Distributed Systems

Consensus

Thoai Nam

High Performance Computing Lab (HPC Lab)

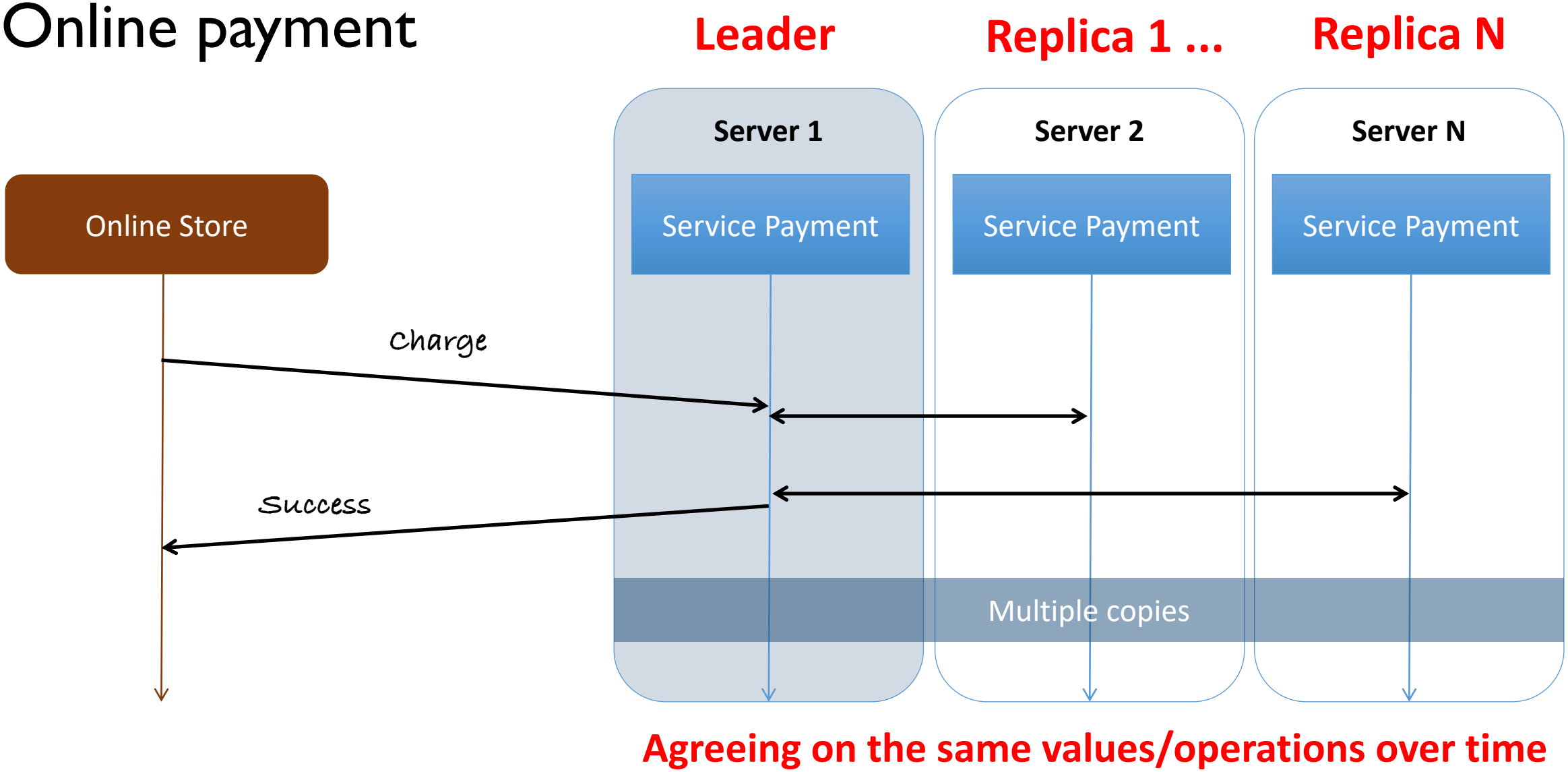
Faculty of Computer Science and Engineering

HCMC University of Technology

Multicast/Broadcast

- FIFO ordering: If a correct process issues multicast/broadcast(*g*,*m*) and then multicast/broadcast(*g*,*m'*), then every correct process that delivers *m'* will have already delivered *m*
- Causal ordering: If multicast/broadcast(*g*,*m*) → multicast/broadcast(*g*,*m'*) then any correct process that delivers *m'* will have already delivered *m*
  - Note that → counts multicast/broadcast messages delivered to the application, rather than all network messages
- Total ordering: If a correct process delivers message *m* before *m'*, then any other correct process that delivers *m'* will have already delivered *m*.

Online payment



Agreeing on the same values/operations over time

State machine replication (1)

- The main idea behind is that a single process (the leader) broadcasts the operations that change its state to other process, the followers (replicas)
- Total order broadcast: every node delivers the same messages in the same order
- The followers execute the same sequence of operation as the leader, then the state of each follower will match the leader

State machine replication (SMR): - every replica acts as SM

- FIFO-total order broadcast: every update to all replicas
- Replica deliver update message: apply to own state
- Applying an update is deterministic – even errors
- Replica is a state machine: starts in fixed initial state, goes through same sequence of state transitions in the same order → all replicas end up in the same state.

State machine replication (2)

Closely related ideas:

- Serializable transactions (execute in delivery order) – Active/Passive replication
- Blockchains, distributed ledgers, smart contracts

Limitations:

- Cannot update state immediately, have to wait for delivery through broadcast
- Need fault-tolerant total order broadcast

Other broadcast

Broadcast	Assumptions about state update function
Total broadcast	Deterministic (SMR)
Causal	Deterministic, concurrent updates commute
Reliable	Deterministic, all updates commute
Best-effort	Deterministic, commutative, idempotent, tolerate message loss

When updates are commutative, replicas can process updates in different orders and still end up in the same state.

Consensus

- A fundamental problem studied in distributed systems, which requires a set of processes to agree on a value in fault tolerant way so that:
  - Every non-faulty process eventually agrees on a value
  - The final decision of every non-faulty process is the same everywhere
  - The value that has been agreed on has been proposed by a process
- Consensus has a large number of practical applications
  - Commit transactions, Decisions in general (votes are involved)
  - Hold a lock (Mutual exclusion), Failure detection (Byzantine)
- Any problem that requires consensus can be solved with a state machine replication.

Consensus and total order broadcast

- Leader regulates the consensus with the nodes via total order broadcast
  - Single point of failure
  - Failover: human operator chooses a new leader, e.g., databases
- Election algorithms can automate the selection of the leader (properties?)
- Consensus and total order broadcast are formally equivalent
- Common consensus algorithms:
  - Paxos: single-value consensus
  - Multi-paxos: generalization to total order broadcast
  - Raft, Viewstamped replication, Zab: FIFO-total order broadcast

Consensus system models

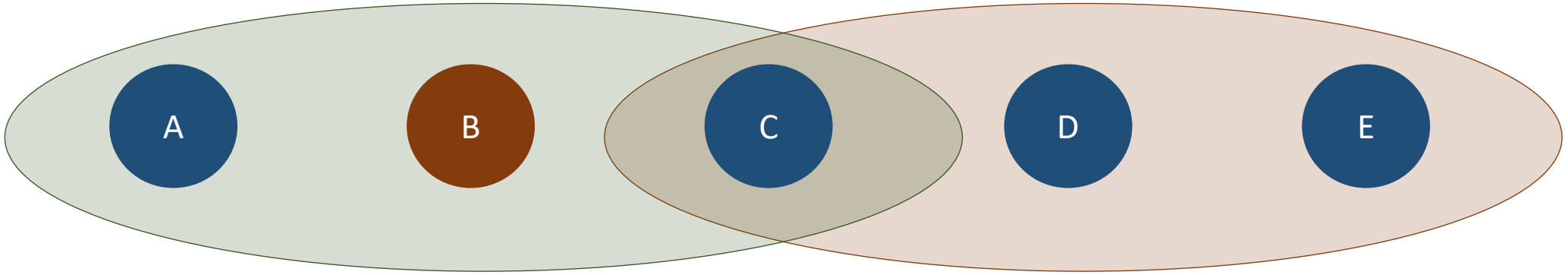
- Paxos, Raft, etc., assume a partially synchronous crash-recovery model.
- Why not asynchronous?
  - FLP result (Fisher, Lynch, Paterson): There is no deterministic consensus algorithm that is guaranteed to terminate in an asynchronous crash-stop system model
  - Paxos, Raft and others, use clocks only used for timeouts/failures detector to ensure progress. Safety (correctness) does not depend on timing
- There are also consensus algorithms for a partially synchronous Byzantine system model (used in blockchains)

- Practical considerations
  - ZooKeeper (https://zookeeper.apache.org/)
  - etcd (https://etcd.io/)

