Failure Detection: Worth it? Masking vs Concealing Faults

Ken Birman

Cornell University. CS5410 Fall 2008.

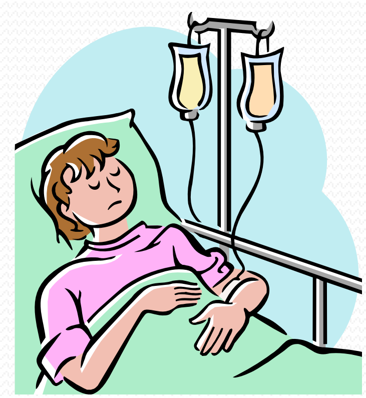
Failure detection… vs Masking

- Failure detection: in some sense, "weakest"
  - Assumes that failures are rare and localized
  - Develops a mechanism to detect faults with low rates of false positives (mistakenly calling a healthy node "faulty")
  - Challenge is to make a sensible "profile" of a faulty node
- Failure masking: "strong"
  - Idea here is to use a group of processes in such a way that as long as the number of faults is below some threshold, progress can still be made
- Self stabilization: "strongest".
  - Masks failures and repairs itself *even after arbitrary faults*

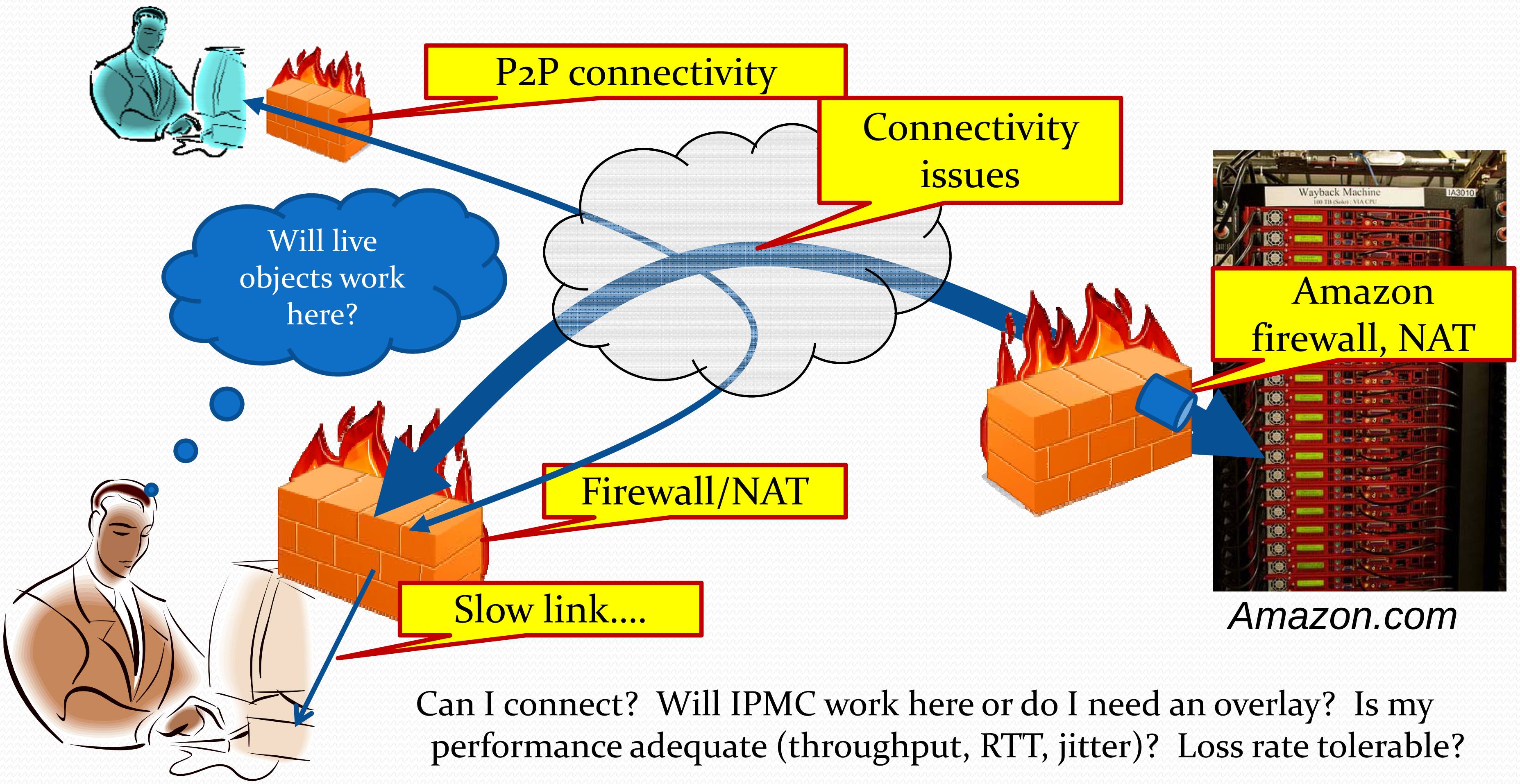
First must decide what you mean by failure

