

# **DISTRIBUTED SYSTEMS**

## **CS6421**

# **FAULT TOLERANCE SYSTEMS**

Prof. Tim Wood and Prof. Roozbeh Haghnazan

Includes material adapted from Van Steen and Tanenbaum's Distributed Systems book

# FINAL PROJECT

Questions?

- Design Document
  - Proposed Design
  - UML Diagrams describing architecture and communication
  - Work timeline with breakdown by team member
- Timeline
  - Milestone 0: Form a Team - 10/12
  - Milestone 1: Select a Topic - 10/19
  - **Milestone 2: Literature Survey - 10/29**
  - **Milestone 3: Design Document - 11/5**
  - Milestone 4: Final Presentation - 12/14

<https://gwdistsys20.github.io/project/>

# LAST TIME...

- Distributed Coordination
  - Distributed Locking
  - Elections
  - State Machine Replication
  - Blockchain

# THIS TIME...

- Fault Tolerance
  - Types of Failures
  - Two Generals Problem
  - Fault Tolerance Algorithms
  - Centralized FT: Raft/Paxos
  - Distributed FT: Blockchain

Next Time: Replication and Consistency Properties

# DISTSYS CHALLENGES

- Heterogeneity
- Openness
- Security
- **Failure Handling**
- Concurrency
- Quality of Service
- Scalability
- Transparency

Any questions about these? You will need to relate your project to them and they will be on the exam!

# BASIC CONCEPTS

- Being fault tolerant is strongly related to what are called dependable systems
- Dependability is a term that covers a number of useful requirements for distributed systems including the following
  - Availability
  - Reliability
  - Safety
  - Maintainability

# DIFFERENT TYPES OF FAILURES

Type of failure	Description of server's behavior
Crash failure ✓	Halts, but is working correctly until it halts
Omission failure ✓ Receive omission Send omission	Fails to respond to incoming requests Fails to receive incoming messages Fails to send messages
Timing failure ✓	Response lies outside a specified time interval
Response failure ✓ Value failure State-transition failure	Response is incorrect The value of the response is wrong Deviates from the correct flow of control
Arbitrary failure ✓	May produce arbitrary responses at arbitrary times

*Wood* *really bad*

# DIFFERENT TYPES OF FAILURES

- What type of failure can be the most problematic one?

The failures that you can not detect it...  
The system thinks that everything works well!!!

***Arbitrary failure***

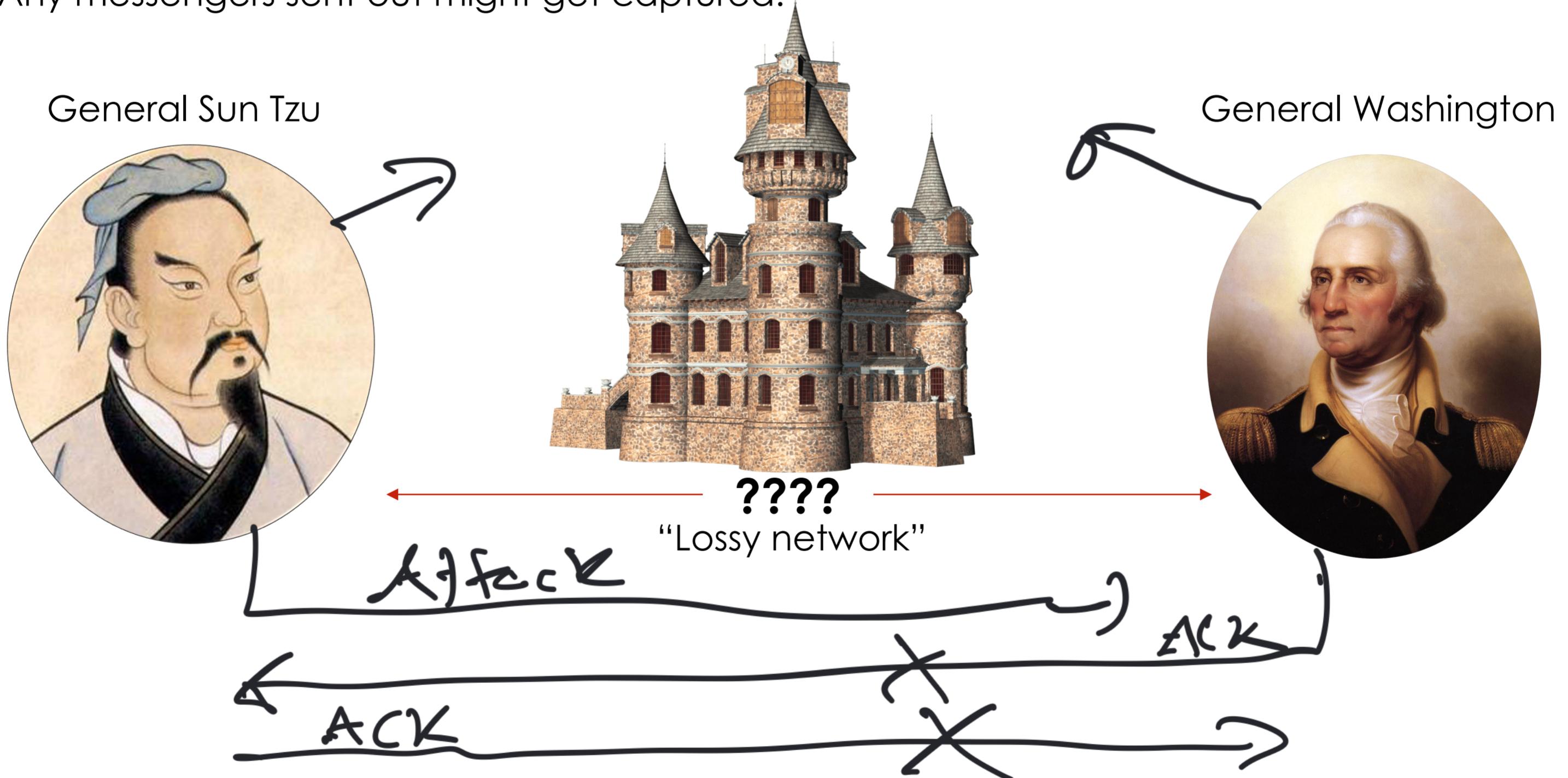
# TWO GENERALS PROBLEM

Two generals are preparing to attack a city

- They will only succeed if **both** attack simultaneously

How can they coordinate their attack?

- Any messengers sent out might get captured!



# TWO GENERALS PROBLEM

Impossible to guarantee agreement in lossy network!

- So usually we will need to assume that network will eventually transmit, or loss can be detected

General Sun Tzu



General Washington



????

# PROPERTIES

- Asynchrony: networks can have unbounded delay
- **Safety**: all nodes agree on the state of the system
  - nothing bad should happen
- **Liveness**: progress is made on incoming requests
  - something good should happen
- **Fault Tolerance**: at least one node can fail

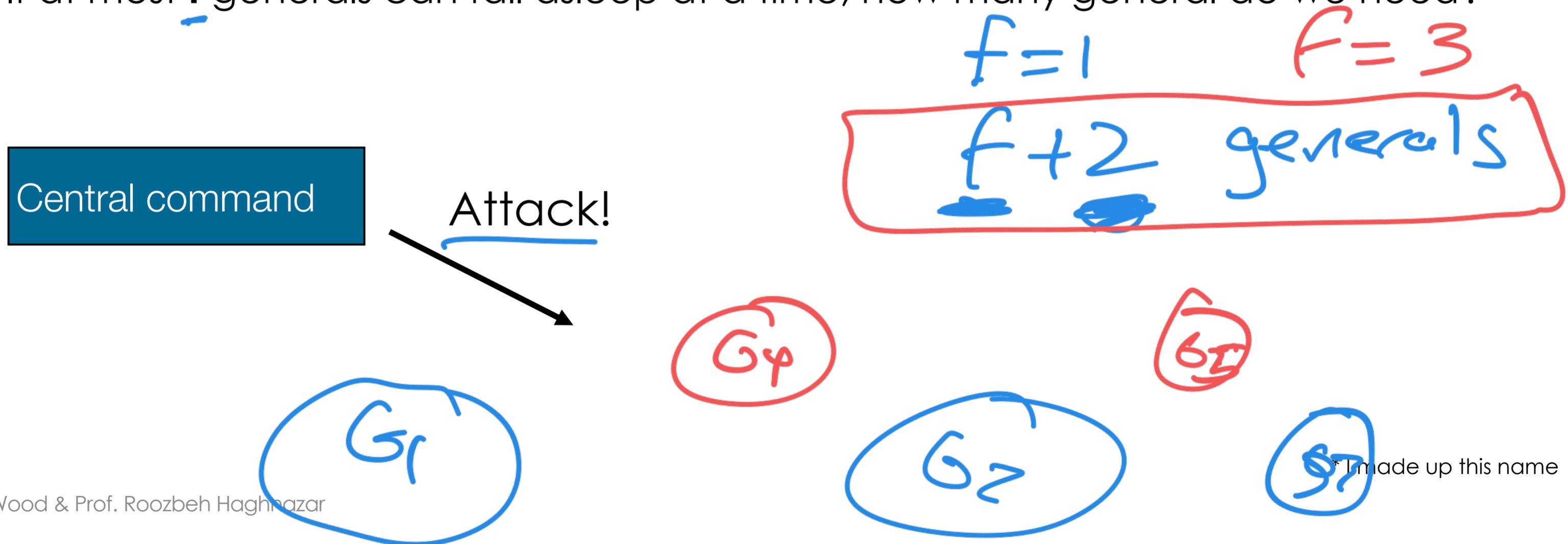
# PROPERTIES

- Asynchrony: networks can have unbounded delay
- **Safety:** all nodes agree on the state of the system
  - nothing bad should happen
- **Liveness:** progress is made on incoming requests
  - something good should happen
- **Fault Tolerance:** at least one node can fail