Leader in consensus

Some consensus uses a leader to sequence messages

- Use a failure detector (timeout) to determine suspected crash or unavailable leader
- On suspected leader crash, a new leader is elected
- Prevent two leaders at the same time “split-brain”



Elects a leader (B)

Cannot elect a different leader as C already voted (B)

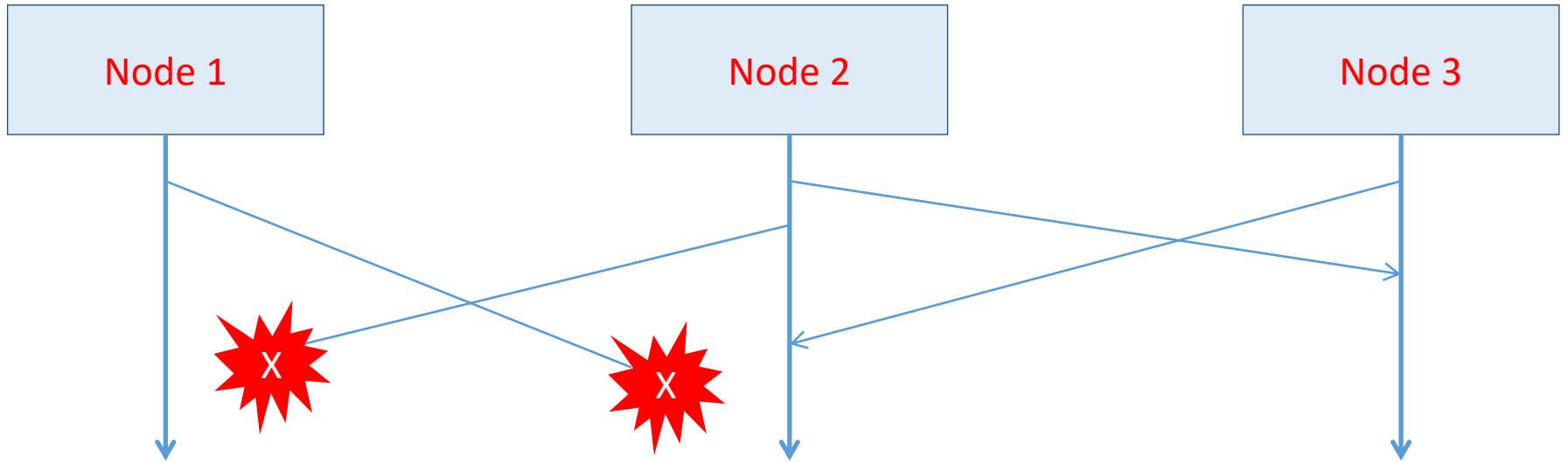
Ensure leader per term:

- Term is incremented every time a leader election is started
- A node can only vote once per term
- Require a quorum of nodes to elect a leader in a term

A single leader?

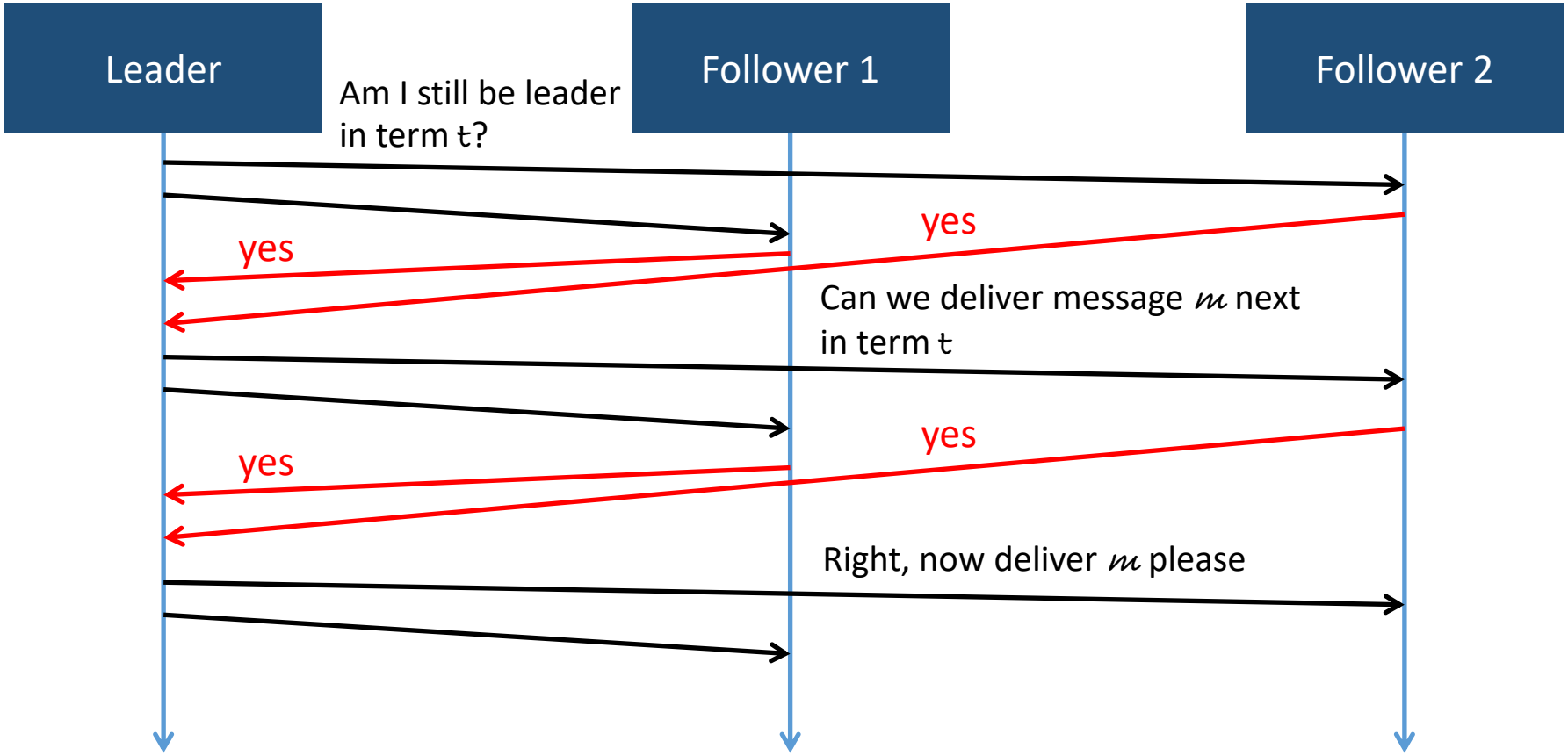
Can guarantee unique leader per term?

- Cannot prevent having multiple leaders from different terms
- Example: Node 1 is leader in *term t*, but due to a network partition it can no longer communication with Node 2 and 3



Node 2 and 3 may elect a new leader in term t + 1
Node 1 may not even know that a new leader has been elected!

Checking if a leader has been voted out



For every decision (message to deliver), the leader must first get acknowledgements from a quorum.

[source] Distributed Systems course given by Dr. Martin Kleppmann (University of Cambridge, UK)

