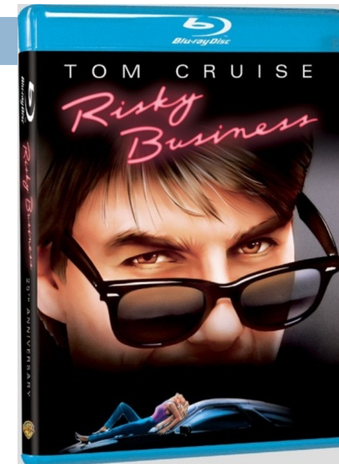
CS5412: THE CLOUD UNDER ATTACK!

Lecture XXIV

Ken Birman

For all its virtues, the cloud is risky!



- Categories of concerns
  - Client platform inadequacies, code download, browser insecurities
  - Internet outages, routing problems, vulnerability to DDoS
  - Cloud platform might be operated by an untrustworthy third party, could shift resources without warning, could abruptly change pricing or go out of business
  - Provider might develop its own scalability or reliability issues
  - Consolidation creates monoculture threats
  - Cloud security model is very narrow and might not cover important usage cases

But the cloud is also good in some ways

- With a private server, DDoS attacks often succeed
  - In contrast, it can be very hard to DDoS a cloud
  - With 100,000 nodes we can shift work and clouds have immense amounts of network bandwidth
  - DDoS "operator" spends money on the attack
  - So... if cloud is able to block the attack, the DDoS-er won't even try

- In fact there have been very few cases of successful DDoS against cloud-hosted services

But the cloud is also good in some ways

- Diversity can compensate for monocultures
- Elasticity represents a unique new technical capability that we can’t replicate in other settings
- Ability to host huge amounts of data, not feasible in a smaller data center, enables us to compute directly on the raw data
- Massive parallelism can benefit if the subtasks are simple and if it isnt hard to assemble the results
- … the list goes on

So the cloud is tempting

- And cheaper, too!

- What's not to love?
  - Imagine that you work for a large company that is healthy and has managed its own story in its own way
  - Now the cloud suddenly offers absolutely unique opportunities that we can't access in any other way
  - Should you recommend that your boss drink the potion?

But how can anyone trust the cloud?

- The cloud seems *so* risky that it makes no sense at all to trust it in any way!

- Yet we seem to trust it in *many* ways

- This puts the fate of your company in the hands of third parties!

The concept of "good enough"

- We've seen that there really isn't any foolproof way to build a computer, put a large, complex program on it, and then run it with confidence
- We also know that with effort, many kinds of systems really start to work very well
- When is a "pretty good" solution good enough?

How they do in avionics

- FAA and NASA have a process that is used for building critical components: things like fly-by-wire control software
  - This process requires very stringent proofs
  - The program must be certified on particular hardware, even specific versions of chips
  - Any change of any kind triggers a recertification task, even sources replacement chips from a new "batch"
- Very costly: a controller 100 lines long may generate 1000 pages of documentation!

How *most* production software is built

- Generally, company develops good specification
- Code is created in teams with code review frequent and much unit testing
- Then code is passed to a “red team” that uses the code, attacks it, tries to find issues
- Cycle continues until adequate assurance is reached and the initial release can take place
- Subsequently must track and fix bugs, repeat Q/A, do periodic patch releases
- Wise to rebuild entire solution every 5 years or so

How was the cloud built?

- There wasn't enough time for proper Q/A
  - So much of the cloud was built in a huge hurry
  - Even today, *race for features* often doesn't leave time for *proper testing*

- Early versions have been rough, insecure, fault-prone
  - Over time, slow improvement
  - Seems to shift a lot of emphasis to patches and upgrades
  - Many cloud systems auto-upgrade frequently

Legacy code

- Not all code fits the "rebuilt periodically" model
  - Many major technologies were important in their day but now live on in isolated settings
  - They work… do something important for some organization… and so nobody touches them
- These legacy systems are often minimally maintained but over time the amount of legacy code can become substantial
- Over time people lose track... big companies often have spaghetti-like structures of old, interdependent components

The parable of Y2K

- Once upon a time many, many systems had dependencies on clocks lacking adequate precision
  - They only kept 2 digits for the years, like a credit card that expires 05/13
  - Thus when we reached 01/00 it looked like time travel 100 years into the past
  - Experiments made it clear that many systems crashed when this happened… and nobody had any idea how to find the "bad apples" in the barrel

So how did things work out?

- Initial cost estimates were terrifying
  - Tens or hundreds of billions of dollars to scan the hundreds of millions of lines of code that do important things
  - Lack of people do even do the work
  - Code in baffling, ancient languages like COBOL
  - Disaster loomed…
- Infosys rode to the rescue!

Infosys in the pre-Y2K period

- A small Indian software company that was known mostly for its work on the Paris Airport luggage transportation system
  - A very complex system, which Infosys was successful building at a fraction the standard cost and with far fewer bugs or delays than France had ever seen
  - Company had a few hundred employees
- Founded in 1981 with $600!