FLP Impossibility Theorem: in an asynchronous network, you can only get 2 out of 3 properties

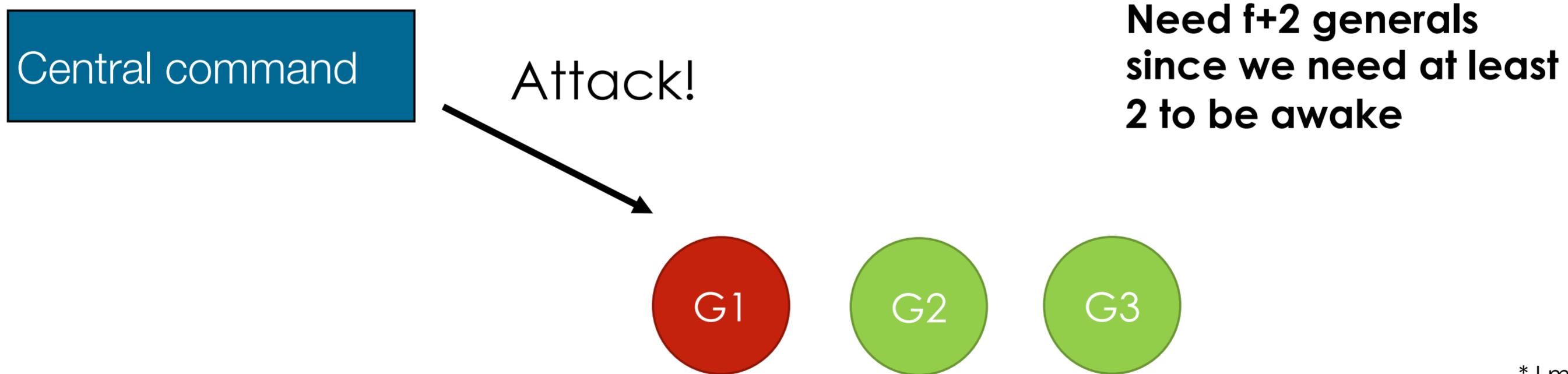
# SLEEPY GENERALS PROBLEM\*

- Our general are tired, but messengers can't die!
- Need 2 generals to be awake and attack for success
  - If at most  $f$  generals can fall asleep at a time, how many general do we need?



# SLEEPY GENERALS PROBLEM\*

- Our general are tired, but messengers can't die!
- Need **2** generals to be awake and attack for success
  - If at most  $f$  generals can fall asleep at a time, how many general do we need?



\* I made up this name

# BUREAUCRATIC GENERALS PROBLEM\*

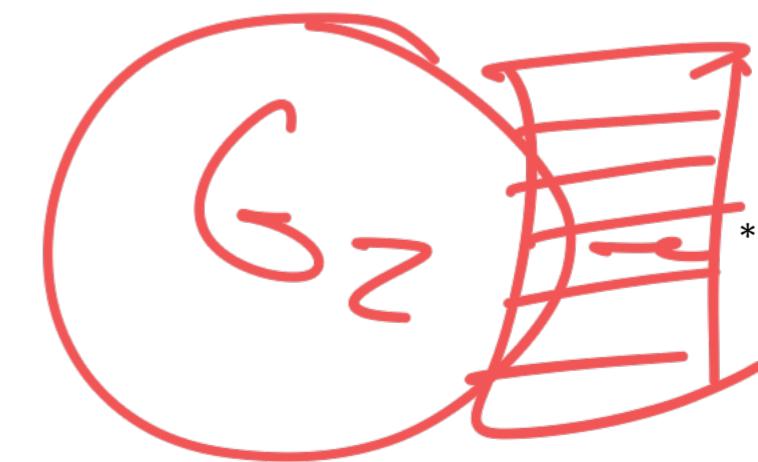
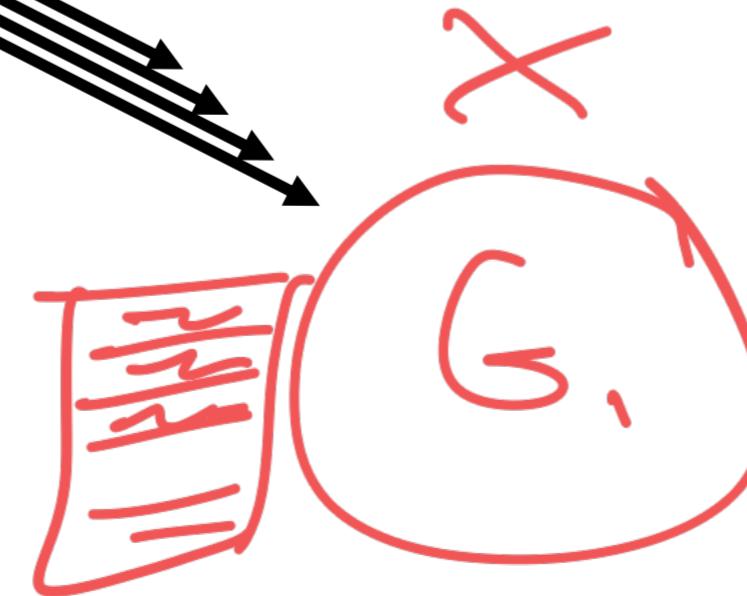
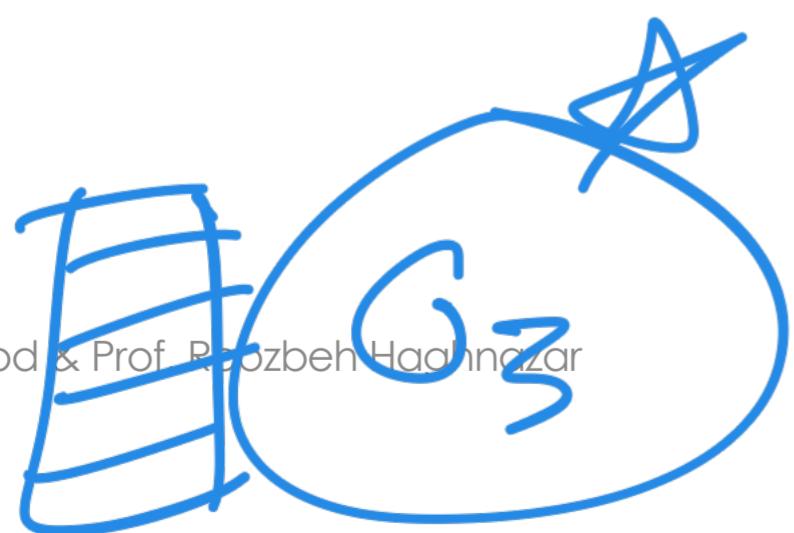
- Our general are tired, but messengers can't die!
- Need **1** general to be awake and attack for success, **f** can fail
- Need to ensure that all paperwork is filled correctly!
  - Need complete history of commands to attack (stateful system)

$f=1$

$2f+1$

Central command

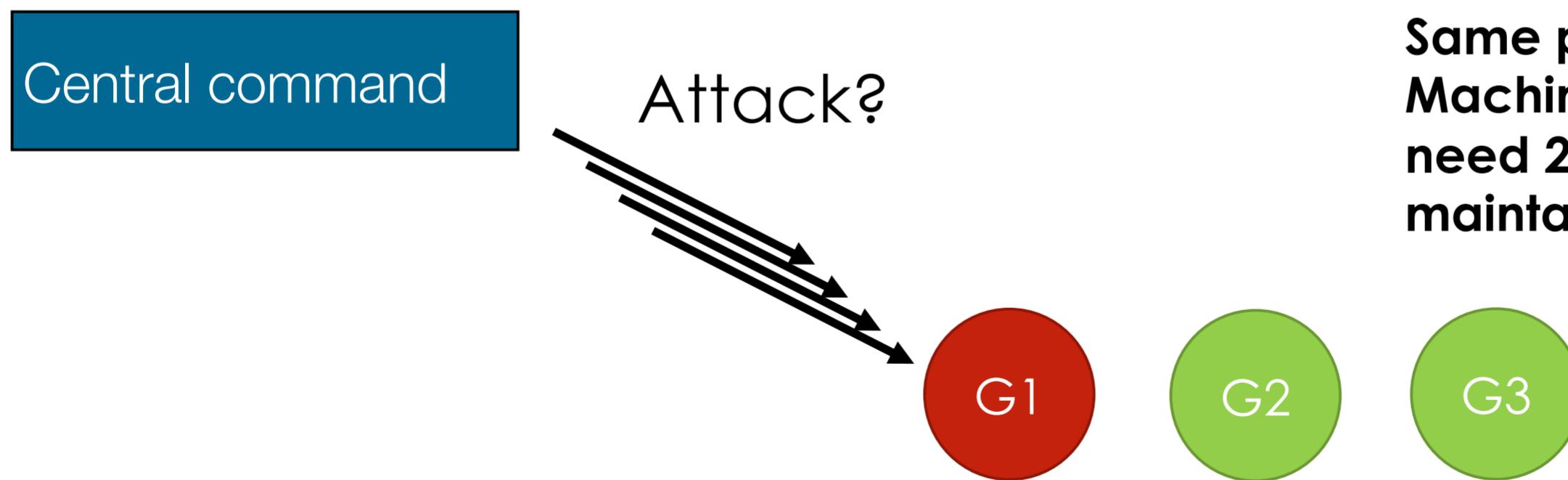
Attack?