- A system can fail in many ways
  - Crash (or halting) failure: silent, instant, clean
  - Sick: node is somehow damaged
  - Compromise: hacker takes over with malicious intent

- But that isn't all....



Also need to know what needs to work!



Can I connect? Will IPMC work here or do I need an overlay? Is my performance adequate (throughput, RTT, jitter)? Loss rate tolerable?

Missing data

- Today, distributed systems need to run in very challenging and unpredictable environments
- We don't have a standard way to specify the required performance and "quality of service" expectations

- So, each application needs to test the environment in its own, specialized way
  - Especially annoying in systems that have multiple setup options and perhaps could work around an issue
  - For example, multicast: could be via IPMC or via overlay

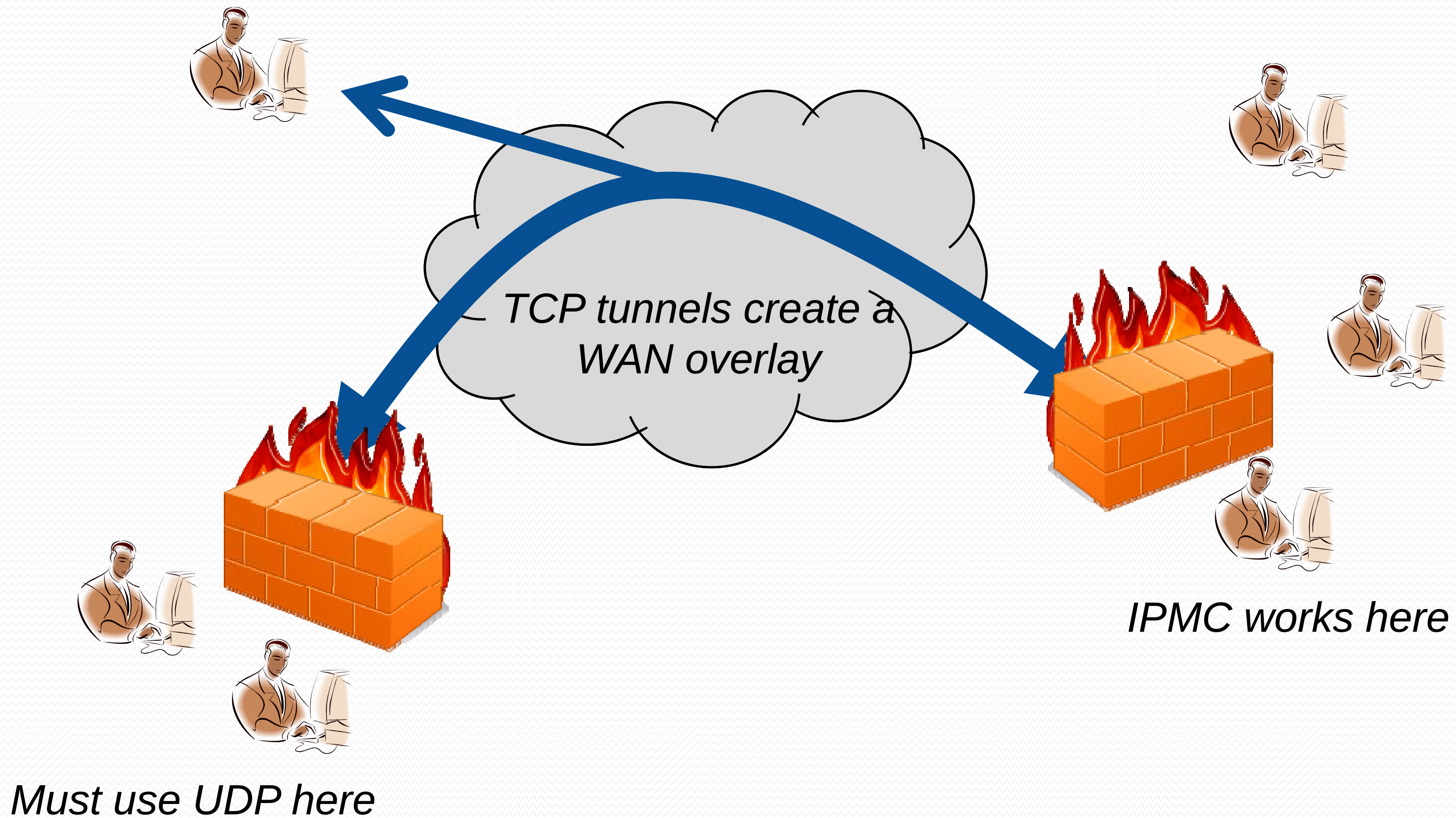
Needed?

