CS5412: THE REALTIME CLOUD

Lecture XXIV

Ken Birman

Can the Cloud Support Real-Time?

- More and more "real time" applications are migrating into cloud environments
  - Monitoring of traffic in various situations, control of the traffic lights and freeway lane limitations
  - Tracking where people are and using that to support social networking applications that depend on location
  - Smart buildings and the smart power grid

- Can we create a real-time cloud?

Core Real-Time Mechanism

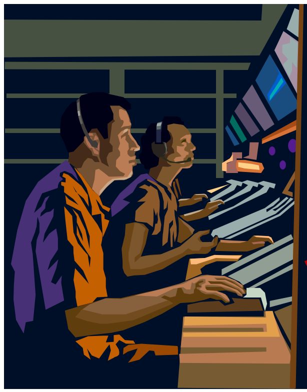
- We've discussed publish-subscribe
  - Topic-based pub-sub systems (like the TIB system)
  - Content-based pub-sub solutions (like Sienna)

- Real-time systems often center on a similar concept that is called a real-time data distribution service
  - DDS technology has become highly standardized
  - It mixes a kind of storage solution with a kind of pub-sub interface but the guarantees focus on real-time

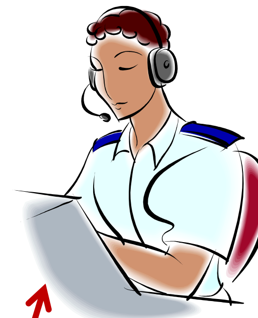
What is the DDS?

- The **Data Distribution Service for Real-Time Systems** (DDS) is an Object Management Group (OMG) standard that aims to enable scalable, real-time, dependable, high performance and interoperable data exchanges between publishers and subscribers.
- DDS is designed to address the needs of applications like financial trading, air traffic control, smart grid management, and other big data applications.

Air Traffic Example



Owner of flight plan updates it…
there can only be one owner.



… Other clients see
real-time read-only updates

DDS makes the update persistent, records the ordering of the event, reports it to client systems

- DDS combines database and pub/sub functionality

Quality of Service options

- Early in the semester we discussed a wide variety of possible guarantees a group communication system could provide

- Real-time systems often do this too but the more common term is *quality of service* in this case
  - Describes the quality guarantees a subscriber can count upon when using the DDS
  - Generally expressed in terms of throughput and latency

CASD (Δ-T atomic multicast)

- Let's start our discussion of DDS technology by looking at a form of multicast with QoS properties
  - This particular example was drawn from the US Air Traffic Control effort of the period 1995-1998
  - It was actually a failure, but there were many issues
  - At the core was a DDS technology that combined the real-time protocol we will look at with a storage solution to make it durable, like making an $Isis^2$ group durable by having it checkpoint to a log file (you use g.SetPersistent() or, with SafeSend, enable Paxos logging)

CASD: **Flaviu Cristian**, Houtan **Aghili**, Ray **Strong** and Danny Dolev.
Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement (1985)

