CS5412: CONSENSUS AND THE FLP IMPOSSIBILITY RESULT

Lecture XII

Ken Birman

Generalizing Ron and Hermione's challenge

- Recall from last time: Ron and Hermione had difficulty agreeing where to meet for lunch
  - The central issue was that Ron wanted to reason in a perfectly logical way. They never knew for sure if email was delivered... and always ended up in the "default" case
- In general we often see cases in which N processes must agree upon something
  - Often reduced to "agreeing on a bit" (0/1)
  - To make this non-trivial, we assume that processes have an input and must pick some legitimate input value
- Can we implement a fault-tolerant agreement protocol?

Connection to consistency

- A system behaves consistently if users can't distinguish it from a non-distributed system that supports the same functionality
  - Many notions of consistency reduce to agreement on the events that occurred and their order
  - Could imagine that our "bit" represents
    - Whether or not a particular event took place
    - Whether event A is the "next" event
- Thus fault-tolerant consensus is deeply related to fault-tolerant consistency

Consensus ≡ Agreement?

- For CS5412 we treat these as synonyms
- The theoretical distributed systems community has detailed definitions and for that group, the terms refer to very similar but not identical problems
- Today we're "really" focused on Consensus, but don't worry about the distinctions

Fischer, Lynch and Patterson

- A surprising result
  - Impossibility of Asynchronous Distributed Consensus with a Single Faulty Process
- They prove that no asynchronous algorithm for agreeing on a one-bit value can guarantee that it will terminate in the presence of crash faults
  - And this is true even if no crash actually occurs!
  - Proof constructs infinite non-terminating runs

Core of FLP result

- They start by looking at an asynchronous system of N processes with inputs that are all the same
  - All 0's must decide 0, all 1's decides 1
- They are assume we are given a correct consensus protocol that will "vote" (somehow) to pick one of the inputs, e.g. perhaps the majority value
  - Now they focus on an initial set of inputs with an uncertain ("bivalent") outcome (nearly a tie)
  - For example: N=5 and with a majority of 0's the protocol picks 0, *but with a tie, it picks 1*. Thus if one of process with a 0 happens to fail, the outcome is different than if all vote

Core of FLP result

- Now they will show that from this bivalent state we can force the system to do some work and yet still end up in an equivalent bivalent state
- Then they repeat this procedure
- Effect is to force the system into an infinite loop!
  - And it works *no matter what correct consensus protocol you started with.* This makes the result very general

