CS514: Intermediate Course in Operating Systems

Professor Ken Birman
Vivek Vishnumurthy: TA

Fault tolerance

- We've been skirting the issue of fault-tolerant distributed computing
  - Fault-tolerance motivates us to use gossip protocols and similar mechanisms
    - Although scalability was also a motivation
  - But in general, what does it mean for a system to "tolerate" failures?
- Today: focus on failure models

Failure models

- Issues related to failures
  - How do systems "fail?"
  - Given a category of failures, are there limits to what can we do about it?
- Today explore this issue
  - Real world studies of failure rates
  - Experience with some big projects that failed
  - Formal models of failure (crash, fail-stop, Byzantine)
  - A famous (but confusing) impossibility result

Who needs failure "models"?

- The problem is that processes can fail in so many ways
  - Hardware failures are rare, but they happen
  - Software bugs can cause a program to malfunction by crashing, corrupting data, or just failing to "do its job"
  - Intruders might inject some form of failure to disrupt or compromise a system
  - A failure *detector* could malfunction, signaling a failure even though nothing is wrong

Bohrbugs and Heisenbugs

- A categorization due to Bruce Lindsey
  - Bohrbugs are dull, boring, debuggable bugs
    - They happen every time you run the program and are easy to localize and fix using modern development tools
    - If "purify" won't find it… try binary search
  - Heisenbugs are hard to pin down
    - Often associated with threading or interrupts
    - Frequently a data structure is damaged but this is only noticed much later
    - Hence hard to reproduce and so hard to fix
    - In mature programs, Heisenbugs dominate

Clean-room development

- Idea is that to write code
  - First, the team develops a good specification and refines it to modules
  - A primary coding group implements them
  - Then the whole group participates in code review
  - Then the primary group develops a comprehensive test suite and runs it
  - Finally passes off to a Q/A group that redoes these last stages (code review, testing)
  - Later, upgrades require same form of Q/A!

Reality?

- Depends very much on the language
  - With Java and C# we get strong type checking and powerful tools to detect many kinds of mistakes
  - Also clean abstraction boundaries
- But with C++ and C and Fortran, we lack such tools
- The methodology tends to require good tools

Why do systems fail?

- Many studies of this issue suggest that
  - Incorrect specifications (e.g. the program just doesn't "work" in the first place)
  - Lingering Heisenbugs, often papered-over
  - Administrative errors
  - Unintended side-effects of upgrades and bug fixes
- … are dominant causes of failures.

What can we do about it?

- Better programming languages, approaches and tools can help
  - For example shift from C to Java and C# has been hugely beneficial
- But we should anticipate that large systems will exhibit problems
- Failures are a side-effect of using technology to solve complex problems!

Who needs failure "models"?

- Role of a failure model
  - Lets us reduce fault-tolerance to a mathematical question
    - In model M, can problem P be solved?
    - How costly is it to do so?
    - What are the best solutions?
    - What tradeoffs arise?
  - And clarifies what we are saying
    - Lacking a model, confusion is common

Categories of failures

- Crash faults, message loss
  - These are common in real systems
  - Crash failures: process simply stops, and does nothing wrong that would be externally visible before it stops
- These faults can't be directly detected

Categories of failures

- Fail-stop failures
  - These require system support
  - Idea is that the process fails by crashing, and the system notifies anyone who was talking to it
  - With fail-stop failures we can overcome message loss by just resending packets, which must be uniquely numbered
  - Easy to work with… but rarely supported

Categories of failures

- Non-malicious Byzantine failures
  - This is the best way to understand many kinds of corruption and buggy behaviors
  - Program can do pretty much anything, including sending corrupted messages
  - But it doesn’t do so with the intention of screwing up our protocols
- Unfortunately, a pretty common mode of failure