- Application comes with a "quality of service contract"
- Presents it to some sort of management service
  - That service studies the contract
  - Maps out the state of the network
  - Concludes: yes, I can implement this
  - Configures the application(s) appropriately
- Later: watches and if conditions evolve, reconfigures the application nodes
- See: Rick Schantz: QuO (Quality of Service for Objects) for more details on how this could work

Example

- Live objects within a corporate LAN
  - End points need multicast… discover that IPMC is working and cheapest option
- Now someone joins from outside firewall
  - System adapts: uses an overlay that runs IPMC within the LAN but tunnels via TCP to the remote node
- Adds a new corporate LAN site that disallows IPMC
  - System adapts again: needs an overlay now…

Example


TCP tunnels create a WAN overlay
IPMC works here
Must use UDP here

Failure is a state transition

- Something that was working no longer works
  - For example, someone joins a group but IPMC can't reach this new member, so he'll experience 100% loss
- If we think of a working application as having a contract with the system (an implicit one), the contract was "violated" by a change of system state
- All of this is very ad-hoc today
  - Mostly we only use timeouts to sense faults

Hidden assumptions

- Failure detectors reflect many kinds of assumptions
  - Healthy behavior assumed to have a simple profile
    - For example, all RPC requests trigger a reply within Xms
  - Typically, minimal "suspicion"
    - If a node sees what seems to be faulty behavior, it reports the problem and others trust it
    - Implicitly: the odds that the report is from a node that was itself faulty are assumed to be very low. If it look like a fault to anyone, then it probably was a fault…
  - For example (and most commonly): timeouts

