures atomicity (the "A" in ACID transactions)
  - A transaction either commits or aborts
  - If it commits, its updates are durable
  - If it aborts, it has no side effects
  - Also consistency (preserving invariants) relies on atomicity

- If the transaction updates data on multiple nodes (partitioned / sharded database)
  - Either all nodes commit or all nodes abort
  - If any node crashes, all must abort

Atomic commit and consensus

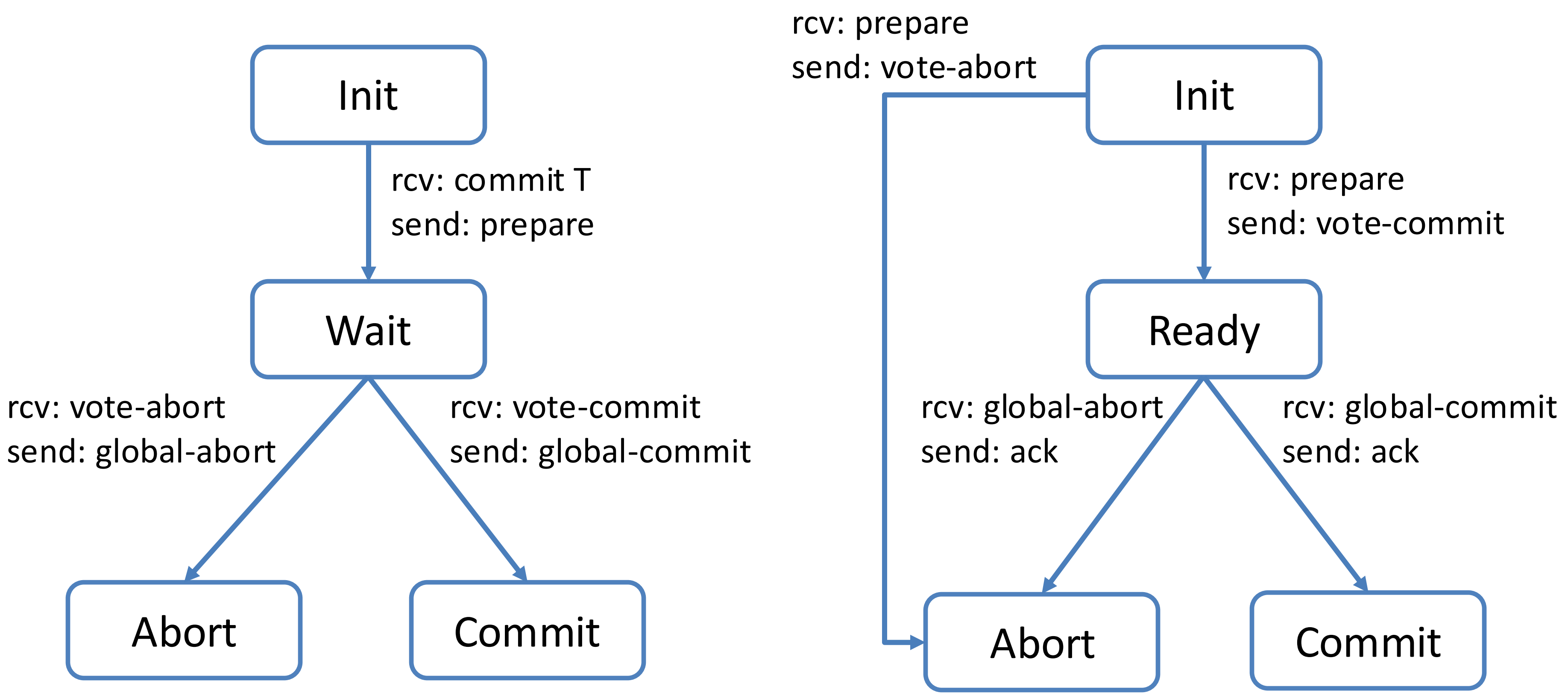
Consensus	Atomic commit
One or more nodes propose a value	Every node votes to commit or abort
Nodes agree on one of the proposed values	Commit if and only if all nodes vote to commit, abort otherwise
Tolerates failures, as long as a majority of nodes is available	Any crash leads to an abort

Two phase commit (2PC)



Two phase commit (2PC)

State graphs: show the state in which the components may be



Coordinator
(Transaction manager)

Participant
(Resource manager)

2PC: possible failures

- A participant fails
  - After a timeout, the coordinator assumes abort
- The coordinator fails
  - Participant waiting for prepare (Init state) → it can decide to abort
    - No participant can have already received a (global commit) decision
  - Participant waiting for global decision (Ready state) → It cannot decide on its own
    - It can wait for the coordinator to recover …
    - … or it can request the decision to another participant, which
      - May have received a reply from the coordinator
      - May be in Init state → coordinator has crashed before completing the prepare phase → assume abort
  - What if everybody is in the same Ready situation?
    - Nothing can be decided until the coordinator recovers
    - 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;
}
```

Log is assumed to be on durable storage.
It survives crashes.

So we know that if coordinator crashes and then recovers, we are sure that he recovers exactly from where he has left.

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;
}
```

2PC is a blocking protocol

- 2PC is safe (never leads to an incorrect state) but it may block

- In the previous slides, we assumed all nodes (including the coordinator) need to reach an agreement
  – In that case, 2PC is vulnerable to a single-node failure (the coordinator)

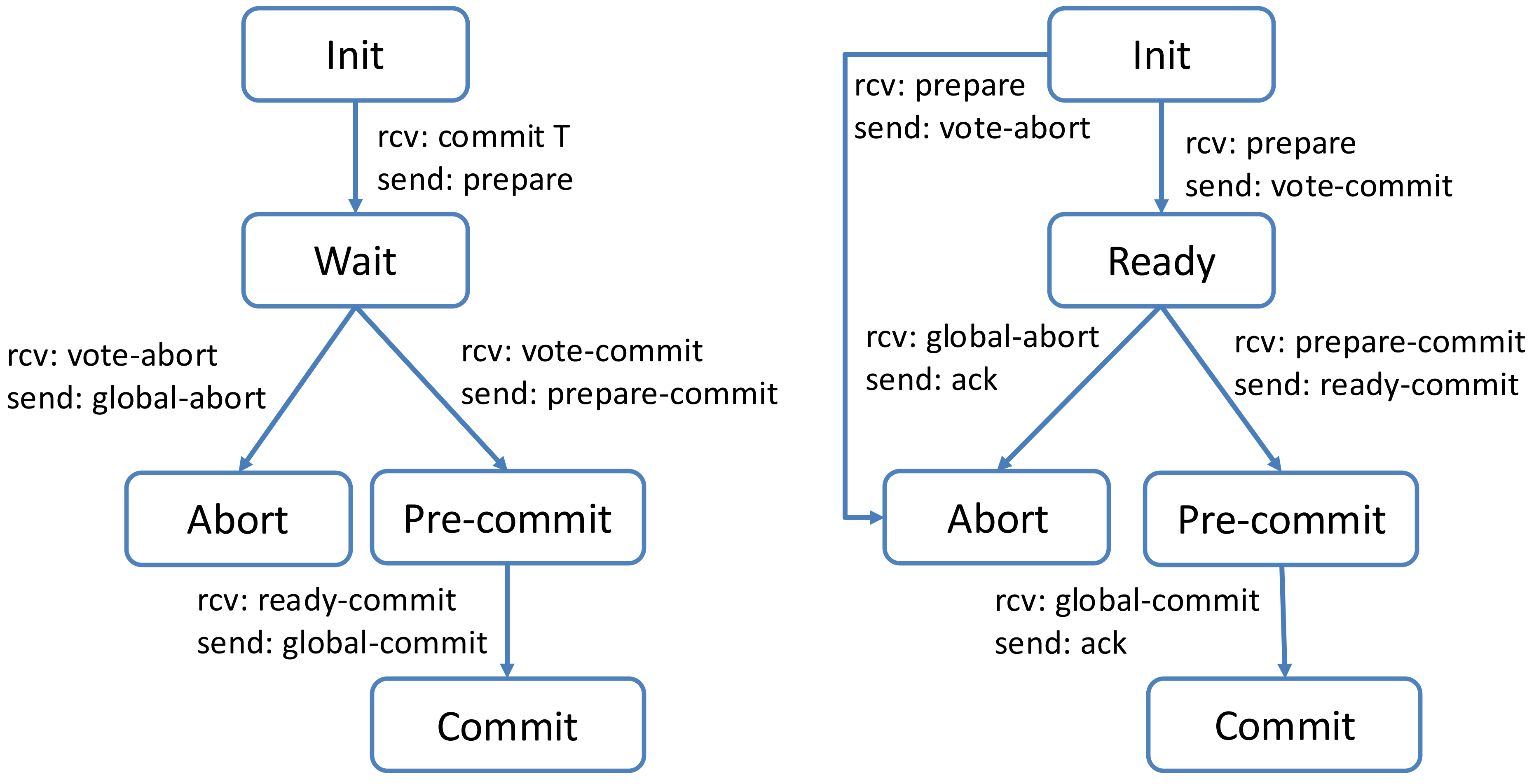
- If it takes time to restore the failed node (e.g., manual procedure), the system remains unavailable