Bivalent state

S_* denotes bivalent state

S_0 denotes a decision 0 state

S_1 denotes a decision 1 state

System
starts in S_*

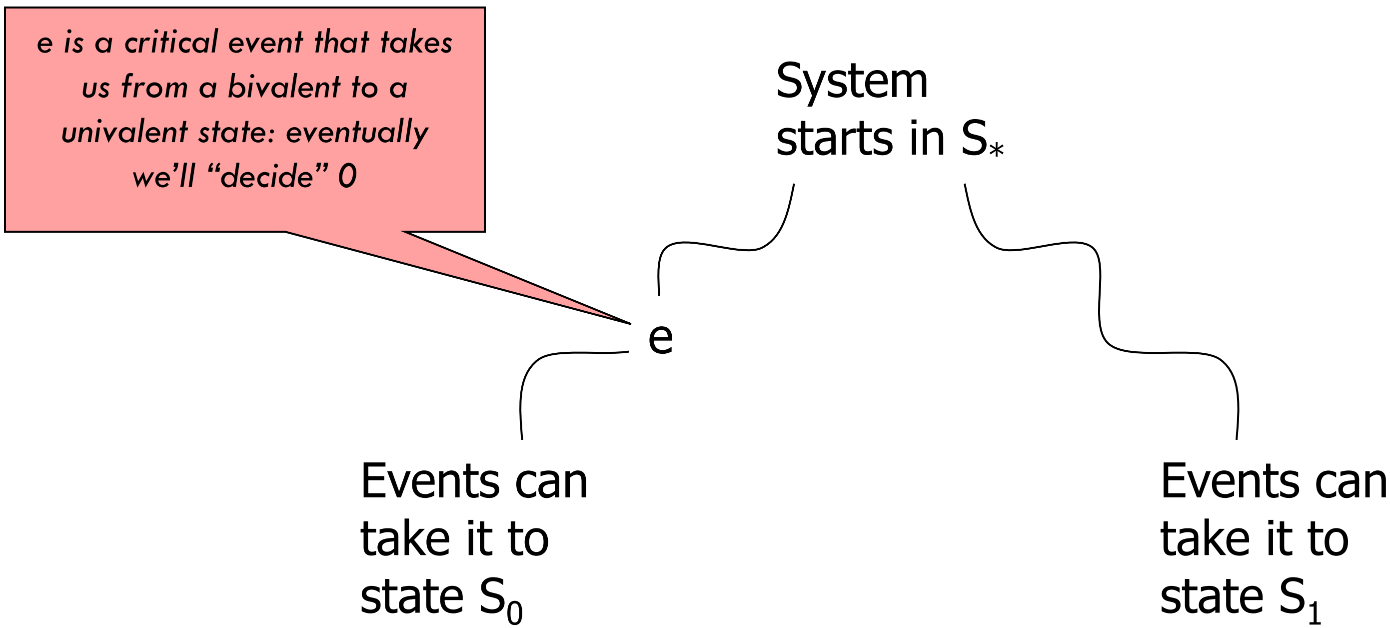
Events can
take it to
state S_0

Sooner or later all executions
decide 0

Events can
take it to
state S_1

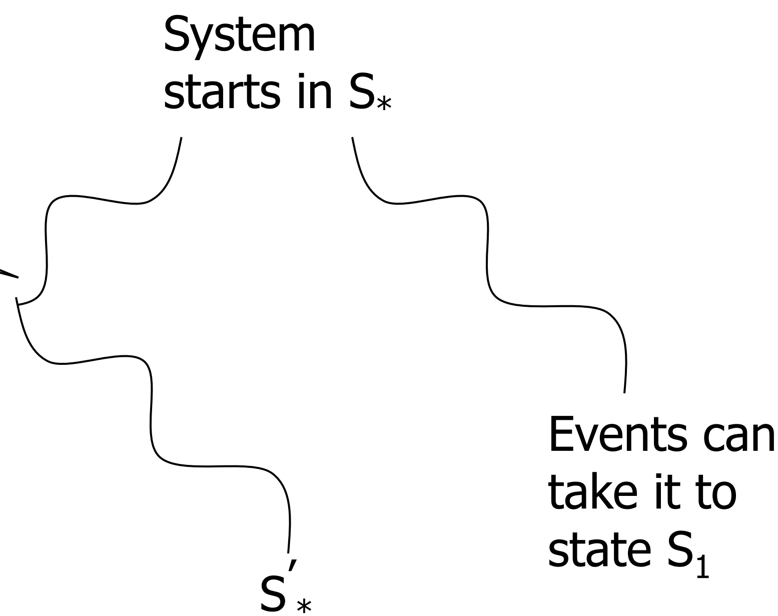
Sooner or later all executions
decide 1

Bivalent state

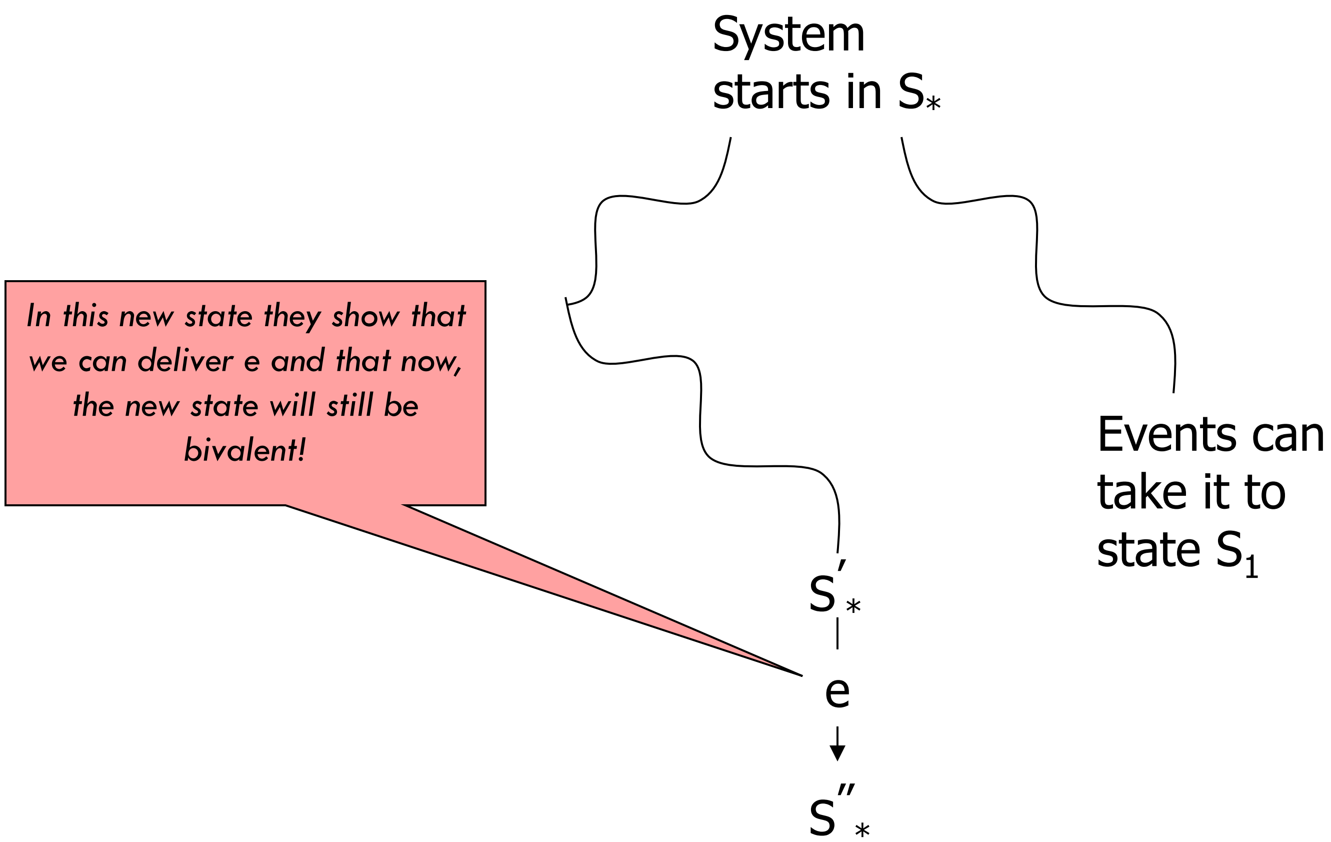


Bivalent state

They delay e and show that there is a situation in which the system will return to a bivalent state

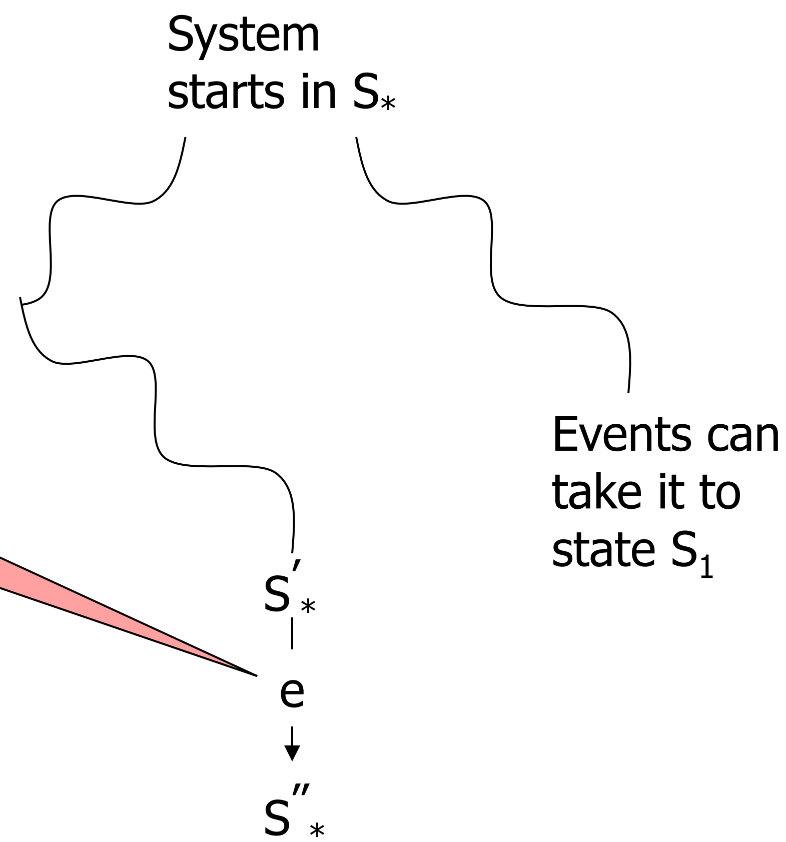


Bivalent state



Bivalent state

Notice that we made the system do some work and yet it ended up back in an "uncertain" state. We can do this again and again



Core of FLP result in words

- In an initially bivalent state, they look at some execution that would lead to a decision state, say "0"
  - At some step this run switches from bivalent to univalent, when some process receives some message *m*
  - They now explore executions in which *m* is delayed

Core of FLP result

- So:
  - Initially in a bivalent state
  - Delivery of *m* would make us univalent but we delay *m*
  - They show that if the protocol is fault-tolerant there must be a run that leads to the <u>other</u> univalent state
  - And they show that you can deliver *m* in this run without a decision being made
- This proves the result: they show that a bivalent system can be forced to do some work and yet remain in a bivalent state.
  - If this is true once, it is true as often as we like
  - In effect: we can delay decisions indefinitely

But how did they “really” do it?

- Our picture just gives the basic idea
- Their proof actually proves that there is a way to force the execution to follow this tortured path
- But the result is very theoretical…
  - … to much so for us in CS5412
  - So we’ll skip the real details

Intuition behind this result?