Timeouts: Pros and Cons

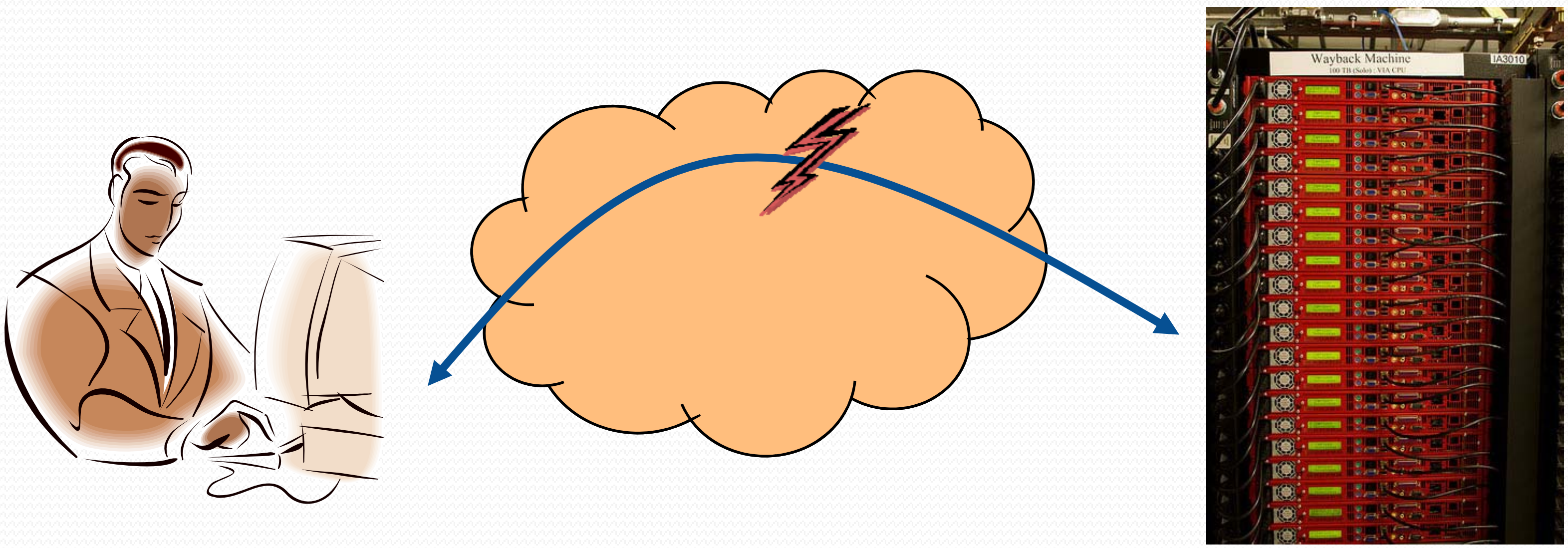
Pros

- Easy to implement
- Already used in TCP
- Many kinds of problems manifest as severe slowdowns (memory leaks, faulty devices…)
- Real failures will usually render a service "silent"

Cons

- Easily fooled
- Vogels: If your neighbor doesn't collect the mail at 1pm like she usually does, would you assume that she has died?
- Vogels: Anyhow, what if a service hangs but low-level pings still work?

A “Vogels scenario” (one of many)



- Network outage causes client to believe server has crashed and server to believe client is down
- Now imagine this happening to thousands of nodes all at once… triggering chaos

Vogels argues for sophistication

- Has been burned by situations in which network problems trigger massive flood of "failure detections"
- Suggests that we should make more use of indirect information such as
  - Health of the routers and network infrastructure
  - If the remote O/S is still alive, can check its management information base
  - Could also require a "vote" within some group that all talk to the same service – if a majority agree that the service is faulty, odds that it is faulty are way higher

Other side of the picture

- Implicit in Vogels' perspective is view that failure is a real thing, an "event"
  - Suppose my application is healthy but my machine starts to thrash because of some other problem
  - Is my application "alive" or "faulty"?
- In a data center, normally, failure is a cheap thing to handle.
- Perspective suggests that Vogels is
  - Right in his worries about the data center-wide scenario
  - But too conservative in normal case

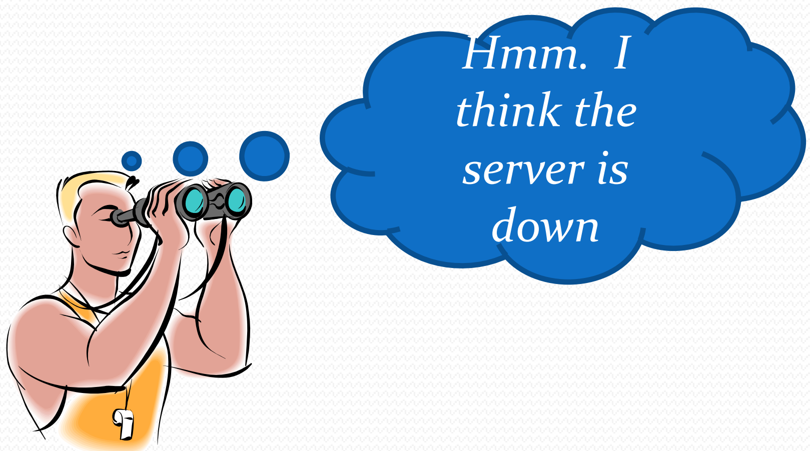
Other side of the picture

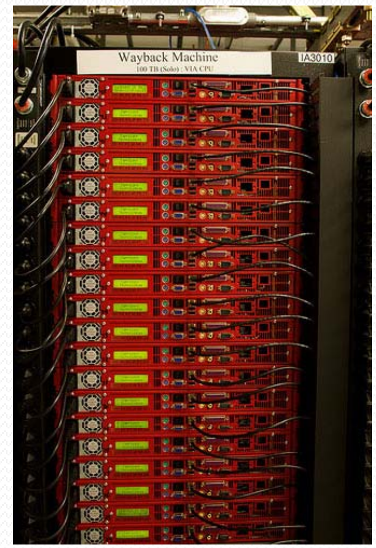
- Imagine a buggy network application
  - Its low-level windowed acknowledgement layer is working well, and low level communication is fine
  - But at the higher level, some thread took a lock but now is wedged and will never resume progress
- That application may respond to "are you ok?" with "yes, I'm absolutely fine".... Yet is actually dead!
  - Suggests that applications should be more self-checking
  - But this makes them more complex... self-checking code could be buggy too! (Indeed, certainly is)

