CIS 5050
Software Systems

Linh Thi Xuan Phan

Department of Computer and Information Science
University of Pennsylvania

Lecture 16: Byzantine Fault Tolerance
April 4+9, 2024

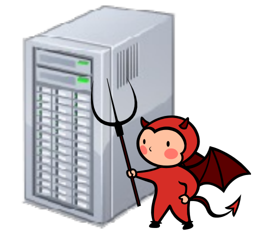
Announcement

- TA Panel on PennCloud
  - When: Thursday, April 11 at 1pm
  - Where: Zoom (details will be announced via Ed)
- Goal: Sharing experience working on the project and Q&As
- Panelists: Many of your fantastic TAs!
- Please do attend and come with questions

Remember: Fault tolerance

- Earlier, we talked about the consensus problem
  - Several nodes want to 'agree' on a single value, from among several different proposals
  - This is a key building block in many distributed systems – e.g., for state-machine replication
- Paxos offers (crash-)fault-tolerant consensus!
  - Excellent!
- But what if the assumption does not hold?
  - In other words, what happens if nodes can fail in other ways?
  - What if we assume a much more 'difficult' fault model?

Recap: Byzantine fault model



- Remember the Byzantine fault model?
- Allows faulty nodes to deviate from the protocol
  - They can send extra messages
  - They can suppress messages they were supposed to send
  - They can tamper with their (local) data
  - They can conspire with each other
  - They can send different messages ("tell lies")
  - They can equivocate, i.e., tell different things to different other nodes
    - What would be an example of that?
  - They can crash (intentionally or unintentionally)
    - How does this compare to the crash fault model?
- What kinds of real-world faults does this represent?

What can we assume?

- Idea: Let's assume that any subset of the nodes in the system can become Byzantine!
- Good: Very conservative assumption!
  - We can "sleep well at night": if our system can handle Byzantine faults, then it can handle pretty much anything that can happen
- Bad: No useful system designs possible!
  - What if all the nodes fail at the same time & destroy their data?
- We need some kind of limit
  - Example: Up to f nodes can be Byzantine at any given time

Questions you may have

- Is Byzantine fault tolerance (BFT) even possible?
  - Yes! There are protocols that can do this.
- What kinds of assumptions do we need?
  - As discussed just now, a limit on the number of faulty nodes
  - Also, synchrony and cryptographic signatures make things easier
- What is the cost, relative to crash fault tolerance?
  - BFT is substantially more expensive than, say, Paxos!

Plan for today

- Motivation
- The Byzantine Generals Problem NEXT
  - Impossibility for N=3f
  - Solution for N=3f+1
  - Solution with signatures
- Byzantine Fault Tolerance
  - PBFT