- Think of a real system trying to agree on something in which process p plays a key role
- But the system is fault-tolerant: if p crashes it adapts and moves on
- Their proof "tricks" the system into thinking p failed
  - Then they allow p to resume execution, but make the system believe that perhaps q has failed
  - The original protocol can only tolerate1 failure, not 2, so it needs to somehow let p rejoin in order to achieve progress
- This takes time… and no real progress occurs

But what did “impossibility” mean?

- In formal proofs, an algorithm is totally correct if
  - It computes the right thing
  - And it *always* terminates
- When we say something is possible, we mean “there is a totally correct algorithm” solving the problem
- FLP proves that any fault-tolerant algorithm solving consensus has runs that never terminate
  - These runs are <u>extremely</u> unlikely (“probability zero”)
  - Yet they imply that we can’t find a totally correct solution
  - And so “consensus is impossible” (“not always possible”)

How did they pull this off?

- A very clever adversarial attack
  - They assume they have perfect control over which messages the system delivers, and when
  - They can pick the exact state in which a message arrives in the protocol
- They use this ultra-precise control to force the protocol to loop in the manner we've described
- In practice, no adversary ever has this much control

In the real world?

- The FLP scenario "could happen"
  - After all, it is a valid scenario.
  - ... And any valid scenario can happen

- But step by step they take actions that are incredibly unlikely. For many to happen in a row is just impossible in practice
  - A "probability zero" sequence of events
  - Yet in a temporal logic sense, FLP shows that if we can prove correctness for a consensus protocol, we'll be unable to prove it live in a realistic network setting, like a cloud system

So…

- Fault-tolerant consensus is…
  - Definitely possible (not even all that hard). Just vote!
  - And we can prove protocols of this kind correct.

- But we can't prove that they will terminate
  - If our goal is just a probability-one guarantee, we actually *can* offer a proof of progress
  - But in temporal logic settings we want perfect guarantees and we can't achieve that goal

Recap

- We have an asynchronous model with crash failures
  - A bit like the real world!
- In this model we know how to do some things
  - Tracking "happens before" & making a consistent snapshot
  - Later we'll find ways to do ordered multicast and implement replicated data and even solve consensus
- But now we also know that there will always be scenarios in which our solutions can't make progress
  - Often can engineer system to make them extremely unlikely
  - Impossibility doesn't mean these solutions are wrong – only that they live within this limit

Tougher failure models

- We’ve focused on crash failures
  - In the synchronous model these look like a “farewell cruel world” message
  - Some call it the “failstop model”. A faulty process is viewed as first saying goodbye, then crashing
- What about tougher kinds of failures?
  - Corrupted messages
  - Processes that don’t follow the algorithm
  - Malicious processes out to cause havoc?