~~$f \neq 1$~~

# BUREAUCRATIC GENERALS PROBLEM\*

- Our general are tired, but messengers can't die!
- Need **1** general to be awake and attack for success, **f** can fail
- Need to ensure that all paperwork is filled correctly!
  - Need complete history of commands to attack (stateful system)



**Same problem as State Machine Replication! We need  $2f+1$  so that we can maintain a log reliably**

\* I made up this name too

# TRAITOROUS GENERALS PROBLEM\*

- One of our generals is a traitor!
- How to make **majority of generals agree** to attack?

Central command

Attack?



# TRAITOROUS GENERALS PROBLEM\*

- One of our generals is a traitor!
- How to make **majority of generals agree** to attack?

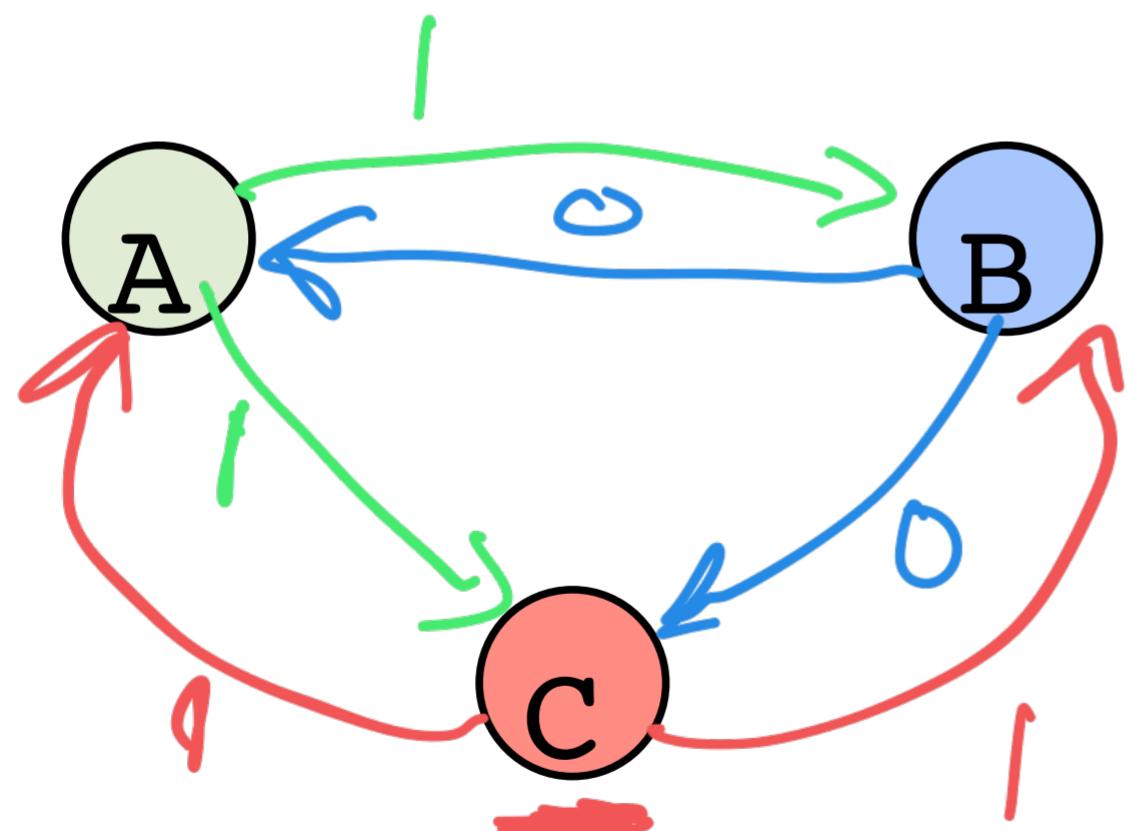


\* I made up this name too

# REACHING AGREEMENT

$$f=1$$

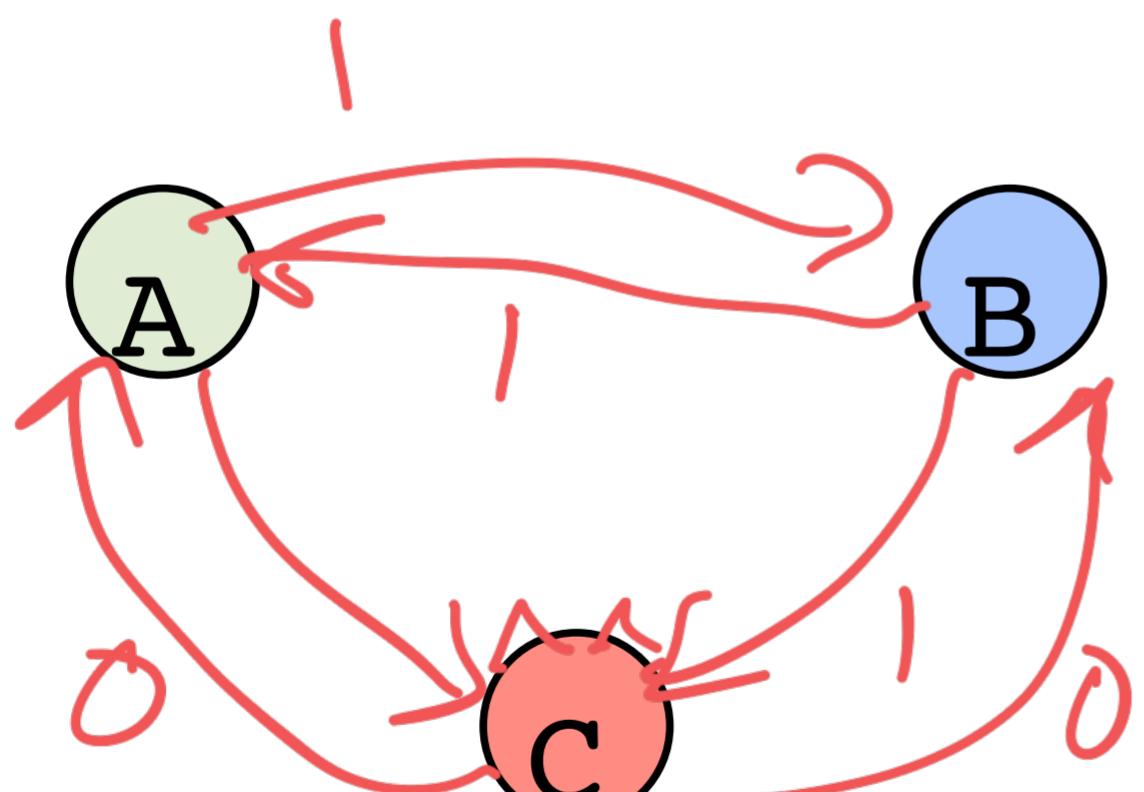
- The assault will only succeed if at least 2 armies attack at the same time ✓
  - I vote we should... 1 = attack, 0 = retreat!



Replica	Receives	Action
A: 1	1 0 1 → 1	
B: 0	1 0 1 → 1	
C: 1	1 0 1 → 1	

# REACHING AGREEMENT

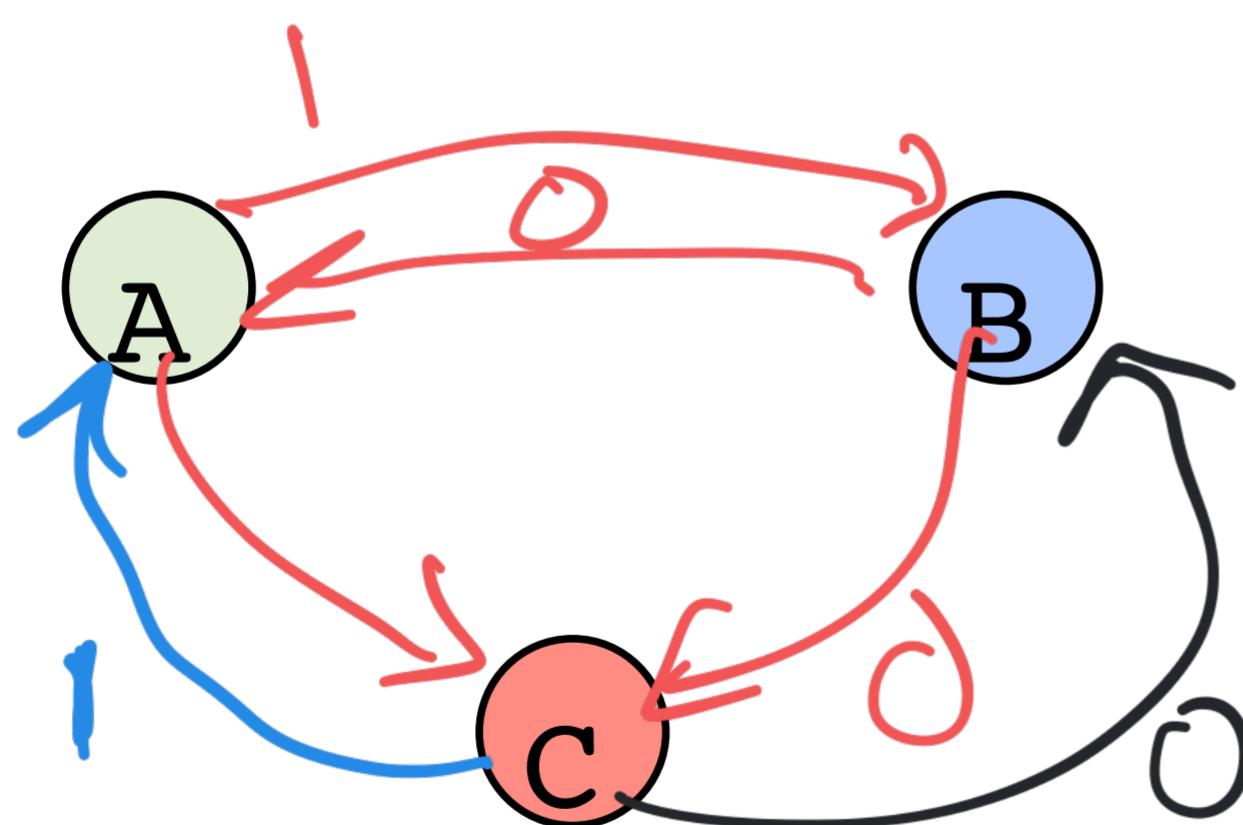
- The assault will only succeed if at least 2 armies attack at the same time
  - I vote we should... 1 = attack, 0 = retreat!