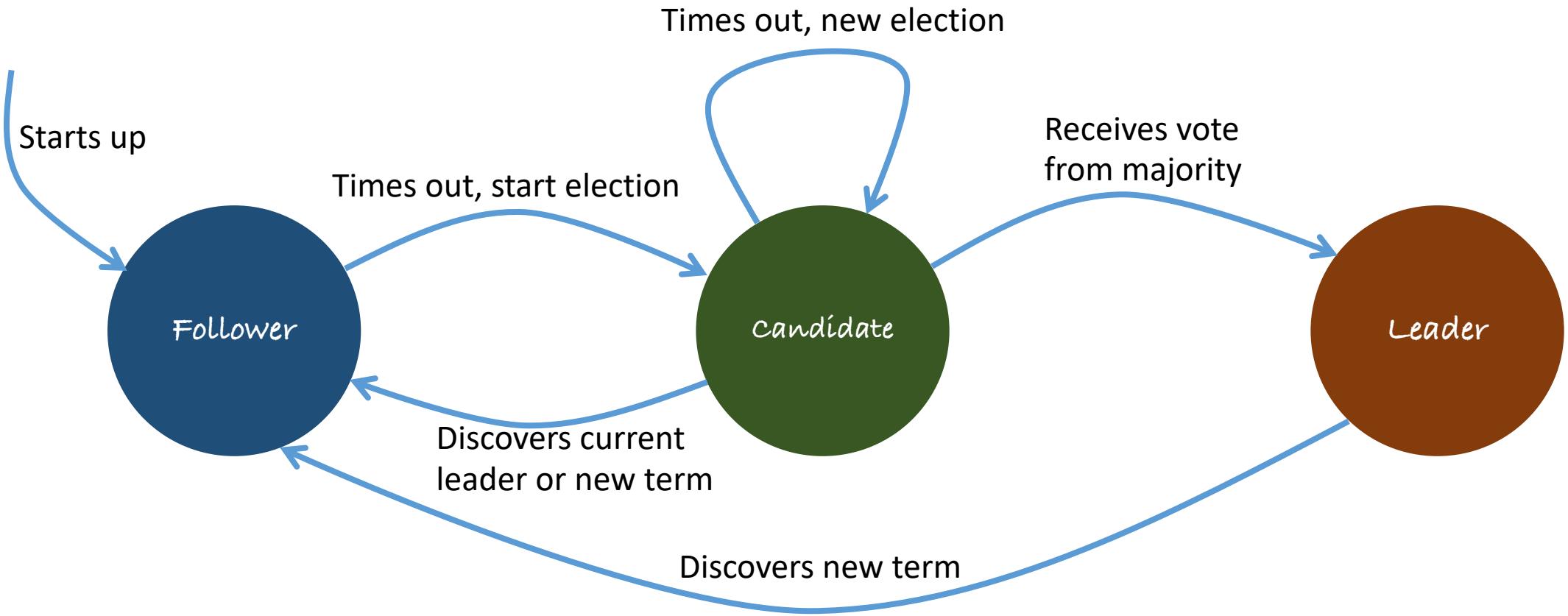
Raft

https://raft.github.io/

Raft

- Modern solution the problem of consistency
- An algorithm that guarantees the strongest consistency possible
- Raft is based on state machine replication
- In Raft, time is divided into election term
- A term is depicted by a logical clock and just increases forward
- The term starts by an election to decide who becomes a leader
- Raft guarantees that for any term there is at most one leader.

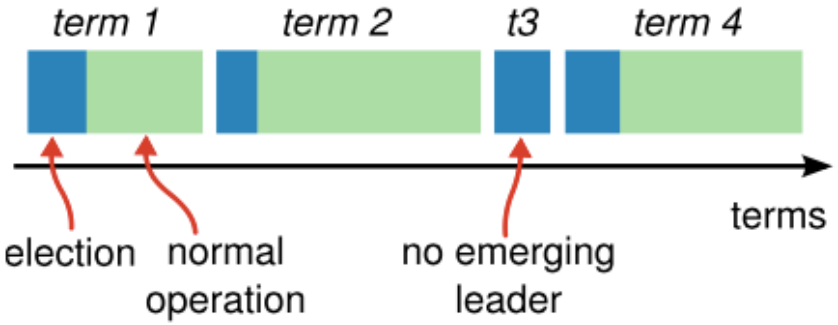
State machine replication in Raft (Raft algorithm)



Raft algorithm overview

- Every process starts as follower
- A follower expects to receive a periodic heartbeat from the leader containing the election term the leader was elected in
- If the follower does not receive any heartbeat within a certain period of time, a timeout fires and the leader is presumed dead
- The follower starts a new election by increment the current election term and transitioning to candidate state
- It then votes for itself and sends a request to all processes in the system to vote for it, stamping the request with the current election term.

Raft



Outcome

- The candidate wins the election: the candidate becomes a leader and starts sending out heartbeats to the other processes
- Another process wins the election: In this case, terms between process are compared, if another process claims to be the leader with a term greater or equal the candidate's term, it accepts the new leader and returns to the follower state
- A period of time goes by with no winner: very unlikely, but if it happens, then candidate will eventually time-out and starts a new election process.

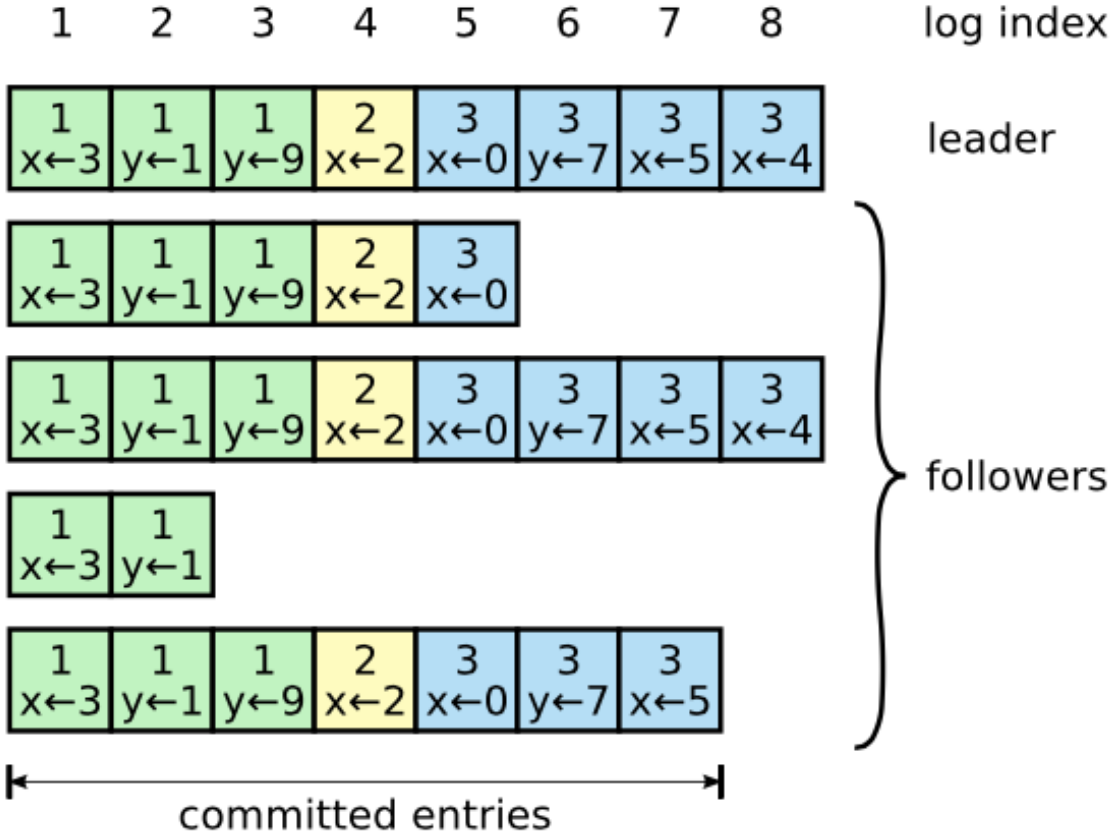
One single leader guarantee is enough?

- On way to avoid dynamic leaders is by using a fencing token (a number that increases every time a distributed lock is acquired - a logical clock).

Raft

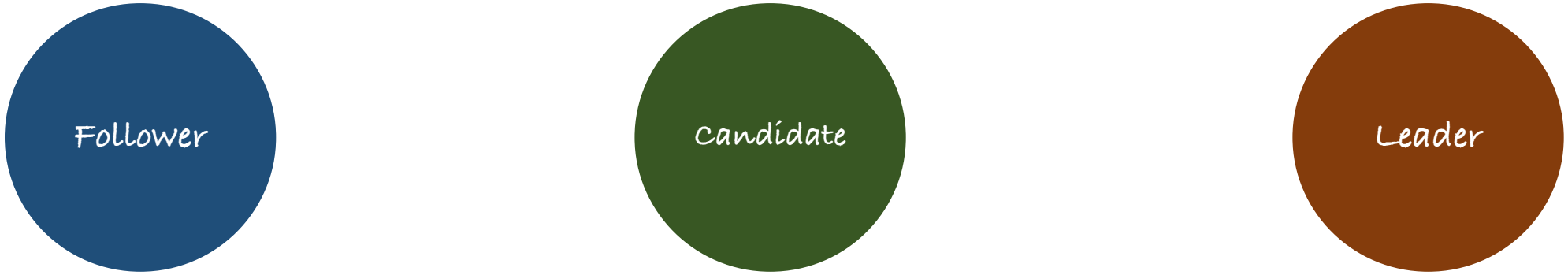
Log replication

- The leader is the only one that can make changes to the replica states
- A log is created inside the leader and then replicated across the followers (log replication)
- When the leader applies an operation to its local state, it appends a new log entry into its own log (operation is logged).



