Distributed Systems CS6421

Fault Tolerance Systems

Prof. Roozbeh Haghnazar

Slides Credit:

Prof. Tim Wood and Prof. Roozbeh Haghnazar

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

Final Project

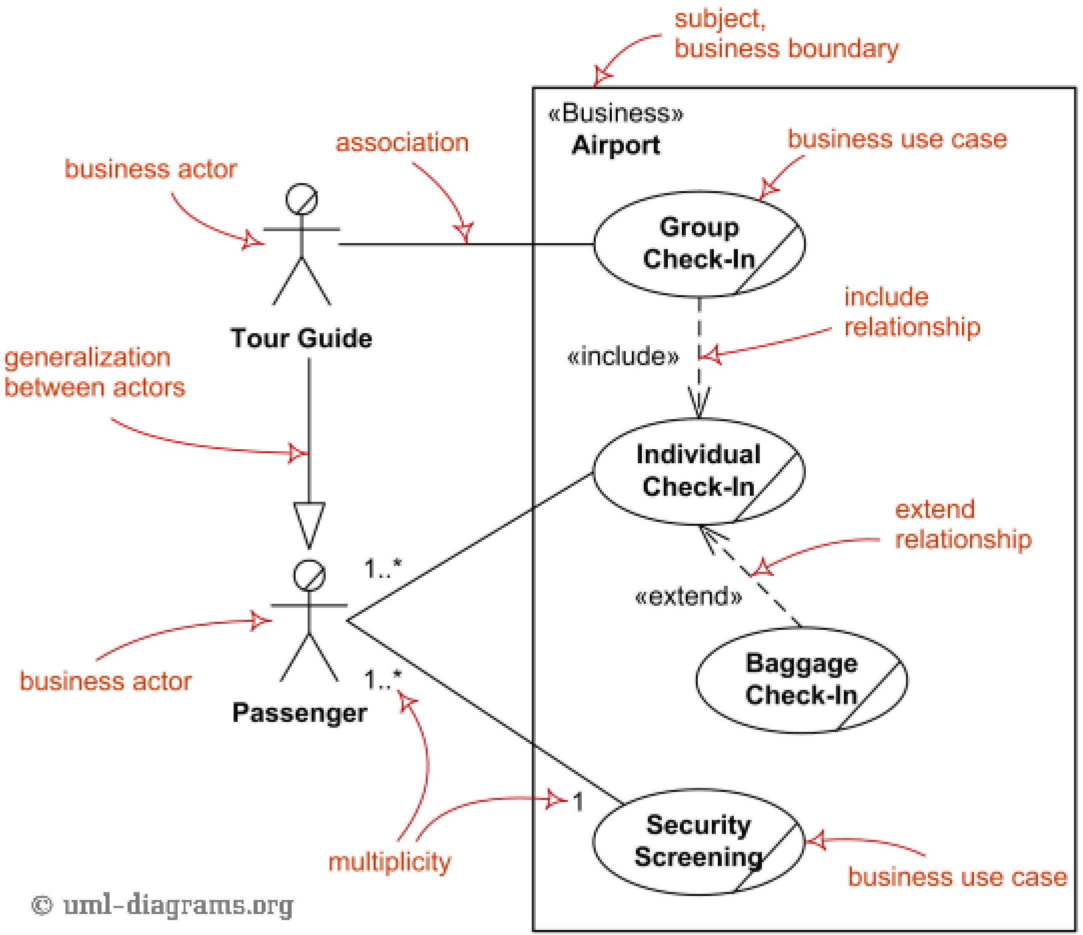
Questions?

- Design Document
  - Proposed Design
  - UML Diagrams describing architecture and communication
  - Work timeline with breakdown by team member

- Timeline
  - Milestone 0: Form a Team
  - Milestone 1: Select a Topic
  - Milestone 2: Literature Survey
  - **Milestone 3: Design Document**
  - Milestone 4: Final Presentation