Three phase commit (3PC)

- Attempt to solve the problems of 2PC by adding another phase to the protocol
  - No state leading directly to both commit and abort
  - No state where final decision is impossible can lead to commit
- The above conditions are satisfied if and only if the protocol is non-blocking [1]
- Idea: split the commit/abort phase into two phases
  - Communicate the outcome to all nodes
  - Let them commit only after everyone knows the outcome

[1] D. Skeen, M. Stonebraker "A formal model of crash recovery in distributed systems" IEEE TSE, 1983

Three phase commit (3PC)



Coordinator

Participant

Three-phase commit: possible failures

- A participant fails
  - Coordinator blocked waiting for vote (Wait state) can assume abort decision (no participant can be in Pre-commit)
  - Coordinator blocked in Pre-commit state can safely commit and tell the failed participant to commit when it recovers
- The coordinator fails
  - Participant blocked waiting for prepare (Init state) can decide to abort
  - Participant blocked waiting for global decision (Ready state) can contact another participant:
    - Abort (at least one) → Abort
    - Commit (at least one) → Commit
    - Init (at least one) → Abort
    - Pre-commit (at least one), #Pre-commit + #Ready form a majority → Commit
    - Ready (majority), no-one in Pre-commit → Abort
  - No two participants can be one in Pre-commit and the other in Init

D. Skeen "A quorum-based commit protocol", 1982

3PC: possible failures

- 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

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 and safety in presence of network partitions in an asynchronous system

Fischer, Lynch, Paterson “Impossibility of distributed consensus with one faulty process”, 1985

CAP theorem

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

Raft

REPLICATED STATE MACHINES

Replicated state machine

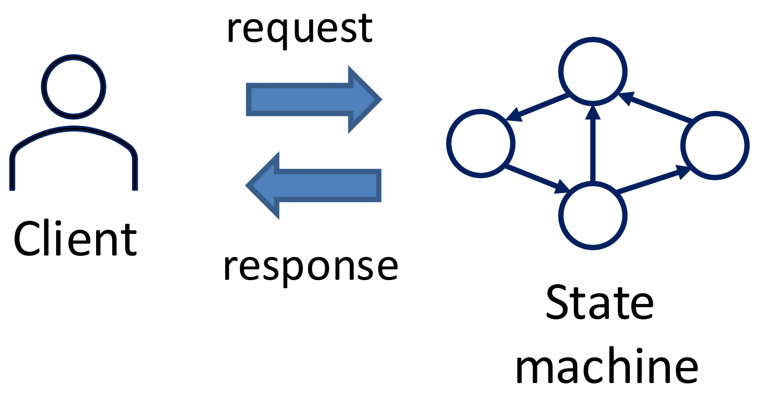
- General purpose consensus algorithm
  - Allows a collection of machines (servers) to work as a coherent group
  - They operate on identical copies of the same state
  - They offer a continuous service, even if some machines fail
  - Clients see them as a single (fault-tolerant) machine

Slides inspired by and adapted from: Talk on Raft at CS@Illinois Distinguished Lecture Series
by John Ousterhout, August 2016

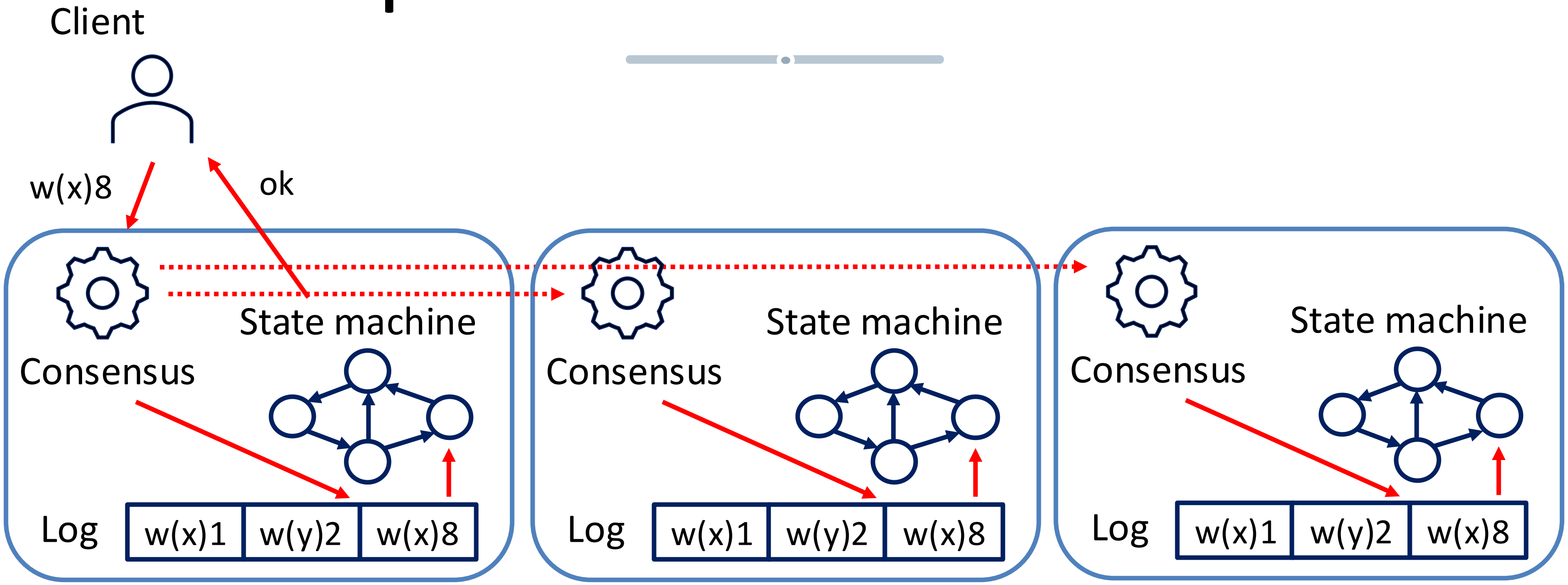
Replicated state machine

- State machine
  - Manages internal state
  - Responds to external requests (commands)

- Example: storage system
  - Internal state: set of variables
  - External requests: read / write operations



Replicated state machine



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

Failure model

Only 2 assumptions

- 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

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

L. Lamport "The Part-Time Parliament" ACM Transactions on Computer Systems, 1998

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

D. Ongaro, J. Ousterhout “In Search of an Understandable Consensus Algorithm” USENIX Annual Technical Conference, 2014
https://raft.github.io

Raft decomposition

- Log replication (normal operation)
  - Leader accepts commands from clients, appends to its log
  - Leader replicates its log to other servers
- Leader election
  - Select one server to act as leader
  - Detect crashes, choose new leader
- Safety
  - Keep log consistent
  - Only servers with up-to-date logs can become leader