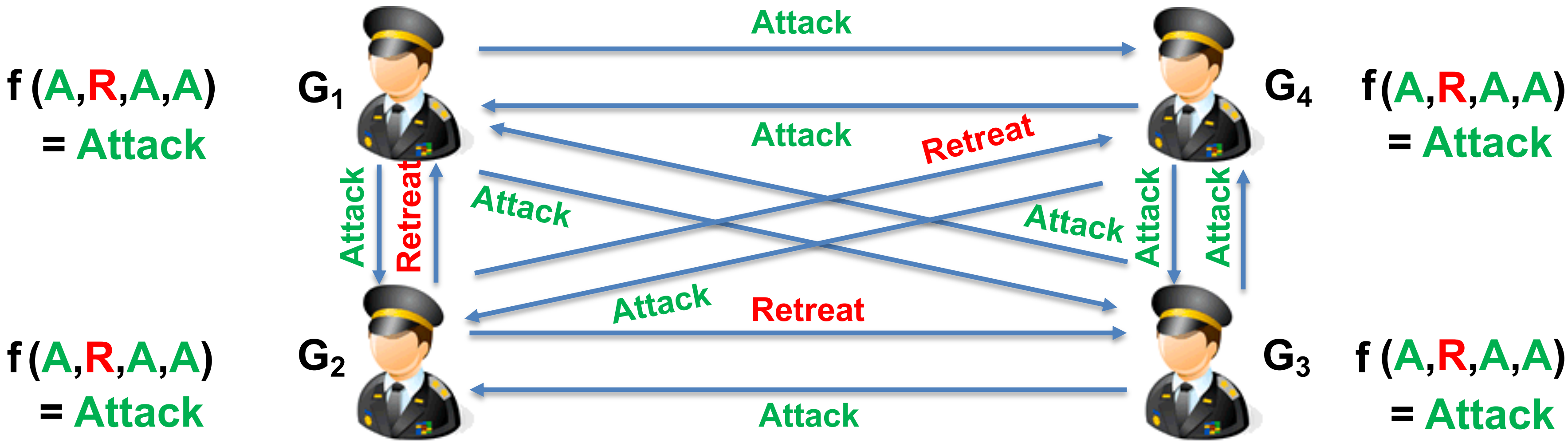
Motivation: Byzantine Generals

- Suppose several divisions of the Byzantine army are camped outside an enemy city
  - The generals need to come up with a common plan of action
    - Example: "Attack at dawn", "Retreat"
  - They can only succeed if they all follow the plan ← **Consensus!**
  - However, they can only communicate by messenger
  - Some of the generals may be traitors ← **Byzantine faults!**

How can we solve this?

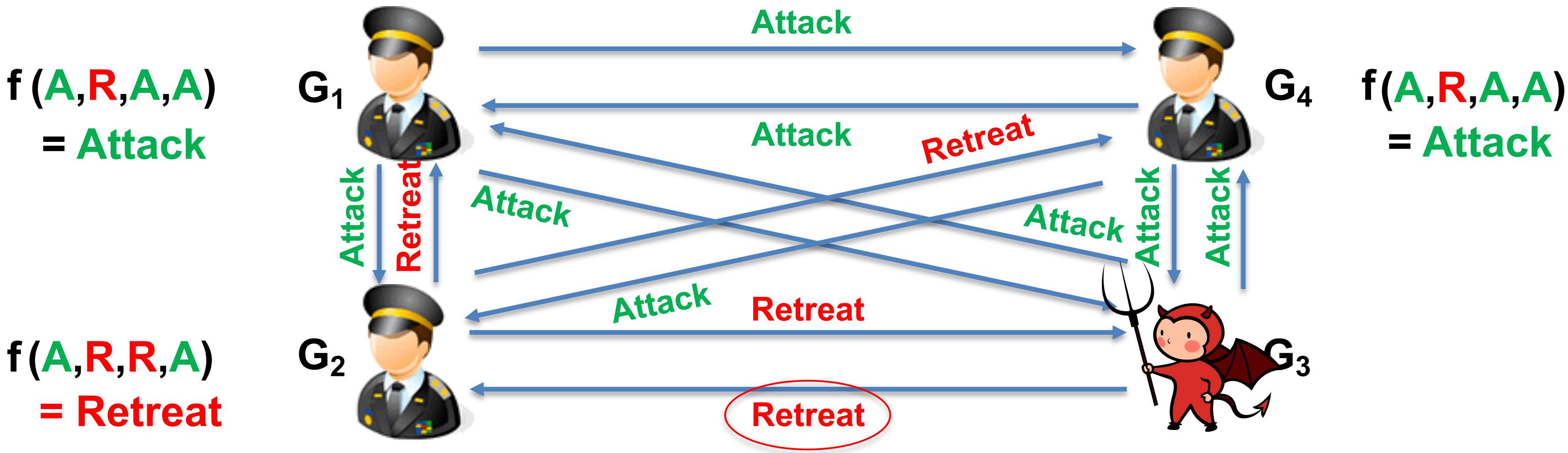
- Goal #1: All loyal generals decide on the same plan
  - Best we can hope for - we can't really control what the traitors do
- Goal #2: A small number of traitors should not cause the loyal generals to adopt a bad plan
  - For instance, if all the generals do what General #17 says, they will be in trouble if General #17 is a traitor!
- How can we accomplish this?
  - Idea: Suppose each of the N generals has an opinion v_i about what they should do, and suppose all the generals know all the $v_1,...,v_N$
  - If they all apply some deterministic function $f(v_1,...,v_N)$ to decide what to do, then we have reached goal #1
  - If f(...) is 'robust' (say, the majority), then we have reached goal #2!

Strawman solution



- How can the generals learn the $v_1,\ldots,v_N$?
  - Idea: They could send messengers to each other
  - If every general G_i sends v_i to every other general G_j, then they all know the vector of 'opinions'
  - They can then each apply their deterministic function f(...) to decide what the joint decision is

Problem: Equivocation



- But: Traitors can send different v_i to each general!
  - Result: Loyal generals could come to different conclusions!
  - This is called equivocation

- What can we do about this?