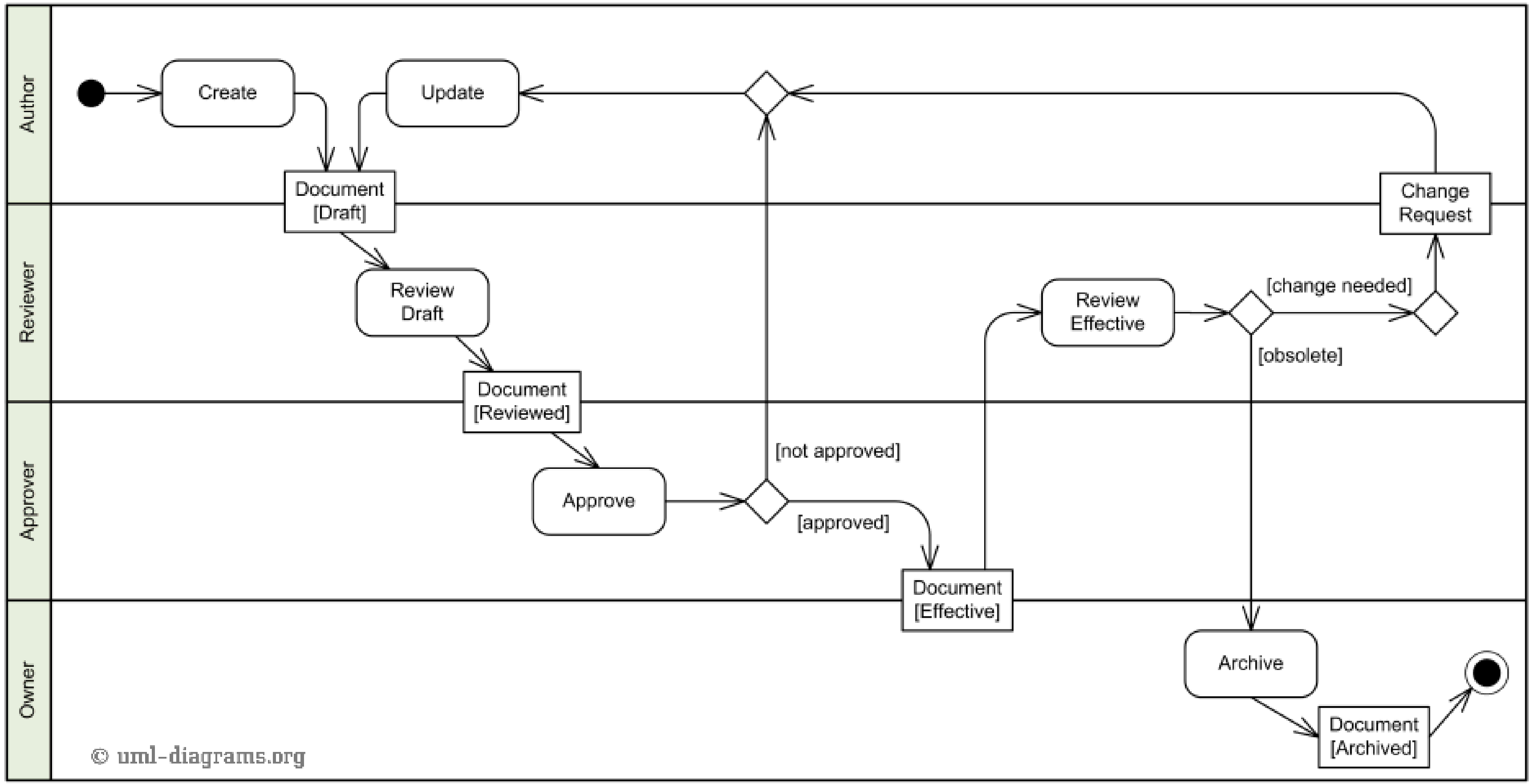
DIAGRAMS

- Use Case Diagram



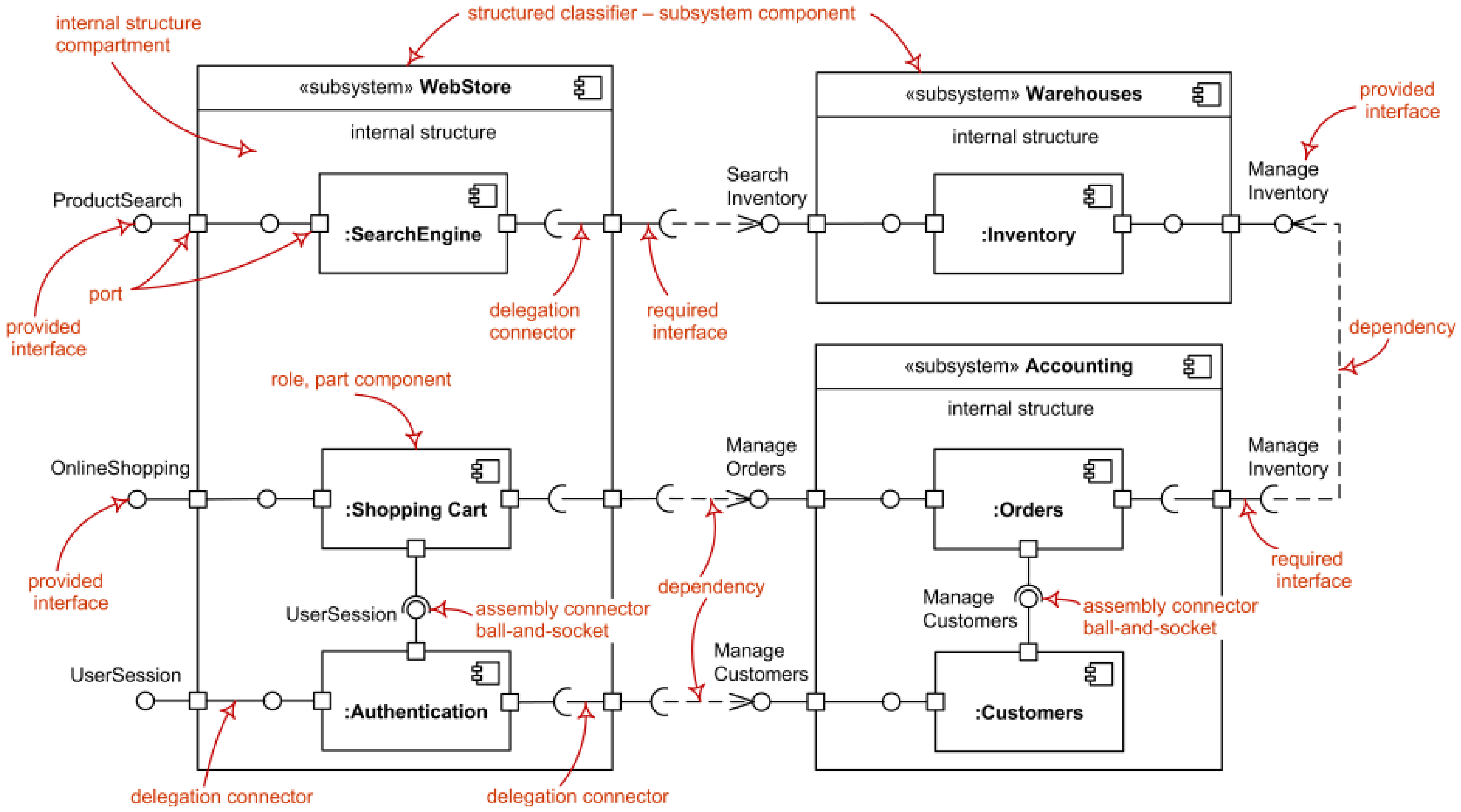
DIAGRAMS

- Activity Diagram



DIAGRAMS

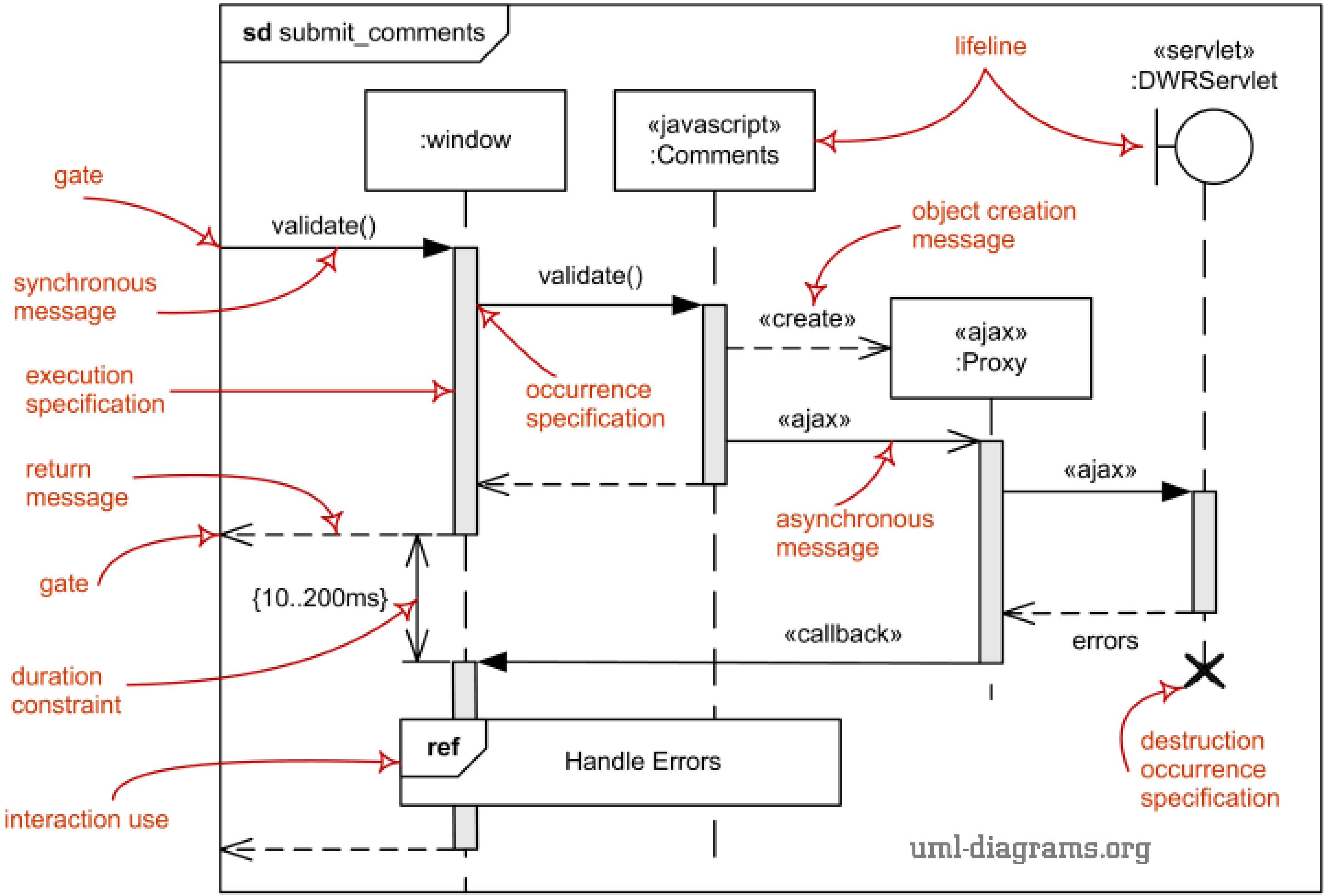
- Component Diagram



Diagrams

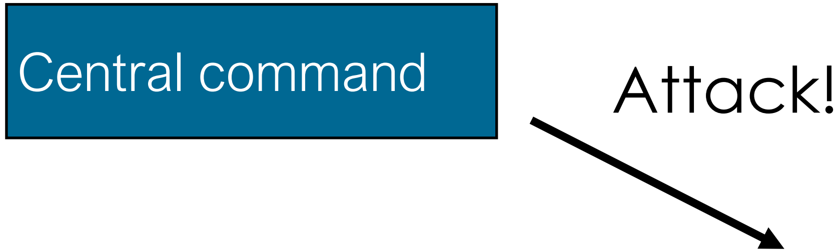
- Sequence Diagram