Real-time multicast: Problem statement

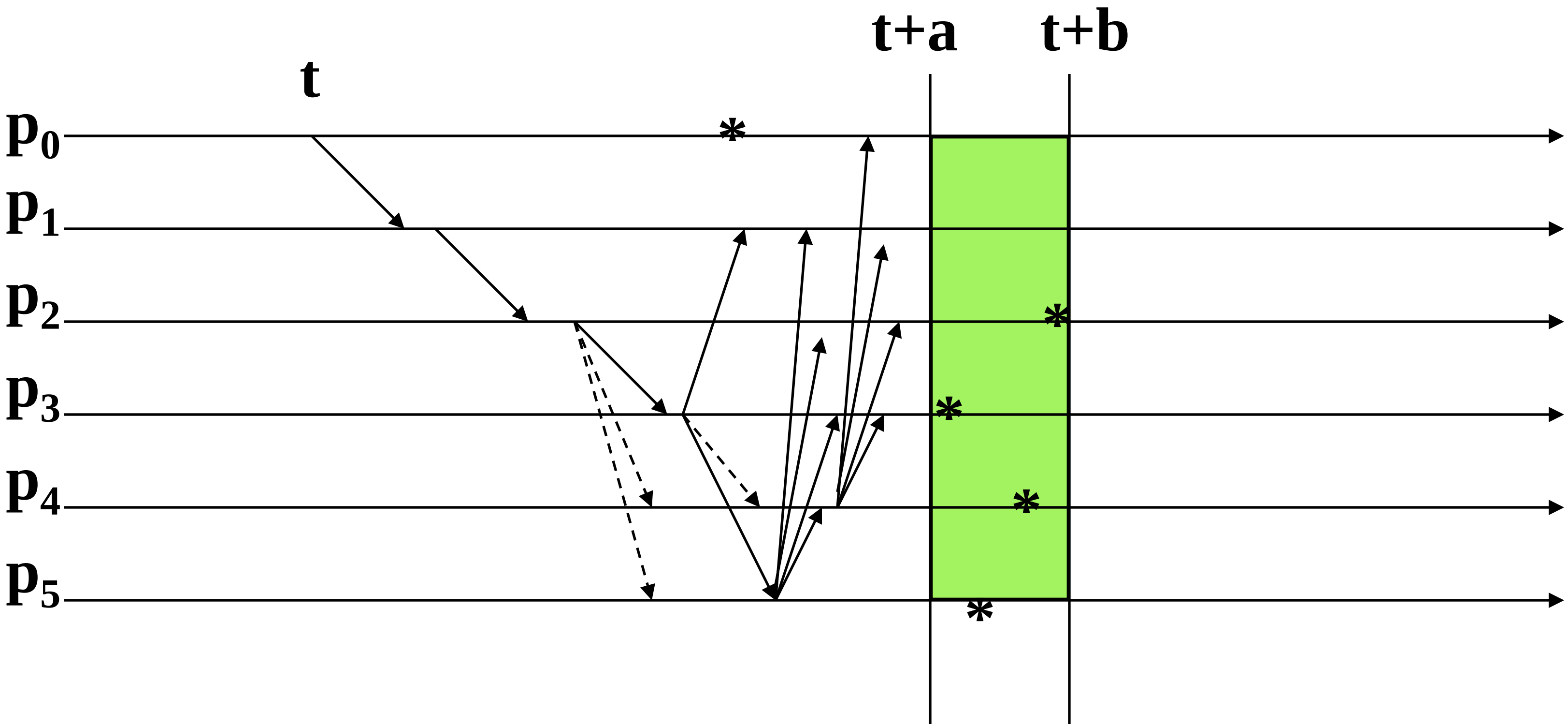
- The community that builds real-time systems favors proofs that the system is *guaranteed* to satisfy its timing bounds and objectives

- 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?