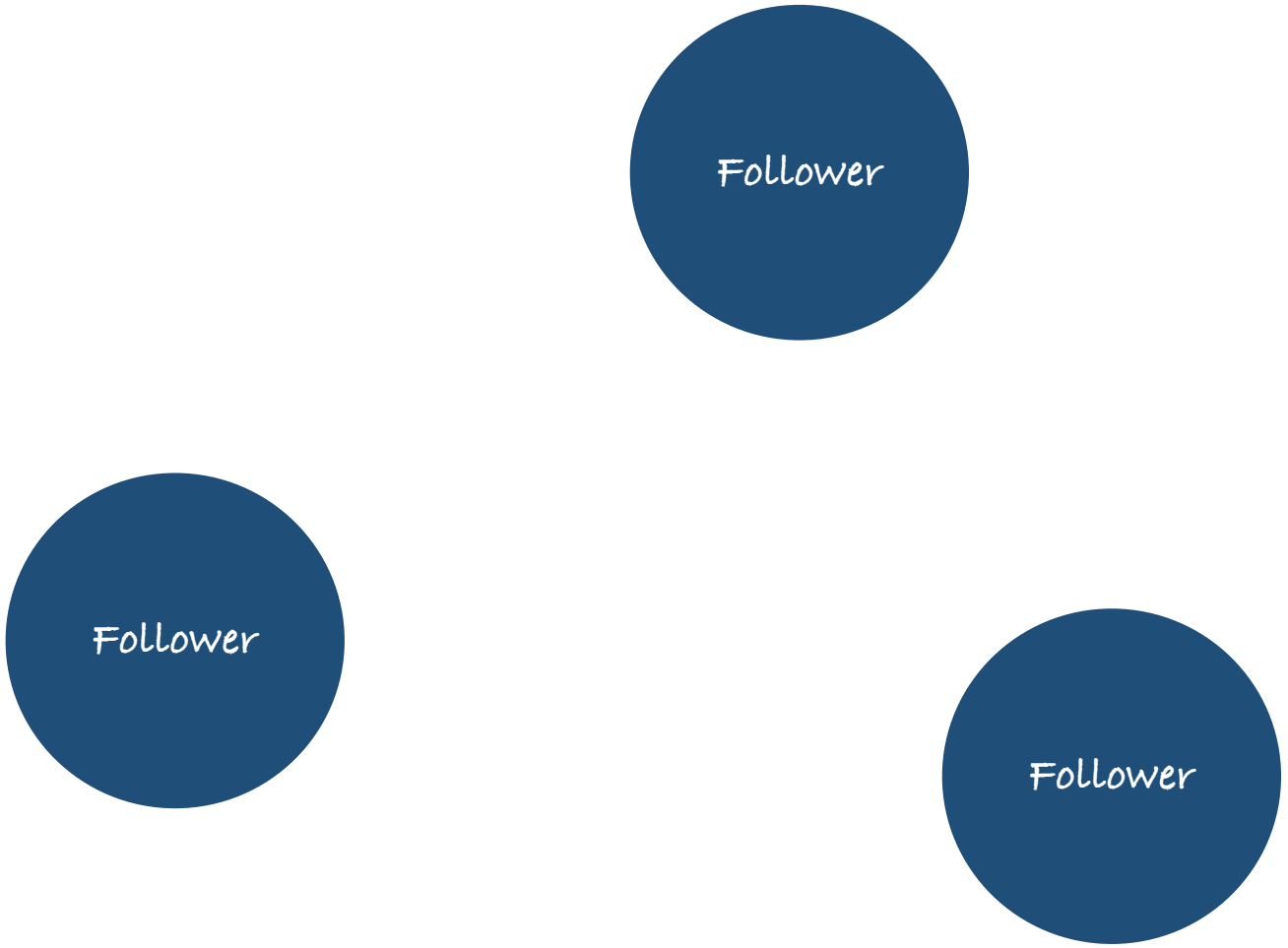
Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