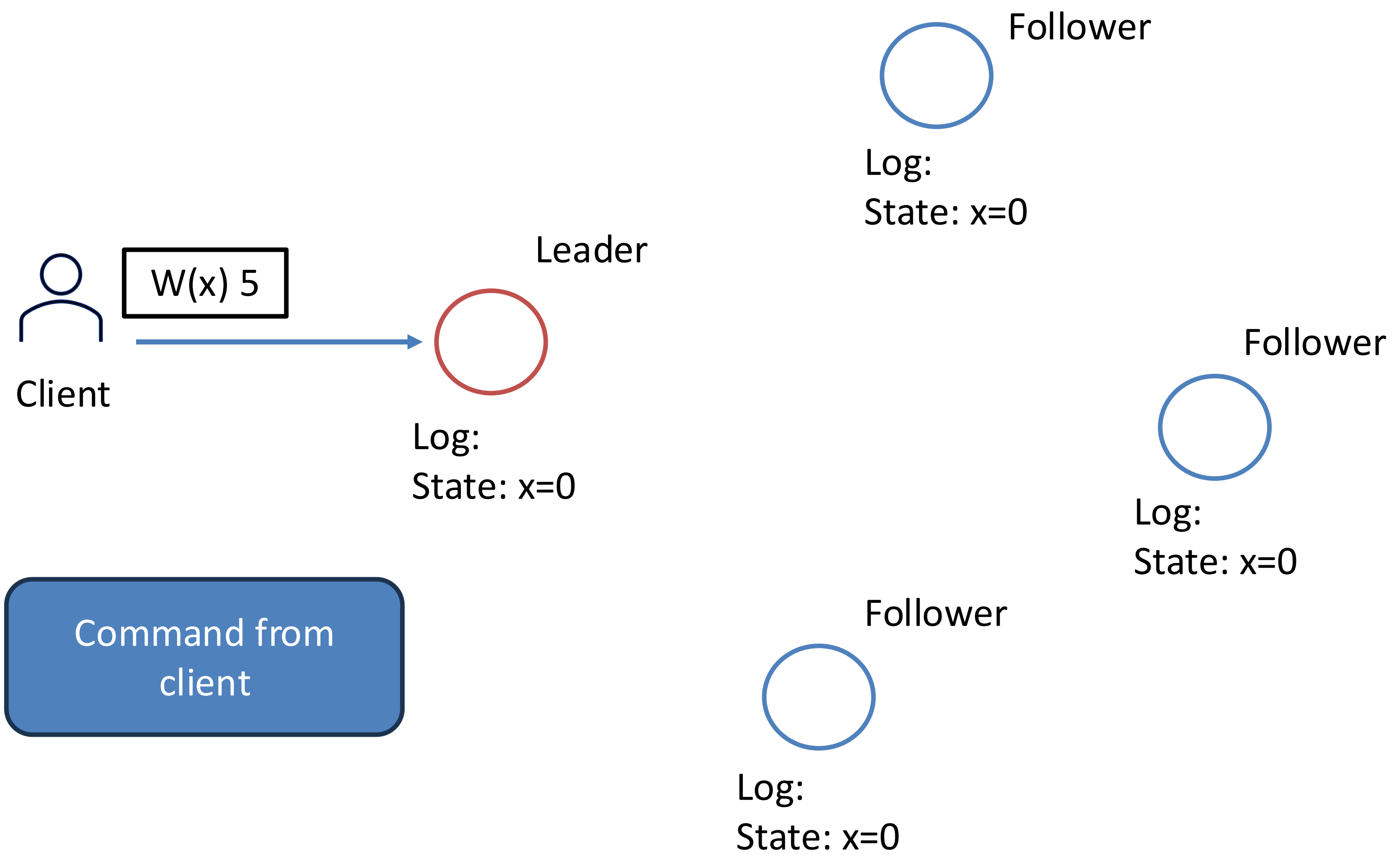
Raft overview

- Nodes can be in three states: follower, leader, candidate
  - All nodes start as followers
- If followers don't hear from a leader for a while, they become candidate
- A candidate runs an election: if it wins, it becomes the leader
- All commands go through the leader, who is responsible for committing and propagating them

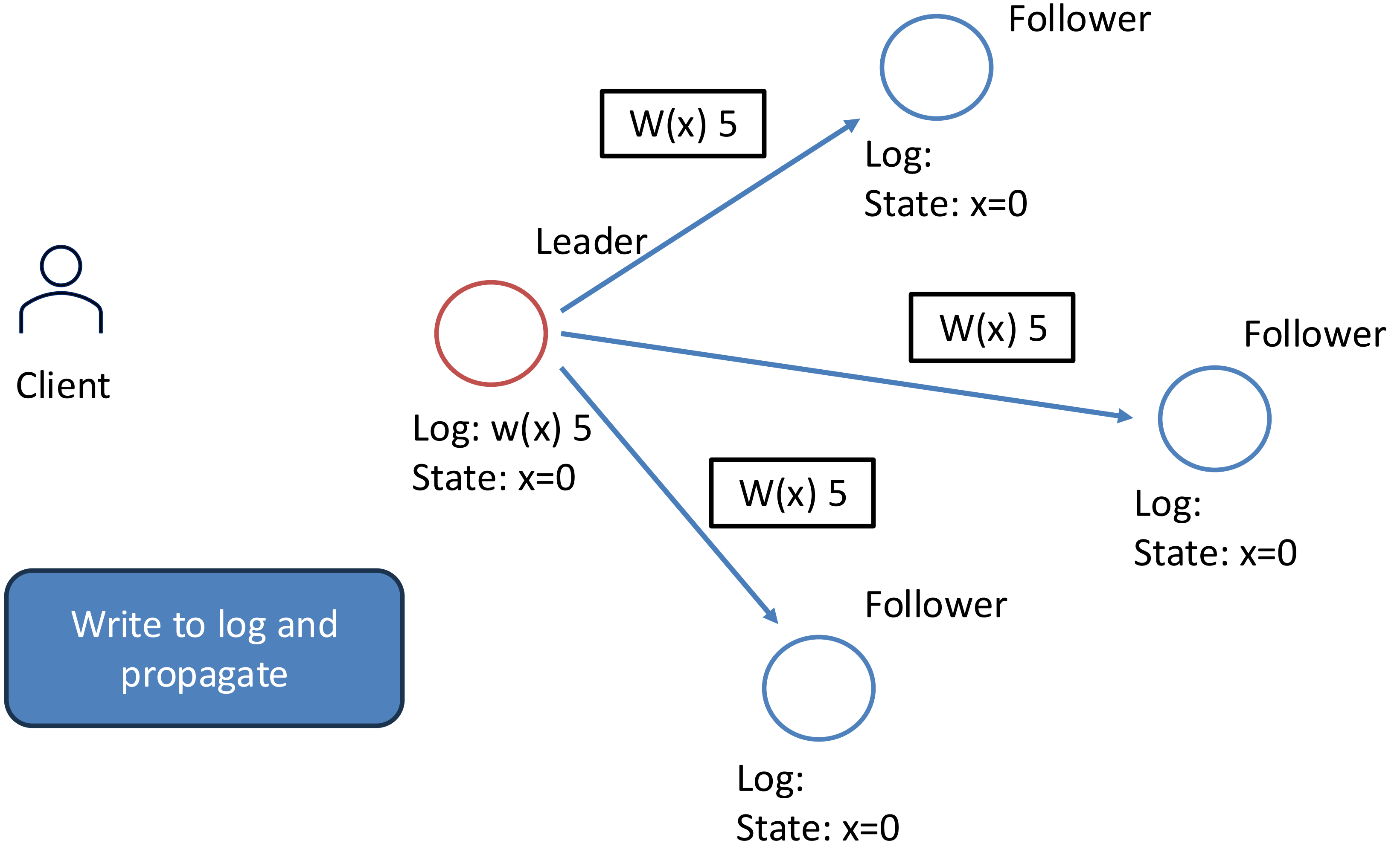
Normal operation

- 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
  - 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