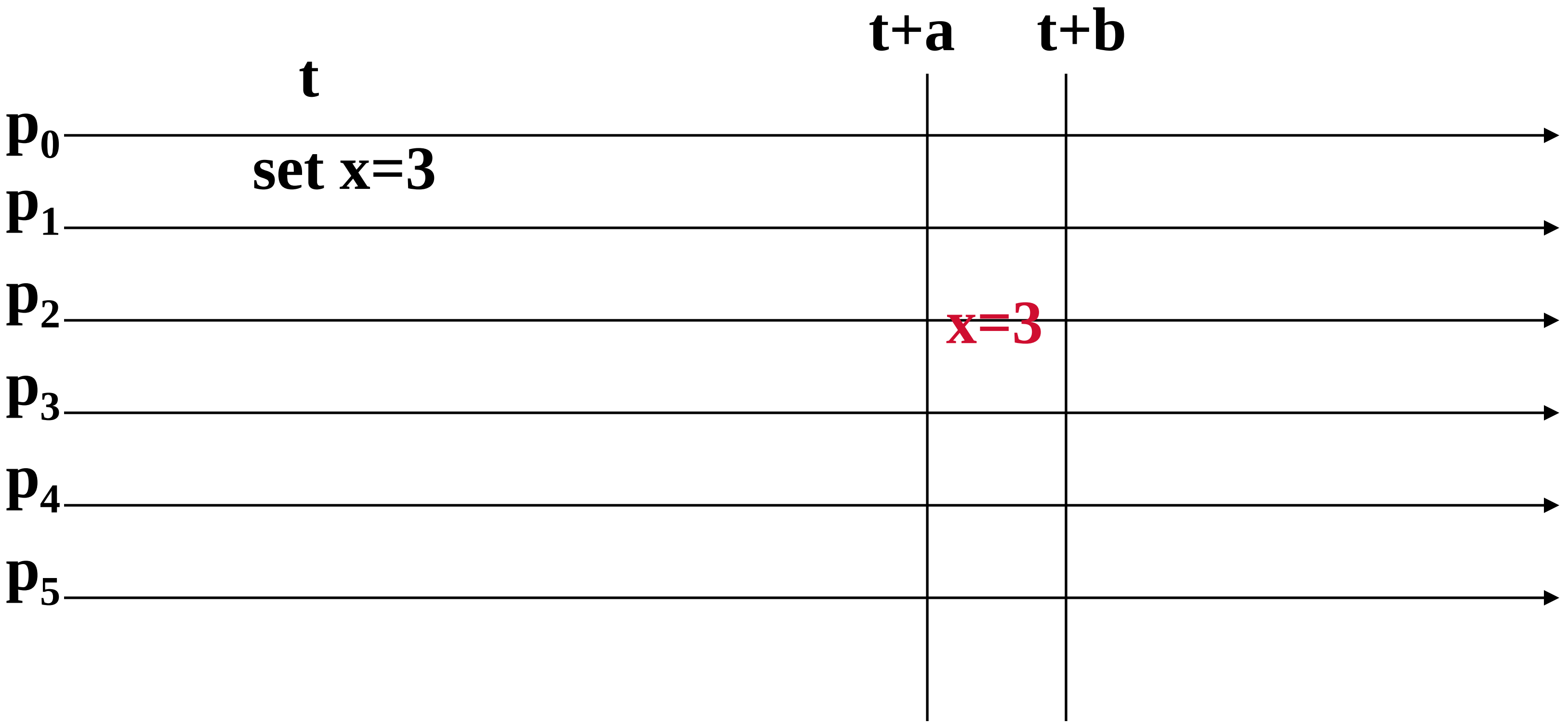
- Suppose we want to implement broadcast protocols that make direct use of temporal information
- Examples:
  - Broadcast that is delivered at same time by all correct processes (plus or minus the clock skew)
  - Distributed shared memory that is updated within a known maximum delay