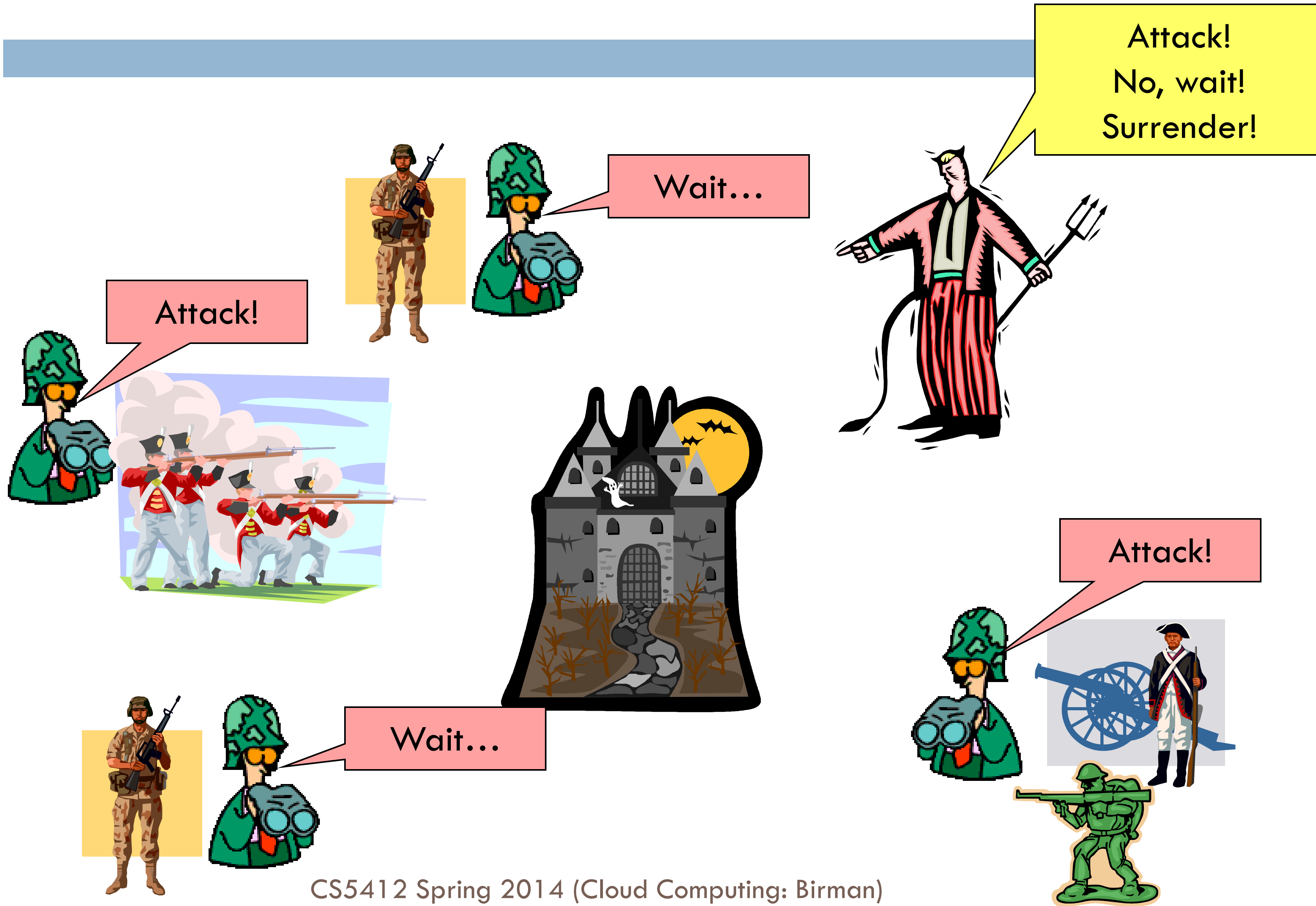
Here the situation is much harder

- Generally we need at least 3f+1 processes in a system to tolerate f Byzantine failures
  - For example, to tolerate 1 failure we need 4 or more processes
- We also need f+1 "rounds"
- Let's see why this happens

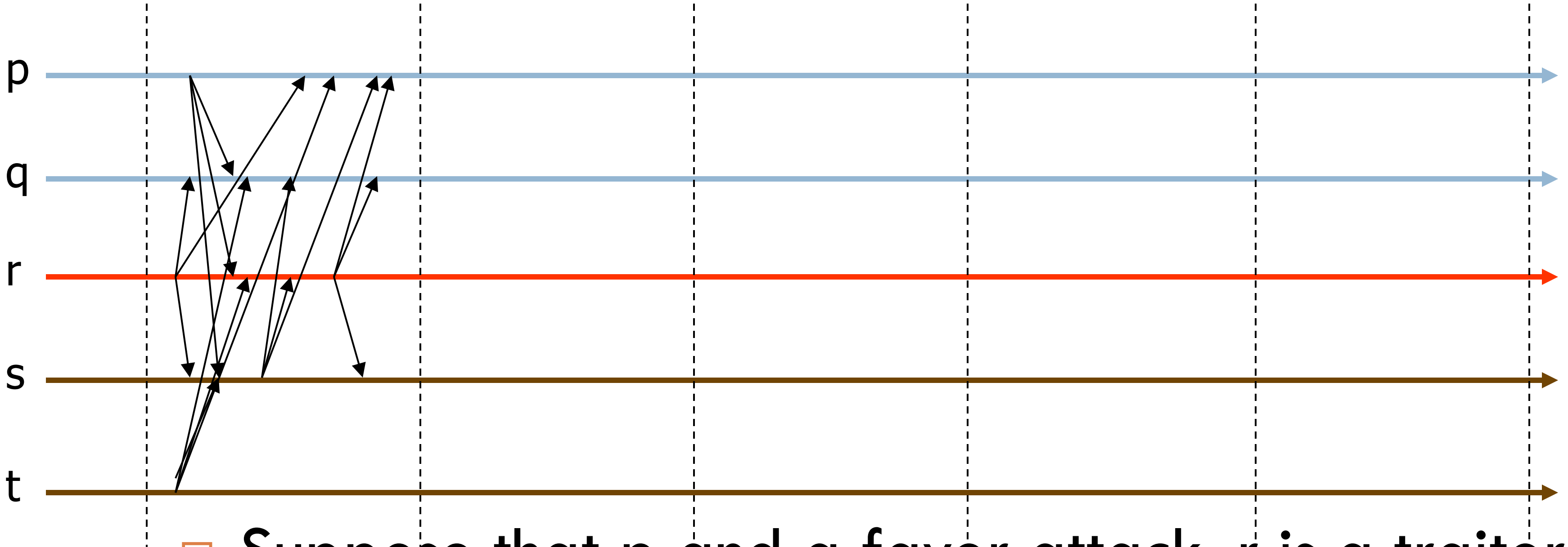
Byzantine scenario

- Generals (N of them) surround a city
  - They communicate by courier
- Each has an opinion: “attack” or “wait”
  - In fact, an attack would succeed: the city will fall.
  - Waiting will succeed too: the city will surrender.
  - But if some attack and some wait, disaster ensues
- Some Generals (f of them) are traitors… it doesn’t matter if they attack or wait, but we must prevent them from disrupting the battle
  - Traitor can’t forge messages from other Generals