Normal operation

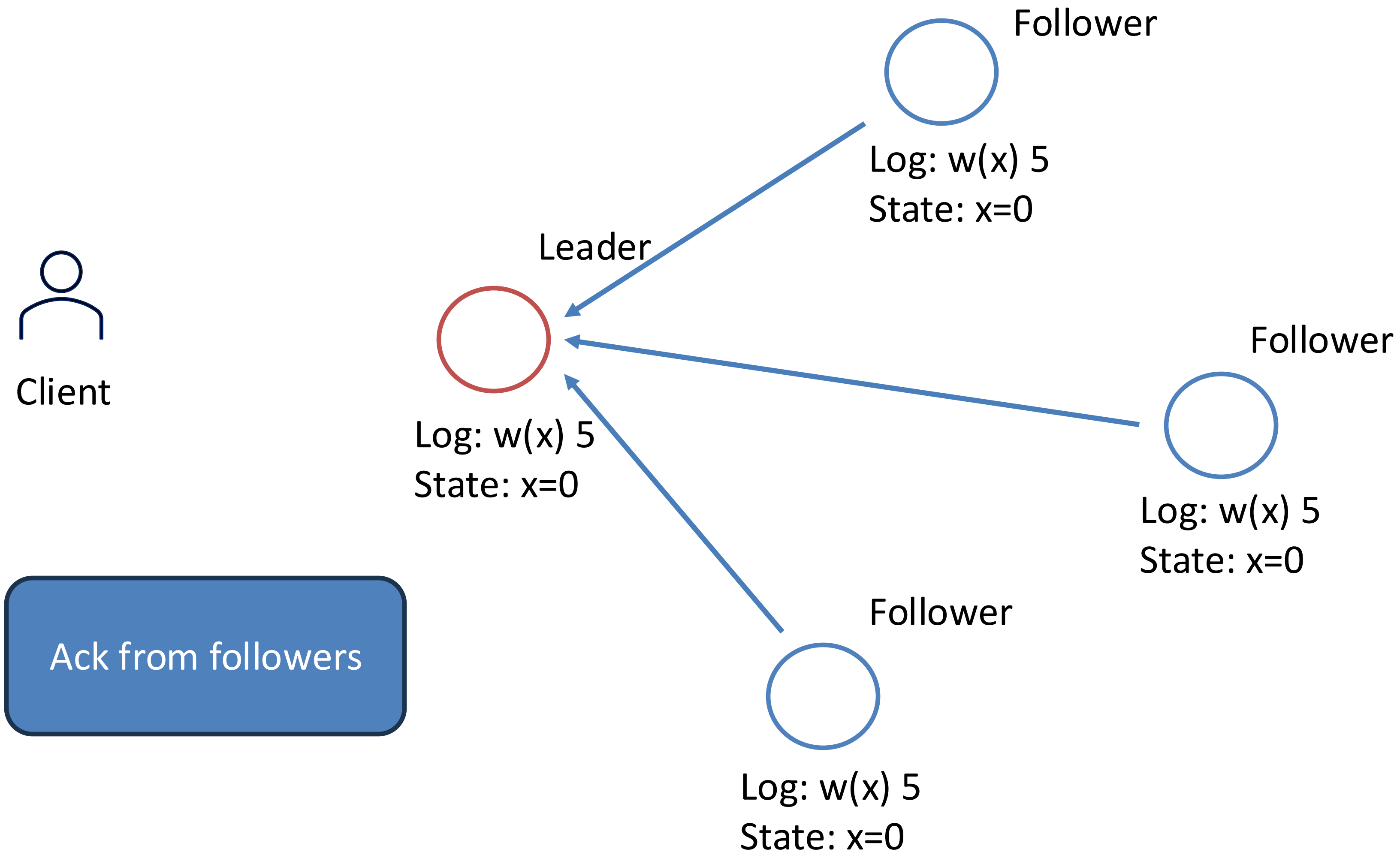


Normal operation

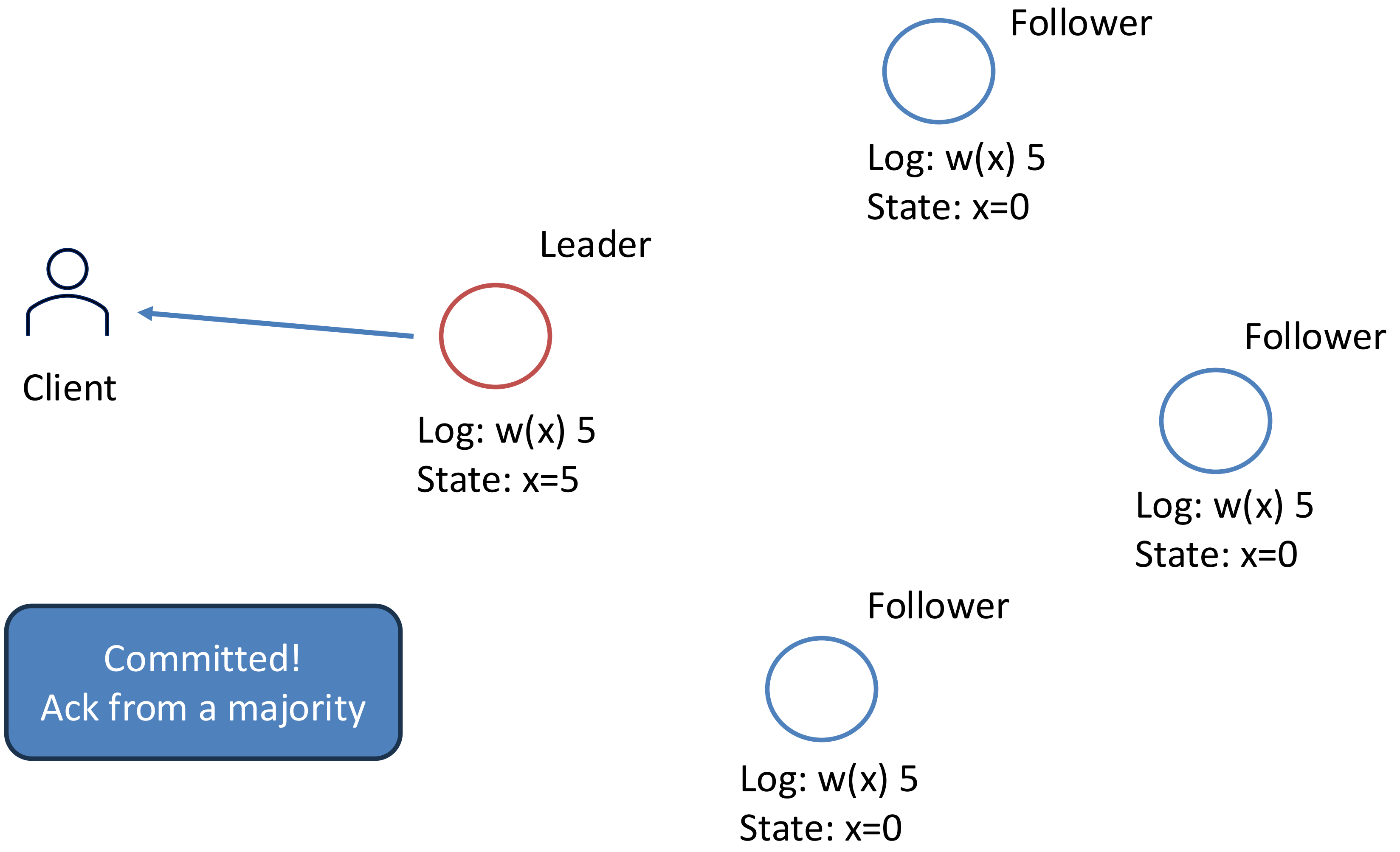


Write to log and propagate

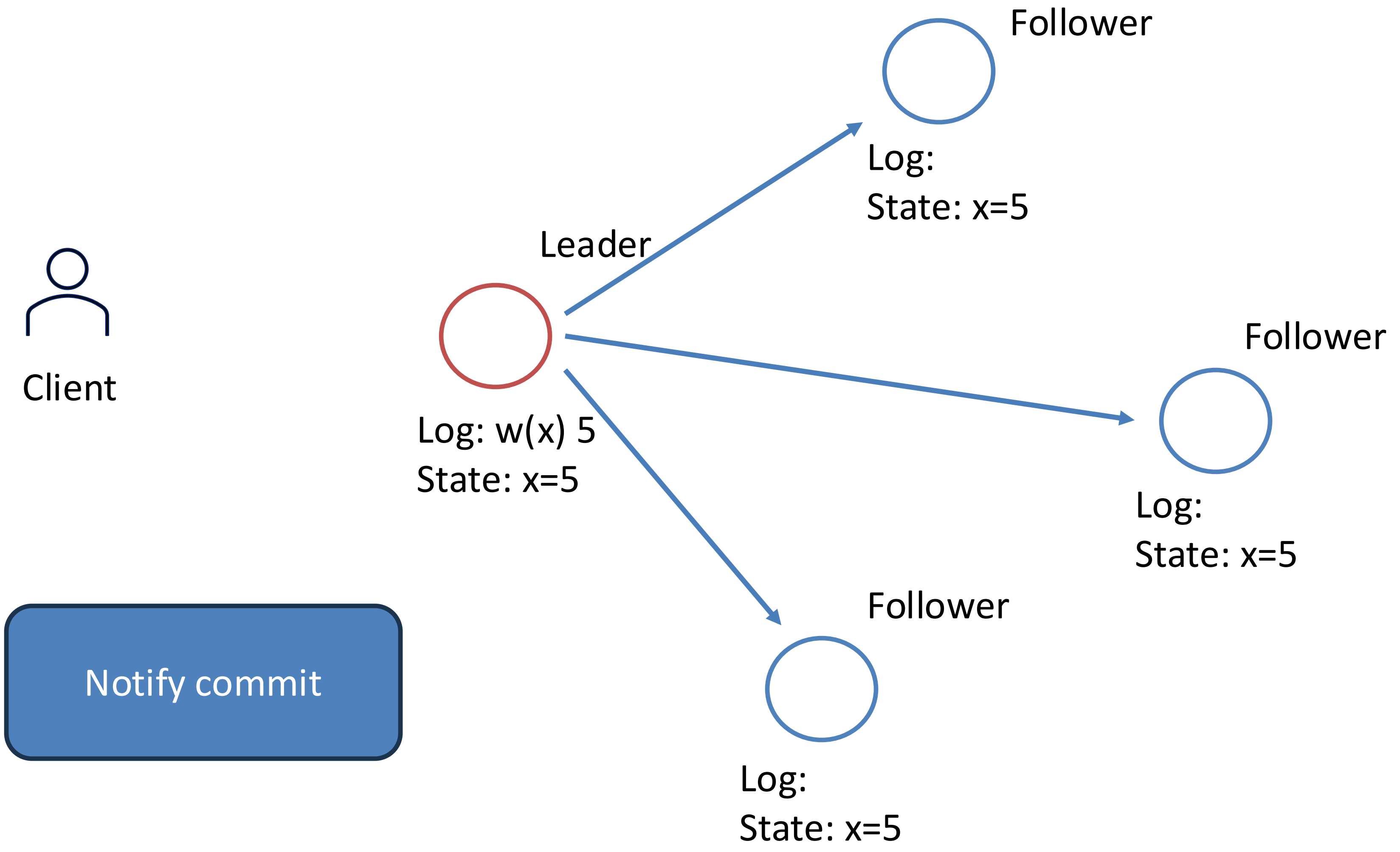
Normal operation



Normal operation



Normal operation

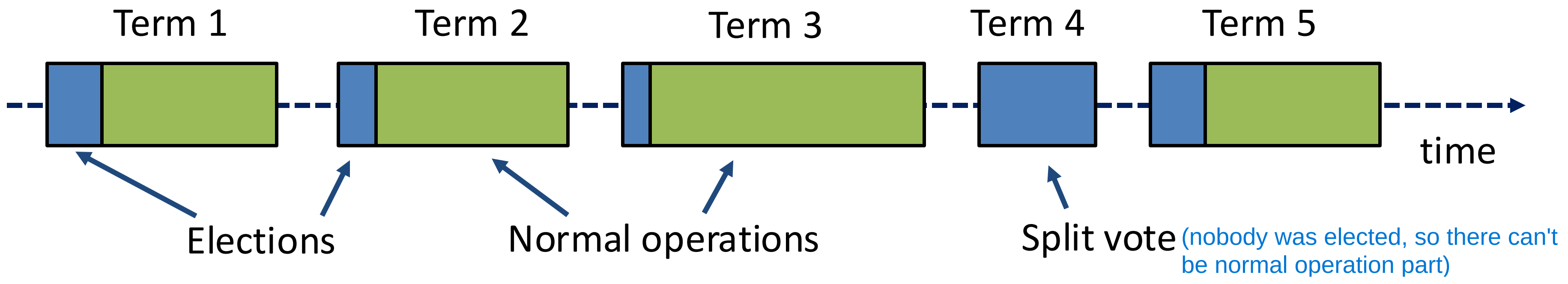


Leader election

- The leader periodically sends AppendEntries messages with its unacknowledged log entries
  - Possibly empty (works like an earth-bit)