  - Group of processes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

A real-time broadcast



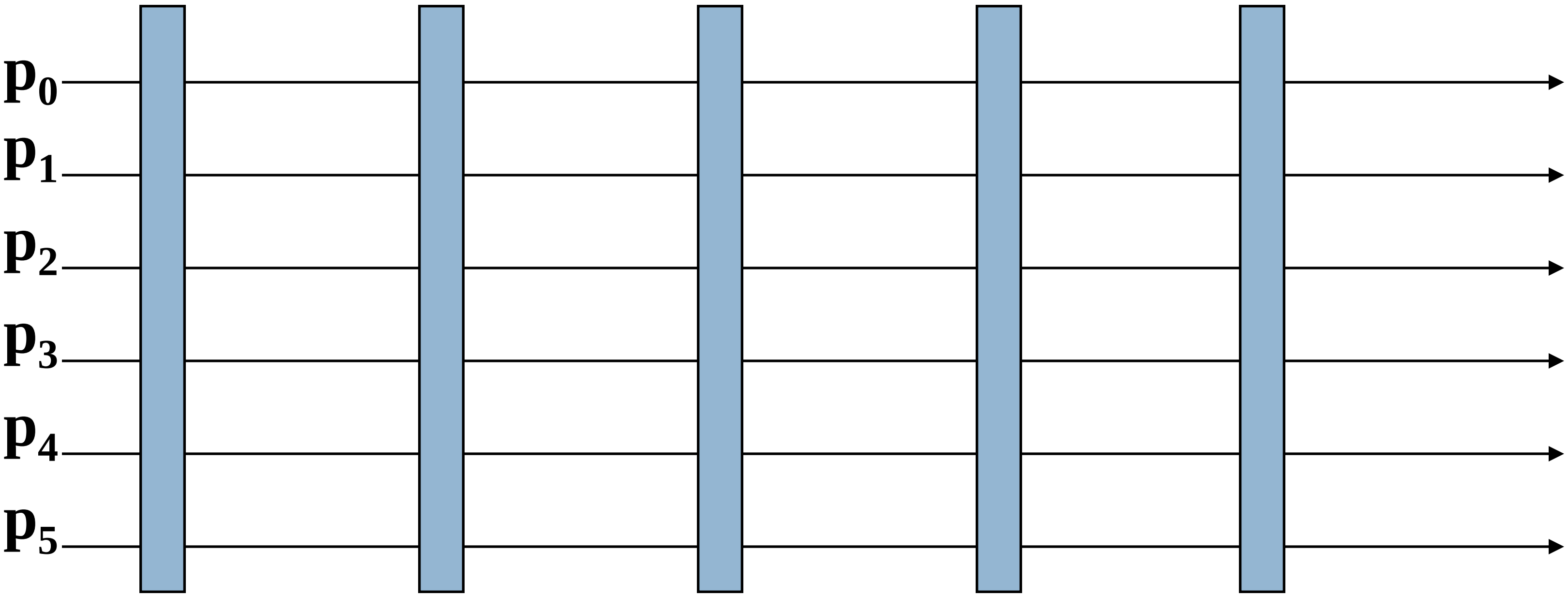
Message is sent at time *t* by p_0. Later both p_0 and p_1 fail. But message is still delivered atomically, after a bounded delay, and within a bounded interval of time (at non-faulty processes)