- https://www.uml-diagrams.org/



SLEEPY GENERALS PROBLEM*

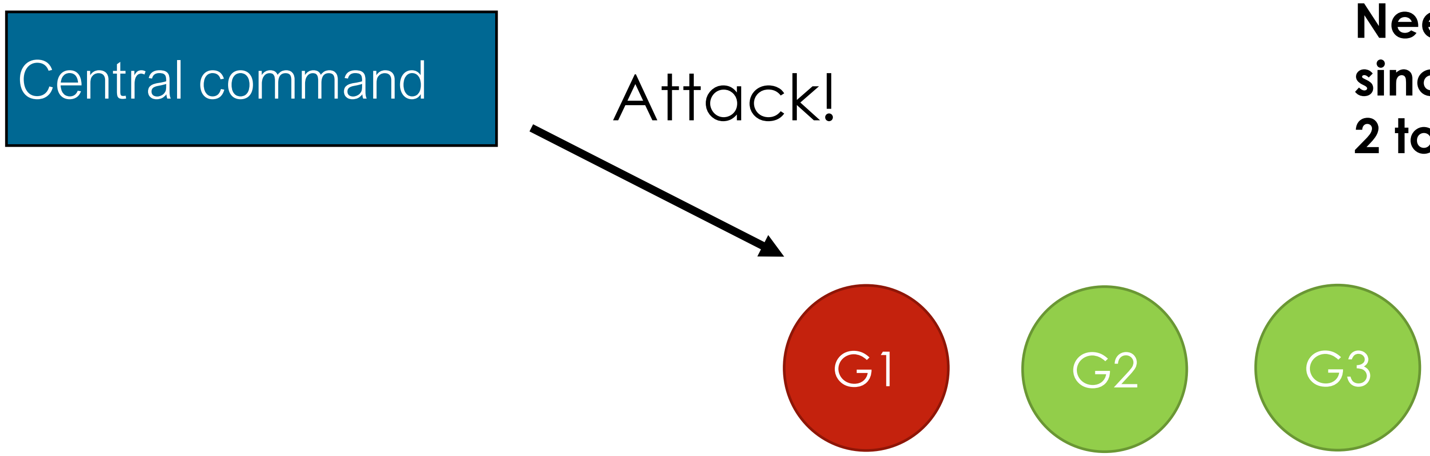
- Our general are tired, but messengers can’t die!
- Need **2** generals to be awake and attack for success
  - If at most **f** generals can fall asleep at a time, how many general do we need?