Reduction to commander+generals

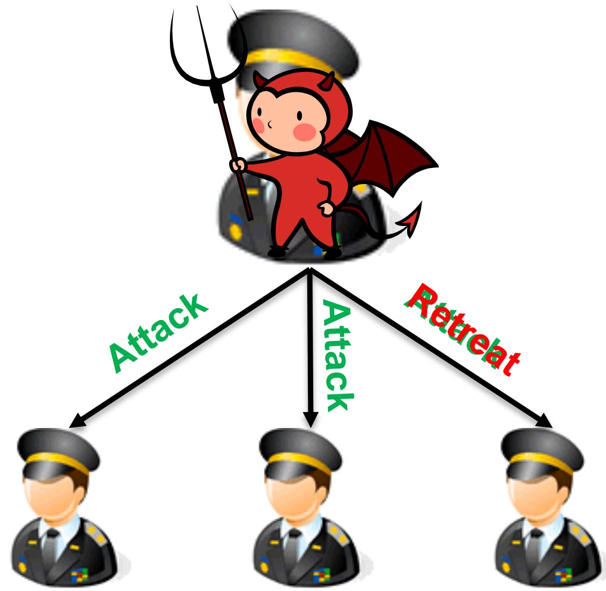
- We need to make sure that every loyal general uses the same $v_1,...,v_N$!
  - Is this enough?
  - No! Trivial solution: Use "Retreat" for all v_i!
  - We also need to make sure that, if G_i is loyal, then its actual value will be used as v_i by all the loyal generals!

- What is the core of the problem?
  - We need to make sure that all the loyal generals use the same v_i!
  - Same problem for each i (no dependencies between generals)
  - Let's look at a somewhat simpler problem that just considers what some particular general is proposing (we'll call him the "commander")
  - Once we can solve that, we can easily assemble a full solution, as discussed just now

Plan for today

- Motivation
- The Byzantine Generals Problem NEXT
  - Impossibility for N=3f
  - Solution for N=3f+1
  - Solution with signatures
- Byzantine Fault Tolerance
  - PBFT

The Byzantine Generals Problem



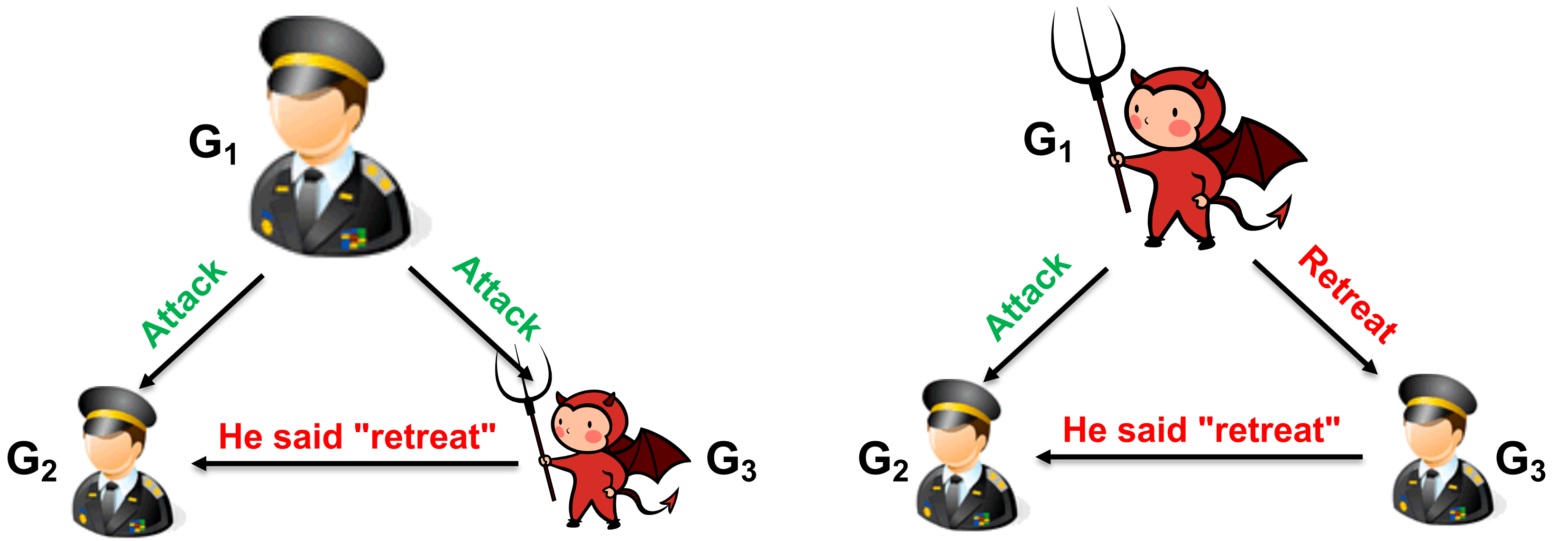
- A commanding general must send an order to his N-1 lieutenant generals, such that
  - IC1: All loyal lieutenants obey the same order, and
  - IC2: If the commanding general is loyal, then every loyal lieutenant obeys the order that he sends
- Let's think about this a bit...