- 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 high probability than only one election is started in a given time, so one candidate, so fast election
  - Range: 150 – 300 ms
    much larger than a typical network delay.

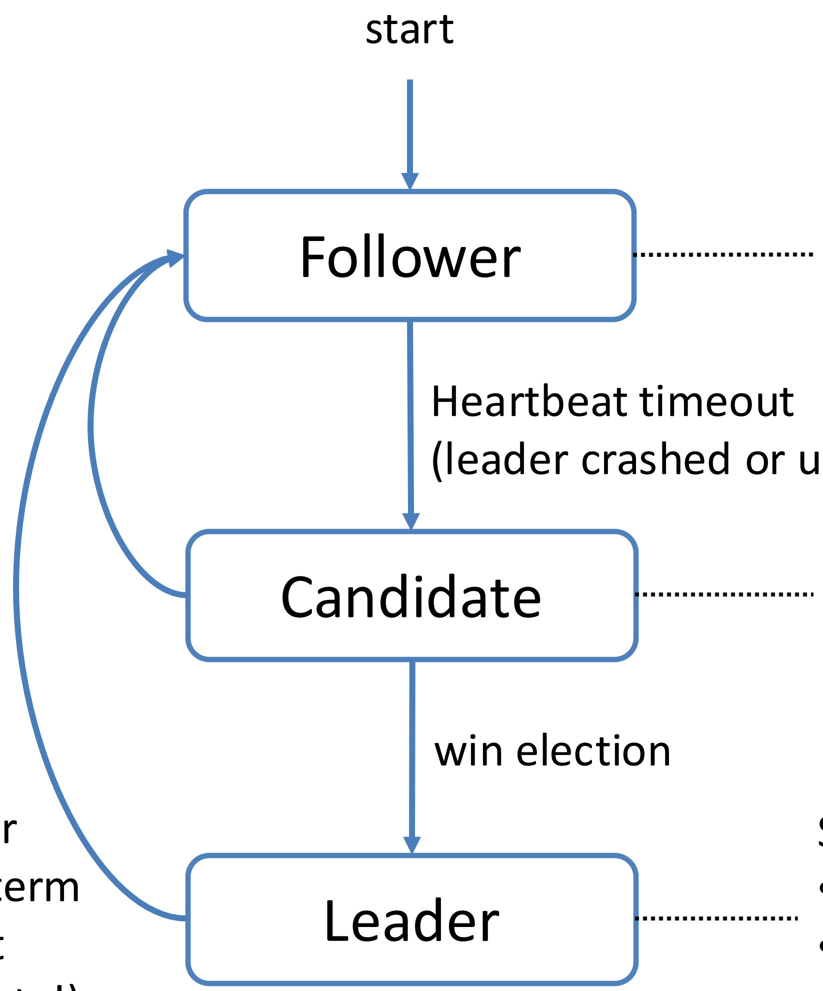
Terms



- 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

--> in practise it highly improbable. Notice that in THEORY inifinite split vote terms can happen, but it never happens.

Server states and messages



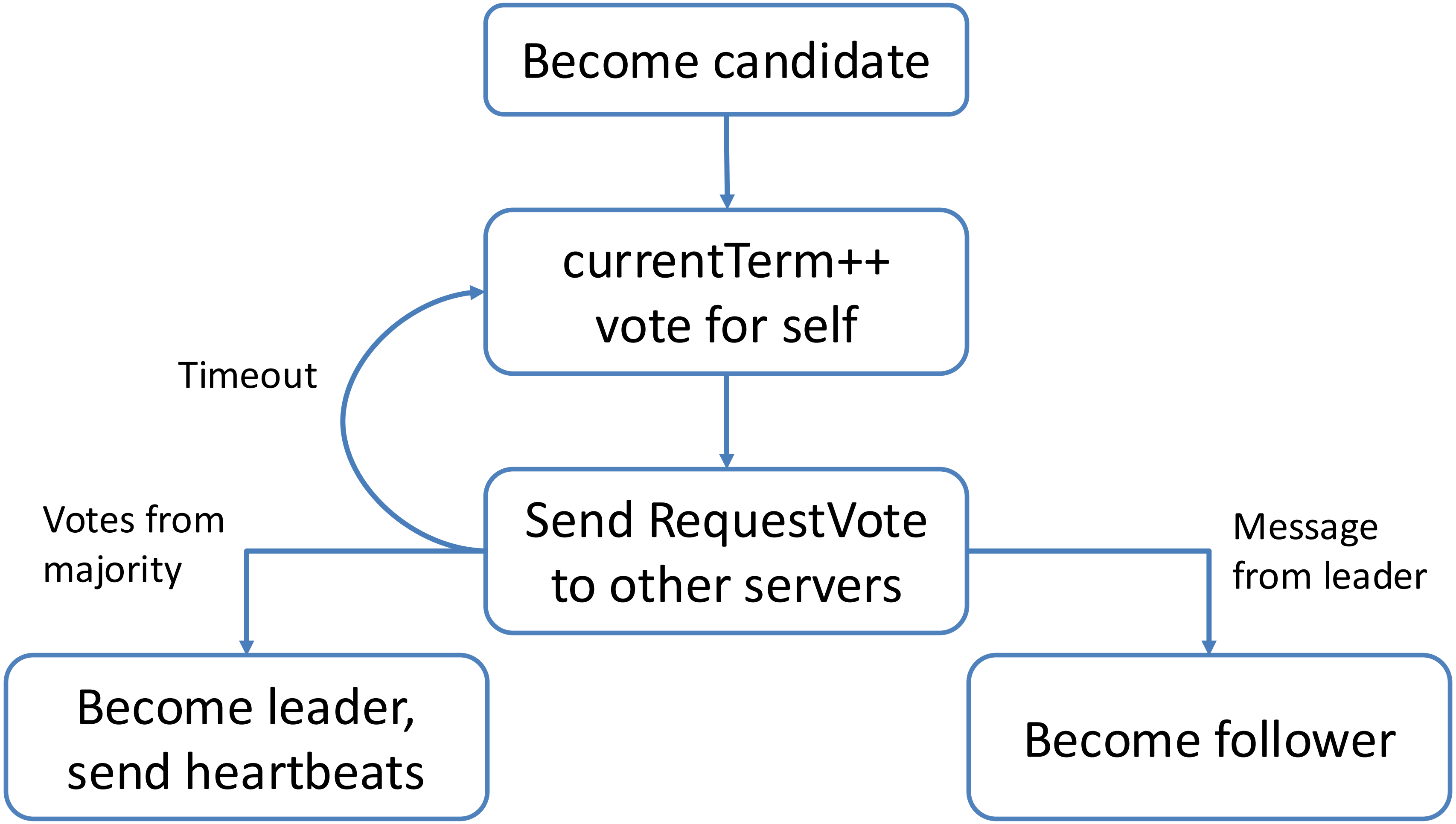
Passive: does not send messages
But waits for regular heartbeats