Recall lessons from eBay, MSFT

- Design with *weak consistency models* as much as possible. Just restart things that fail
- Don't keep persistent state in these expendable nodes, use the file system or a database
  - And invest heavily in file system, database reliability
  - Focuses our attention on a specific robustness case…
- If in doubt… restarting a server is cheap!

Recall lessons from eBay, MSFT





- Cases to think about
  - One node thinks three others are down
  - Three nodes think one server is down
  - Lots of nodes think lots of nodes are down

Recall lessons from eBay, MSFT

- If a healthy node is "suspected", watch more closely
- If a watched node seems faulty, reboot it
- If it still misbehaves, reimage it
- If it still has problems, replace the whole node

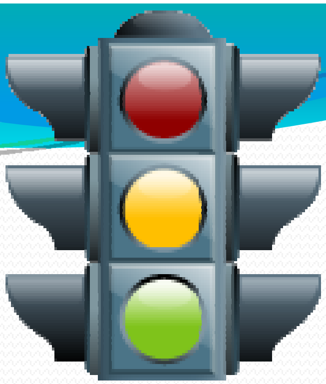
Assumptions?

- For these cloud platforms, restarting is cheap!
  - When state is unimportant, relaunching a node is a very sensible way to fix a problem
  - File system or database will clean up partial actions because we use a transactional interface to talk to it
  - And if we restart the service somewhere else, the network still lets us get to those files or DB records!
- In these systems, we just want to avoid thrashing by somehow triggering a globally chaotic condition with everyone suspecting everyone else

Rule of thumb

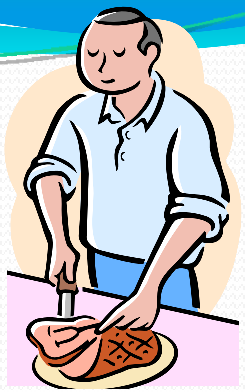
- Suppose all nodes have a "center-wide status" light
  - Green: all systems go
  - Yellow: signs of possible disruptive problem
  - Red: data center is in trouble
- In green mode, could be quick to classify nodes as faulty and quick to restart them
  - Marginal cost should be low
- As mode shifts towards red… become more conservative to reduce risk of a wave of fault detections

Thought question



- How would one design a data-center wide traffic light?
  - Seems like a nice match for gossip
  - Could have every machine maintain local “status”
    - Then use gossip to aggregate into global status
    - Challenge: how to combine values without tracking precisely who contributed to the overall result
      - One option: use a “slicing” algorithm
    - But solutions to exist… and with them our light should be quite robust and responsive
  - Assumes a benign environment

Slicing



- Gossip protocol explored by Gramoli, Vigfussen, Kermarrec, Cornell group
- Basic idea is related to sorting
  - With sorting, we create a rank order and each node learns who is to its left and its right, or even its index
  - With slicing, we rank by attributes into *k* slices for some value of *k* and each node learns its own slice number
- For small or constant *k* can be done in time $\Omega(\log n)$
  - And can be continuously tracked as conditions evolve

Slicing protocol

- Gossip protocol in which, on each round
  - Node selects a random peer (uses random walks)
  - Samples that peer's attribute values