Replica	Receives	Action
A: 1	110	1
B: 1	110	(
C: 0	110	~0

# REACHING AGREEMENT

- The assault will only succeed if at least 2 armies attack at the same time
  - I vote we should... 1 = attack, 0 = retreat!



Replica	Receives	Action
A: 1	1 0 1 → 1	<del>1</del>
B: 0	1 0 0 → 0	<del>0</del>
C: ???	1 0 ?	0

# BYZANTINE FAULT

- Is a condition of a computer system, particularly distributed computing systems, where components may fail and there is **imperfect information** on whether a component has failed
- Further, a component can fail in a **malicious way**, i.e., at the worst possible time and in the worst possible way
- Related terms: **interactive consistency**, **source congruency**, **error avalanche**, **Byzantine agreement problem**, **Byzantine generals problem**, and **Byzantine failure**

## BYZANTINE GENERALS PROBLEM

$$f + 1$$

$$\cancel{2f + 1} \leftarrow$$

$$\boxed{3f + 1}$$

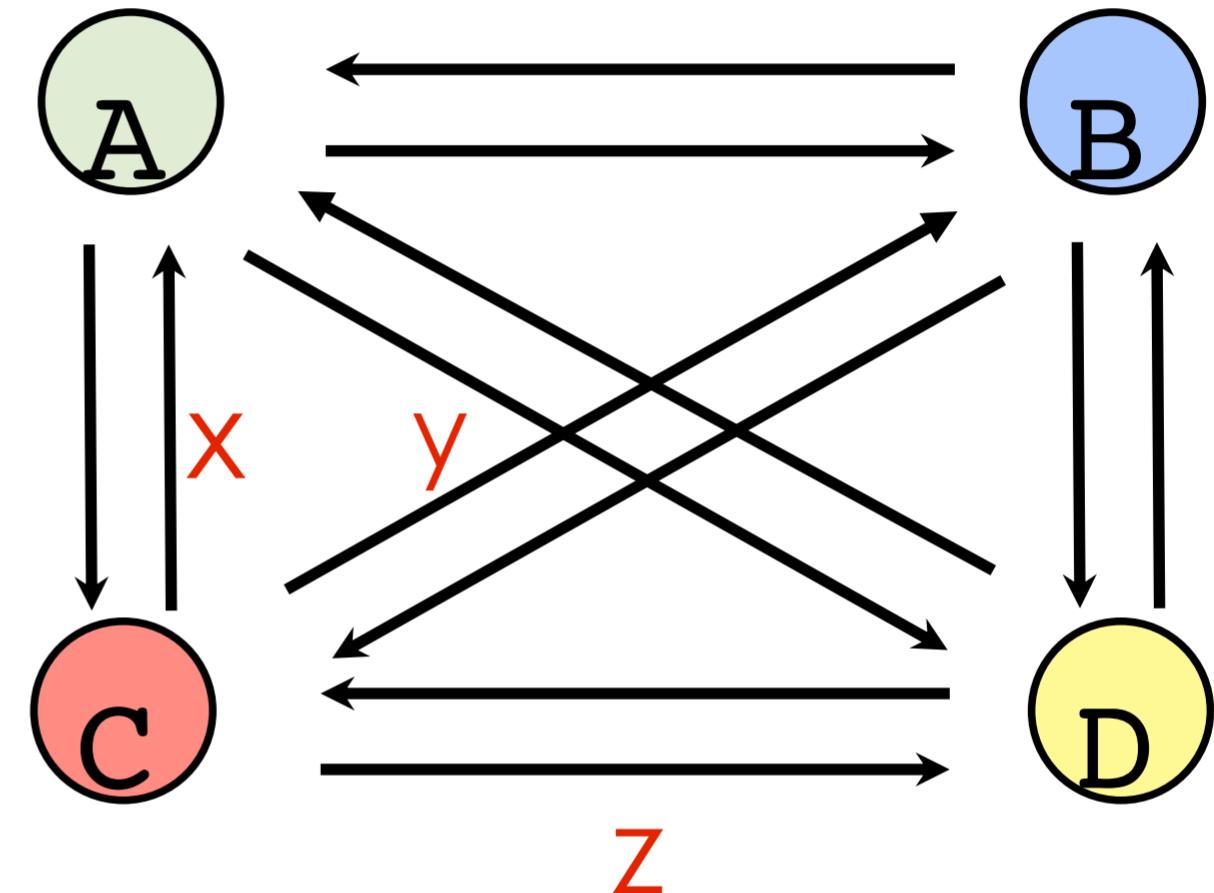


Prof. Tim Wood & Prof. Roozbeh Haghnazan

| malicious node  $\rightarrow$  4 replicas

# BYZANTINE GENERALS SOLVED<sup>\*</sup>!

- Need more replicas to reach consensus
- Requires  $3f+1$  replicas to tolerate  $f$  byzantine faults
- Step 1: Send your plan to everyone
- Step 2: Send learned plans to everyone
- Step 3: Detect conflicts and use majority



Replica	Receives	Majority
A	<b>A:</b> (1, 0, <u>1</u> , 1) <b>B:</b> (1, 0, <u>0</u> , 1) <b>C:</b> ( <u>1</u> , <u>1</u> , 1, <u>1</u> ) <b>D:</b> (1, 0, <u>1</u> , 1)	<b>A:</b> 1 <b>B:</b> 1 <b>C:</b> 1 <b>D:</b> 1
B	<b>A:</b> (1, 0, <u>1</u> , 1) <b>B:</b> (1, 0, <u>0</u> , 1) <b>C:</b> ( <u>0</u> , <u>0</u> , 0, <u>0</u> ) <b>D:</b> (1, 0, <u>1</u> , 1)	<b>A:</b> 1 <b>B:</b> 1 <b>C:</b> 0 <b>D:</b> 1

# PROBLEM SUMMARY

- Two Generals Problem
  - If network can arbitrarily lose messages, then it is impossible to guarantee two (or more) nodes can reach agreement
- Sleepy Generals Problem
  - If  $f$  nodes can fail, you need  $f+x$  replicas to guarantee  $x$  correct responses from a **stateless** system (typically  $x=1$ )
- Bureaucratic Generals Problem
  - If  $f$  nodes can fail, you need  $2f+1$  replicas to guarantee a correct response from a **stateful** system
- Byzantine Generals Problem
  - If  $f$  nodes can be arbitrarily malicious, you need  $3f+1$  replicas to guarantee a correct response (stateful or stateless)

# PROBLEM SUMMARY

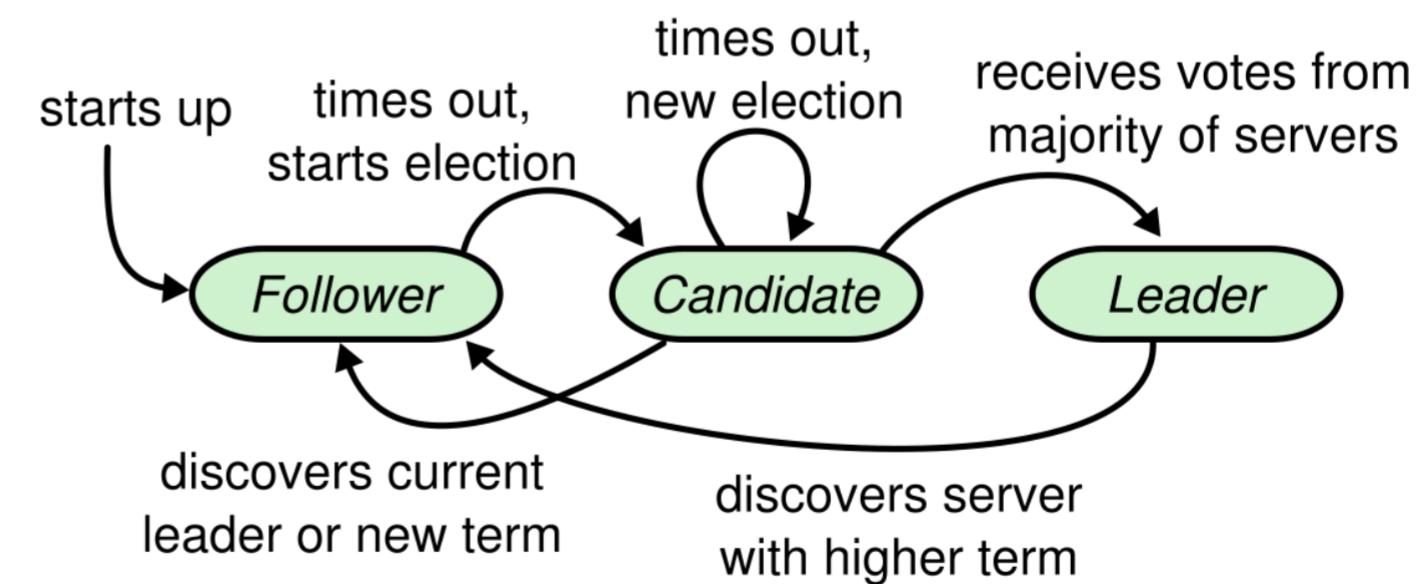
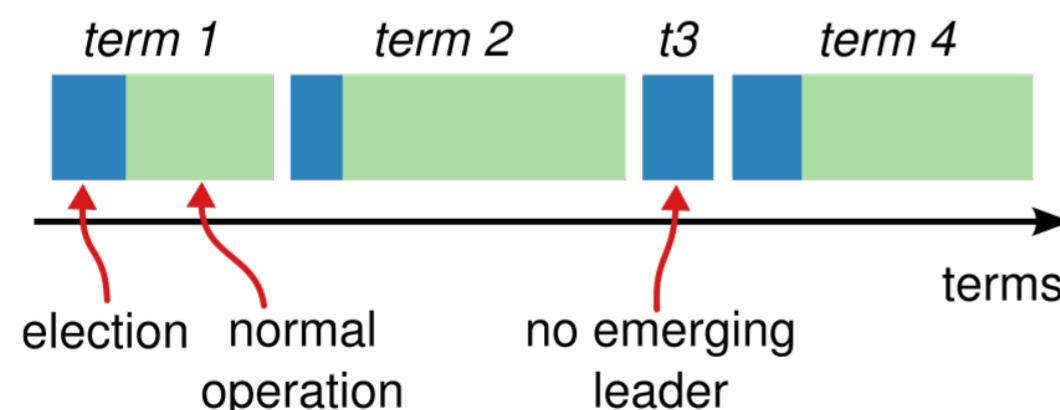
- Two Generals Problem
    - If network can arbitrarily lose messages, then it is impossible to guarantee two (or more) nodes can reach agreement
  - Sleepy Generals Problem
    - If  $f$  nodes can fail, you need  $f+x$  replicas to guarantee  $x$  correct responses from a **stateless** system (typically  $x=1$ )
  - Bureaucratic Generals Problem
    - If  $f$  nodes can fail, you need  $2f+1$  replicas to guarantee a correct response from a **stateful** system
  - Byzantine Generals Problem
    - If  $f$  nodes can be arbitrarily malicious, you need  $3f+1$  replicas to guarantee a correct response (stateful or stateless)
- Paxos, Raft*
- PBFT, Zyzzyva, Blockchain*
- BFT*