Sends a RequestVote message to get elected as leader

Sends AppendEntries messages:

- To replicate its log
- Empty messages also used as heartbeats

Leader election



Election correctness

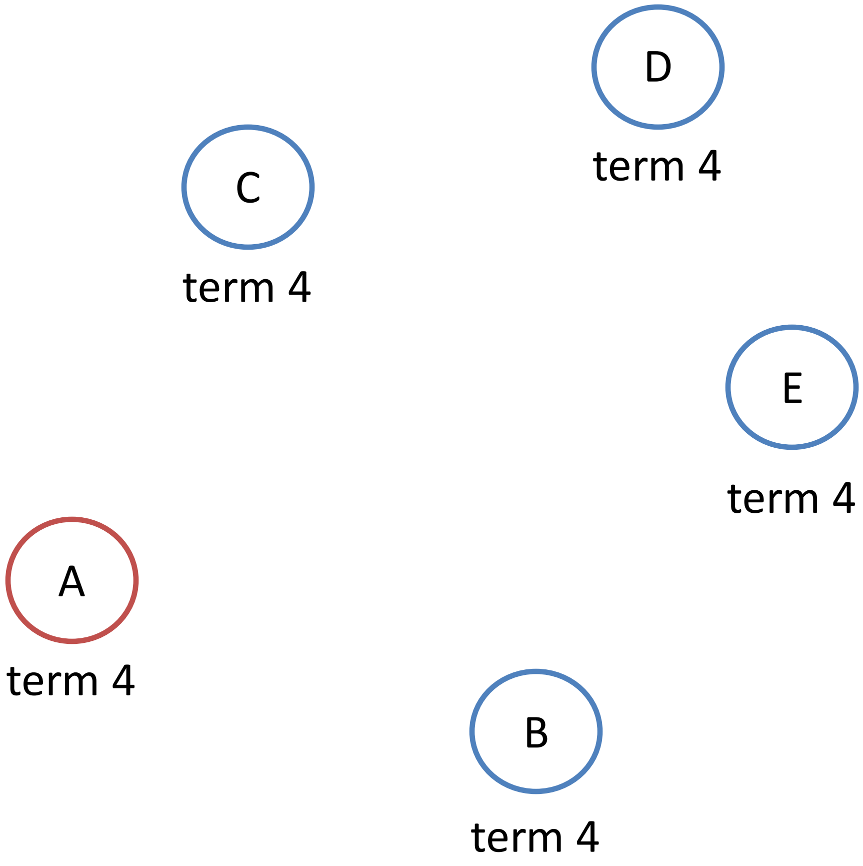
- Safety: allow at most one winner per term
  - Each server gives only one vote per term (persist on disk)
  - Majority required to win election

- 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

since we operate in asynchronous assumptions

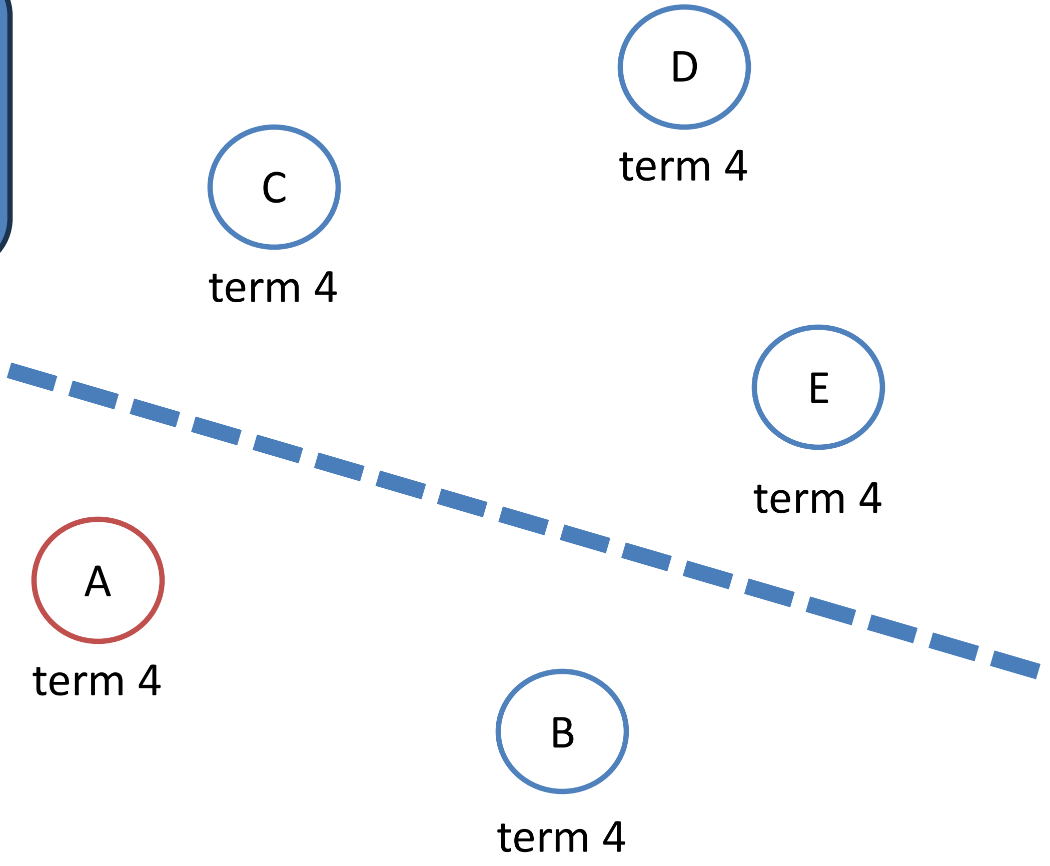
Network partitions

Consider a network of 5 nodes.
A is the leader for term 4.



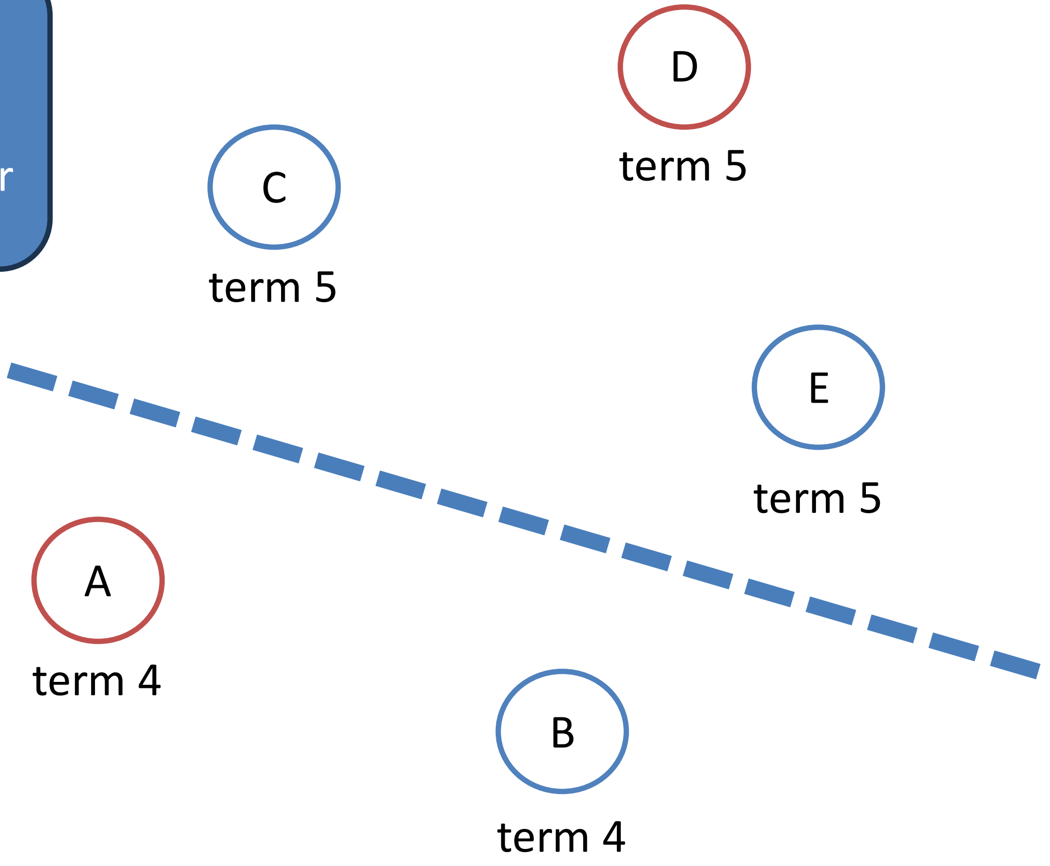
Network partitions

Now there is a network partition. A and B still consider A as the leader.