Infosys was an unusual company

- Founders were all very socially pro-active and very concerned about the situation of India’s poor

- Extremely high ethical standard: A decision to never pay bribes or in any way rig the outcome of business decisions

- When many company executives were paying themselves big bonuses, the founders reinvested

1987: A big event

- Infosys got a toehold in the United States when it landed its first US corporate client
  - A company named Data Basics Corporation

- The Infosys “angle”?
  - Hire smart kids from all over India
  - Offer them additional training at a corporate campus in Mysore
  - Form them into a highly qualified workforce

Financial angle?

- In the early days, Infosys was paying highly qualified employees $5,000/year
- In the US highly qualified technology workers were earning $125,000/year in that time period
  - Skill sets weren't so different…

- Today the gap is a little smaller, but not hugely so

How Y2K helped

- Companies like Infosys tackled the Y2K challenge for "pennies on the dollar" relative to estimates
  - A company facing a $50M bill to review all the corporate code base saw it shrink to perhaps $1M
  - And Infosys often finished these tasks early

- …. January 1, 2000 arrived and the world didn't end. Instead the world of outsourcing began!
  - A few minor issues occurred, but nothing horrible

Lessons one learns

- Cheaper isn't necessarily inferior!
  - In fact over time, cheaper but "good enough" wins
  - This is a very important lesson that old companies miss
- Earlier adopters often accept risks
  - … risks that can be managed
  - And those good-enough solutions sometimes catch up later
- Bad stuff (lots of it) lurks deep within the cool new stuff that we all love

Fast forward to 2012

- Today cloud computing has a similar look and feel
  - It works really well for the things we use it to do today
    - How often does an iPhone service malfunction?
      - Pretty often, actually, but not often enough to bother anyone
    - The cloud is fast, scalable, has amazing capabilities, and yes, it has a wide variety of issues
- Is the cloud really worse than what came before it?
  - Given that the cloud evolved from what came earlier, is this even a sensible question?
  - When has any technology ever been "assured"?

Life with technology is about tradeoffs

- Clearly, we err if we use a technology in a dangerous or inappropriate way
  - Liability laws need to be improved: they let software companies escape pretty much all responsibility
  - Yet gross negligence is still a threat to those who build things that will play critical roles and yet fail to take adequate steps to achieve assurance

Another parable: Real-time multicast

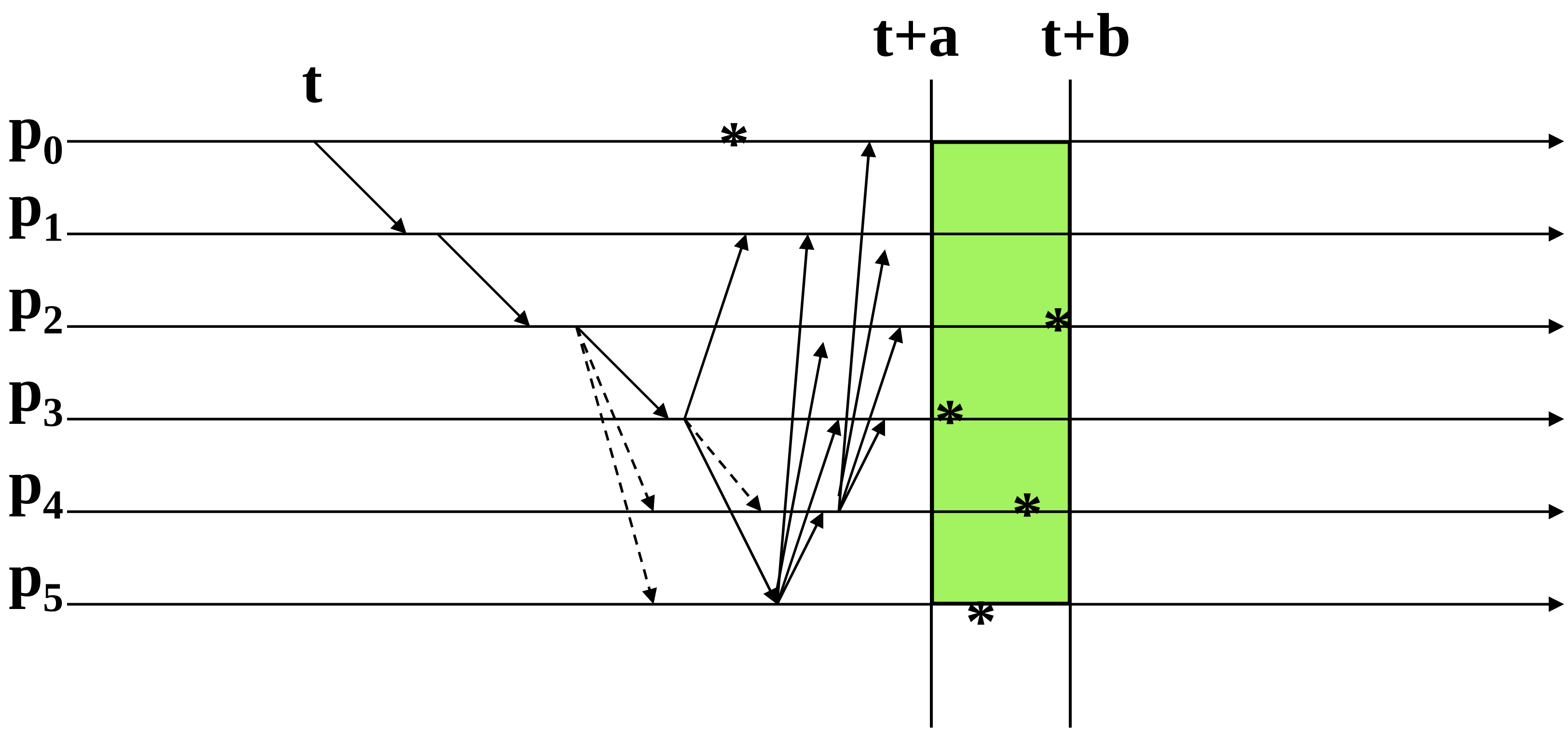
- The community that builds real-time systems favors proofs that the system is *guaranteed* to satisfy its timing bounds and objectives