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

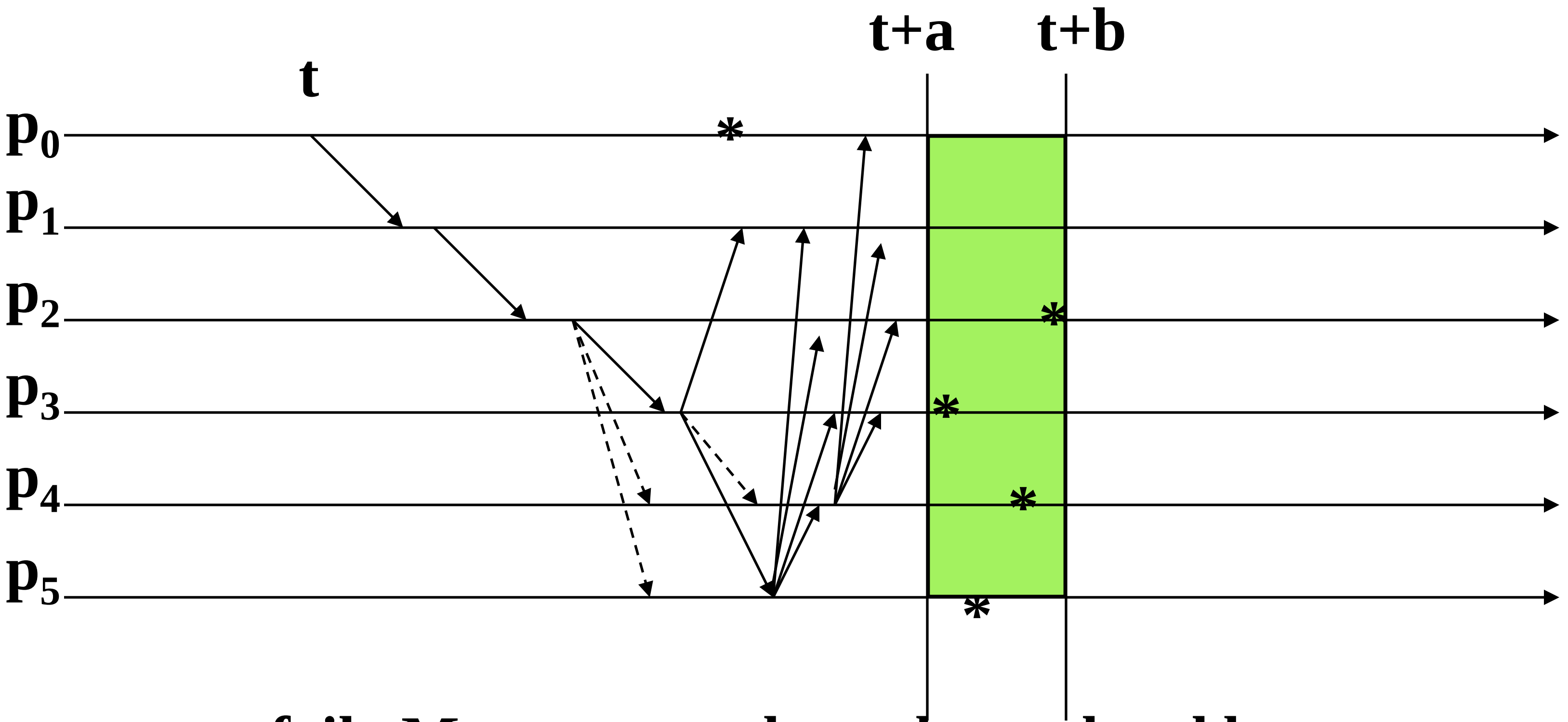
The CASD protocol suite

- 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

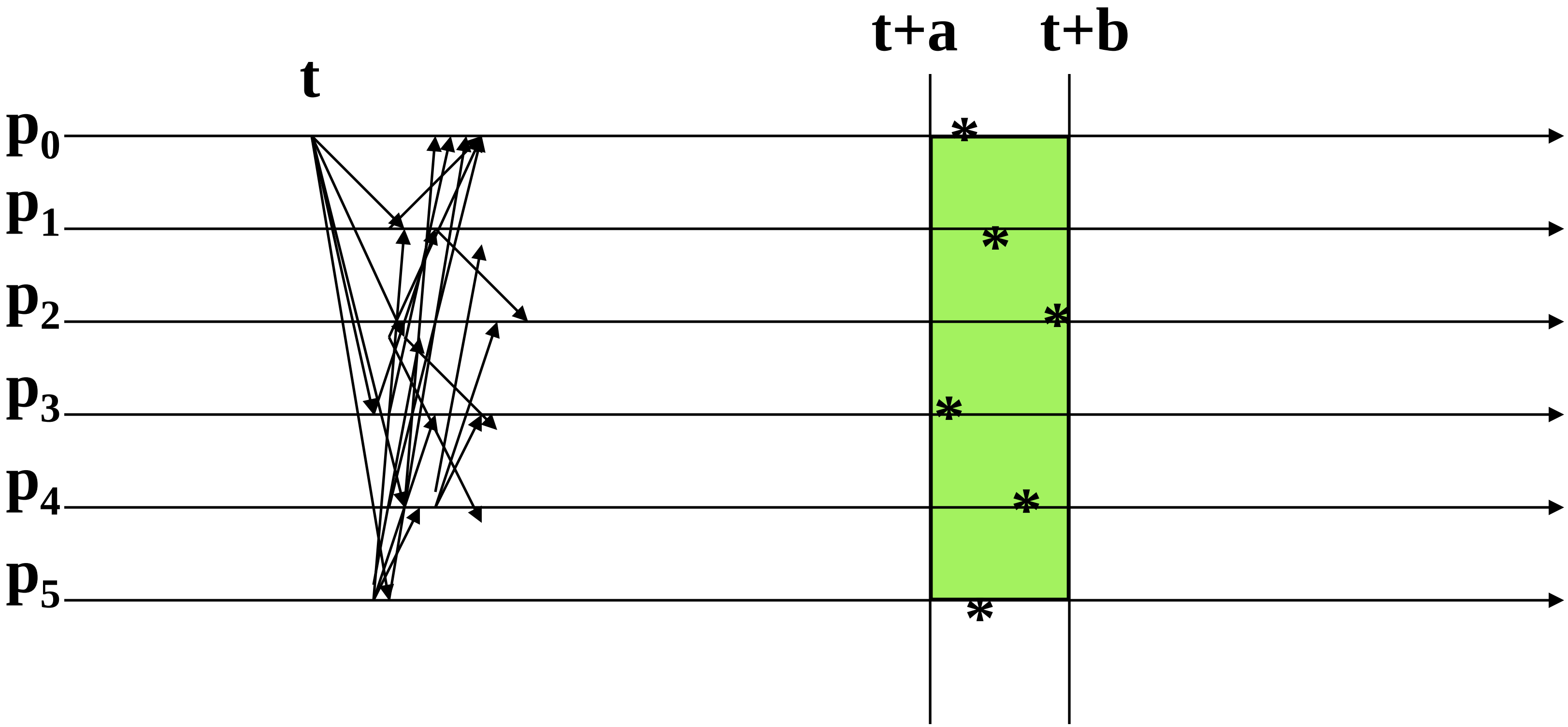