Raft

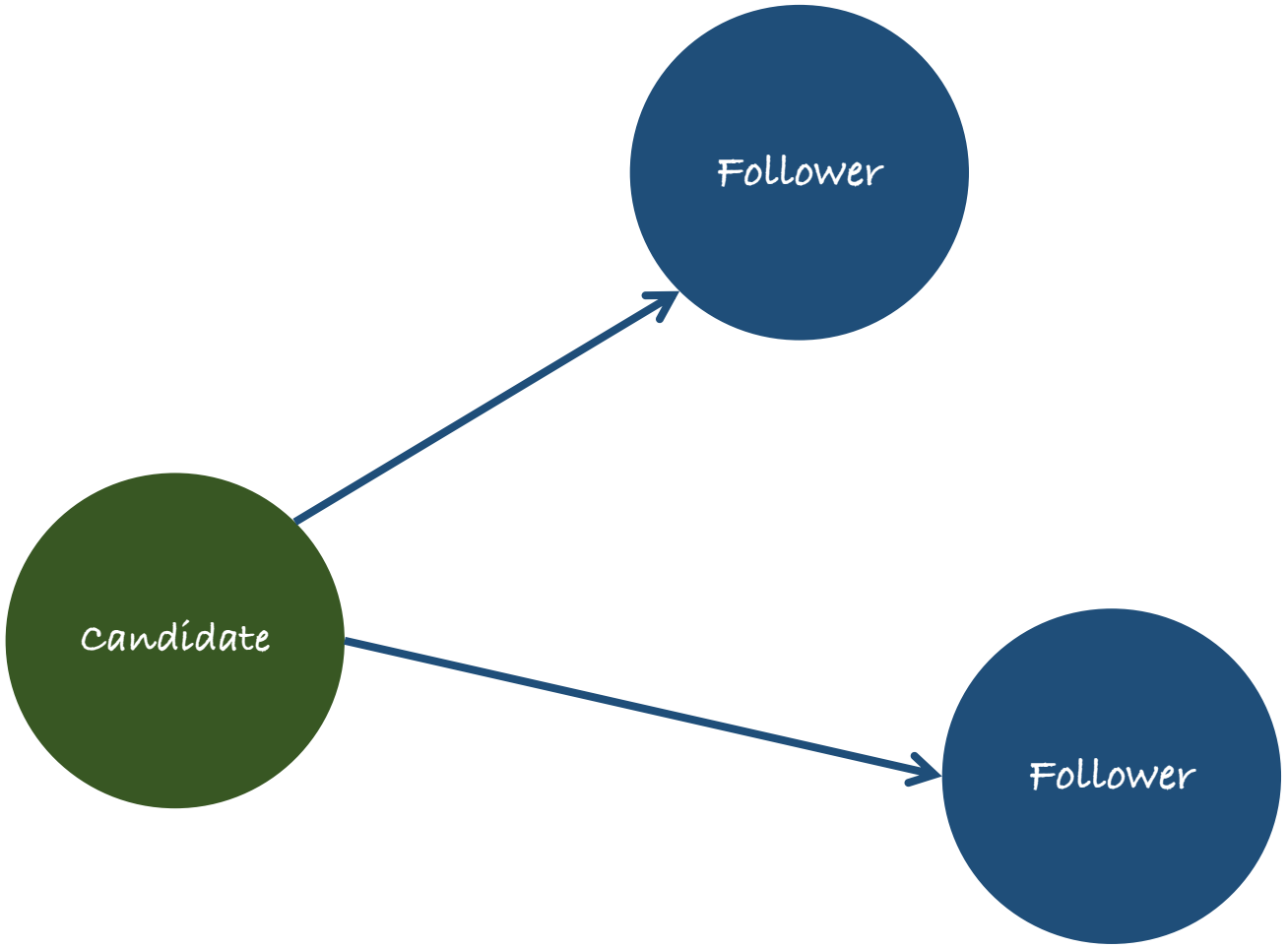


We need a protocol structure to handle data consistency across multiple nodes

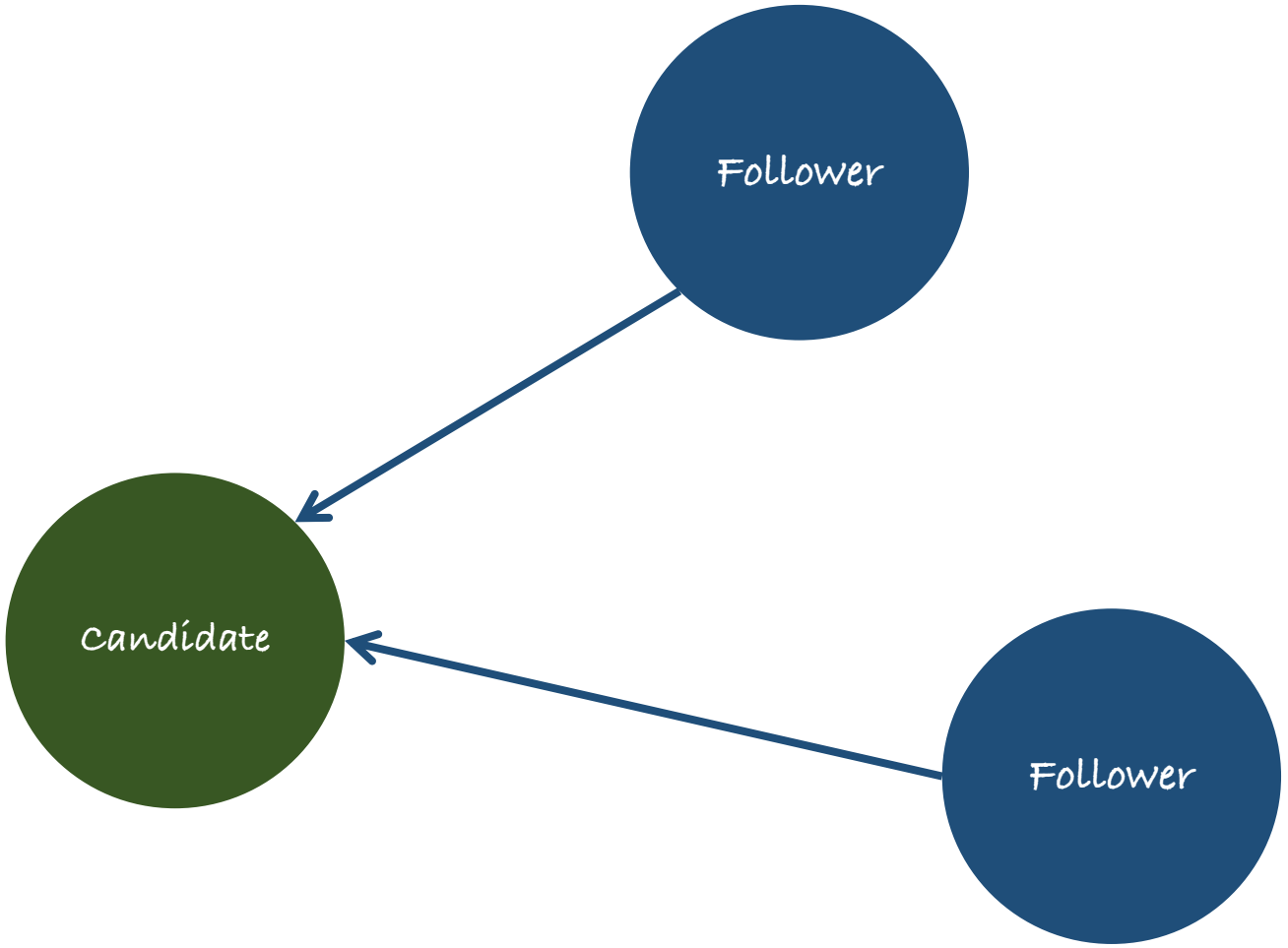
Raft



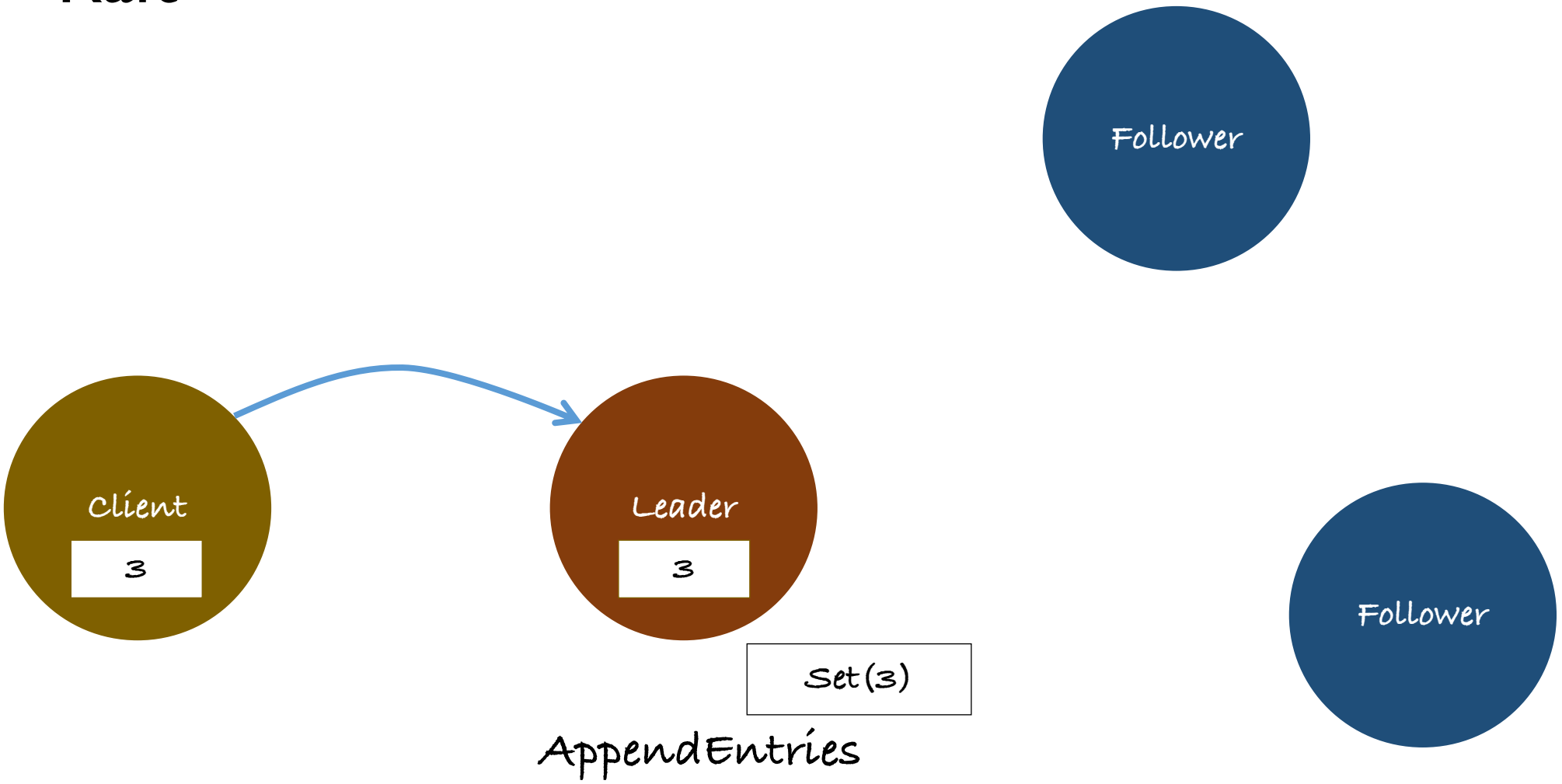
Raft



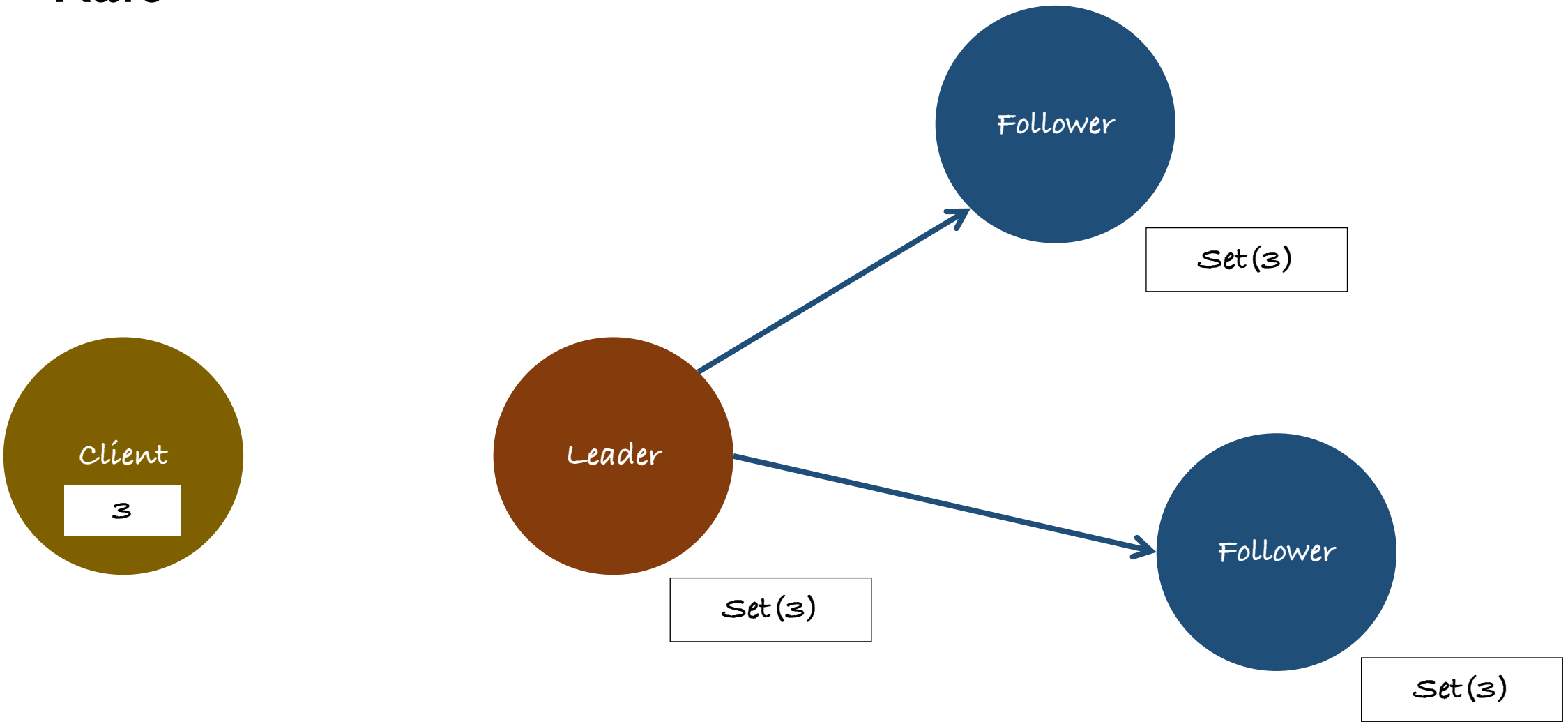
Raft



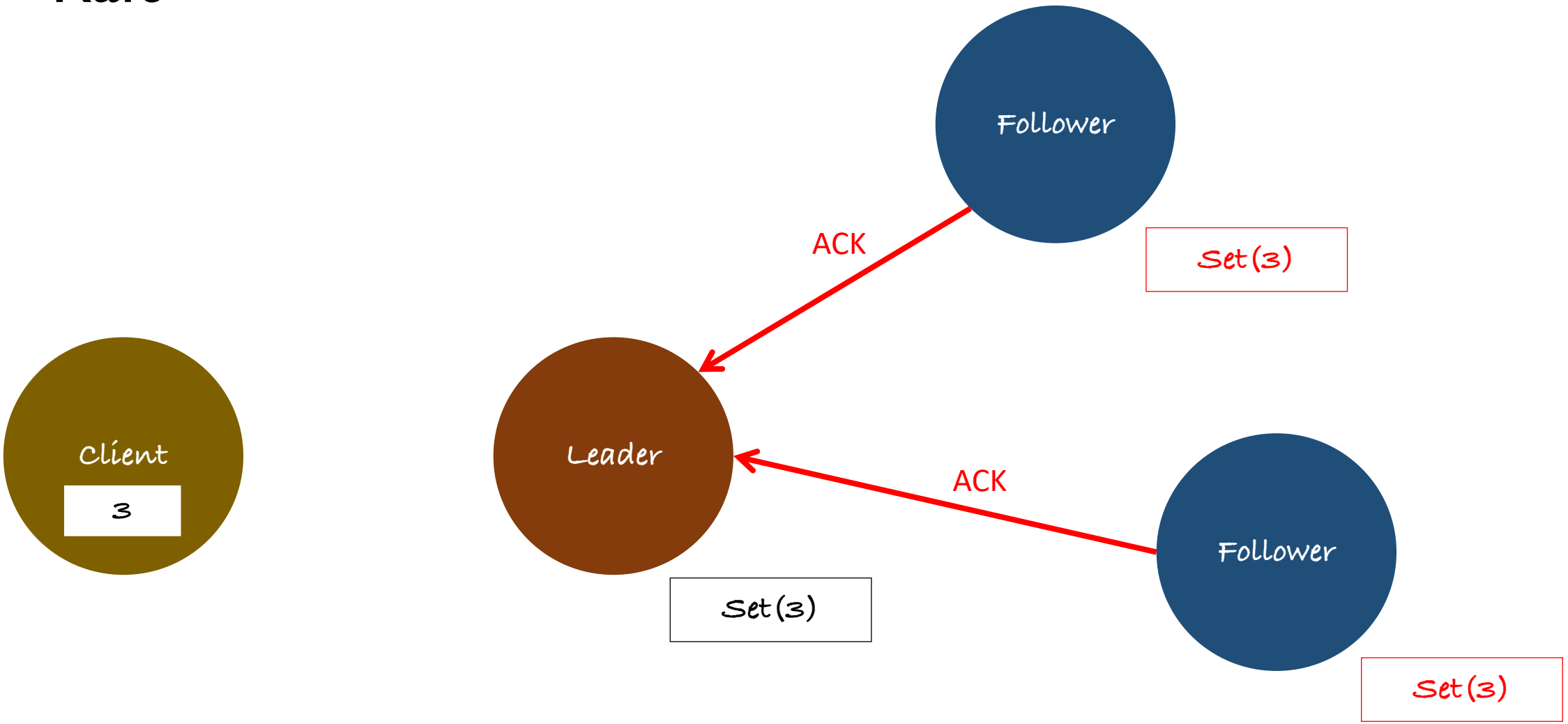
Raft



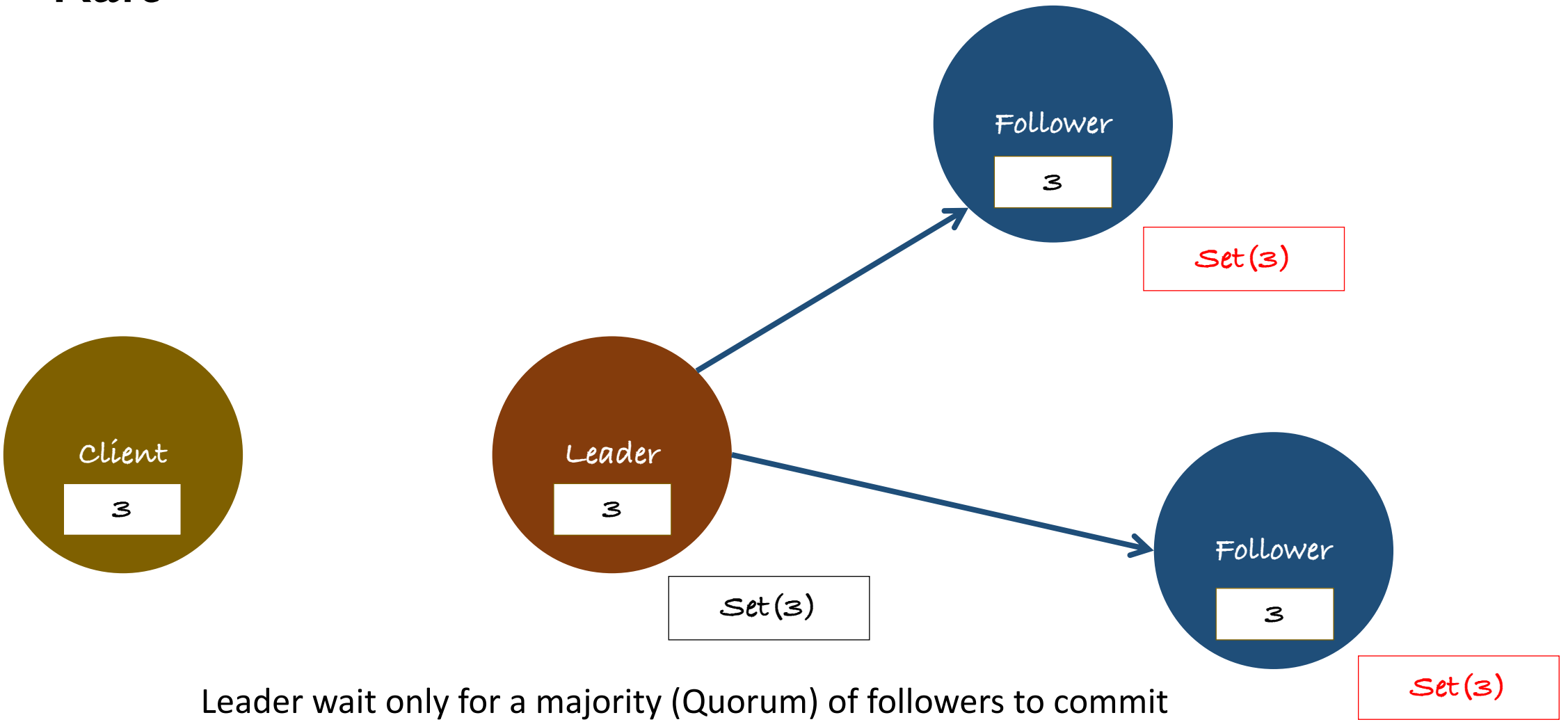
Raft



Raft

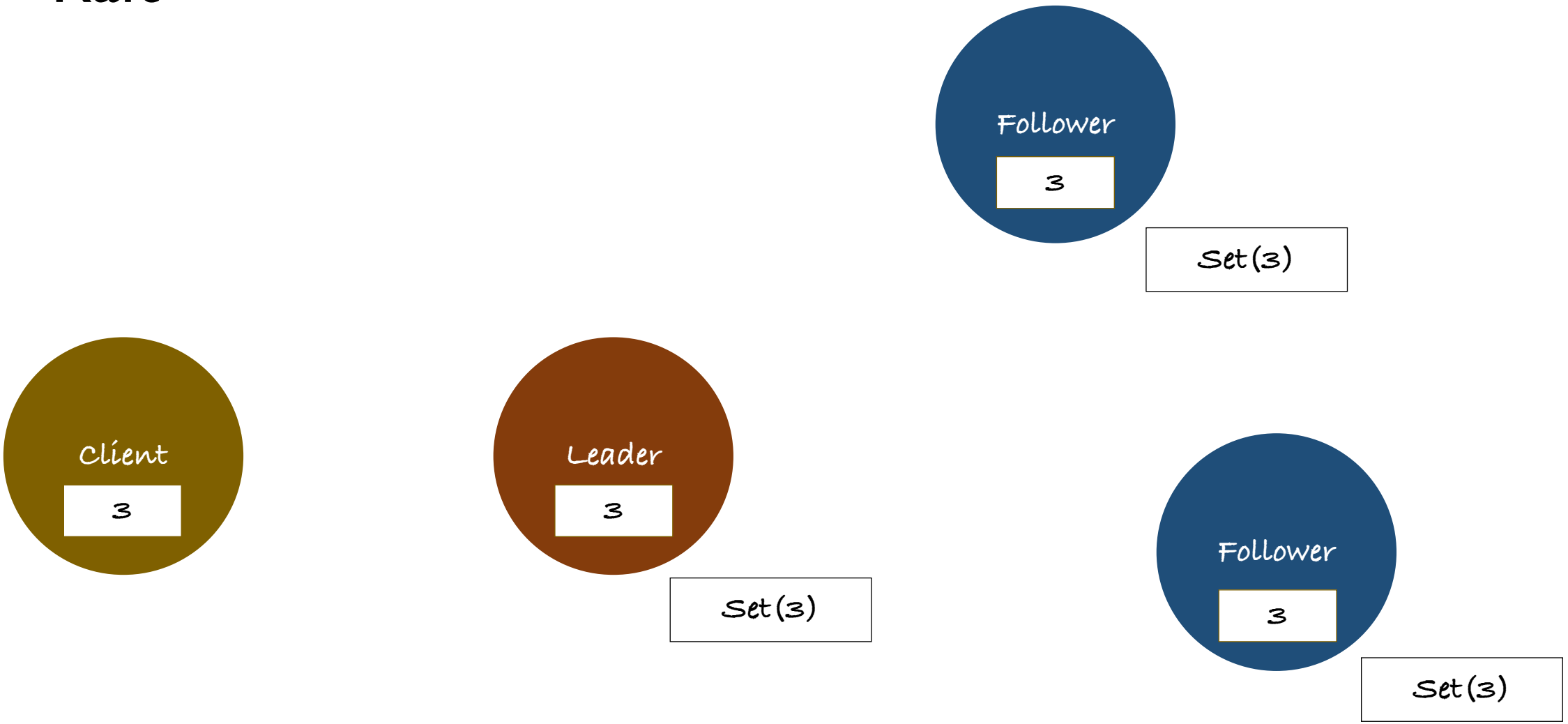


Raft



Leader wait only for a majority (Quorum) of followers to commit

Raft



Byzantine

Byzantine generals

- The problem [Lamport 1982]
  - three or more generals are to agree to **attack** or **retreat**
  - one (commander) issues the order
  - the others (lieutenants) decide
  - one or more generals are treacherous (= faulty!)
    - ✧ Propose **attack**ing to one general, and **retreat**ing to another
    - ✧ either commander or lieutenants can be treacherous!
- Requirements
  - Termination, Agreement as before