Byzantine scenario

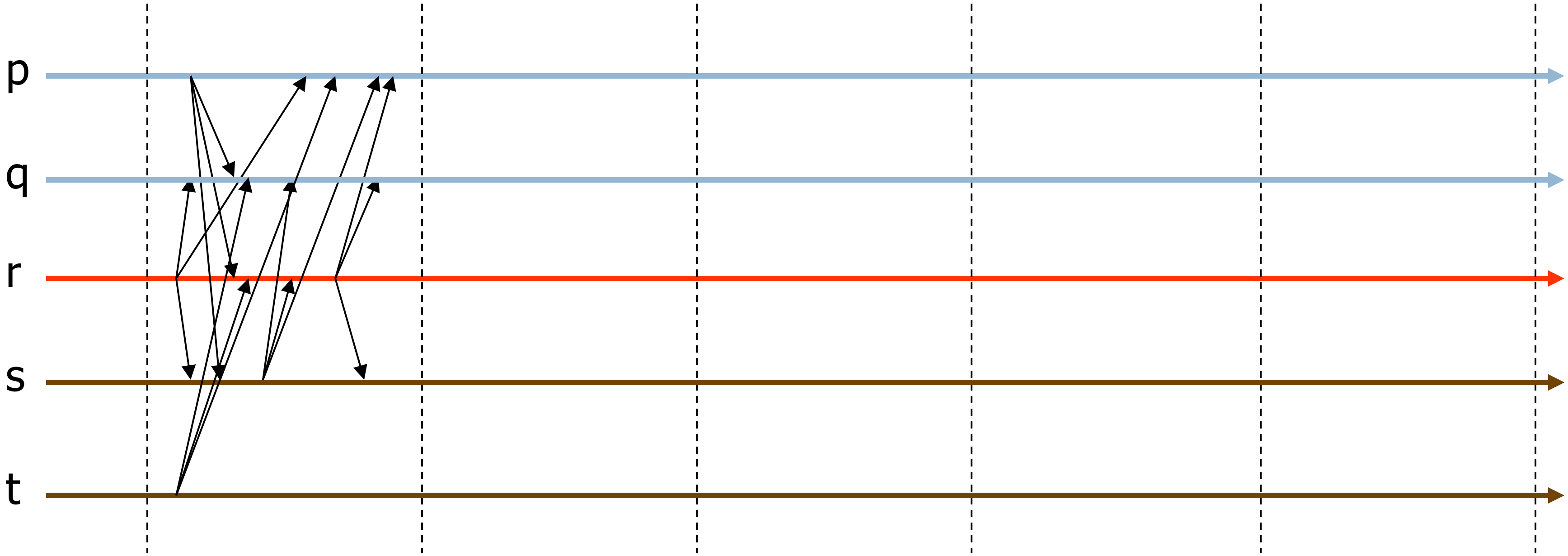


A timeline perspective



- Suppose that p and q favor attack, r is a traitor and s and t favor waiting… assume that in a tie vote, we attack

A timeline perspective



- After first round collected votes are:
  - {attack, attack, wait, wait, traitor's-vote}

What can the traitor do?

- Add a legitimate vote of "attack"
  - Anyone with 3 votes to attack knows the outcome
- Add a legitimate vote of "wait"
  - Vote now favors "wait"
- Or send different votes to different folks
- Or don't send a vote, at all, to some

Outcomes?

- Traitor simply votes:
  - Either all see {a,a,a,w,w}
  - Or all see {a,a,w,w,w}
- Traitor double-votes
  - Some see {a,a,a,w,w} and some {a,a,w,w,w}
- Traitor withholds some vote(s)
  - Some see {a,a,w,w}, perhaps others see {a,a,a,w,w,} and still others see {a,a,w,w,w}
- Notice that traitor can't manipulate votes of loyal Generals!

What can we do?