Idea of CASD

- 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

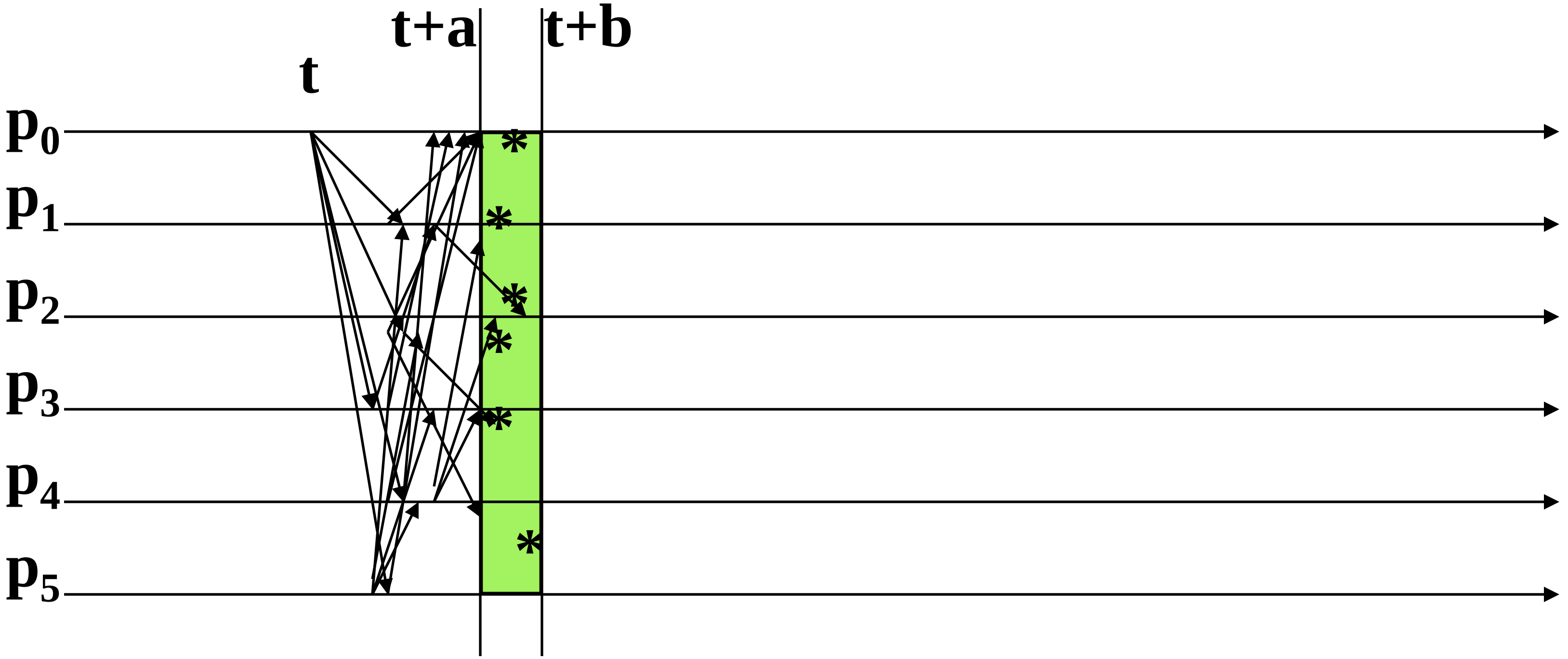
The problems with CASD

- 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?