Assumptions

- Let's be more specific about what we assume
  - Bounded faults: At most f generals can be traitors at any given time
  - Reliable network: Messages are not lost in transit
  - Authentication: Recipient can tell who (directly) sent a message
  - Synchrony: Transmission delay is bounded

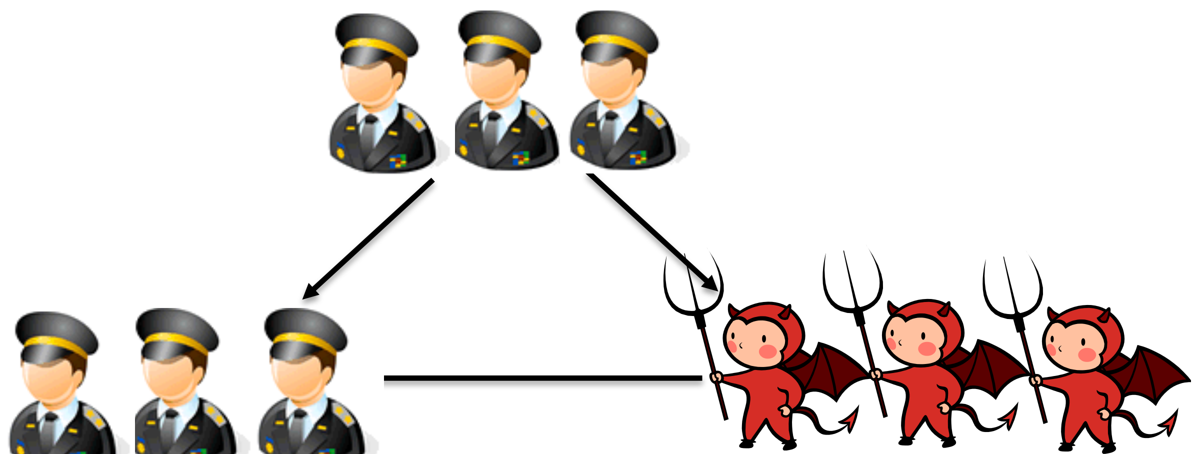
- What does this mean?
  - Authentication is not the same as cryptography: We only assume that we can tell who the direct sender is!
    - Cryptography makes things a bit easier! We'll get to this later.
  - What happens when a general doesn't send a message at all?
    - In what situation can this happen?
    - Can the recipient tell when it happens?

How many traitors can we tolerate?



- Let's assume that there are just N=3 generals.
  - Is there a solution that 'works' if there is f=1 traitor?
- No! Consider the two scenarios above:
  - On the left, the commander is loyal, so IC2 requires that G_2 attacks
  - On the right, IC1 requires that G_2 does the same as G_3 (retreat)
  - But the two look exactly alike to G_2!