  - Integrity: If the commander is correct then all correct processes decide on the value proposed by commander.

Consensus in synchronous system (1)

- Uses basic multicast
  - guaranteed delivery by correct processes assuming the sender does not crash
- Admits process crash failures
  - assume up to f of the N processes may crash
- How it works...
  - $f+1$ rounds
  - relies on synchrony (timeout!)

Consensus in synchronous system (2)

- Initially
  - each process proposes a value from a set D
- Each process
  - maintains the set of values V_r known to it at round r
- In each round r, where $1 \leq r \leq f+1$,each process
  - multicasts the values to each other (only values not sent before, $V_r - V_{r-1}$)
  - receives multicast messages, recording any new value in V_r
- In round $f+1$
  - each process chooses minimum V_{f+1} as decision value

Consensus in synchronous system (3)

Crash failure:
A server halts, but working correctly until it halts

Arbitrary failure:
A server may produce arbitrary responses at arbitrary times

- Why it works?
  - sets timeout to maximum time for correct process to multicast message
  - can conclude process crashed if no reply
  - if process crashes, some value not forwarded...
- At round $f+1$
  - all correct process arrive at the same set of values
  - hence reach the same decision value (minimum)
  - at least $f+1$ rounds needed to tolerate f crash failures
- What about arbitrary failures?

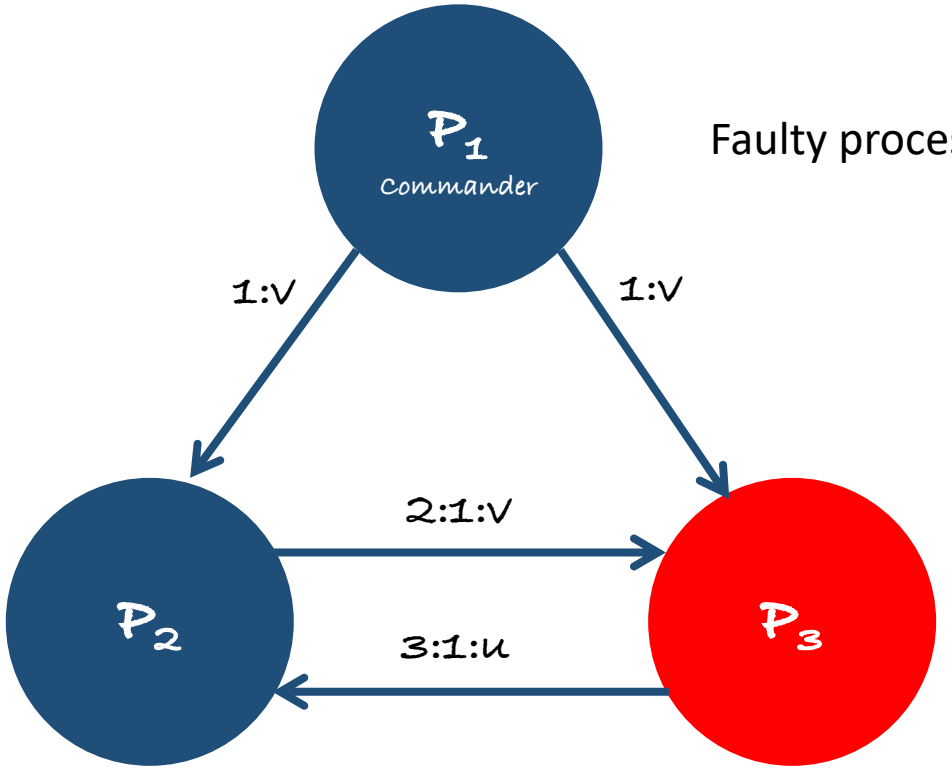
Byzantine generals...

- Processes exhibit arbitrary failures
  - up to f of the N processes faulty
- In asynchronous system
  - can use timeout to detect absence of a message
  - cannot conclude process crashed if no reply
  - impossibility with $N \leq 3f$
- In asynchronous system
  - cannot use timeout to reliably detect absence of a message
  - impossibility with even one failure!!!
  - hence impossibility of reliable totally ordered multicast...