* I made up this name

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?



Need f+2 generals since we need at least 2 to be awake

* I made up this name

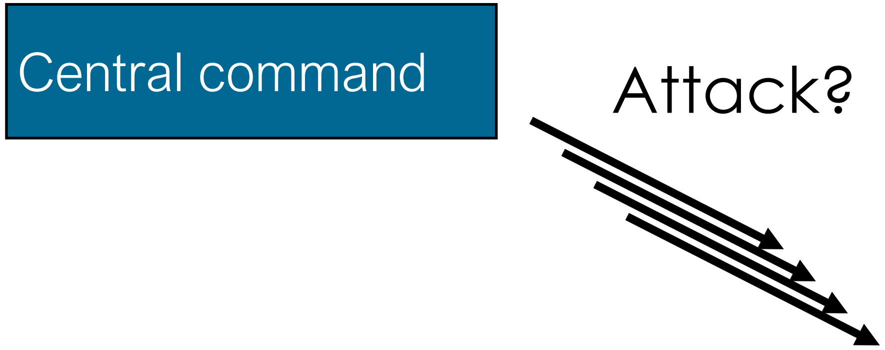
Bureaucratic Generals Problem*

- **Stateless Coordination (Sleepy Generals):**
  - In the Sleepy Generals Problem, the only requirement is that at least two generals must be awake to launch an attack. The system is “stateless” in the sense that you don't need to remember the sequence of orders; Need to ensure that all paperwork is filled correctly!
- **complete history of commands to attack (stateful system):**
  - In contrast, the Bureaucratic Generals Problem adds a layer of complexity by requiring that all generals maintain a complete log—or “paperwork”—of the commands. This means that the coordination isn’t just about having enough generals awake at the moment of action; it’s also about ensuring that the entire history of commands is accurately recorded and can be verified.

* I made up this name too

Bureaucratic Generals Problem[*]

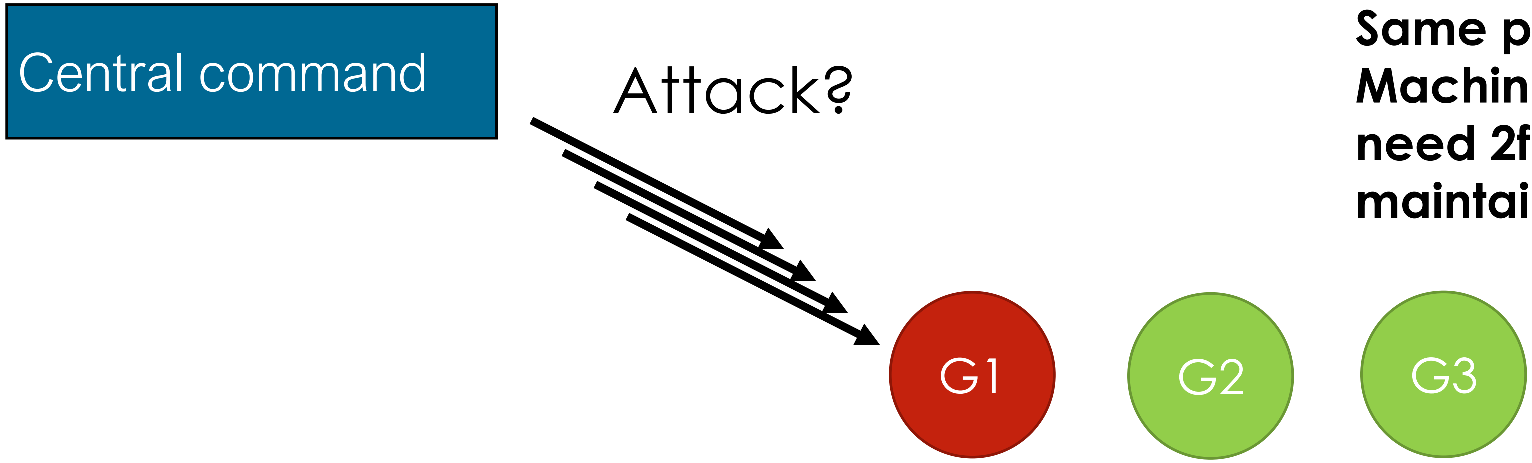
- Our general are tired, but messengers can’t die!
- Need **1** general to be awake and attack for success, **f** can fail
- Need to ensure that all paperwork is filled correctly!
  - Need complete history of commands to attack (stateful system)



[*] I made up this name too

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)

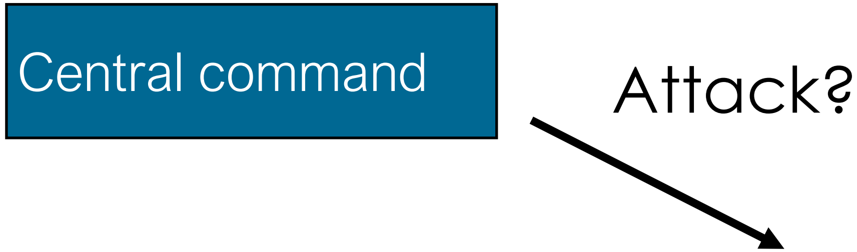


Same problem as State Machine Replication! We need 2f+1 so that we can maintain a log reliably