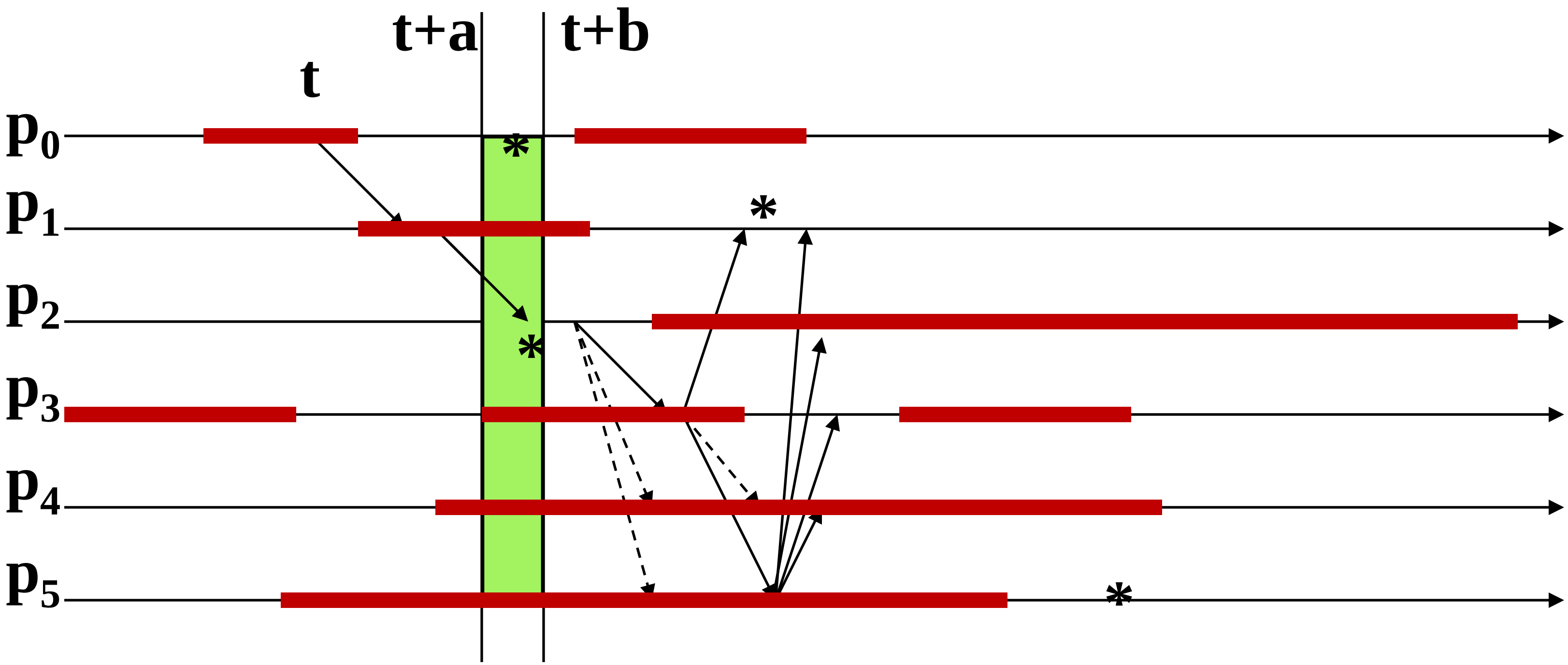
CASD in a more typical run



… 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?

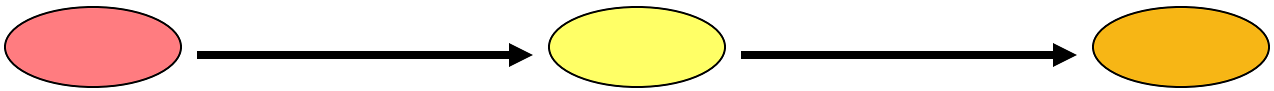
- In fact, we have seen that CASD can do just this, if the parameters are set aggressively
- Moreover, the problem is not detectable either by "technically faulty" processes or "correct" ones
- Thus, DSM can become inconsistent and we lack any obvious way to get it back into a consistent state