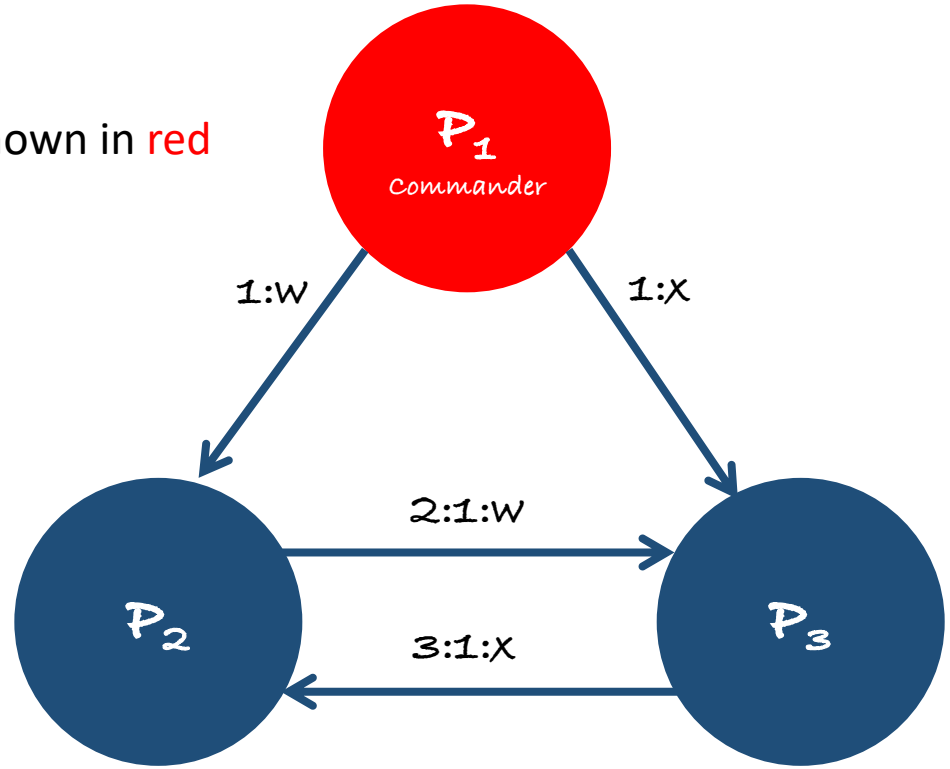
Impossibility with three generals

- Assume synchronous system
  - 3 processes, one faulty
  - if no message received, assume $\perp$
  - proceed in rounds
  - messages ‘3:1:u’ meaning ‘3 says 1 says u’
- Problem! ‘1 says v’ and ‘3 says 1 says u’
  - cannot tell which process is telling the truth!
  - goes away if digital signatures used...
- Show
  - no solution to agreement for $N=3$ and $f=1$
- Can generalize to impossibility for $N \leq 3f$

Three Byzantine generals



Faulty processes are shown in red



P_3 sends illegal value to P_2
P_2 cannot tell which value sent commander faulty
P_2 cannot tell which value sent by commander

Commander faulty
P_2 cannot tell which value sent by commander

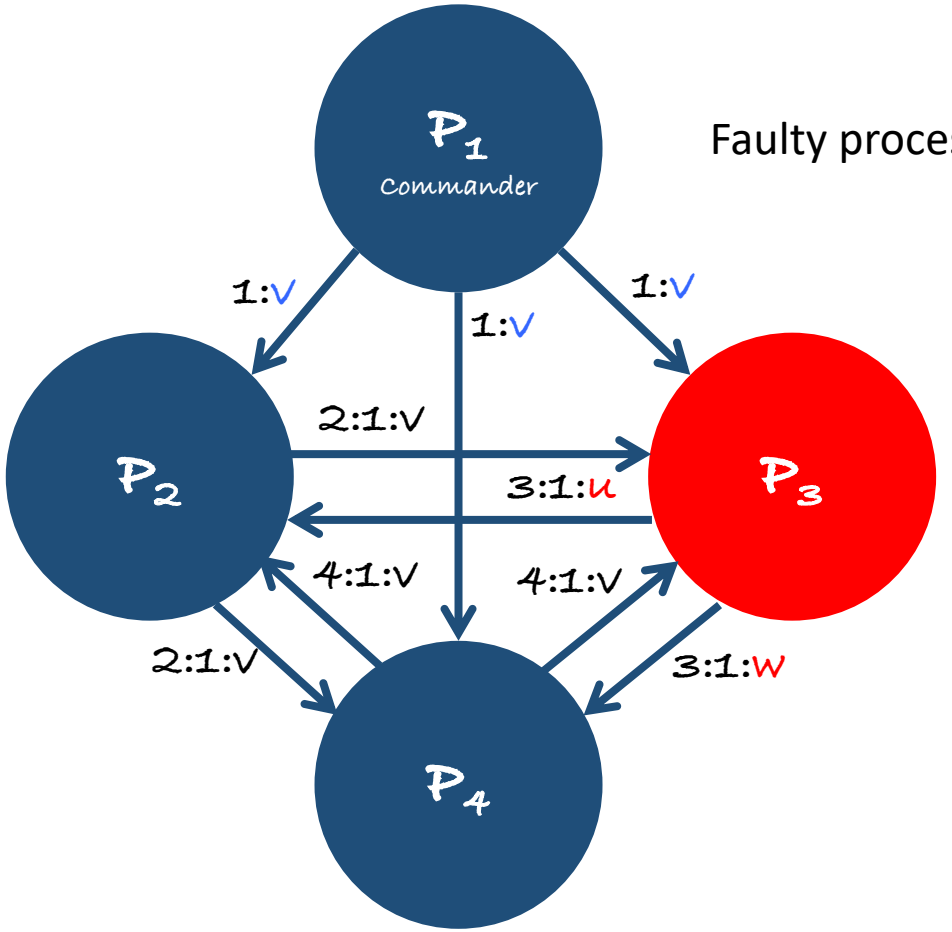
Impossibility with three generals

- So, if the solution exists
  - p_2 decides on value sent by commander (v) when the commander is correct
  - and also when commander faulty (w), since cannot distinguish between the two scenarios
- Apply the same reasoning to p_3
  - obtain p_3 must decide on x when commander faulty
- Thus
  - contradiction! since p_2 decides on w, p_3 on x if commander faulty
  - no solution exists

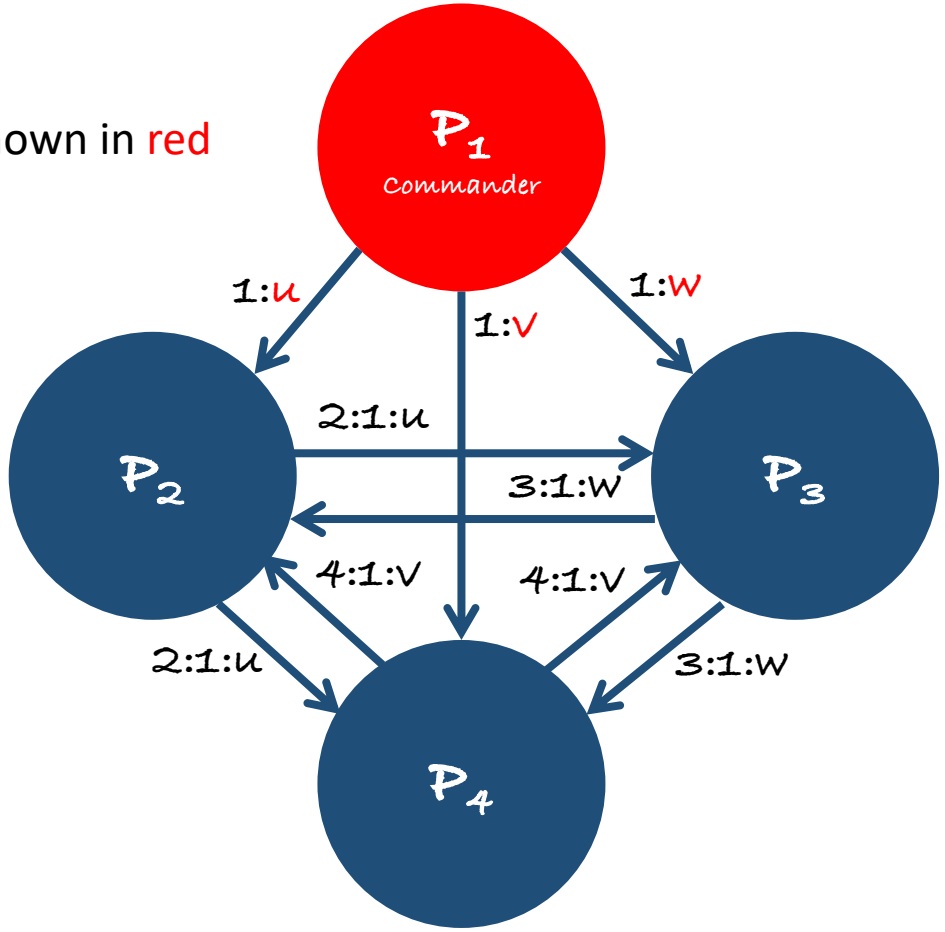
But...

- Solution exists for 4 processes with one faulty
  - commander sends value to each of the lieutenants
  - each lieutenant sends value it received to its peers
  - if commander faulty, then correct lieutenants have gathered all values sent by the commander
  - if one lieutenant faulty, the each correct lieutenant receives 2 copies of the value from the commander
- Thus
  - correct lieutenants can decide on majority of the values received
- Can generalize to $N \geq 3f+1$

Four Byzantine generals



Faulty processes are shown in red



P_2 decides majority(v,u,v) = v
P_4 decides majority(v,u,v) = v

P_2, P_3 and P_4 decide ⊥

(no majority exists)

In asynchronous systems...

- No guaranteed solution exists even for one failure!!!

  [Fisher, Lynch, Paterson ‘85]
  - does not mean never reach consensus in presence of failures
  - but that can reach it with positive probability
- But...
  - Internet asynchronous, exhibits arbitrary failures and uses consensus?
- Solutions exist using
  - partially synchronous systems
  - randomization [Aspnes & Herlihy, Lynch, etc]

Chain replication

Blockchain consensus mechanisms