* I made up this name too

Traitorous Generals Problem*

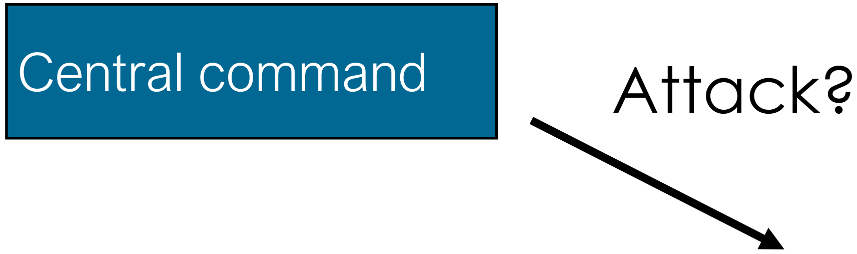
- One of our generals is a traitor!
- How to make **majority of generals agree** to attack?



* 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?

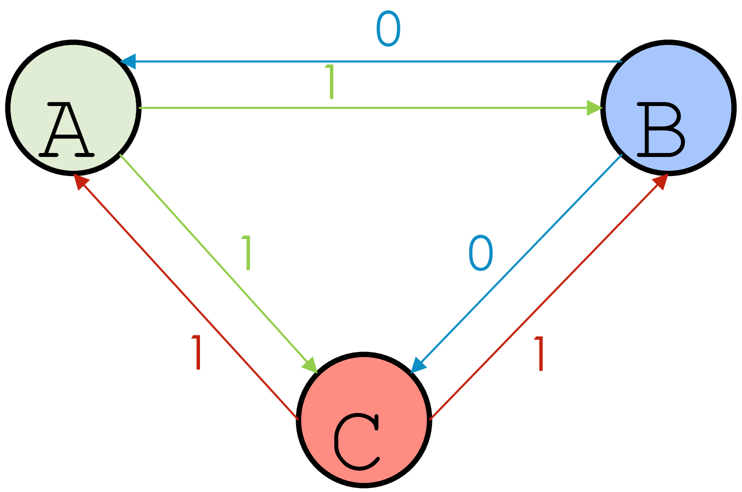


Need more than f+1 replicas!
Can't have a trusted primary anymore!
Replicas need to talk to each other to reach agreement on the decision
Vote and take the majority?

* I made up this name too

Reaching Agreement

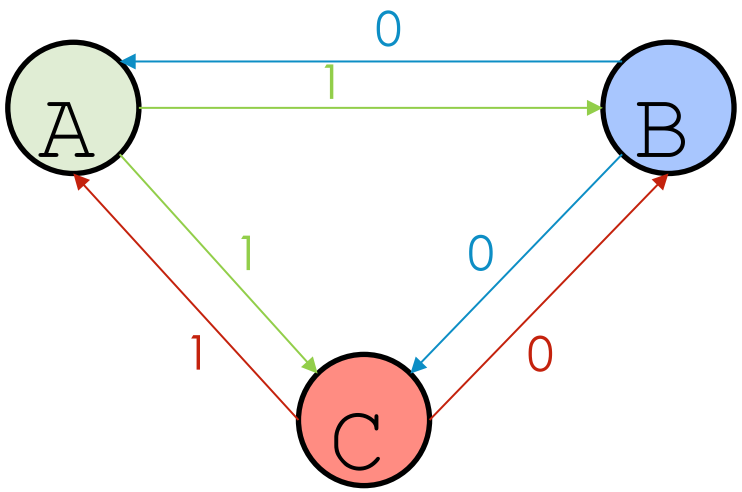
- The assault will only succeed if at least 2 armies attack at the same time
  - I vote we should… 1 = attack, 0 = retreat!



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

REACHING AGREEMENT

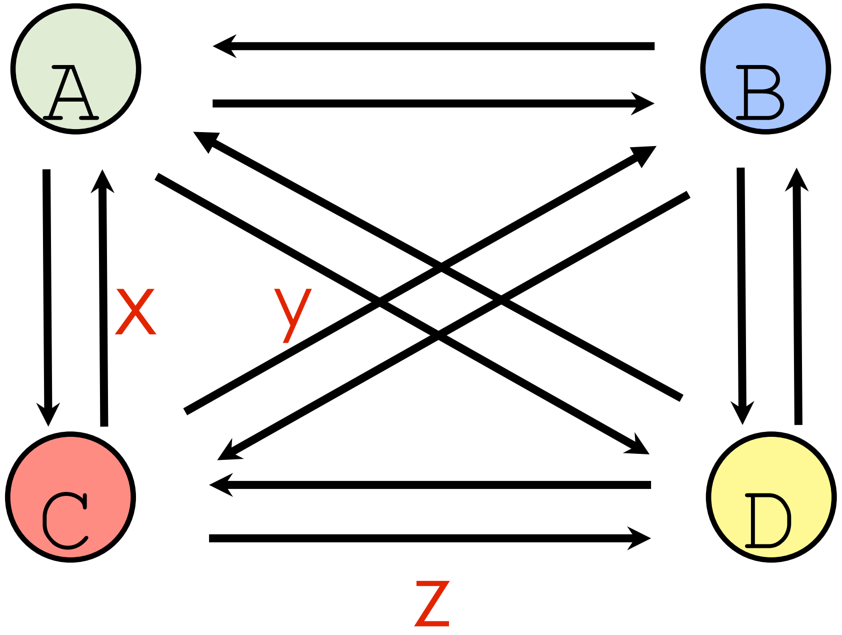
- The assault will only succeed if at least 2 armies attack at the same time
    - I vote we should… 1 = attack, 0 = retreat!



Replica	Receives	Action
A: 1	1-0-1	1
B: 0	1-0-0	0
C: ???	1-0-?	0

Byzantine Generals Solved*!