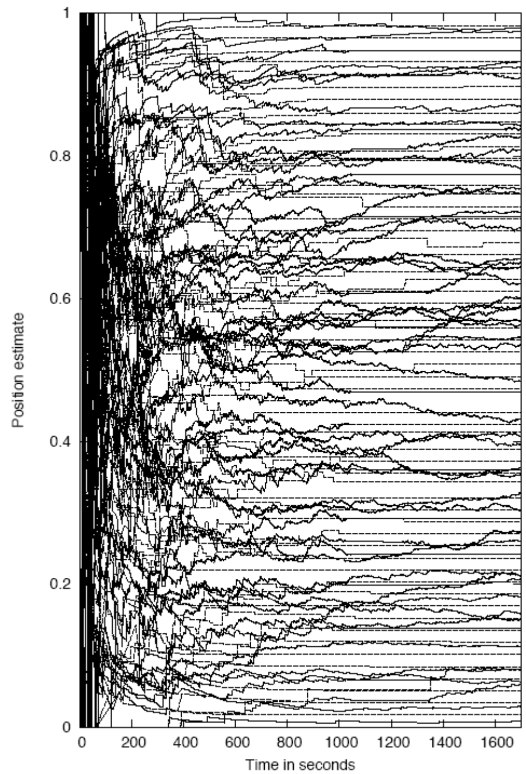
  - Over time, node can estimate where it sits on an ordered list of attribute values with increasing accuracy
- Usually we want k=2 or 3 (small, constant values)
  - Nodes close to boundary tend to need longer to estimate their slice number accurately

Slicing protocol: Experiment

Comparison experiment

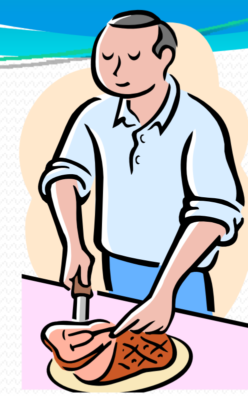
- Two protocols
  - Sliver
  - Ranking: an earlier one
- Major difference: Sliver is careful not to include values from any single node twice
- Also has some minor changes

- Sliver converges quickly… Ranking needs *much* longer



Sliver: dashed lines
Ranking: solid

Slicing



- So, hypothetically, a service could
  - Use a local scheme to have each node form a health estimate for itself and the services it uses
  - Slice on color with, say, k=3, then aggregate to compute statistics. Ideally, *<u>no</u>* yellows or reds in upper 2 slices…

- Aggregation is easy in this case: yes/no per-color
- As yellows pervade system and red creeps to more nodes, we quickly notice it system-wide (log n delay)

Caution about feedback

- Appealing to use system state to tune the detector thresholds used locally
  - If I think the overall system is healthy, I use a fine-grained timeout
  - If the overall system enters yellow mode, I switch to a longer timeout, etc
- But this could easily oscillate… important to include a damping mechanism in any solution!
  - Eg switching back and forth endlessly would be bad
  - But if we always stay in a state for at least a minute…

Reputation

- Monday we discussed reputation monitoring
  - Nodes keep records documenting state (logs)
  - Audit of these logs can produce proofs prove that peers are misbehaving
  - Passing information around lets us react by shunning nodes that end up with a bad reputation

- Reputation is a form of failure detection!
  - Yet it only covers “operational” state: things p actually did relative to q

Reputation has limits

- Suppose q asserts that "p didn't send me a message at time t, so I believe p is down"
  - P could produce a log "showing" that it sent a message
  - But that log only tells us what the application thinks it did (and could also be faked)
- Unless p broadcasts messages to a group of witnesses we have no way to know if p or q is truthful
  - In most settings, broadcasts are too much overhead to be willing to incur... but not always

Leading to "masking"

- Systems that mask failures
  - Assume that faults happen, may even be common
  - Idea is to pay more all the time to ride out failures with no change in performance
- Could be done by monitoring components and quickly restarting them after a crash…
- … or could mean that we form a group, replicate actions and state, and can tolerate failures of some of the group members

Broad schools of thought

- Quorum approaches
  - Group itself is statically defined
    - Nodes don't join and leave dynamically
    - But some members may be down at any particular moment
  - Operations must touch a majority of members
- Membership-based approaches
  - Membership actively managed
  - Operational subset of the nodes collaborate to perform actions with high availability
  - Nodes that fail are dropped and must later rejoin

Down the Quorum road

- Quorum world is a world of
  - Static group membership
  - Write and Read quorums that must overlap
    - For fault-tolerance, $Q_w < n$ hence $Q_r > 1$
  - Advantage: progress even during faults and no need to worry about "detecting" the failures, provided quorum is available.
  - Cost: even a read is slow. Moreover, writes need a 2-phase commit at the end, since when you do the write you don't yet know if you'll reach a quorum of replicas

Down the Quorum road

- Byzantine Agreement is basically a form of quorum fault-tolerance
  - In these schemes, we assume that nodes can crash but can also behave maliciously
  - But we also assume a bound on the number of failures
  - Goal: server as a group must be able to overcome faulty behavior by bounded numbers of its members
- We'll look at modern Byzantine protocols on Nov 24