Generalizing to larger N

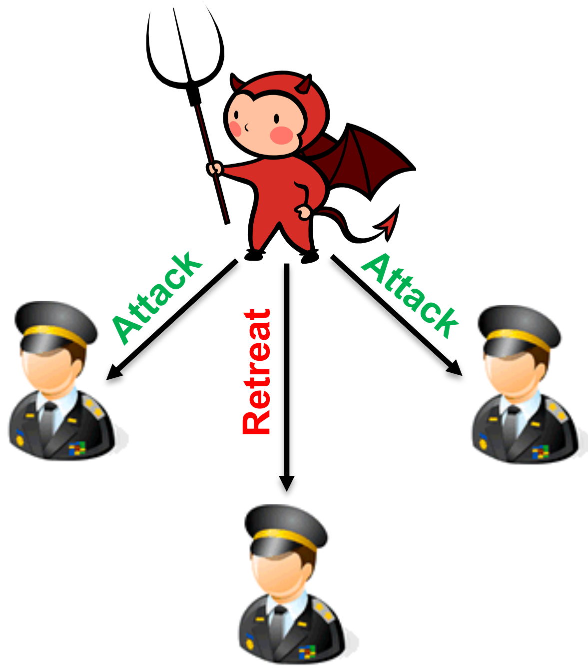


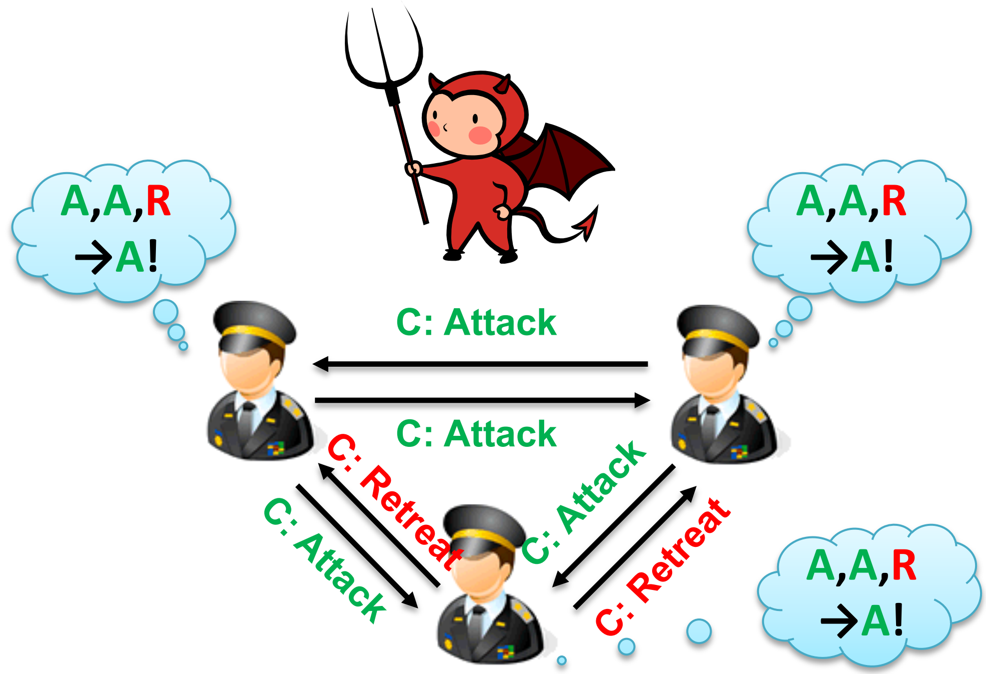
- What if we had more generals?
  - We can replace each Byzantine general in the scenario just now with m other ("Albanian") generals
  - A similar argument applies: if m generals are traitors, they can produce two scenarios just like on the previous slide
- We need at least 3f+1 generals to tolerate f traitors!

Plan for today

- Motivation
- The Byzantine Generals Problem
  - Impossibility for N=3f
  - Solution for N=3f+1 NEXT
  - Solution with signatures
- Byzantine Fault Tolerance
  - PBFT

Strawman solution





- Can we solve the problem with 3f+1 generals?
- Idea: Lieutenants exchange information
  - Each lieutenant tells the others what he heard from the commander
    - Need a default value (say, "Retreat") if a traitor fails to send a message
  - Once they all know what everyone heard, they go with the command that they heard the most often

Strawman solution

- Will this work?
- Works fine for f=1 traitors!
  - If the traitor is a lieutenant, he will be 'outvoted' by the other two
  - If the traitor is the commander, the lieutenants all have the same information, and will come to the same conclusion
- But what if f>1 (and N>3f)?
  - Suppose the commander is a traitor, and also a lieutenant
  - The commander could send a different value to each lieutenant (say, "Attack", "Retreat", "Wait", "Panic", ...)
  - The traitor among the lieutenants could then forward different values to the other lieutenants (say, "Attack" to some, and "Retreat" to others)
  - Each lieutenant goes with the value he receives twice – and that can be different for different loyal lieutenants!

What can we do?

- We did make some progress, however!
  - The protocol works fine for f=1 traitors!
- Idea: Invoke the protocol recursively
  - Say, OM(1) is the protocol we discussed just now (for f=1)
- We can construct a protocol OM(2) as follows:
  - Each lieutenant uses OM(1) to tell the other lieutenants about the value he heard
  - If the commander is a traitor, this will work fine: Since we have f=2, there can be at most one traitor among the lieutenants
  - If the commander is loyal, OM(1) can produce a discrepancy of up to +/- 1 (because of the traitor), but that's ok, since every loyal lieutenant already has one (consistent) value from the commander, so the majority value will still be the correct one!
- Similar constructions are possible for OM(f), f>2
  - However, high message complexity!