- Clearly we can't decide yet; some loyal Generals might have contradictory data
  - In fact if anyone has 3 votes to attack, they can already "decide".
  - Similarly, anyone with just 4 votes can decide
  - But with 3 votes to "wait" a General isn't sure (one could be a traitor…)
- So: in round 2, each sends out "witness" messages: here's what I saw in round 1
  - General Smith send me: "attack$_{\text{(signed) Smith}}$"

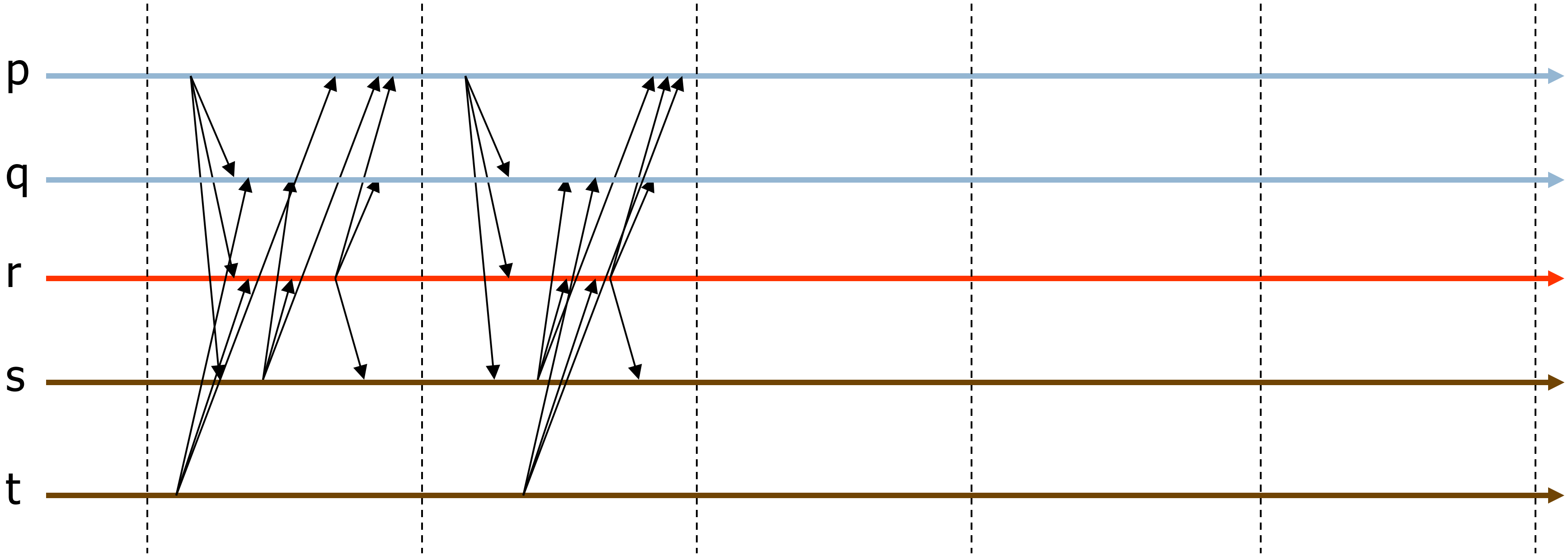
Digital signatures

- These require a cryptographic system
  - For example, RSA
  - Each player has a secret (private) key K^{-1} and a public key K.
    - She can publish her public key
  - RSA gives us a single "encrypt" function:
    - $Encrypt(Encrypt(M,K),K^{-1}) = Encrypt(Encrypt(M,K^{-1}),K) = M$
    - Encrypt a hash of the message to "sign" it