Network partitions

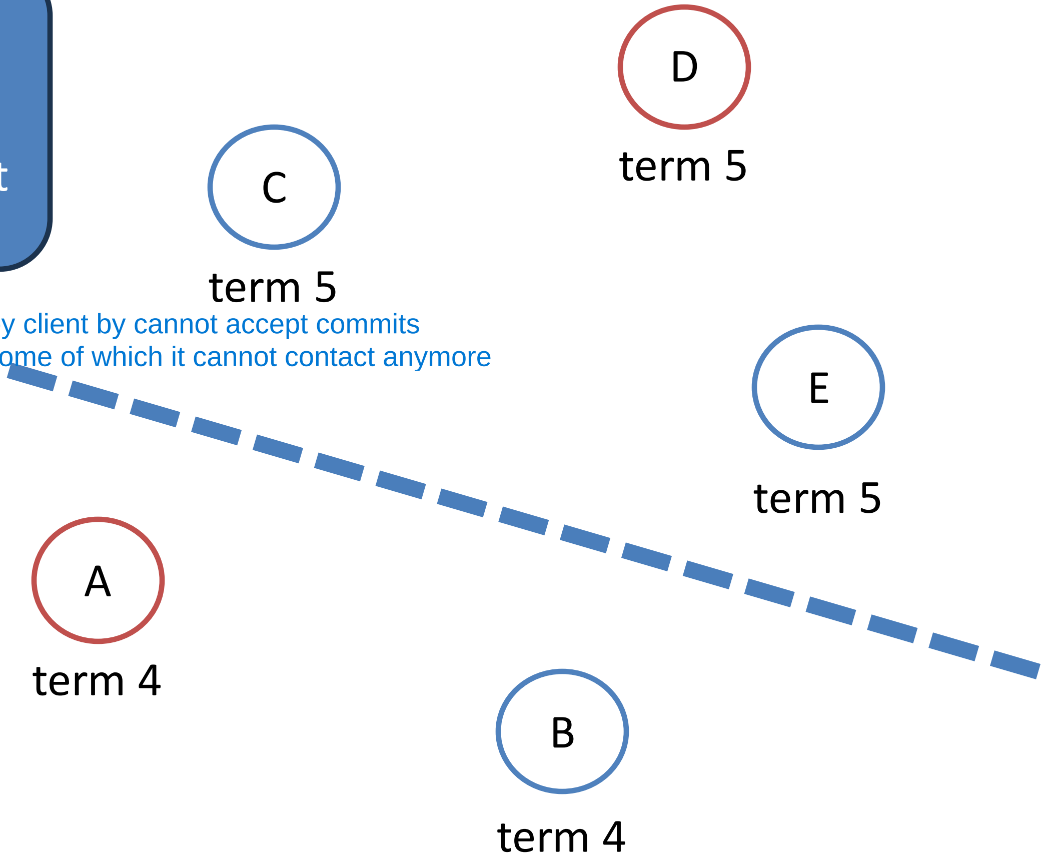
C, D, E will eventually start an election. Let's say that D wins and becomes the leader (for term 5).



Network partitions

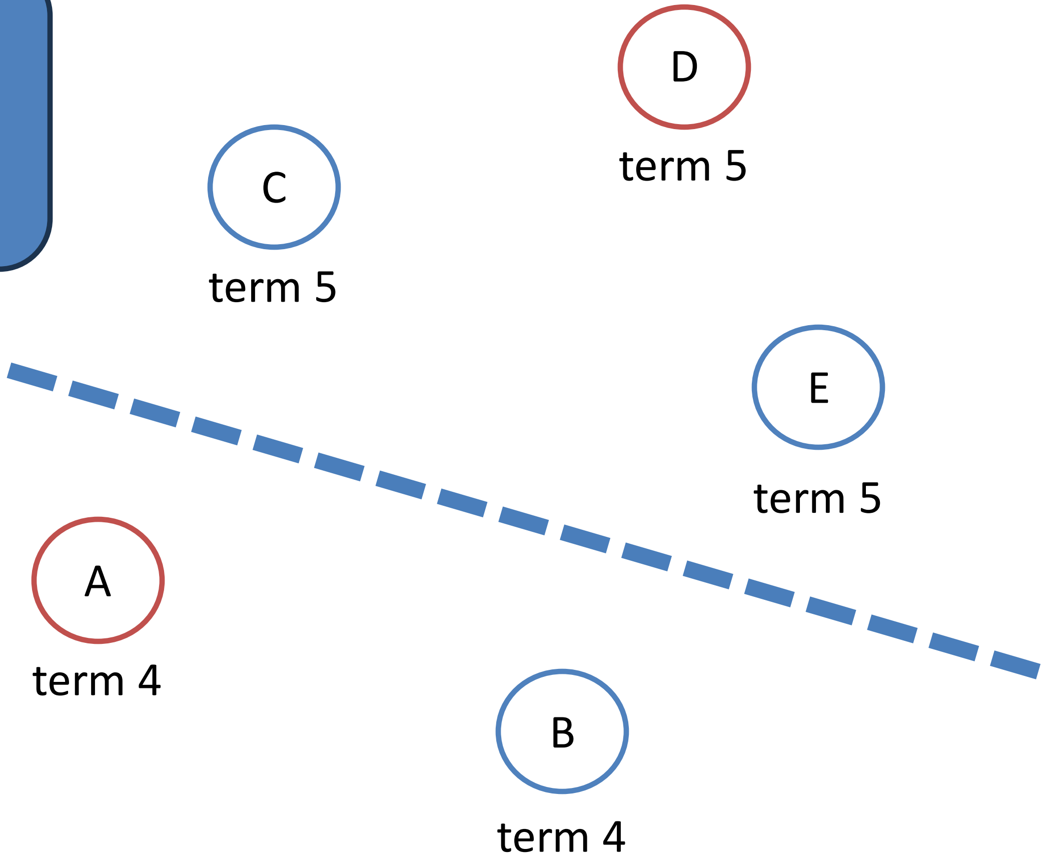
Clients may still send commands to A, but it will not be able to commit them (it cannot reach a majority).

In this phase, A can be still reached by client by cannot accept commits because it need ack from all nodes, some of which it cannot contact anymore



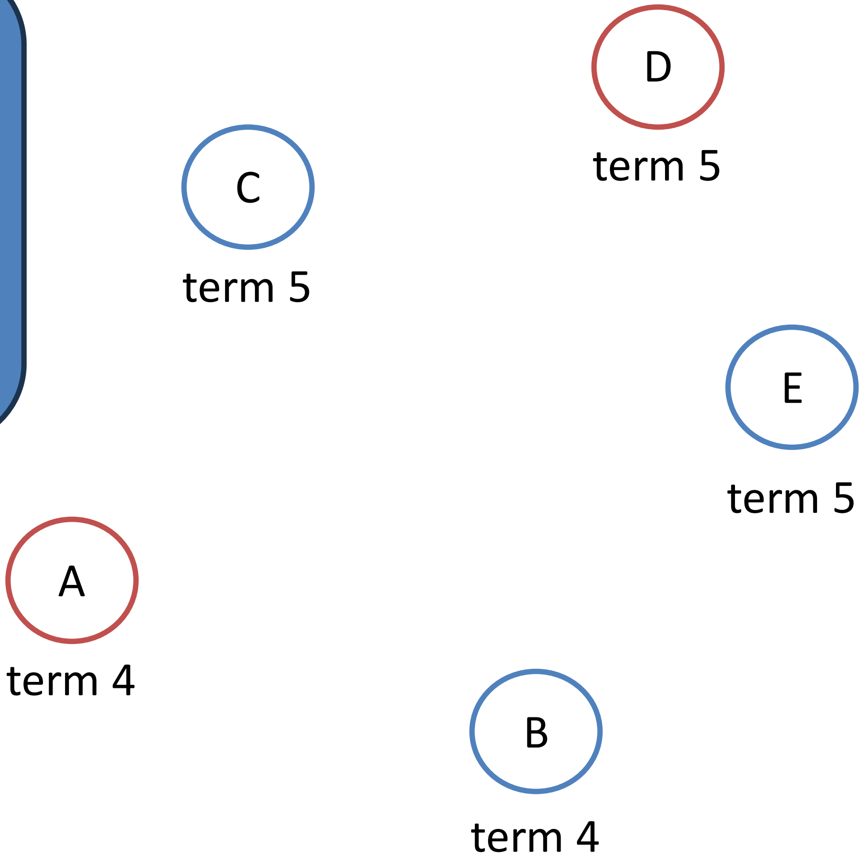
Network partitions

Clients may also send commands to D, which will be able to commit them.