Micro-reboot

- Byzantine thinking
  - Attacker managed to break into server *i*
  - Now he knows how to get in and will perhaps manage to compromise more servers
- So… reboot servers at some rate, even if nothing seems to be wrong
  - With luck, we repair server *i* before server *j* cracks
  - Called "proactive micro-reboots" (Armondo Fox, Miguel Castro, Fred Schneider, others)

Obfuscation

- Idea here is that if we have a population of nodes running some software, we don't want them to share identical vulnerabilities
- So from the single origin software, why not generate a collection of synthetically diversified versions?
  - Stack randomization
  - Code permutation
  - Deliberately different scheduling orders
  - Renumbered system calls
  - … and the list goes on

An extreme example

- French company (GEC-Alstrom) doing train brakes for TGV was worried about correctness of the code
  - So they used cutting-edge automated proof technology (the so-called *B*-method)
  - But this code must run on a *platform* they don't trust
- Their idea?
  - Take the original code and generate a family of variants
  - Run the modified program (a set of programs)
  - Then external client compares outputs
- *"I tell you three times: It is safe to not apply the brakes!"*

An extreme example

- Separation of service from client becomes a focus
  - Client must check the now-redundant answer
  - Must also make sure parts travel down independent pathways, if you worry about malicious behavior
- Forces thought about the underlying fault model
  - Could be that static messed up memory
  - Or at other extreme, agents working for a terrorist organization modified the processor to run the code incorrectly
  - GEC-Alstrom never really pinned this down to my taste

Byzantine model: pros and cons

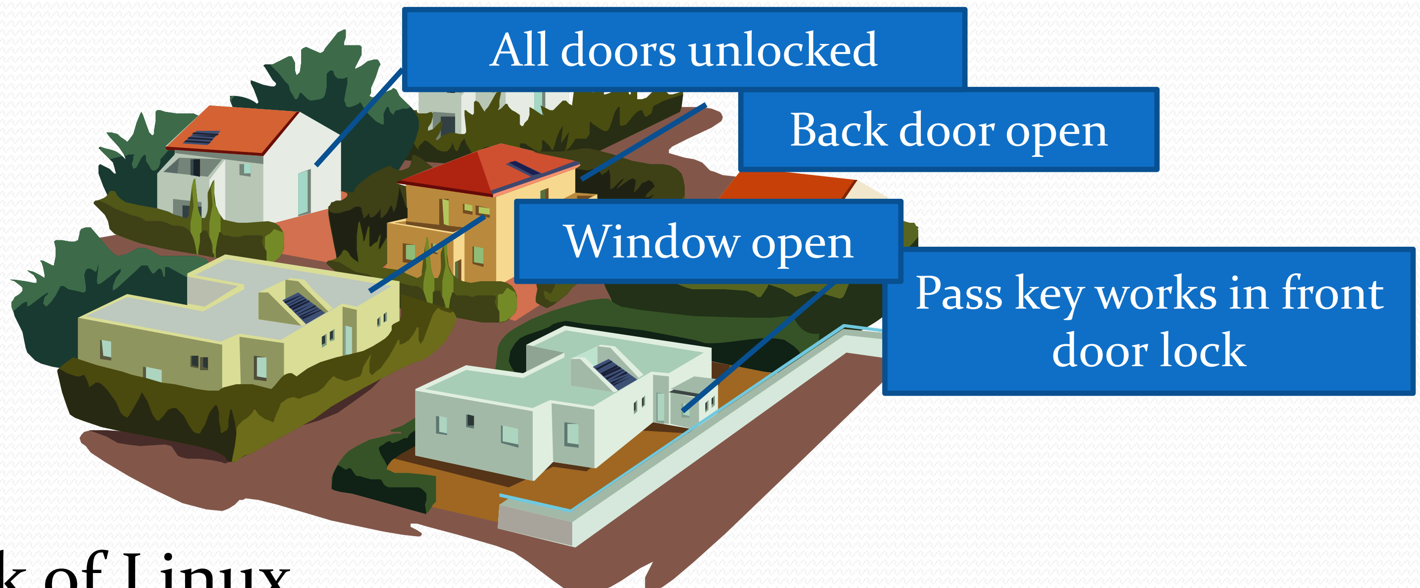
- On the positive side, increasingly practical
  - Computers have become cheap, fast… cost of using 4 machines to simulate one very robust system tolerable
  - Also benefit from wide availability of PKIs: Byzantine protocols are much cheaper if we have signatures
  - If the service manages the crown jewels, much to be said for making that service very robust!
- Recent research has shown that Byzantine services can compete reasonably well with other forms of fault-tolerance (but obviously BFT is still more expensive)