- The community that does things like data replication in the cloud tends to favor speed
  - We want the system to be fast
  - Guarantees are great unless they slow the system down

Can a guarantee slow a system down?

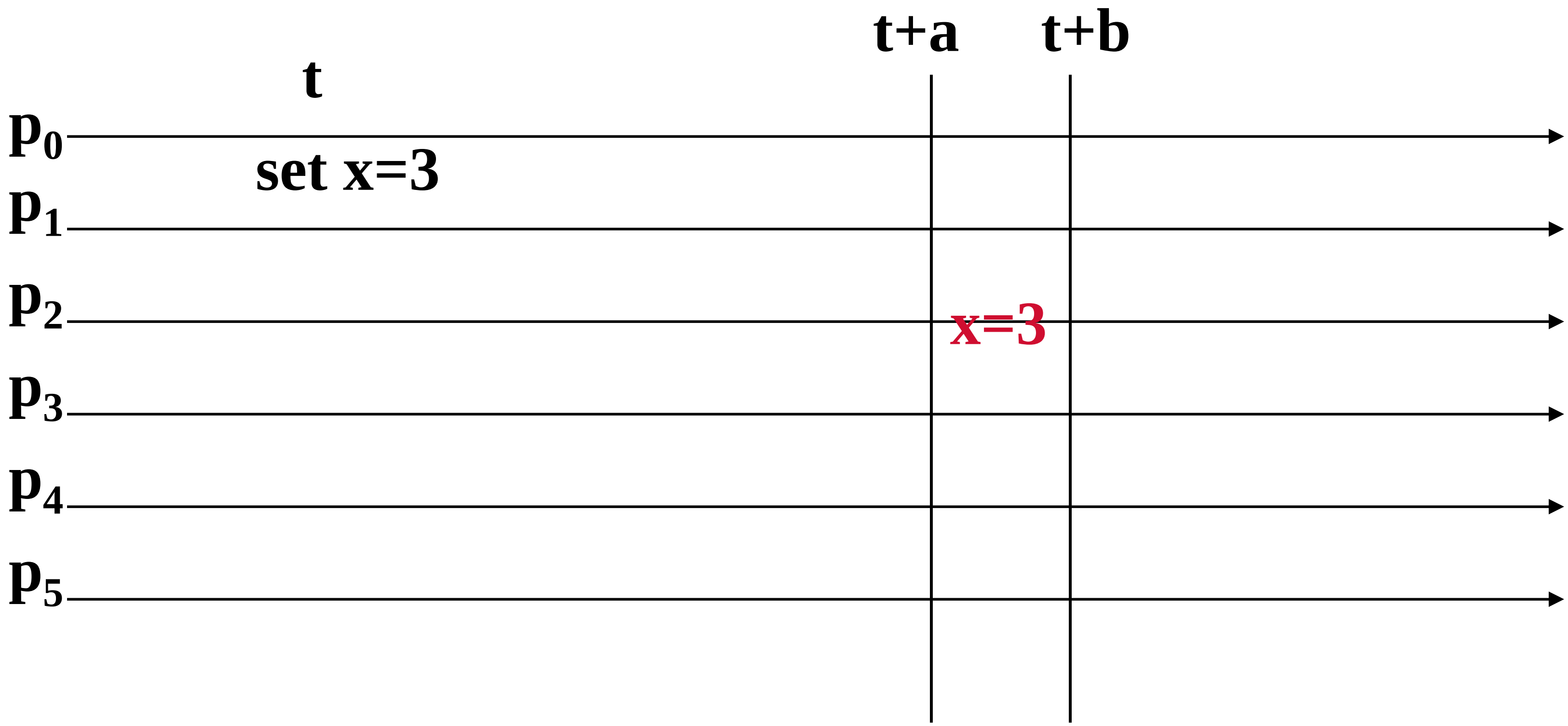
- Suppose we want to implement broadcast protocols that make direct use of temporal information
- Examples:
  - Broadcast that is delivered at same time by all correct processes (plus or minus the clock skew)
  - Distributed shared memory that is updated within a known maximum delay
  - Group of processes that can perform periodic actions