With such a system

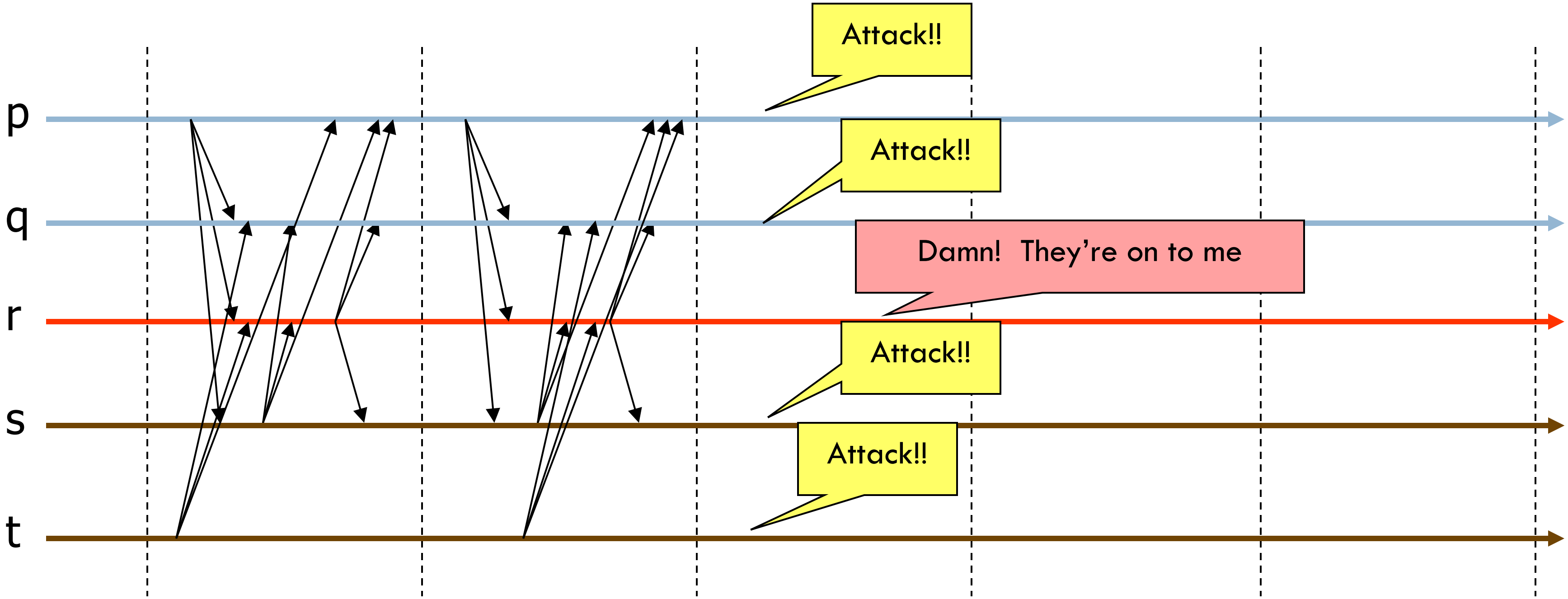
- A can send a message to B that only A could have sent
  - A just encrypts the body with her private key
- … or one that only B can read
  - A encrypts it with B’s public key
- Or can sign it as proof she sent it
  - B can recompute the signature and decrypt A’s hashed signature to see if they match
- These capabilities limit what our traitor can do: he can’t forge or modify a message

A timeline perspective



- In second round if the traitor didn't behave identically for all Generals, we can weed out his faulty votes

A timeline perspective



- We attack!

Traitor is stymied

- Our loyal generals can deduce that the decision was to attack
- Traitor can't disrupt this…
  - Either forced to vote legitimately, or is caught
  - But costs were steep!
    - $(f+1)*n^2$,messages!
    - Rounds can also be slow….
  - "Early stopping" protocols: min(t+2, f+1) rounds; t is true number of faults

Recent work with Byzantine model

- Focus is typically on using it to secure particularly sensitive, ultra-critical services
  - For example the “certification authority” that hands out keys in a domain
  - Or a database maintaining top-secret data
- Researchers have suggested that for such purposes, a “Byzantine Quorum” approach can work well
- They are implementing this in real systems by simulating rounds using various tricks

Byzantine Quorums

- Arrange servers into a $\sqrt{n} \times \sqrt{n}$ array
  - Idea is that any row or column is a quorum
  - Then use Byzantine Agreement to access that quorum, doing a read or a write
- Separately, Castro and Liskov have tackled a related problem, using BA to secure a file server
  - By keeping BA out of the critical path, can avoid most of the delay BA normally imposes

Split secrets

- In fact BA algorithms are just the tip of a broader "coding theory" iceberg
- One exciting idea is called a "split secret"
  - Idea is to spread a secret among *n* servers so that any *k* can reconstruct the secret, but no individual actually has all the bits
  - Protocol lets the client obtain the "shares" without the servers seeing one-another's messages
  - The servers keep but can't read the secret!
- Question: In what ways is this better than just encrypting a secret?

How split secrets work

- They build on a famous result
  - With k+1 distinct points you can uniquely identify an order-k polynomial
    - i.e 2 points determine a line
    - 3 points determine a unique quadratic
  - The polynomial is the "secret"
  - And the servers themselves have the points – the "shares"
  - With coding theory the shares are made just redundant enough to overcome n-k faults

Byzantine Broadcast (BB)

- Many classical research results use Byzantine Agreement to implement a form of fault-tolerant multicast
  - To send a message I initiate "agreement" on that message
  - We end up agreeing on content and ordering w.r.t. other messages
- Used as a primitive in many published papers

Pros and cons to BB

- On the positive side, the primitive is very powerful
  - For example this is the core of the Castro and Liskov technique
- But on the negative side, BB is slow
  - We'll see ways of doing fault-tolerant multicast that run at 150,000 small messages per second
  - BB: more like 5 or 10 per second
- The right choice for infrequent, very sensitive actions… but wrong if performance matters

Take-aways?

- Fault-tolerance matters in many systems
  - But we need to agree on what a "fault" is
  - Extreme models lead to high costs!
- Common to reduce fault-tolerance to some form of data or "state" replication
  - In this case fault-tolerance is often provided by some form of broadcast
  - Mechanism for *detecting* faults is also important in many systems.
    - Timeout is common… but can behave inconsistently
    - "View change" notification is used in some systems. They typically implement a fault agreement protocol.