Plan for today

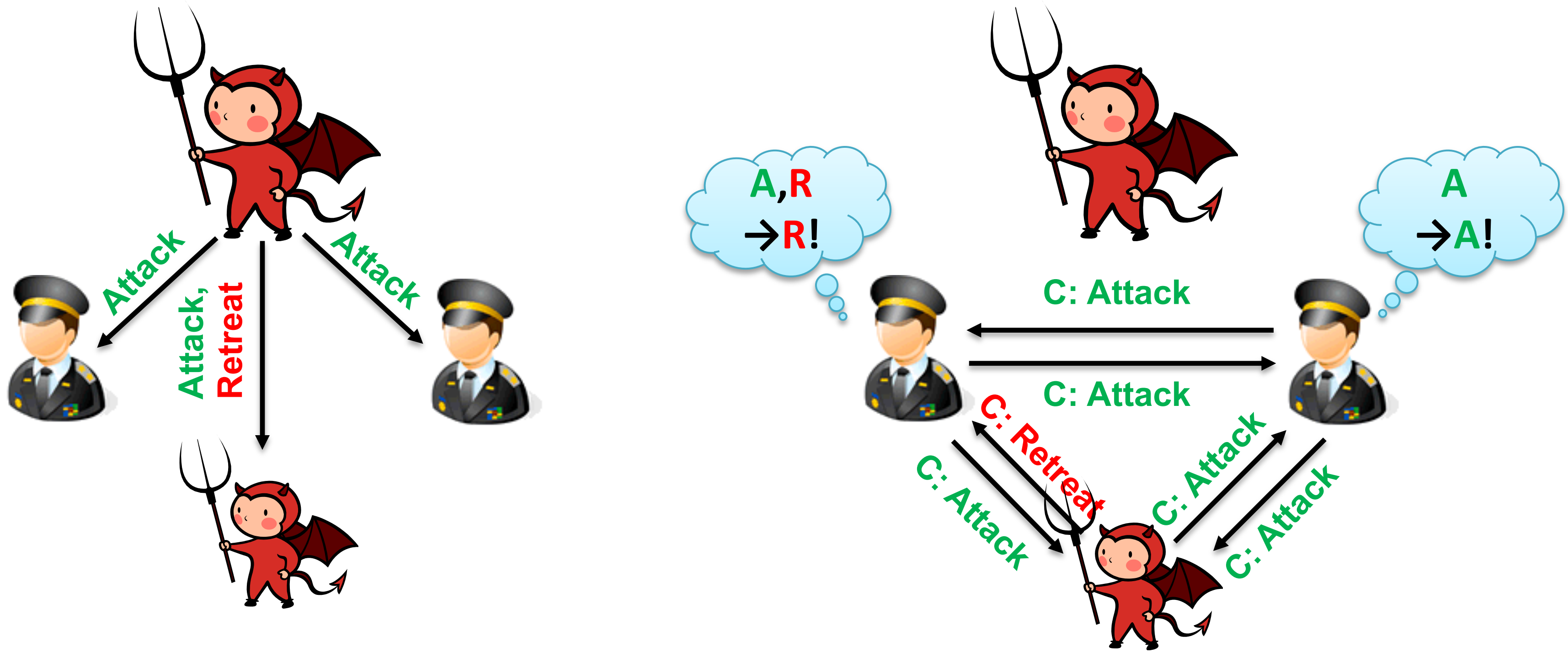
- Motivation
- The Byzantine Generals Problem
  - Impossibility for N=3f
  - Solution for N=3f+1
  - Solution with signatures NEXT
- Byzantine Fault Tolerance
  - PBFT

Solution for N=3 with signatures

- What if the generals can sign their messages?
  - Assumption #1: Loyal generals can recognize each other's signature
  - Assumption #2: Traitors cannot forge the signature of loyal generals

- Then a solution for N=3 is possible!
  - The commander can send a signed command to the lieutenants
  - The lieutenants can exchange the messages they received
    - The lieutenants can no longer tell lies (need to forge commander's signature!)
  - If they have...
    - no signed command at all: Go with some default decision (e.g., "Retreat")
    - a single signed command: Go with that command
    - more than one signed command: The commander is a traitor! Use a deterministic decision function to pick one of the commands (or simply go with "Retreat")
  - However, if there are N>3 nodes, we have to be a little more careful about the forwarding part

One final problem



- Suppose we have N=4, and f=2
  - The commander signs two conflicting orders and sends the first to all loyal lieutenants, and sends both only to the other traitor
  - The traitor then forwards the second order only to one of the loyal lieutenants, and the first order to the other → conflicting decisions possible!

General solution with signatures

- What do we have to do to prevent this?
  - If some signed order O has been received by any loyal lieutenant, then it must also be received by all other loyal lieutenants!