A real-time broadcast



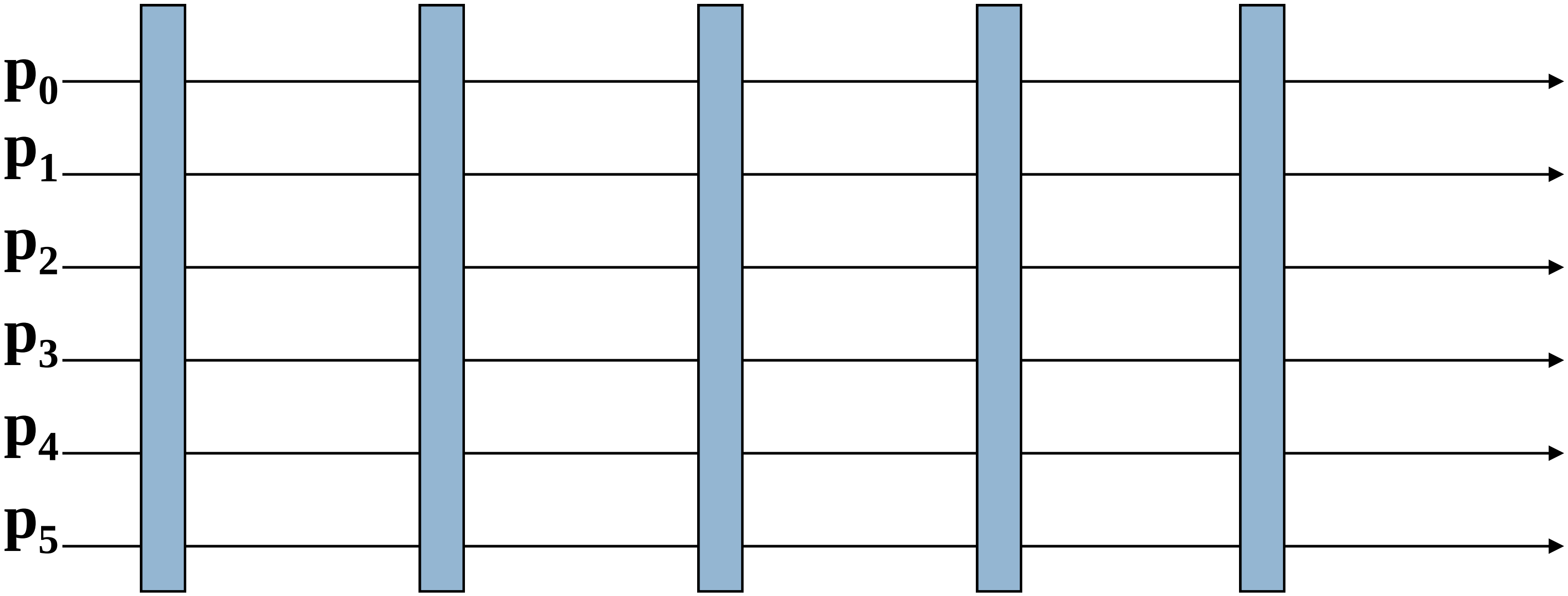
Message is sent at time *t* by p_0. Later both p_0 and p_1 fail. But message is still delivered atomically, after a bounded delay, and within a bounded interval of time (at non-faulty processes)

A real-time distributed shared memory



At time *t* p_0 updates a variable in a distributed shared memory. All correct processes observe the new value after a bounded delay, and within a bounded interval of time.

Periodic process group: Marzullo



Periodically, all members of a group take some action.
Idea is to accomplish this with minimal communication