- 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	A: $(1,0,\underline{1},1)$ B: $(1,0,\underline{0},1)$ C: $(\underline{1},\underline{1},\underline{1},\underline{1})$ D: $(1,0,\underline{1},1)$	A: 1 B: 1 C: 1 D: 1
B	A: $(1,0,\underline{1},1)$ B: $(1,0,\underline{0},1)$ C: $(\underline{0},\underline{0},\underline{0},\underline{0})$ D: $(1,0,\underline{1},1)$	A: 1 B: 1 C: 0 D: 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 _______ replicas to guarantee **x** correct responses from a **stateless** system (typically x=1)
- Bureaucratic Generals Problem
  - If **f** nodes can fail, you need _________ replicas to guarantee a correct response from a **stateful** system
- Byzantine Generals Problem
  - If **f** nodes can be arbitrarily malicious, you need ________ 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

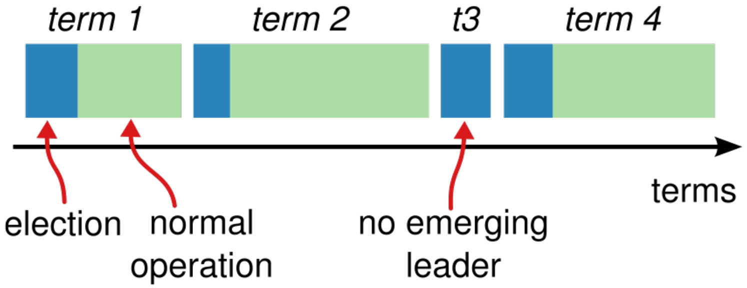
PAXOS AND RAFT

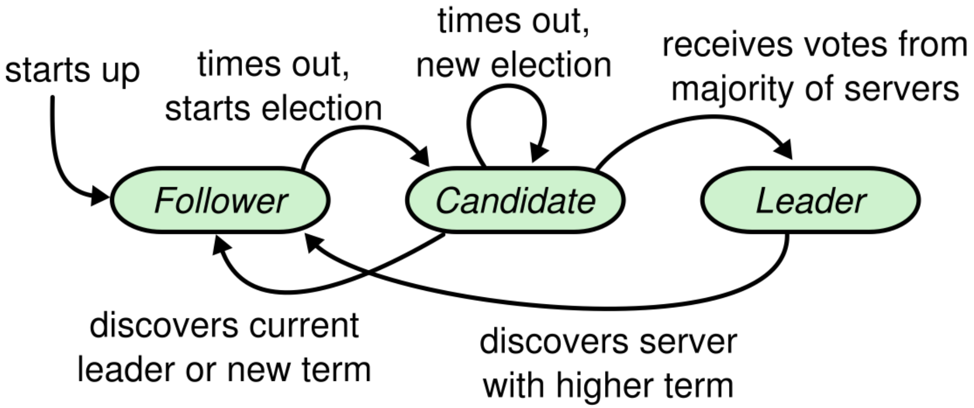
- Goal: Achieve state machine replication for crash fault tolerance (non-byzantine, stateful, reliable network)
- Paxos: Lamport '90, published '98 (interesting history)
  - Consensus algorithm presented in a paper pretending to describe how a fictitious ancient greek civilization wrote laws
  - Used 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)

Following slides were adapted from slides by Diego Ongaro and John Ousterhout

RAFT – KEY IDEAS FOR SMR

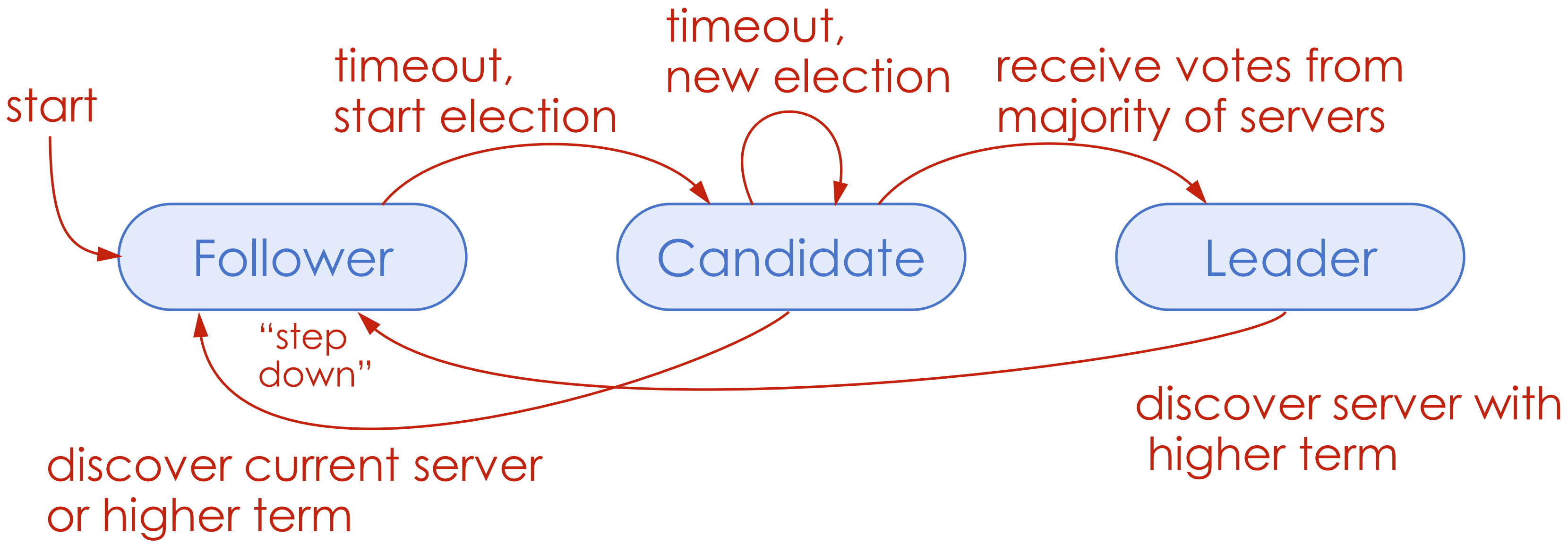
- **Leader election:** Elections periodically occur in case the primary fails
- **Terms:** Help track avoid inconsistent state after recovery
- **Ordered Logs:** All incoming requests pass through leader to be ordered




starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

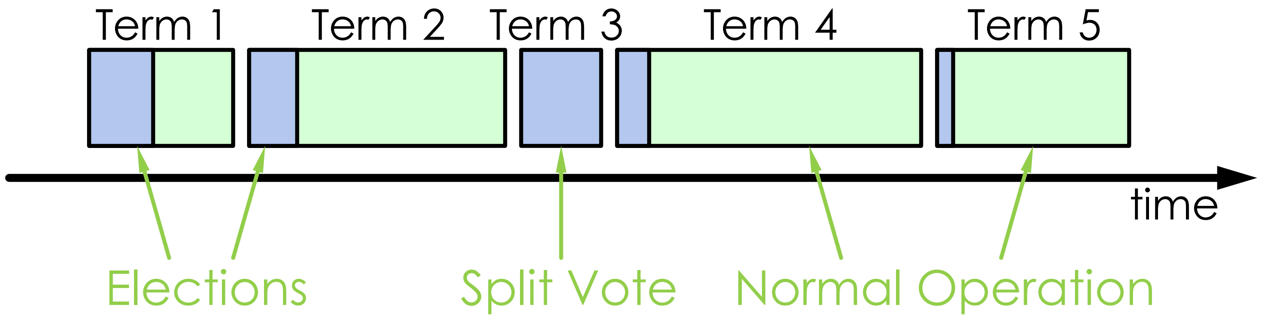
Server States

- 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



Raft Protocol Summary

Followers

- Respond to RPCs from candidates and leaders.
- Convert to candidate if election timeout elapses without either:
  - Receiving valid AppendEntries RPC, or
  - Granting vote to candidate

Candidates

- Increment currentTerm, vote for self
- Reset election timeout
- Send RequestVote RPCs to all other servers, wait for either:
  - Votes received from majority of servers: become leader
  - AppendEntries RPC received from new leader: step down
  - Election timeout elapses without election resolution: increment term, start new election
  - Discover higher term: step down

Leaders

- Initialize nextIndex for each to last log index + 1
- Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts
- Accept commands from clients, append new entries to local log
- Whenever last log index ≥ nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful
- If AppendEntries fails because of log inconsistency, decrement nextIndex and retry
- 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
- Step down if currentTerm changes

RequestVote RPC

Invoked by candidates to gather votes.

Arguments:

candidateId	candidate requesting vote
term	candidate's term
lastLogIndex	index of candidate's last log entry
lastLogTerm	term of candidate's last log entry

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Implementation:

1. If term > currentTerm, currentTerm ← 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

Persistent State

Each server persists the following to stable storage synchronously before responding to RPCs:

currentTerm	latest term server has seen (initialized to 0 on first boot)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries

Log Entry

term	term when entry was received by leader
index	position of entry in the log
command	command for state machine

AppendEntries RPC

Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat .

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat)
commitIndex	last entry known to be committed

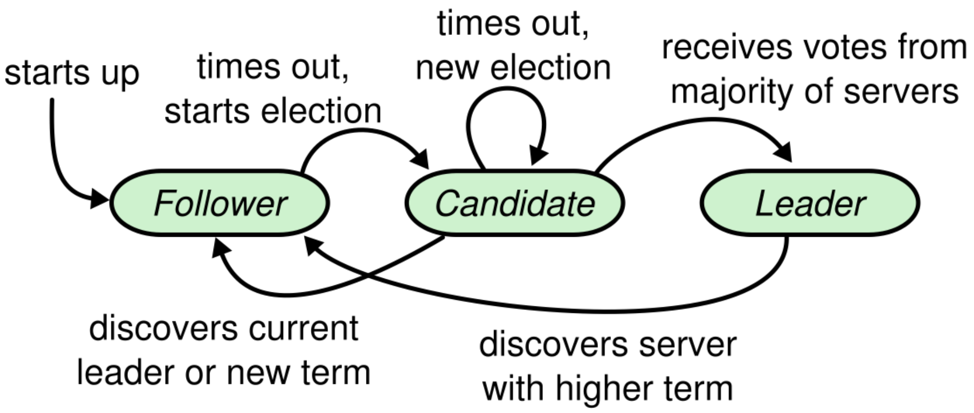
Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Implementation:

1. Return if term < currentTerm
2. If term > currentTerm, currentTerm ← 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

Election Basics


starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

- 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

Let's Run an Election!

Leader:

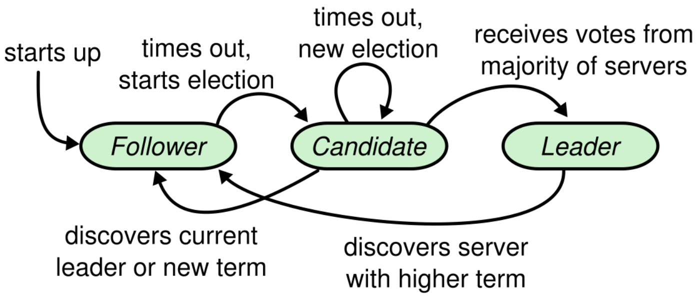
- Sends <**Hello X>** as heartbeat for term X every 5 seconds

Followers:

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

Candidate:

- Send **<Elect ID>**
  - 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**


starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

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

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!