Categories of failure

- Malicious, true Byzantine, failures
  - Model is of an attacker who has studied the system and wants to break it
  - She can corrupt or replay messages, intercept them at will, compromise programs and substitute hacked versions
- This is a worst-case scenario mindset
  - In practice, doesn’t actually happen
  - Very costly to defend against; typically used in very limited ways (e.g. key mgt. server)

Recall: Two kinds of models

- We tend to work within two models
  - **Asynchronous model** makes no assumptions about time
    - Processes have no clocks, will wait indefinitely for messages, could run arbitrarily fast/slow
    - Distributed computing at an “eons” timescale
  - **Synchronous model** assumes a lock-step execution in which processes share a clock

Failures in the asynchronous model

- Network is assumed to be reliable
- But processes can fail
  - A failed process “crashes:” it stops doing anything
  - Notice that in this model, a failed process is indistinguishable from a delayed process
  - In fact, the decision that something has failed takes on an arbitrary flavor
    - Suppose that at point *e* in its execution, process *p* decides to treat *q* as faulty….”

What about the synchronous model?

- Here, we also have processes and messages
  - But communication is usually assumed to be reliable: any message sent at time t is delivered by time $t+\delta$
  - Algorithms are often structured into rounds, each lasting some fixed amount of time Δ, giving time for each process to communicate with every other process
  - In this model, a crash failure is easily detected

Neither model is realistic

- Value of the asynchronous model is that it is so stripped down and simple
  - If we can do something “well” in this model we can do at least as well in the real world
  - So we’ll want “best” solutions
- Value of the synchronous model is that it adds a lot of “unrealistic” mechanism
  - If we can’t solve a problem with all this help, we probably can’t solve it in a more realistic setting!
  - So seek impossibility results

Examples of results

- It is common to look at problems like agreeing on an ordering
  - 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

---

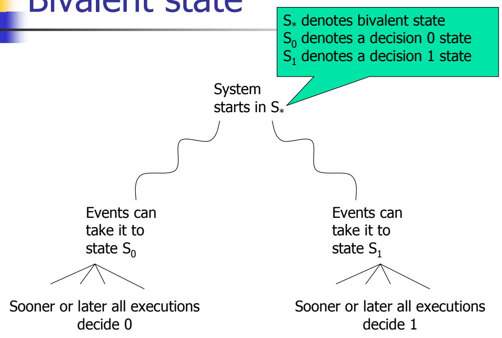
Fischer, Lynch and Patterson

- 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 a system with inputs that are all the same
  - All 0’s must decide 0, all 1’s decides 1
- Now they explore mixtures of inputs and find some initial set of inputs with an uncertain (“bivalent”) outcome
- They focus on this bivalent state

Bivalent state



S_* denotes bivalent state
S_0 denotes a decision 0 state
S_1 denotes a decision 1 state

System starts in S_*

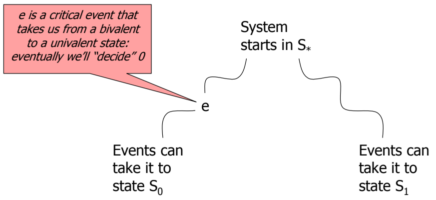
Events can take it to state S_0

Events can take it to state S_1

Sooner or later all executions decide 0

Sooner or later all executions decide 1

Bivalent state



e is a critical event that takes us from a bivalent to a univalent state: eventually we’ll “decide” 0

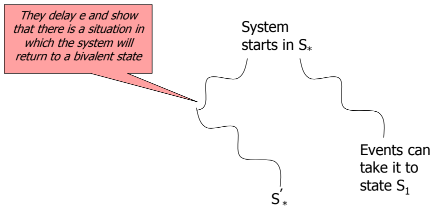
System starts in S_*

e

Events can take it to state S_0

Events can take it to state S_1

Bivalent state



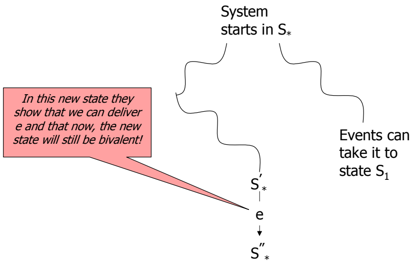
They delay e and show that there is a situation in which the system will return to a bivalent state

System starts in S_*

Events can take it to state S_1

S'_*

Bivalent state



System starts in S_*

In this new state they show that we can deliver e and that now, the new state will still be bivalent!