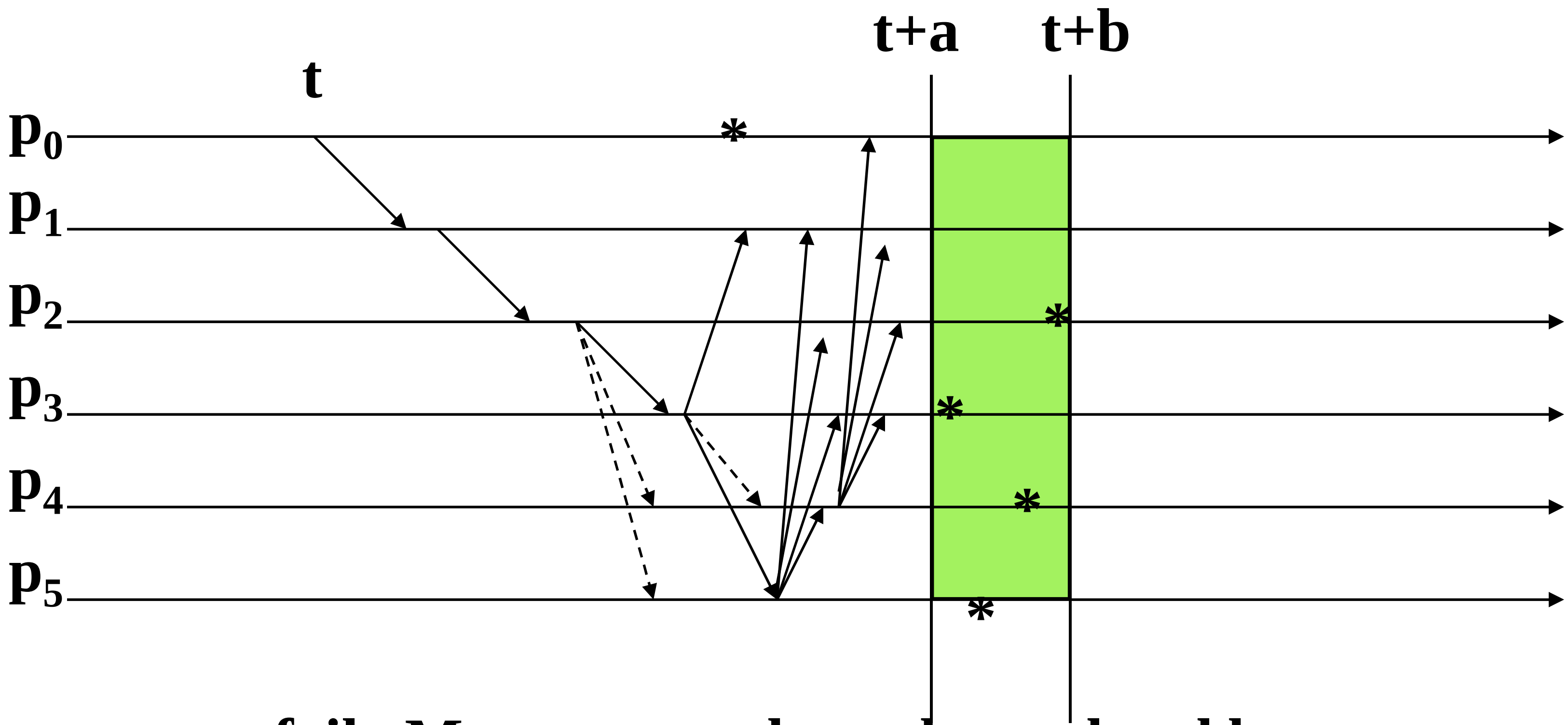
The CASD protocols

- Also known as the "Δ -T" protocols
- Developed by Cristian and others at IBM, was intended for use in the (ultimately, failed) FAA project
- Goal is to implement a timed atomic broadcast tolerant of Byzantine failures

Basic idea of the CASD protocols

- Assumes use of clock synchronization
- Sender timestamps message
- Recipients forward the message using a flooding technique (each echos the message to others)
- Wait until all correct processors have a copy, then deliver in unison (up to limits of the clock skew)