General Sun Tzu

General Washington

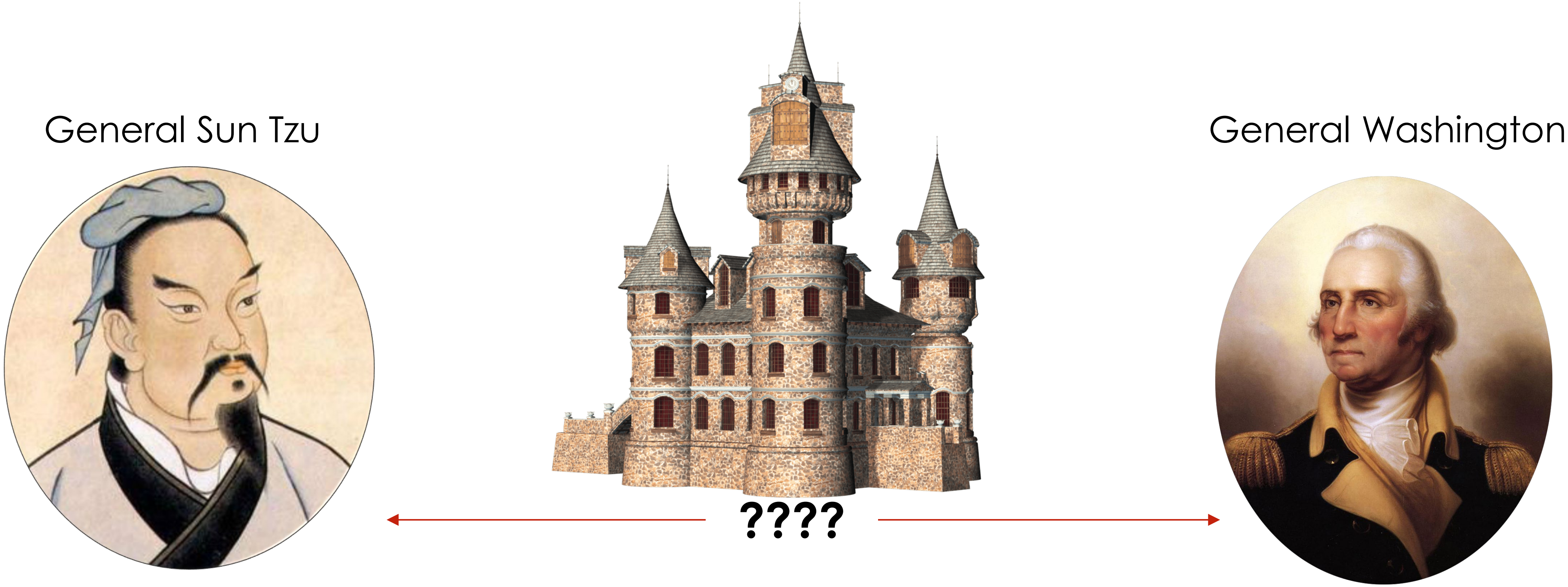
????

"Lossy network"

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



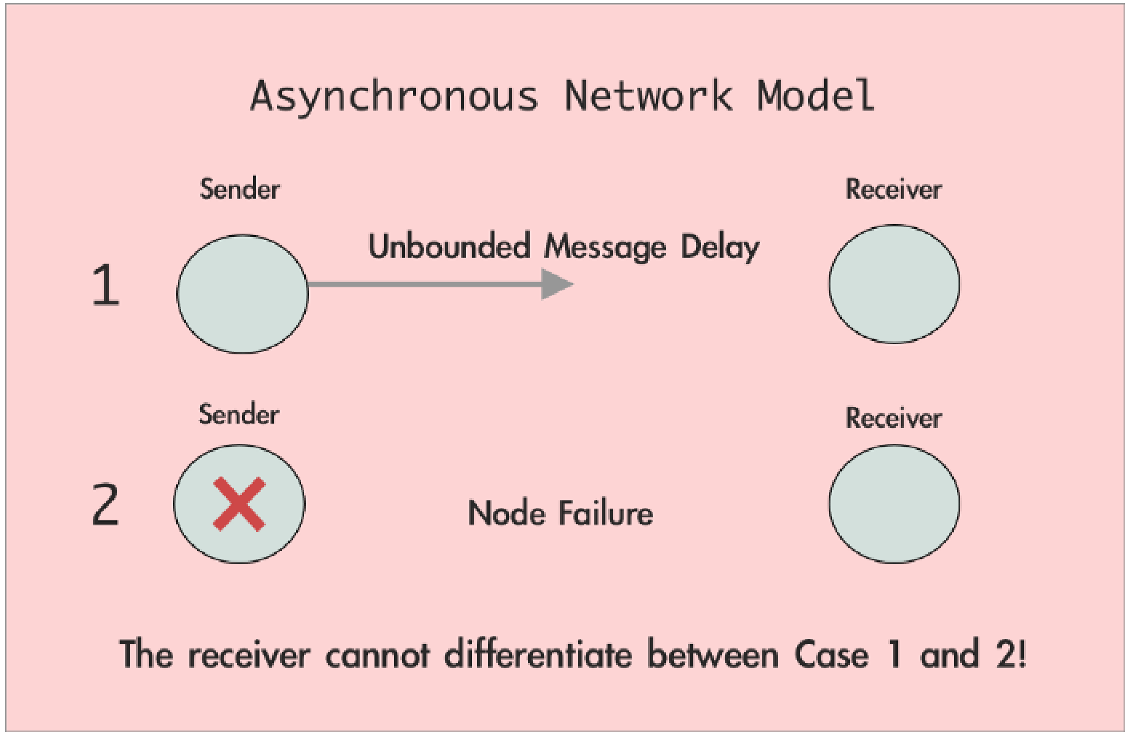
General Sun Tzu

????

General Washington

PROPERTIES

- Asynchrony model □ 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



Asynchronous Network Model

The receiver cannot differentiate between Case 1 and 2!

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

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

There are multiple generals and multiple armies. Success can only be achieved if at least half of them attack at exactly the same time. If they fail to coordinate the timing of the attack, they will be defeated and lose the battle for sure.

They can only communicate with the other generals via their messenger. Moreover, Unfortunately, they have no way to check the authenticity of the message that they receive from the messenger.

So how could an agreement be achieved in this circumstance?

That's your Byzantine Generals' Problem.