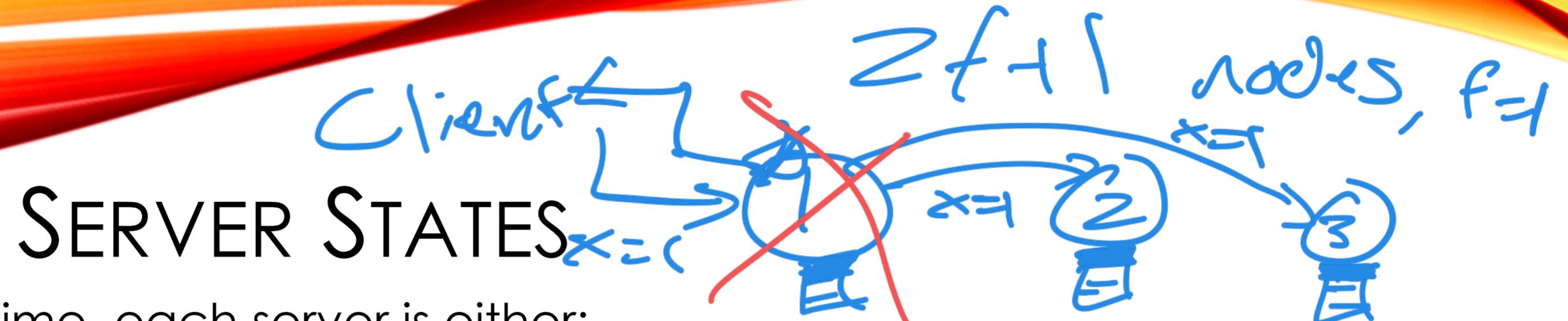
# PAXOS AND RAFT

- Goal: Achieve state machine replication for crash fault tolerance (non-  
byzantine, stateful, reliable network)
- Paxos: Lamport '90, published '98 ([interesting history](#))
  - Consensus algorithm presented in a paper pretending to describe how a fictitious ancient greek civilization wrote laws
  - Used by Google Chubby, Apache Zookeeper, etc
- Raft: Ongaro and Ousterhout '14
  - An “Understandable Consensus Algorithm”. Described as a set of Remote Procedure Calls (RPCs) that need to be implemented, but still provides strong guarantees
  - Dozens of implementations, used in many real products
- Both provide fault tolerance and safety, but are not guaranteed to terminate (no liveness)

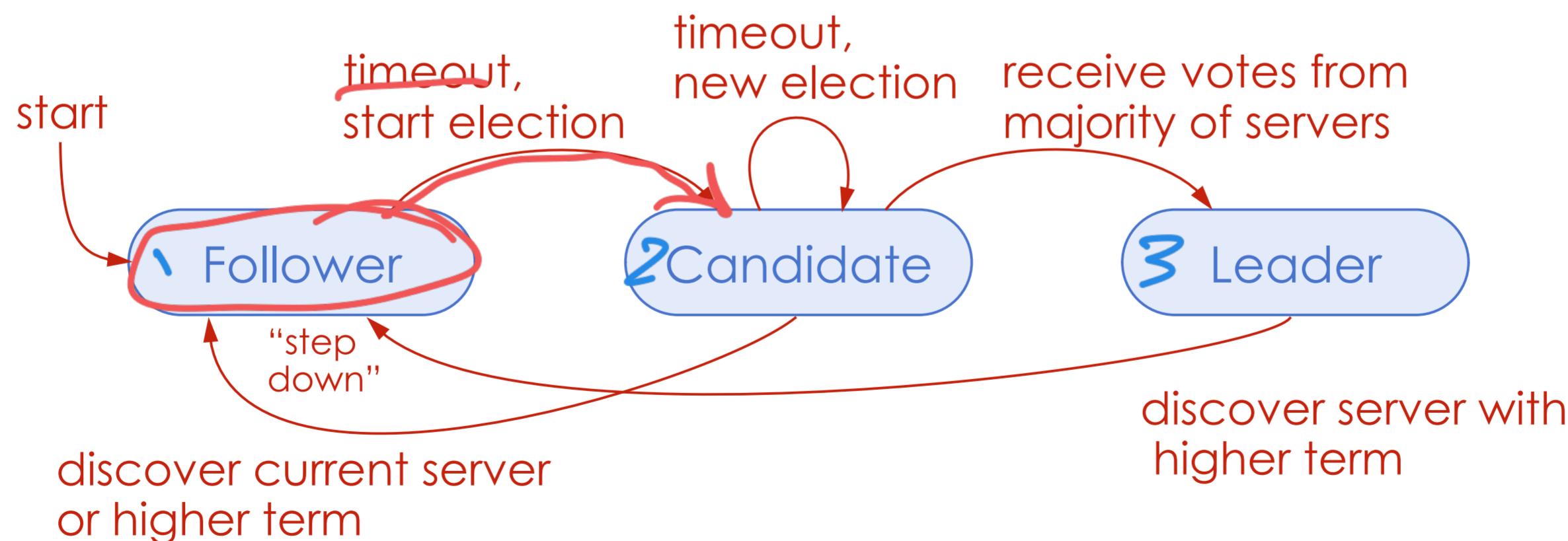
# RAFT – KEY IDEAS FOR SMR

- **Leader election:** Elections periodically occur in case the primary fails
- **Terms:** Help track avoid inconsistent state after recovery
- **Ordered Logs:** All incoming requests pass through leader to be ordered





- At any given time, each server is either:
  - **Leader**: handles all client interactions, log replication, sends heartbeats
    - At most 1 viable leader at a time
  - **Follower**: completely passive (issues no RPCs, responds to incoming RPCs)
  - **Candidate**: used to elect a new leader
- Normal operation: 1 leader, N-1 followers

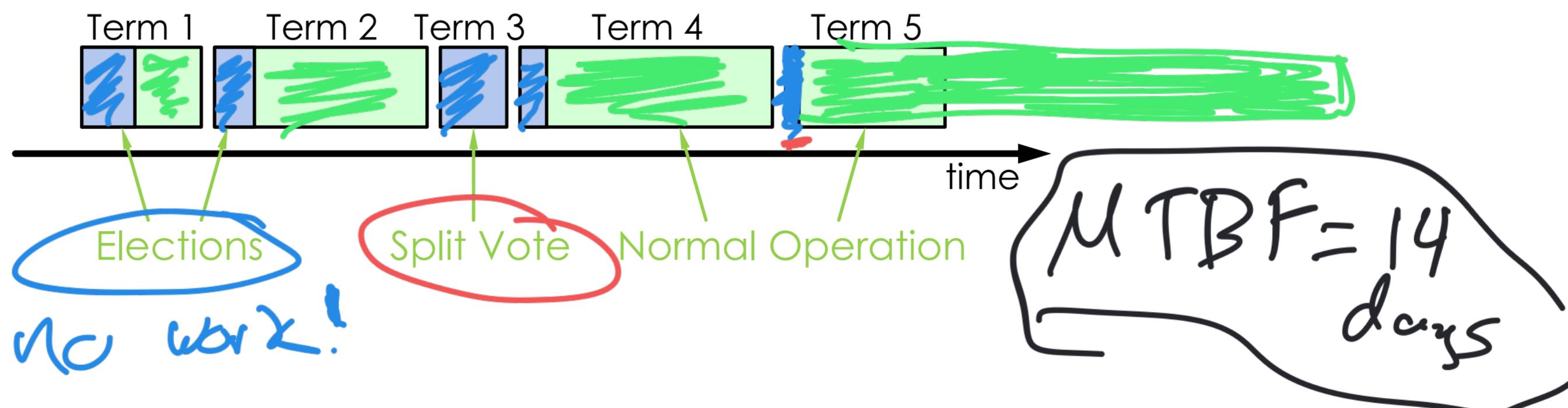


# TERMS

- Time divided into terms:
  - Election
  - Normal operation under a single leader
- At most 1 leader per term
- Some terms have no leader (failed election)
- Each server maintains **current term** value
- **Key role of terms:** identify obsolete information