CASD picture



p_0, p_1 **fail. Messages are lost when echoed by** p_2, p_3

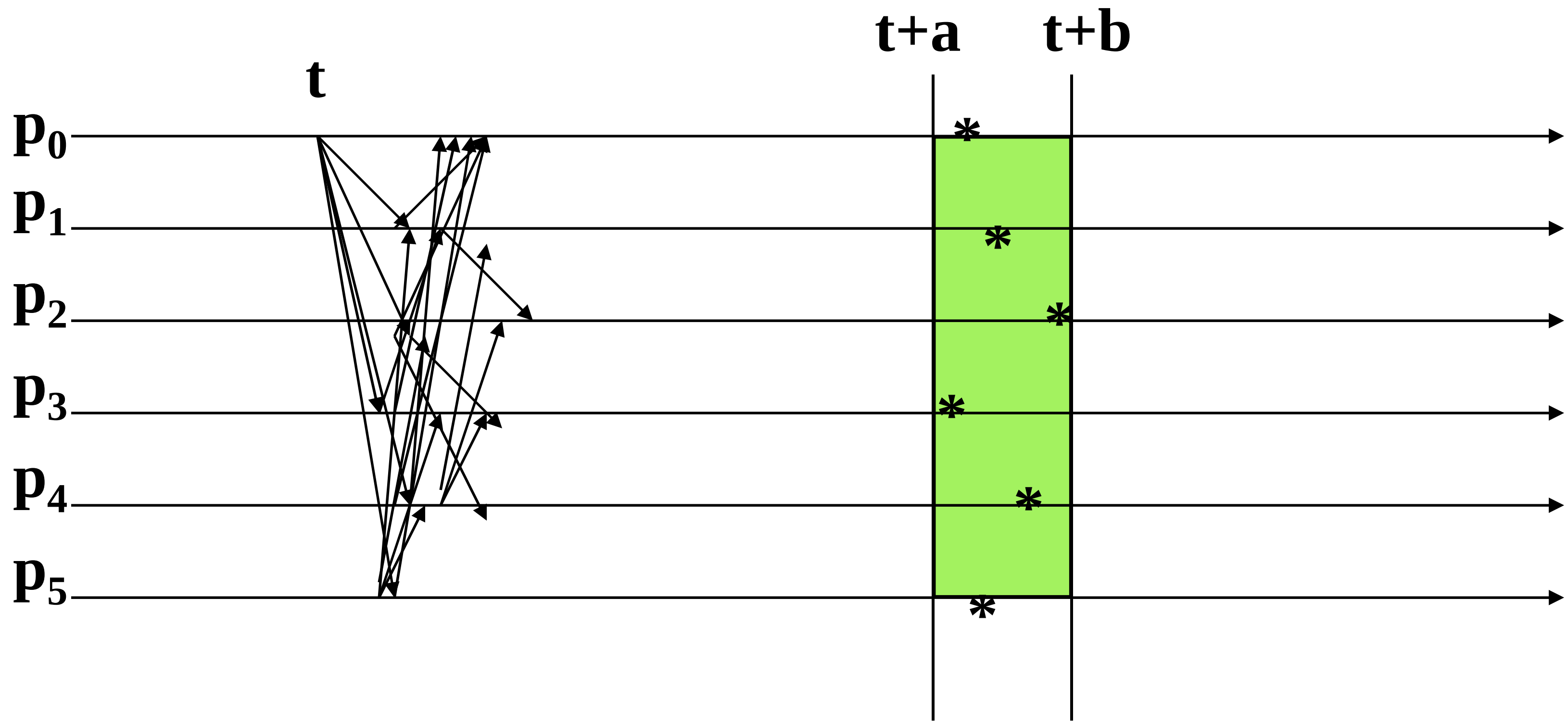
Idea of CASD

- Assume known limits on number of processes that fail during protocol, number of messages lost
- Using these and the temporal assumptions, deduce worst-case scenario
- Now now that if we wait long enough, all (or no) correct process will have the message
- Then schedule delivery using original time plus a delay computed from the worst-case assumptions

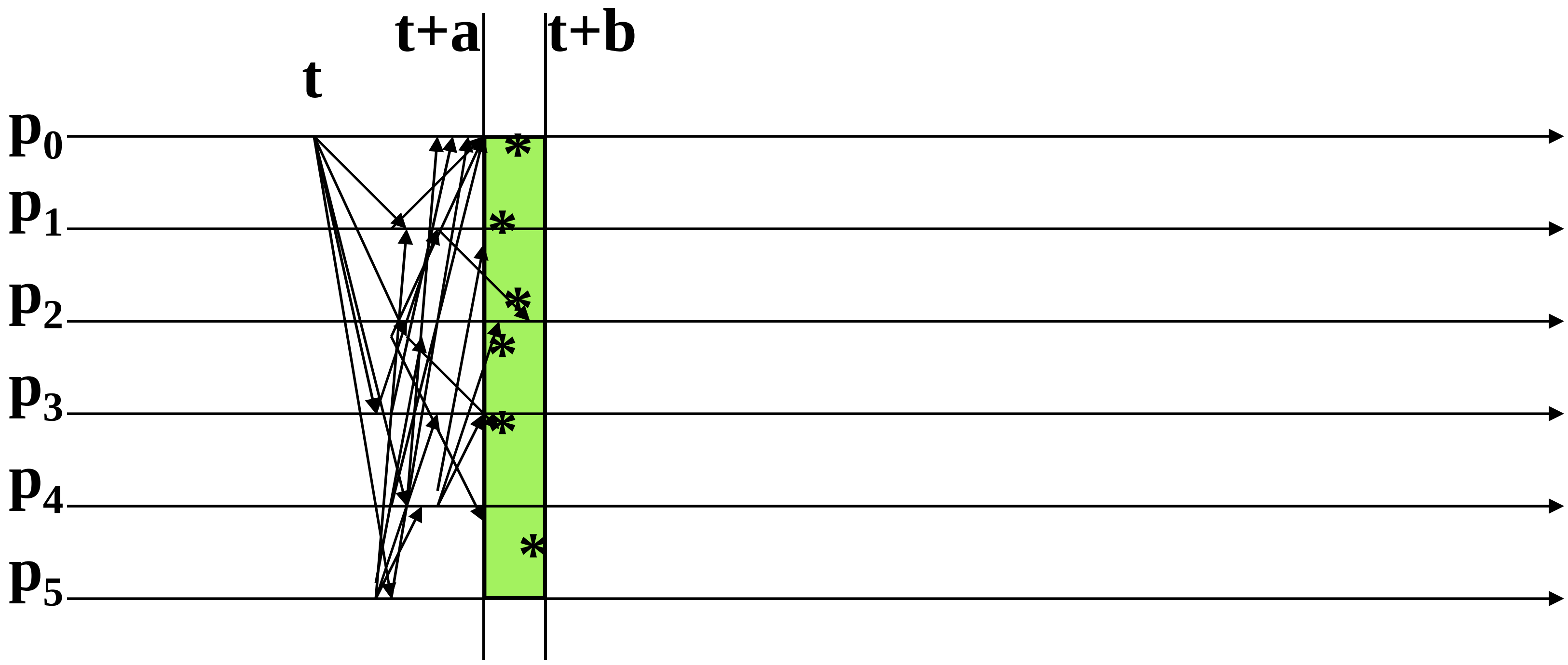
The problems with CASD

- In the usual case, nothing goes wrong, hence the delay can be very conservative
- Even if things do go wrong, is it right to assume that if a message needs between 0 and δms to make one hope, it needs [0,n* δ] to make n hops?
- How realistic is it to bound the number of failures expected during a run?