- Algorithm:
  - The general signs his order and sends it to the lieutenants
  - When a lieutenant receives a properly signed order:
    - If he hasn't seen that order yet, he adds it to his set of valid orders
    - If the order doesn't yet have signed endorsements from f other lieutenants on it, he adds his own endorsement and sends the order to the other lieutenants
  - Once forwarding terminates, each lieutenant uses the deterministic decision function to pick an order (from the set of valid orders) and then follows that order

Why does this work?

- If the commander is loyal, all is well
  - The commander signs only one order, and sends it to all lieutenants
  - Since no lieutenant can forge the signed order, all loyal lieutenants receive a single order (which is signed by the commander)
- What if the commander is a traitor?
  - Consider some order O that the commander has signed, and consider what a loyal lieutenant L should do once he receives O
    - We need to make sure that O is received by all the other loyal lieutenants!
  - Idea: Look at the endorsements
  - If there are endorsements from f other lieutenants:
    - At least one of them must be loyal (since the commander is a traitor, and there are at most f traitors!). In that case, this loyal lieutenant would have previously forwarded O to all the other lieutenants!
  - If there are fewer than f endorsements:
    - L would add its own endorsement and forward O to all other lieutenants!

Plan for today

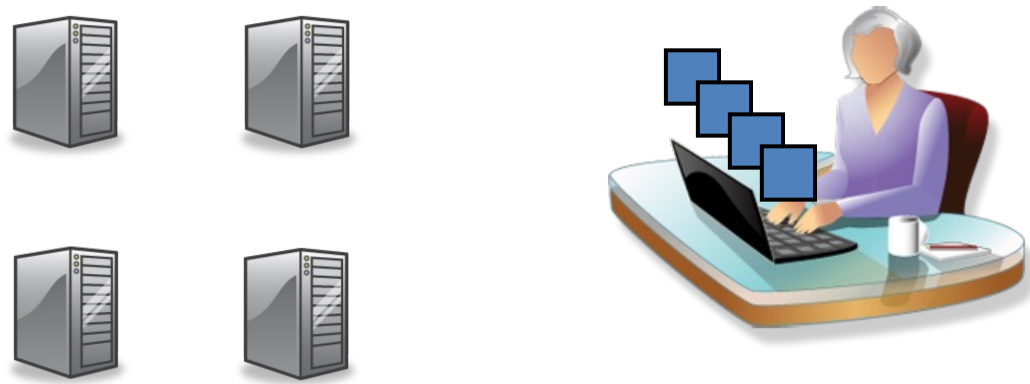
- Motivation
- The Byzantine Generals Problem
  - Impossibility for N=3f
  - Solution for N=3f+1
  - Solution with signatures
- Byzantine Fault Tolerance NEXT
  - PBFT

Why do we care about this?

- Original goal was Byzantine Fault Tolerance (BFT)
  - How can the Byzantine Generals problem help us with this?

- We can use these ideas to implement a form of state-machine replication, just like with Paxos!
  - Remember the original problem definition (without the commander)?
  - Each general had some opinion about what they should all do
  - The goal was for the loyal generals to make a consistent decision
  - This is in essence the mechanism we need for SMR! Each node can be a 'general', and it can 'propose' a next step for the state machine
  - We can then run the protocol, and all the non-faulty nodes will agree on the same next step, which they can then all execute
  - How would a client use this?

Recall: Replicated service



- How would this work?
  - Client sends its requests to each of the nodes
  - The nodes each 'propose' some request as the next one to execute
  - They use a Byzantine-tolerant protocol to agree on one request
    - Requests are executed in a consistent order, starting from the same state
    - If the state machine is deterministic, all correct replicas will be in a consistent state
  - They each execute the request and reply to the client
  - The client goes with the majority response

PBFT

- The classical protocol for this is PBFT
  - From a paper by Castro and Liskov, "Practical Byzantine Fault Tolerance" (OSDI'99)
  - This has sparked a whole line of work on BFT protocols: Q/U, HQ, Zyzzyva, Aardvark, ZZ, ...
- PBFT makes somewhat different assumptions:
  - The nodes can sign their messages (as discussed above)
  - The network is asynchronous – that is, there is no hard limit on how much a message can be delayed in the network
- Why is this assumption useful?
  - Difficult to get a hard bound on message delays in real networks
  - Adversary could try to break the system by delaying messages (which could be easier than hacking into a node directly)

What is so hard about asynchrony?

- Is the problem harder for asynchronous systems?
  - Yes! Now there is no longer a way for a node to decide whether a message hasn't been sent, or whether it is simply taking a long time!