Elections  $100-500\text{ms}$   
- fast & rare  
- successfully pick a leader

~~①~~ ② ③



# Raft Protocol Summary

Followers
<ul style="list-style-type: none"> <li>Respond to RPCs from candidates and leaders.</li> <li>Convert to candidate if election timeout elapses without either: <ul style="list-style-type: none"> <li>Receiving valid AppendEntries RPC, or</li> <li>Granting vote to candidate</li> </ul> </li> </ul>
Candidates
<ul style="list-style-type: none"> <li>Increment currentTerm, vote for self</li> <li>Reset election timeout</li> <li>Send RequestVote RPCs to all other servers, wait for either: <ul style="list-style-type: none"> <li>Votes received from majority of servers: become leader</li> <li>AppendEntries RPC received from new leader: step down</li> </ul> </li> <li>Election timeout elapses without election resolution: increment term, start new election</li> <li>Discover higher term: step down</li> </ul>
Leaders
<ul style="list-style-type: none"> <li>Initialize nextIndex for each to last log index + 1</li> <li>Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts</li> <li>Accept commands from clients, append new entries to local log</li> <li>Whenever last log index <math>\geq</math> nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful</li> <li>If AppendEntries fails because of log inconsistency, decrement nextIndex and retry</li> <li>Mark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of servers</li> <li>Step down if currentTerm changes</li> </ul>

X

RequestVote RPC
Invoked by candidates to gather votes.
<b>Arguments:</b>
<b>candidateId</b> candidate requesting vote
<b>term</b> candidate's term
<b>lastLogIndex</b> index of candidate's last log entry
<b>lastLogTerm</b> term of candidate's last log entry
<b>Results:</b>
<b>term</b> currentTerm, for candidate to update itself
<b>voteGranted</b> true means candidate received vote
<b>Implementation:</b>
1. If term > currentTerm, currentTerm $\leftarrow$ term (step down if leader or candidate)
2. If term == currentTerm, votedFor is null or candidateId, and candidate's log is at least as complete as local log, grant vote and reset election timeout

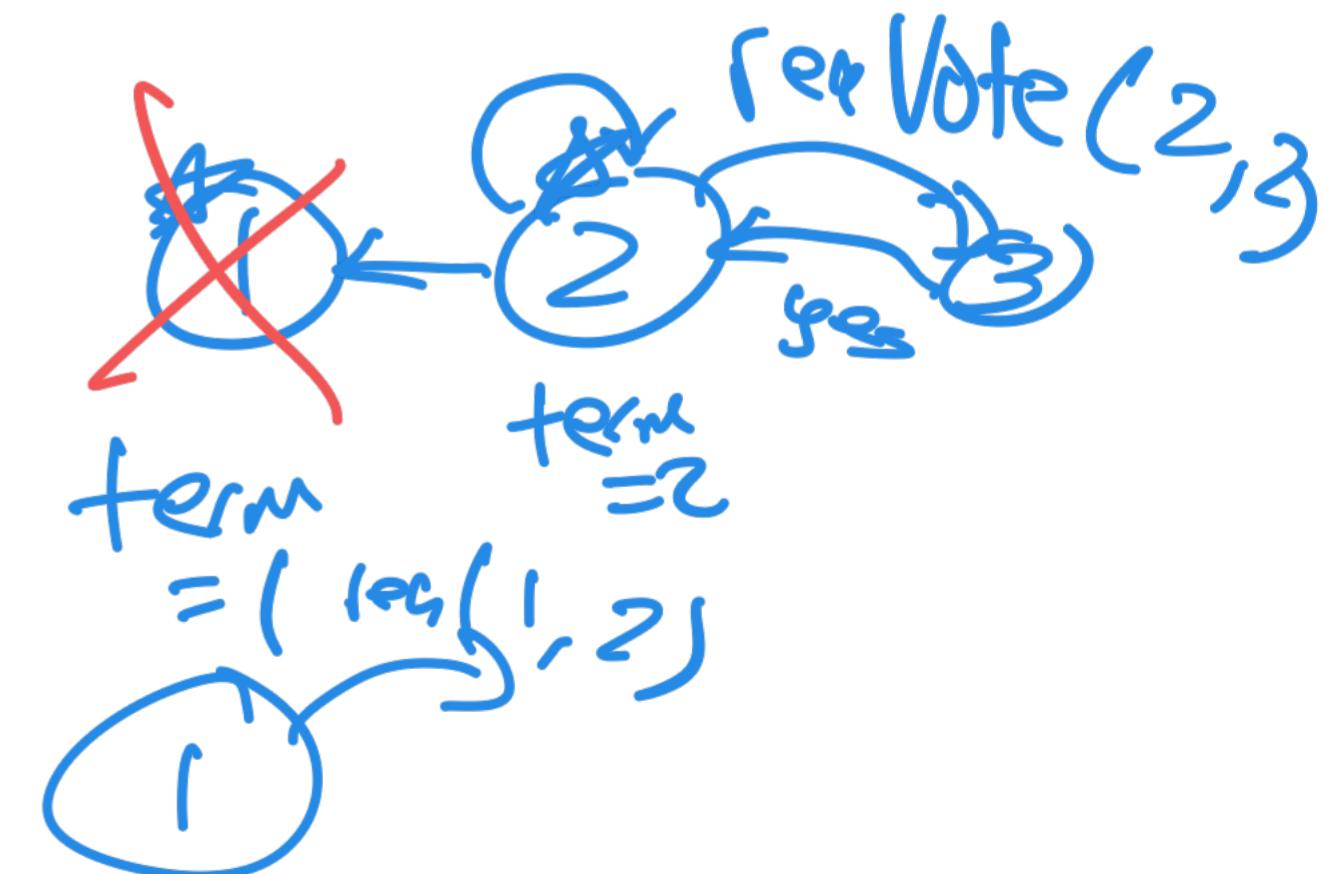
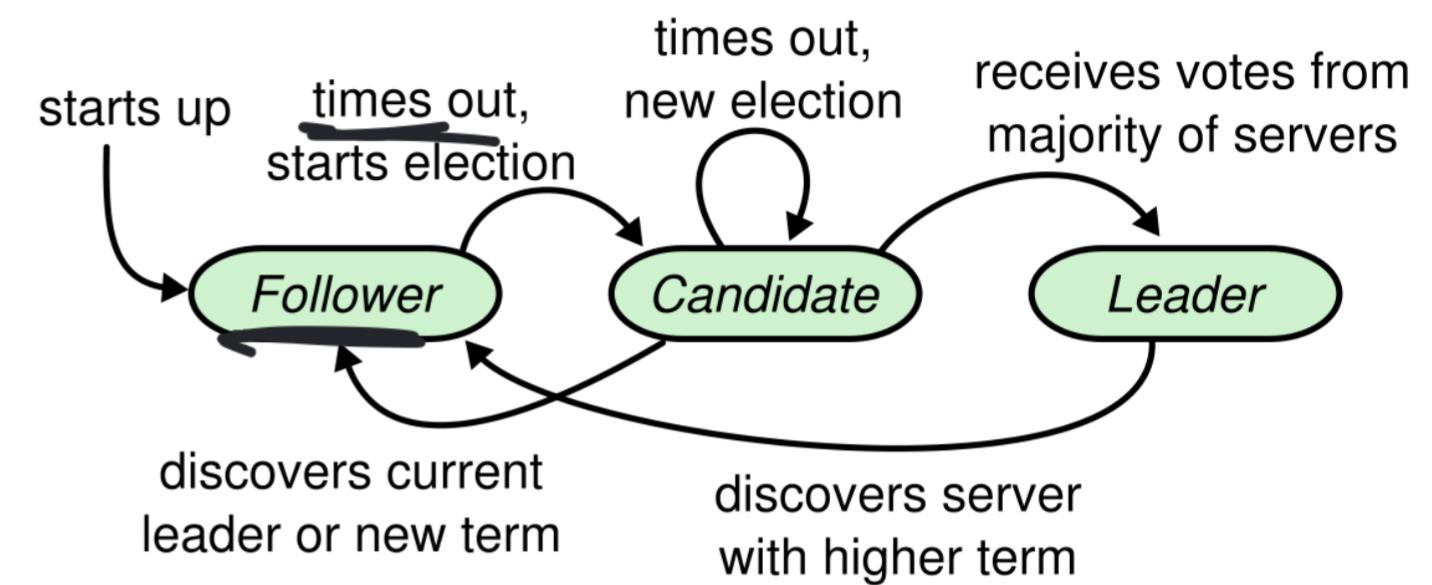
AppendEntries RPC
Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat .
<b>Arguments:</b>
<b>term</b> leader's term
<b>leaderId</b> so follower can redirect clients
<b>prevLogIndex</b> index of log entry immediately preceding new ones
<b>prevLogTerm</b> term of prevLogIndex entry
<b>entries[]</b> log entries to store (empty for heartbeat)
<b>commitIndex</b> last entry known to be committed
<b>Results:</b>
<b>term</b> currentTerm, for leader to update itself
<b>success</b> true if follower contained entry matching prevLogIndex and prevLogTerm
<b>Implementation:</b>
1. Return if term < currentTerm
2. If term > currentTerm, currentTerm $\leftarrow$ term
3. If candidate or leader, step down
4. Reset election timeout
5. Return failure if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry
7. Append any new entries not already in the log
8. Advance state machine with newly committed entries