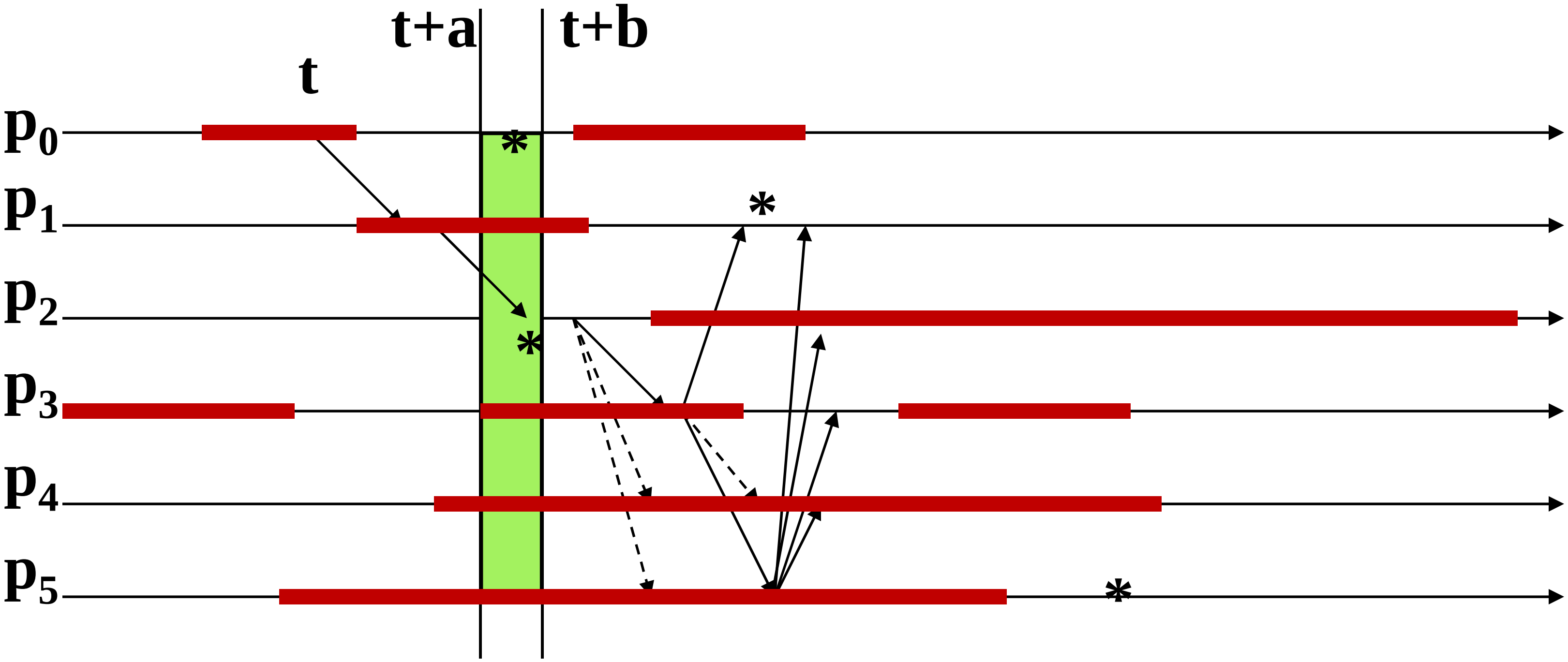
CASD in a more typical run



… leading developers to employ more aggressive parameter settings



CASD with over-aggressive paramter settings starts to "malfunction"



all processes look "incorrect" (red) from time to time

CASD "mile high"

- When run "slowly" protocol is like a real-time version of abcast
- When run "quickly" protocol starts to give probabilistic behavior:
  - If I am correct (and there is no way to know!) then I am guaranteed the properties of the protocol, but if not, I may deliver the wrong messages

How to repair CASD in this case?

- Gopal and Toueg developed an extension, but it slows the basic CASD protocol down, so it wouldn't be useful in the case where we want speed and also real-time guarantees
- Can argue that the best we can hope to do is to superimpose a process group mechanism over CASD (Verissimo and Almeida are looking at this).

Why worry?

- CASD can be used to implement a distributed shared memory ("delta-common storage")
- But when this is done, the memory consistency properties will be those of the CASD protocol itself
- If CASD protocol delivers different sets of messages to different processes, memory will become inconsistent

Why worry?