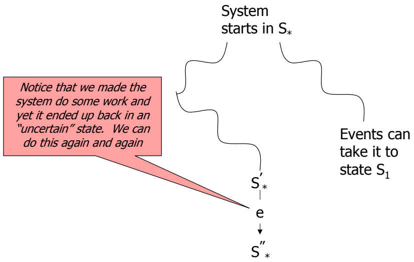
Events can take it to state S_1

S'_*

e

S''_*

Bivalent state



System starts in S_*

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

Events can take it to state S_1

S'_*

e

S''_*

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 CS514
- 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 treating p as if it had failed, but then lets p resume execution and "rejoin"
- 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 extremely unlikely ("probability zero")
  - Yet they imply that we can't find a totally correct solution
  - And so "consensus is impossible" ("not always possible")

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?

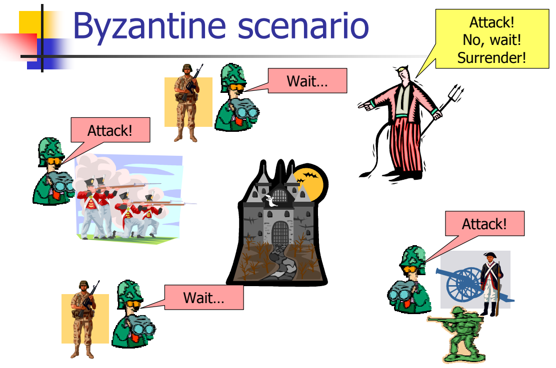
Here the situation is much harder

- 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

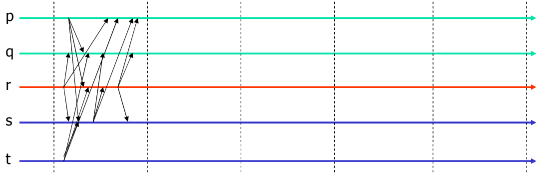
Byzantine scenario

- 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



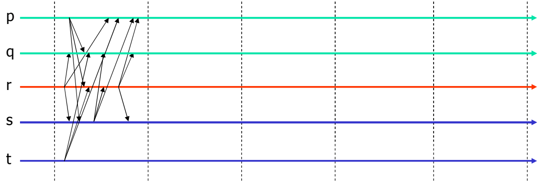
A timeline perspective



- 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$_{\text{(signed) Smith}}$"

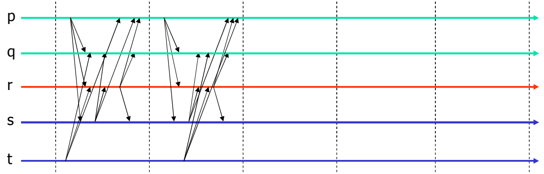
Digital signatures

- 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

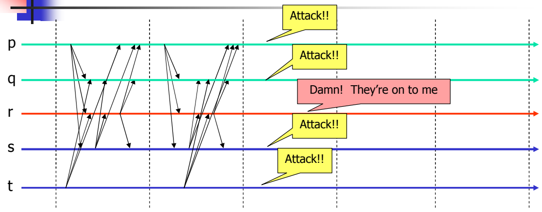
- A can send a message to B that only A could have sent
  - A just encrypts the body with her private key
- … or one that only B can read
  - A encrypts it with B's public key
- Or can sign it as proof she sent it
  - B can recompute the signature and decrypt A's hashed signature to see if they match
- 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.