Persistent State
Each server persists the following to stable storage synchronously before responding to RPCs:
<b>currentTerm</b> latest term server has seen (initialized to 0 on first boot)
<b>votedFor</b> candidateId that received vote in current term (or null if none)
<b>log[]</b> log entries

Log Entry
<b>term</b> term when entry was received by leader
<b>index</b> position of entry in the log
<b>command</b> command for state machine

# ELECTION BASICS

- Triggered by a timeout
- Increment current term
- Change to Candidate state
- Vote for self
- Send RequestVote RPCs to all other servers, retry until either:
  1. Receive votes from majority of servers:
    - Become leader
    - Send AppendEntries heartbeats to all other servers
  2. Receive RPC from valid leader:
    - Return to follower state
  3. No-one wins election (election timeout elapses):
    - Increment term, start new election

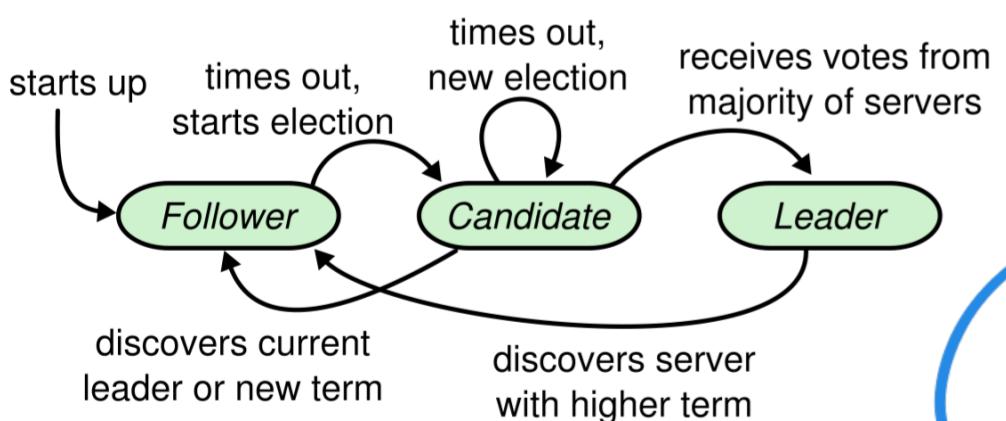


I need 4 volunteers!

# LET'S RUN AN ELECTION!

## Leader:

- Sends **<Hello X>** as heartbeat for term X every 5 seconds



## Followers:

- If no heartbeat in **10 seconds**, become a **Candidate**
- If Receive **<Elect ID T=10>**
- Reply **<VOTE ID>** to first candidate you hear
  - Wait **10 seconds**, if no winner, become **Candidate**



Prof. Tim Wood & Prof. Rozbeh HaghNazan

## Candidate:

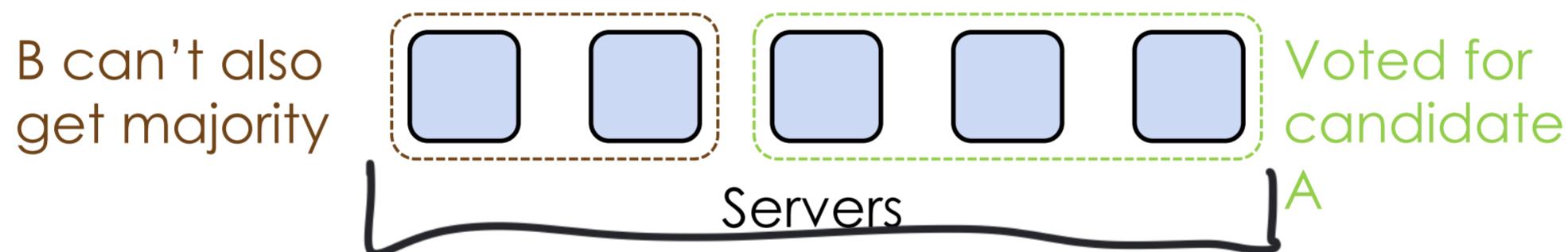
### ~~Recruit~~

- Send **<Elect ID>** **Term**)
- ID is my ID
  - Send **<VOTE ID>** to vote for yourself
  - Wait for VOTE messages
    - If got majority then send **<WIN ID>**
    - If no winner, wait **5-10** seconds and become **Candidate**

5 nodes  
 $\frac{2f+1}{f=2} = 5$

# ELECTIONS, CONT'D

- Safety: allow at most one winner per term
  - Each server gives out only one vote per term (persist on disk)
  - Two different candidates can't accumulate majorities in same term



- Liveness: some candidate should eventually win

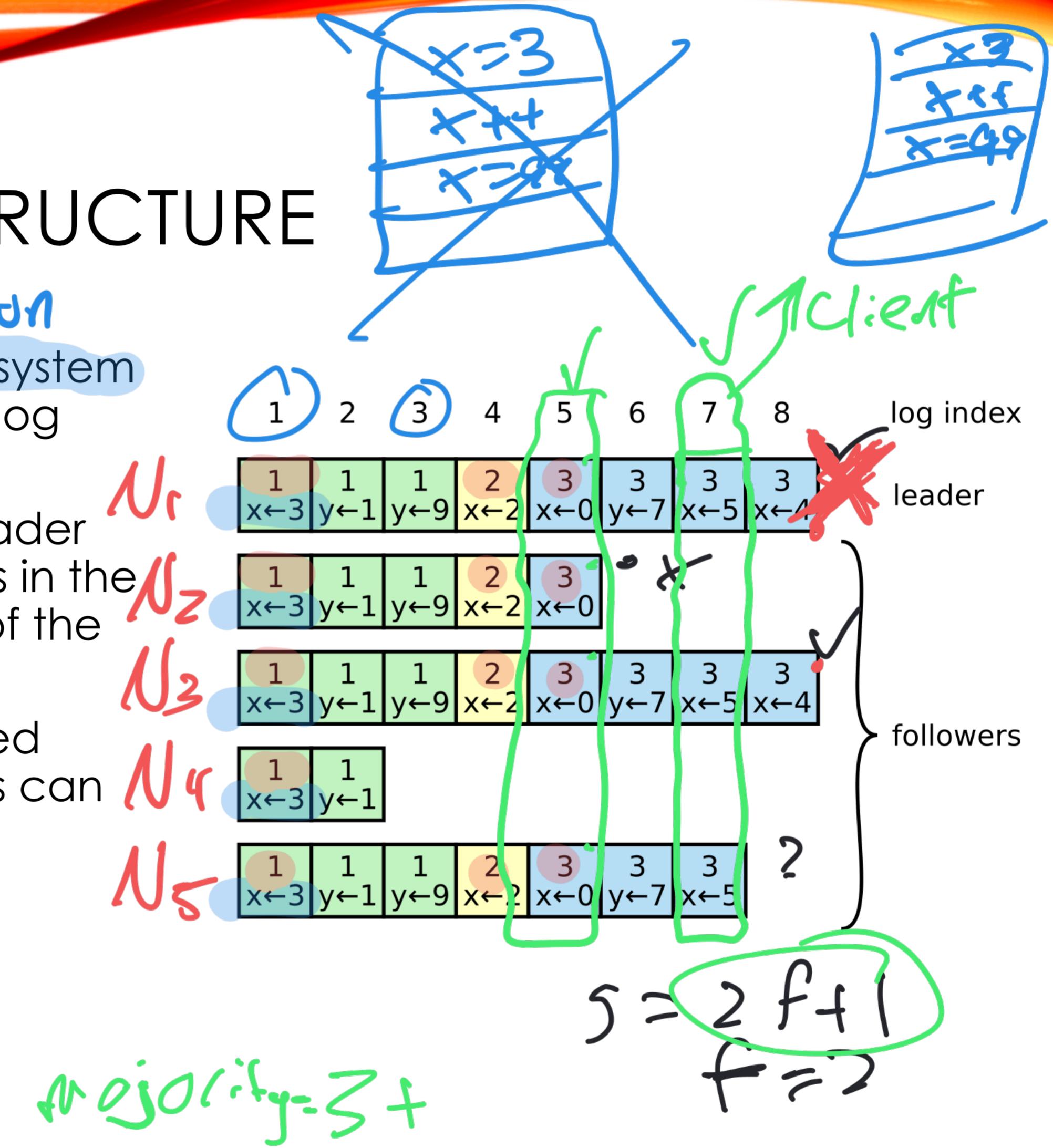
- Choose election timeouts randomly in  $[T, 2T]$
- One server usually times out and wins election before others wake up
- Works well if  $T \gg$  broadcast time **but not guaranteed!**