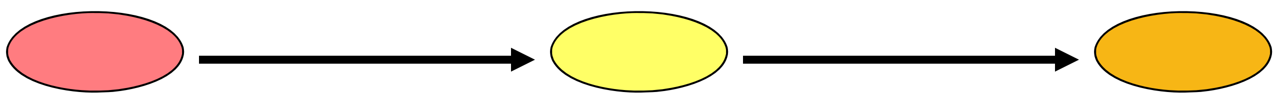
- In fact, we have seen that CASD can do just this, if the parameters are set aggressively
- Moreover, the problem is not detectable either by "technically faulty" processes or "correct" ones
- Thus, DSM can become inconsistent and we lack any obvious way to get it back into a consistent state

Using CASD in real environments

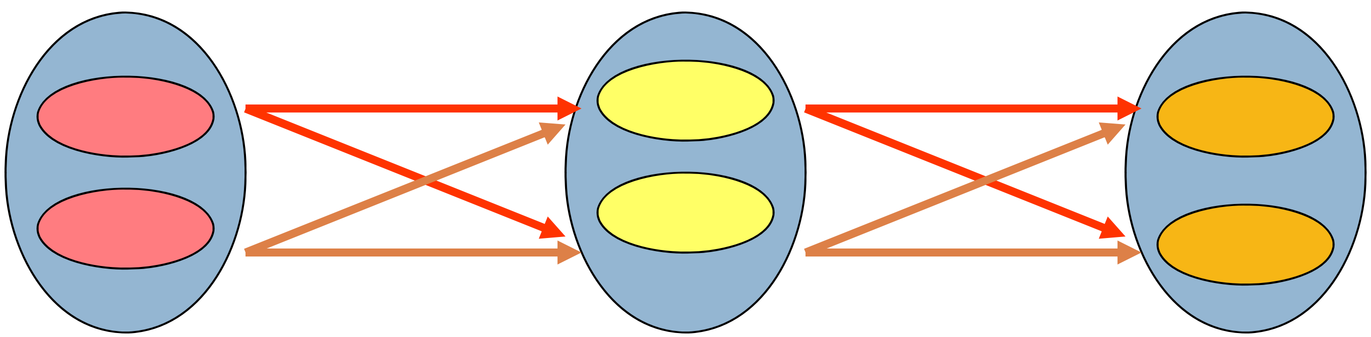
- Once we build the CASD mechanism how would we use it?
  - Could implement a shared memory
  - Or could use it to implement a real-time state machine replication scheme for processes
- US air traffic project adopted latter approach
  - But stumbled on many complexities…

Using CASD in real environments

- Pipelined computation



- Transformed computation



Issues?

- Could be quite slow if we use conservative parameter settings
- But with aggressive settings, either process could be deemed "faulty" by the protocol
  - If so, it might become inconsistent
    - Protocol guarantees don't apply
  - No obvious mechanism to reconcile states within the pair
- Method was used by IBM in a failed effort to build a new US Air Traffic Control system

A comparison

- Virtually synchronous Send is fault-tolerant and very robust, and very fast, but doesn't guarantee real-time delivery of messages
- CASD is fault-tolerant and very robust, but rather slow. But it does guarantee real-time delivery
- CASD is "better" if our concern is absolute confidence that real-time deadlines will be achieved… but only if those deadlines are "slow"

So… which is better for real-time uses?

- Virtually synchronous Send or CASD?
  - CASD may need seconds before it can deliver, but comes with a very strong proof that it will do so correctly
  - Send will deliver within milliseconds unless strange scheduling delays impact a node
    - But actually delay limit is probably ~45 seconds
    - Beyond this, node will be declared to have crashed

Generalizing to the whole cloud

- The cloud has massive scale
- And most of the thing gives incredibly fast responses: sub 100ms is a typical goal
- But sometimes we experience a long delay or a failure

Traditional view of real-time control favored CASD view of assurances

- In this strongly assured model, the assumption was that we need to prove our claims and guarantee that the system will meet goals

- And like CASD this leads to slow systems
  - And to CAP and similar concerns

Back to our puzzle

- So can the cloud do high assurance?
  - Presumably not if we want CASD kinds of proofs
  - But if we are willing to "overwhelm" delays with redundancy, why shouldn't we be able to do well?

- Suppose that we connect our user to two cloud nodes and they perform read-only tasks in parallel
  - Client takes first answer, but either would be fine
  - We get snappier response but no real "guarantee"

A vision: "Good enough assurance"

- Build applications to protect themselves against rare but extreme problems (e.g. a medical device might warn that it has lost connectivity)
  - This is needed anyhow: hardware can fail…
  - So: start with "fail safe" technology

- Now make our cloud solution as reliable as we can without worrying about proofs
  - We want speed and consistency but are ok with rare crashes that might be noticed by the user

Will this do?

- Probably not for some purposes… but some things just don't belong under computer control

- For most purposes, this sort of solution might balance the benefits of the cloud with the kinds of guarantees we know how to provide

- Use redundancy to compensate for delays, insecurity, failures of individual nodes

How the cloud is like Infosys

- The cloud brings huge advantages
  - Lower cost… much better scalability
- And it also brings problems
  - Today's cloud is inconsistent by design, not very secure…
- But why should we assume tomorrow's cloud won't be better? The cloud seems to be winning!
  - Our job: find ways to make the cloud *safely* do more
  - This task seems completely feasible!

Summary: Should we trust the cloud?

- We've identified a tension centering on priorities
  - If your top priority is assurance properties you may be forced to sacrifice scalability and performance in ways that leave you with a *useless* solution
  - If your top priorities center on scale and performance and then you layer in other characteristics it may be feasible to keep the cloud properties and get a good enough version of the assurance properties
- These tradeoffs are central to cloud computing!
- But like the other examples, cloud could win even if in some ways, it isn't the "best" or "most perfect" solution