Byzantine model: pros and cons

- On the negative side:
  - The model is quite "synchronous" even if it runs fast, the end-to-end latencies before actions occur can be high
  - The fast numbers are for throughput, not delay
  - Unable to tolerate malfunctioning *client* systems: is this a sensible line to draw in the sand?
    - You pay a fortune to harden your file server…
    - But then allow a compromised client to trash the contents!

NSA perspective

- There are many ways to attack a modern computer
- Think of a town that has very relaxed security



- Now think of Linux,
  Windows, and the apps that run on them…

NSA perspective

- Want to compromise a computer?
  - Today, simple configuration mistakes will often get you in the door
    - Computer may lack patches for well known exploits
    - May use “factory settings” for things like admin passwords
    - Could have inappropriate trust settings within enclave
  - But suppose someone fixes those. This is like locking the front door.
    - What about the back door? The windows? The second floor?
    - In the limit, a chainsaw will go right through the wall

NSA perspective

- Can attack
  - Configuration
  - Known OS vunerabilities
  - Known application vulnerabilities
  - Perhaps even hardware weaknesses, such as firmware that can be remotely reprogrammed
- Viewed this way, not many computers are secure!

- BFT in a service might not make a huge difference

Mapping to our computer system

- Choice is between a "robust" fault model and a less paranoid one, like crash failures
  - Clearly MSFT was advocating a weaker model
- Suppose we go the paranoia route
  - If attacker can't compromise data by attacking a server…
  - … he'll just attack the host operating system
  - … or the client applications
- Where can we draw the line?

All bets off on top

BFT below

Rings of protection

- Model favored by military (multi-level security)
  - Imagine our system as a set of concentric rings
  - Data “only flows in” and inner ones have secrets outer ones can’t access. (But if data can flow in… perhaps viruses can too… so this is a touchy point)
- Current approach
  - External Internet, with ~25 gateways
  - Military network for “most” stuff
  - Special network for sensitive work is physically disconnected from the outside world

The issue isn't just computers

- Today the network itself is an active entity
  - Few web pages have any kind of signature
  - And many platforms scan or even modify inflight pages!
  - Goal is mostly to insert advertising links, but implications can be far more worrying
- Longer term perspective?
  - A world of Javascript and documents that move around
  - Unclear what security model to use in such settings!

Javascript/AJAX

- Creates a whole new kind of distributed “platform”
  - Unclear what it means when something fails in such environments
  - Similar issue seen in P2P applications
    - Nodes p and q download the same thing
    - But will it behave the same way?
- Little is understood about the new world this creates
- And yet we need to know
  - In many critical infrastructure settings, web browsers and webmail interfaces will be ubiquitous!

Vision for the future

- Applications (somehow) represent their needs
  - "I need a multicast solution to connect with my peers"
  - "... and it needs to carry 100kb/s with maximum RTT 25ms and jitter no more than 3ms."
- Some sort of configuration manager tool maps out the options and makes a sensible selection (or perhaps constructs a solution by snapping together some parts, like a WAN tunnel and a local IPMC layer)
- Then monitors status and if something changes, adapts (perhaps telling application to reconfigure)

Vision for future

- Forces us to think in terms of a "dialog" between the application and its environment
  - For example, a multicast streaming system might adjust the frame rate to accommodate the properties of an overlay, so that it won't overrun the network

- And yet we also need to remember all those "cloud computing lessons learned"
  - Consistency: "as weak as possible"
  - Loosely coupled… locally autonomous…. etc

Summary

- Fault tolerance presents us with a challenge
  - Can faults be detected?
  - Or should we try and mask them?
- Masking has some appeal, but the bottom line is that it seems both expensive and somewhat arbitrary
  - A capricious choice to draw that line in the sand…
  - And if the faults aren't well behaved, all bets are off
- Alternatives reflect many assumptions and understanding them is key to using solutions in sensible ways….