Using CASD in real environments

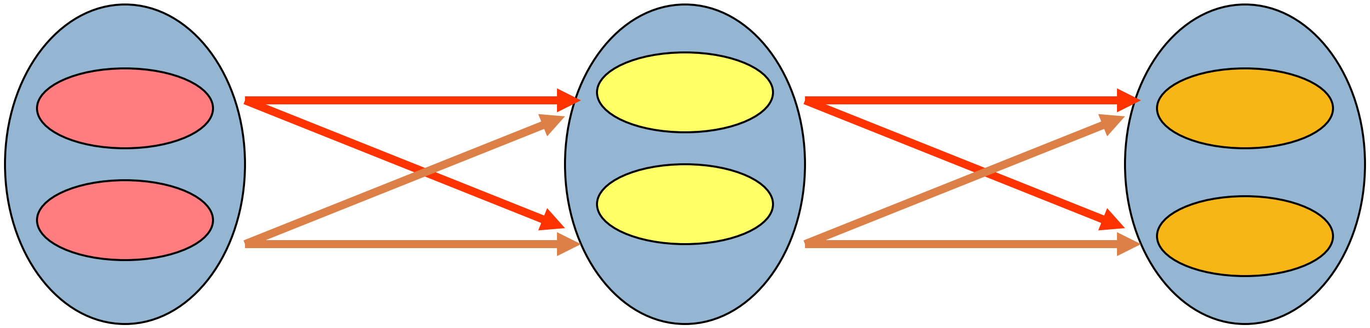
- Once we build the CASD mechanism how would we use it?
  - Could implement a shared memory
  - Or could use it to implement a real-time state machine replication scheme for processes
- US air traffic project adopted latter approach
  - But stumbled on many complexities…

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

Can we combine CASD with consensus?

- Consensus-based mechanisms (Isis^2, Paxos) give strong guarantees, such as "there is one leader"
- CASD overcomes failures to give real-time delivery if parameterized correctly (clearly, *not* if parameterized incorrectly!)
- Why not use both, each in different roles?

A comparison

- Virtually synchronous Send is fault-tolerant and very robust, and very fast, but doesn't guarantee realtime 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 application requires absolute confidence that real-time deadlines will be achieved... but only if those deadlines are "slow"

Weird insight

- If a correctly functioning version of CASD would be way too slow for practical use, then a protocol like Send might be better even for the real-time uses!
- The strange thing is that Send isn't designed to provide guaranteed real-time behavior
- But in practice it is incredibly fast, compared to CASD which can be incredibly slow…

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 ~10 seconds
    - Beyond this, if ISIS_DEFAULT_TIMEOUT is set to a small value like 5s, node will be declared to have crashed

Back to the DDS concept

- In a cloud setting, a DDS is typically
  - A real-time protocol, such as CASD
  - Combined with a database technology, generally transactional with strong durability
  - Combined with a well defined notion of "objects", for example perhaps in the IBM Air Traffic Control project something like "Flight Data Records"
  - Combined with a rule: when the FDR is updated, we will also use the DDS to notify any "subscribers" to that object. So the object name is a topic in pub-sub terms.

IBM Air Traffic Concept

- Everyone uses the Δ-Common storage abstraction and maintains a local "copy" of all FDRs relevant to the current air traffic control state

- To update an FDR, there should be a notion of an *owner* who is the (***single***) controller allowed to change the FDR.
  - Owner performs some action, this updates the durable storage subsystem
  - Then when update is completely final, Δ-T atomic multicast is used to update all the Δ-Common storage records
  - Then this updates applications on all the controller screens

Safety needs?

- Clearly there needs to be a well defined guarantee of a single controller for each FDR
  - There must *always* be an assigned controller
  - … but there can only be one per FDR
- Also we need the DDS to be reliable; CASD could be used, for example
- But we also need a certain level of speed and latency guarantees

What makes it hard?

- As we see with CASD, sometimes the analysis used to ensure reliability "fights" the QoS properties needed for safety in the application as a whole

- Moreover, we didn't even consider delays associated with recovering the DDS storage subsystem when a failure or restart disrupts it
  - E.g. bringing a failed DDS storage element back online
  - We need to be sure that every FDR goes through a single well-defined sequence of "states"

If a system is too slow…

- … it may not be useable even if the technology that was used to build it is superb!
- With real DDS solutions in today's real cloud settings this entire issue is very visible and a serious problem for developers
- They constantly struggle between application requirements and what the cloud can do *quickly*

Generalizing to the whole cloud

- 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

And this leads back to our question

- 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

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

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?

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