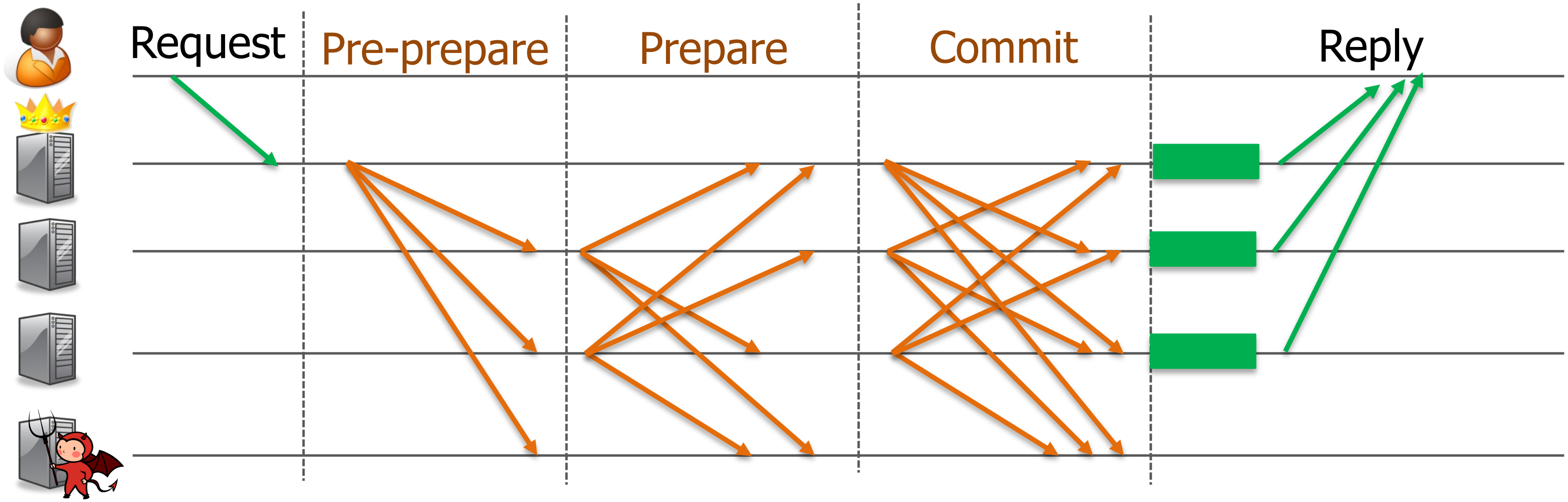
- The limit for asynchronous systems is 3f+1, even with signatures!
  - It has to be possible to proceed after talking to N-f replicas (since f replicas could be faulty and may not respond at all)
  - But the protocol also has to work if f of the replicas that do respond are faulty (and the other messages were simply delayed)
  - To get a correct majority in that case, we need N-2f > f, or N>3f

- PBFT requires 3f+1 replicas to tolerate f faults
  - So, it is optimal!

What guarantees does PBFT give?

- PBFT guarantees safety
  - Think of a safety property as saying "bad things will never happen"
  - As long as no more than f of the 3f+1 replicas are faulty, the replicated service satisfies linearizability
    - In other words, it behaves like a single, centralized implementation
- PBFT also guarantees liveness
  - Think of a liveness property as "good things will happen eventually"
  - In this case, the guarantee says that the system will keep making progress and process requests
- Can it guarantee both?
  - No! It needs at least a little bit of synchrony for liveness
  - Classical case of having your cake and eating it too (FLP impossibility)
  - However, losing liveness temporarily is often less bad than losing safety

How does PBFT work?



- At a high level:
  - Client sends (signed) request to a specific node (the "primary")
  - Primary multicasts the request to the other replicas
    - This uses a multi-round protocol (PRE-PREPARE, PREPARE, COMMIT)
    - Somewhat like Paxos, but with more rounds
    - Guarantees that requests are totally ordered even if the primary is Byzantine
  - Replicas execute the request & send responses to the client
  - Client waits for f+1 matching responses & uses this as the result
  - Details are a little complicated (see the paper for that!)

Recap: PBFT

- Implements a replicated state machine
  - Many distributed systems can use this as a building block!
- Stronger than Paxos, but also more costly
  - Safety & liveness even when there are Byzantine faults!
  - However, needs more rounds (latency!) and lots of messages
- Needs 3f+1 replicas to tolerate f Byzantine nodes
  - Optimal for asynchronous systems!