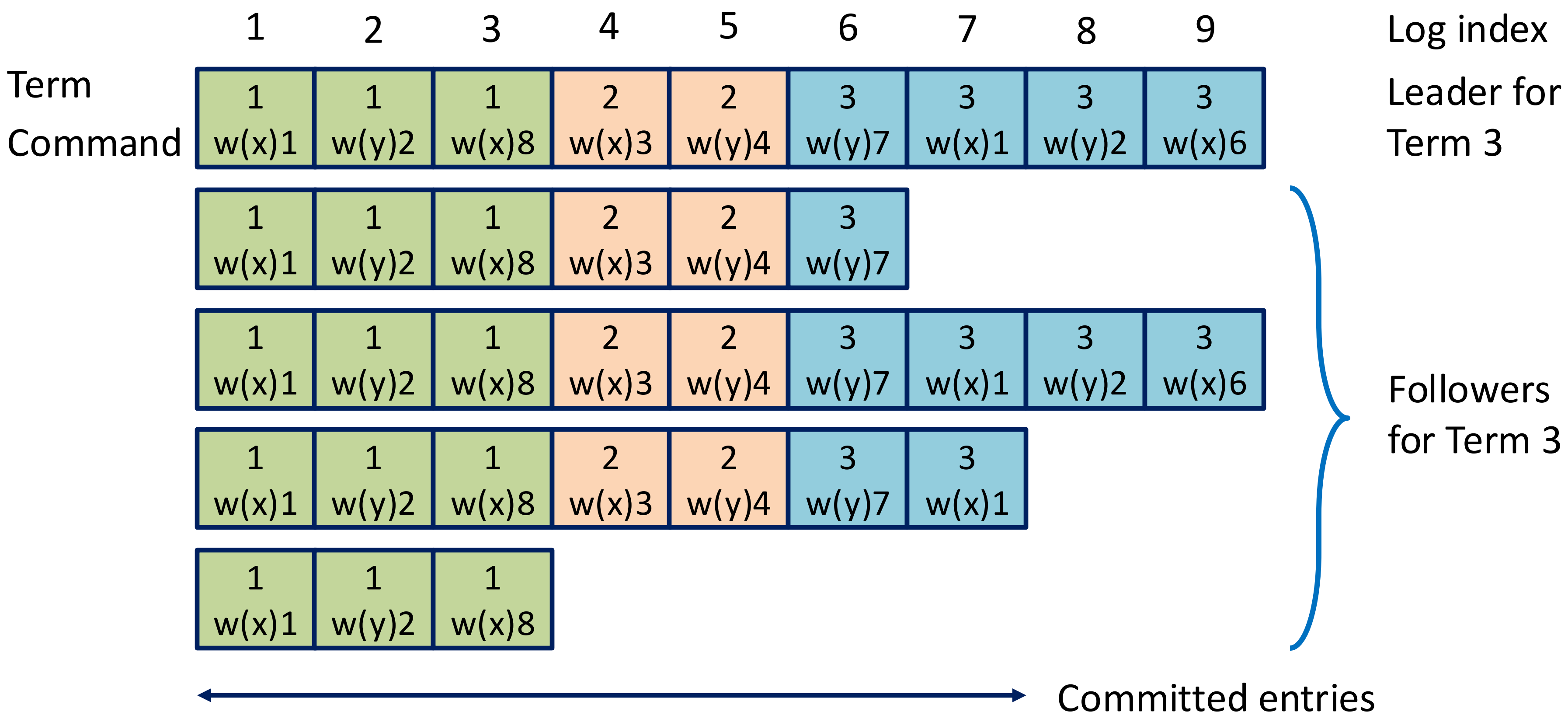


Network partitions

When we remove the network partition, A will eventually receive messages from term 5, learn that it is not up to date and step back as a leader.



Log structure



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

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

Log inconsistencies

- 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

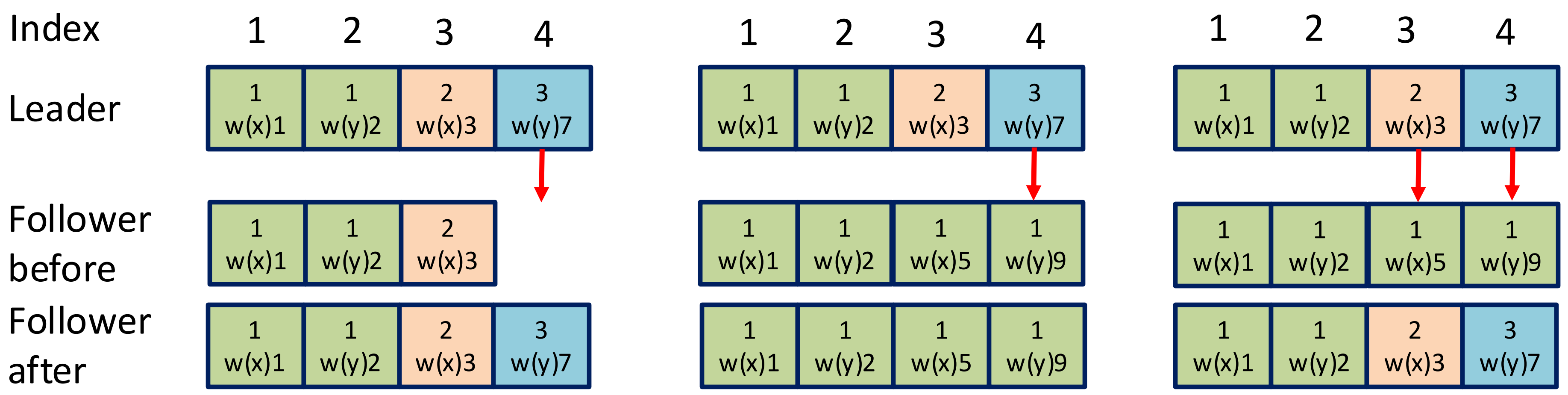
Log matching property

- 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



Example 1: ok

Example 2: mismatch

Example 3: ok

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

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

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? (garbage collection)
  - Change membership: how to consistently add/remove servers from the group? part of the state of the state machine include membership
  - 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

Mostof database replicate the partition (first layer) and within partition use raft (for replicating components, secon layer). In this setup you don't need 3PC.

Use in distributed DBMS

- Two layers
  - Replicated state machine to guarantee that individual partitions do not fail
    - Each partition is not managed by a single machine, …
    - … but by a set of machines that behave as one
      - E.g., 5 machines ensure that the partition is up even if 2 machines fail simultaneously
  - 2PC executes atomic commit across partitions
    - Coordinator (transaction manager) and participants (partitions) are assumed not to fail as they are replicated
    - Channels are assumed to be reliable: it is sufficient that a majority of nodes in a given partition is reachable

JC Corbett et al "Spanner's globally distributed database", ACM TOCS, 2013

Blockchains

BYZANTINE CONDITIONS

BFT consensus

--> protocols that can tolerate byzantine failures.

- 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

- We will not consider these protocols in this course …
- … but we will see a strictly related technology

Blockchains

- Cryptocurrencies can be seen as replicated state machines Distributed consensus of account balances
- State: current balance of each user
- Stored in a replicated ledger (log)
  - Records all operations (transactions/payments)
- Byzantine environment
  - A misbehaving user may try to "double spend" their money
  - Creating inconsistent copies of the log

Like raft, the fact to be "stored" is the sequence of operations done. It is needed to agree on the order of operations

Assumptions and guarantees

- Assumptions
  - Very large number of nodes
  - The set of participating nodes is unknown upfront
  - No single entity owns the majority of compute resources

permissionless membership: everyone can be part of the group.

- 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

Bitcoin blockchain

- 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

Bitcoin blockchain

- 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!

Bitcoin blockchain

- 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!