$T \sim 250\text{ms}$

# LOG STRUCTURE

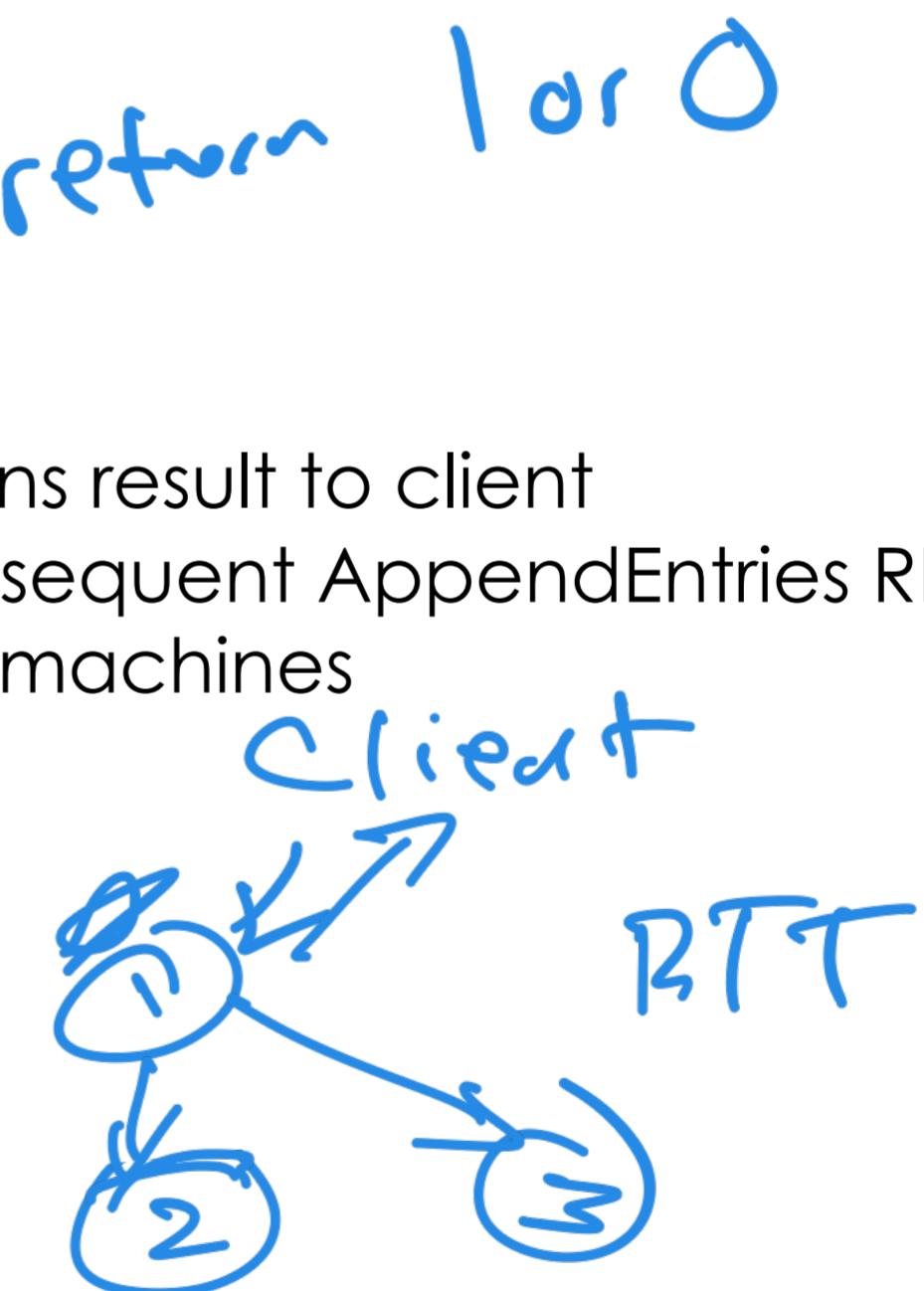
*operation*

- **Logs:** Store a change to the system state, a term number and a log index.
- A log is **committed** by the leader once a majority of the nodes in the system have stored a copy of the entry.
- **Recovery and changes:** Failed nodes or newly joining nodes can be brought up to date by synchronizing log



# NORMAL OPERATION

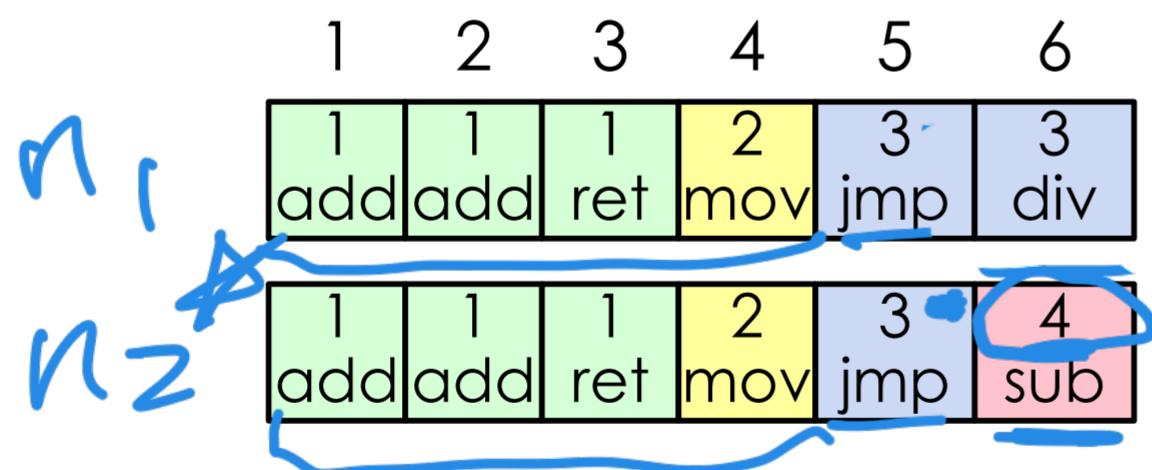
- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to followers
- Once new entry committed:
  - Leader passes command to its state machine, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers pass committed commands to their state machines
- Crashed/slow followers?
  - Leader retries RPCs until they succeed
- Performance is optimal in common case:
  - One successful RPC to any majority of servers



# LOG CONSISTENCY

High level of coherency between logs:

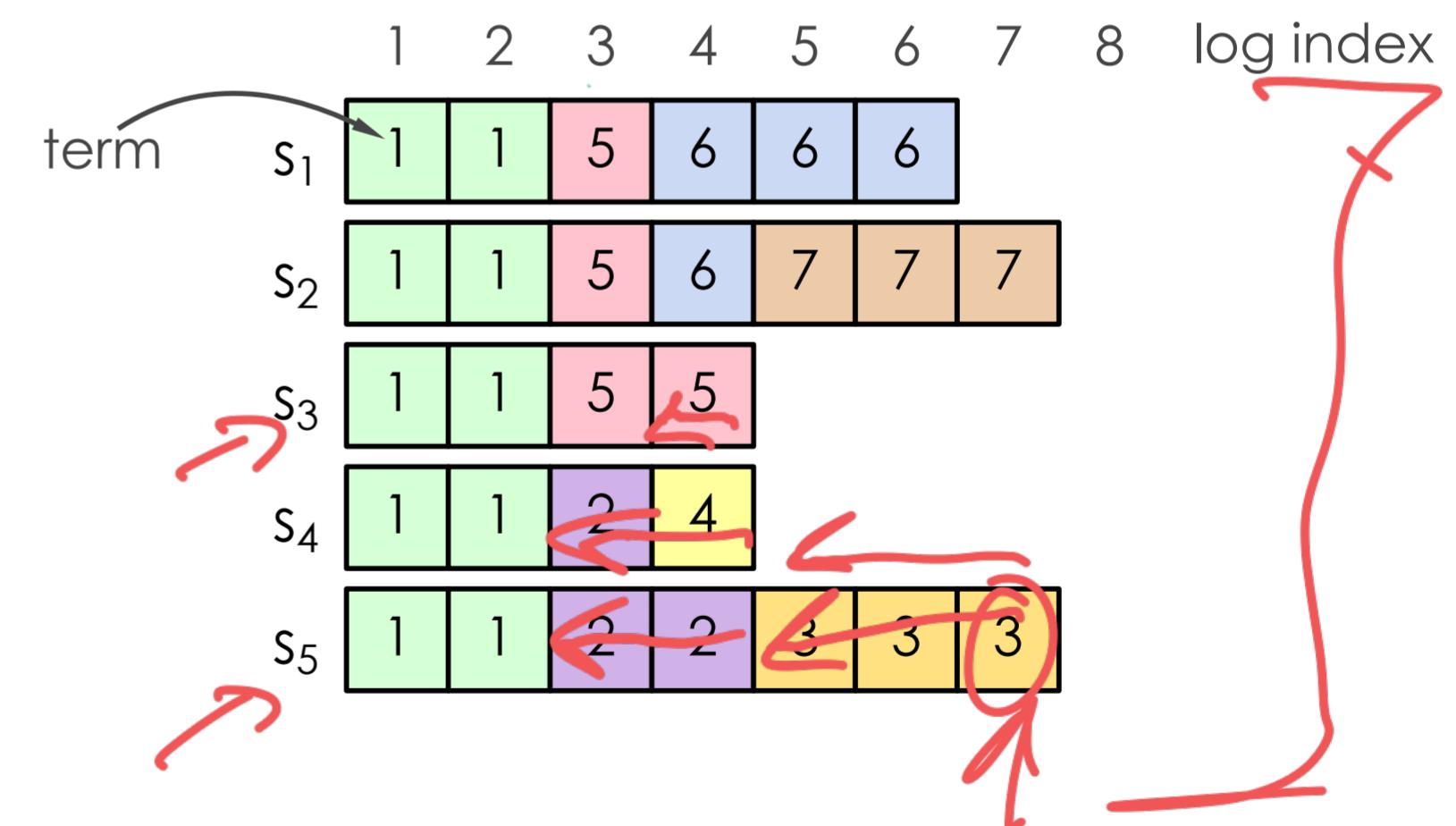
- If log entries on different servers have **same index and term**:
  - They store the same command
  - The logs are identical in all preceding entries



- If a given entry is **committed**, all preceding entries are also committed

# ROLLBACK

- A failed leader may cause some nodes to have **uncommitted logs**
- When new electing a new leader, pick the one with the largest log!
- Roll back logs that weren't committed
- Logs can get into strange states if nodes fail/reconnect
- But if a majority of nodes hold a log we can treat it as committed and it will never be rolled back



# RAFT SUMMARY

1. Leader election: uses randomness to resolve quickly
2. Normal operation: Leader orders requests, commit with majority
3. Safety and consistency: Client only gets response if committed
4. Rollback: Uncommitted logs can be removed if a leader fails

Guarantees consistent responses to client and no loss of state using  **$2f+1$**  nodes

Used by consistent data stores like **etcd**