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?

Many ways to ask this question

- Can the data center network itself be improved to have great predictability and support fast failure sensing? Leads to "F10" concept (U. Washington)
- Can we build file systems better suited to capturing data from real-time sources? Leads to "Freeze Frame FS" idea (Cornell)
- Today: Can we do data replication with good real-time properties?

Core Real-Time Mechanism

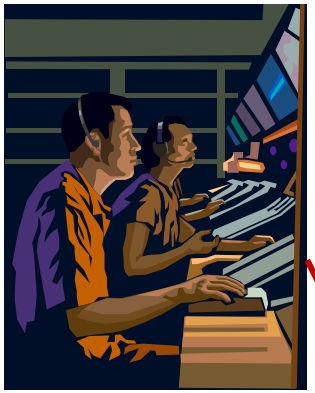
- We've discussed publish-subscribe
  - Topic-based pub-sub systems (like the TIB system)
  - Content-based pub-sub solutions (like Sienna)

- 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)

Real-time multicast: Problem statement

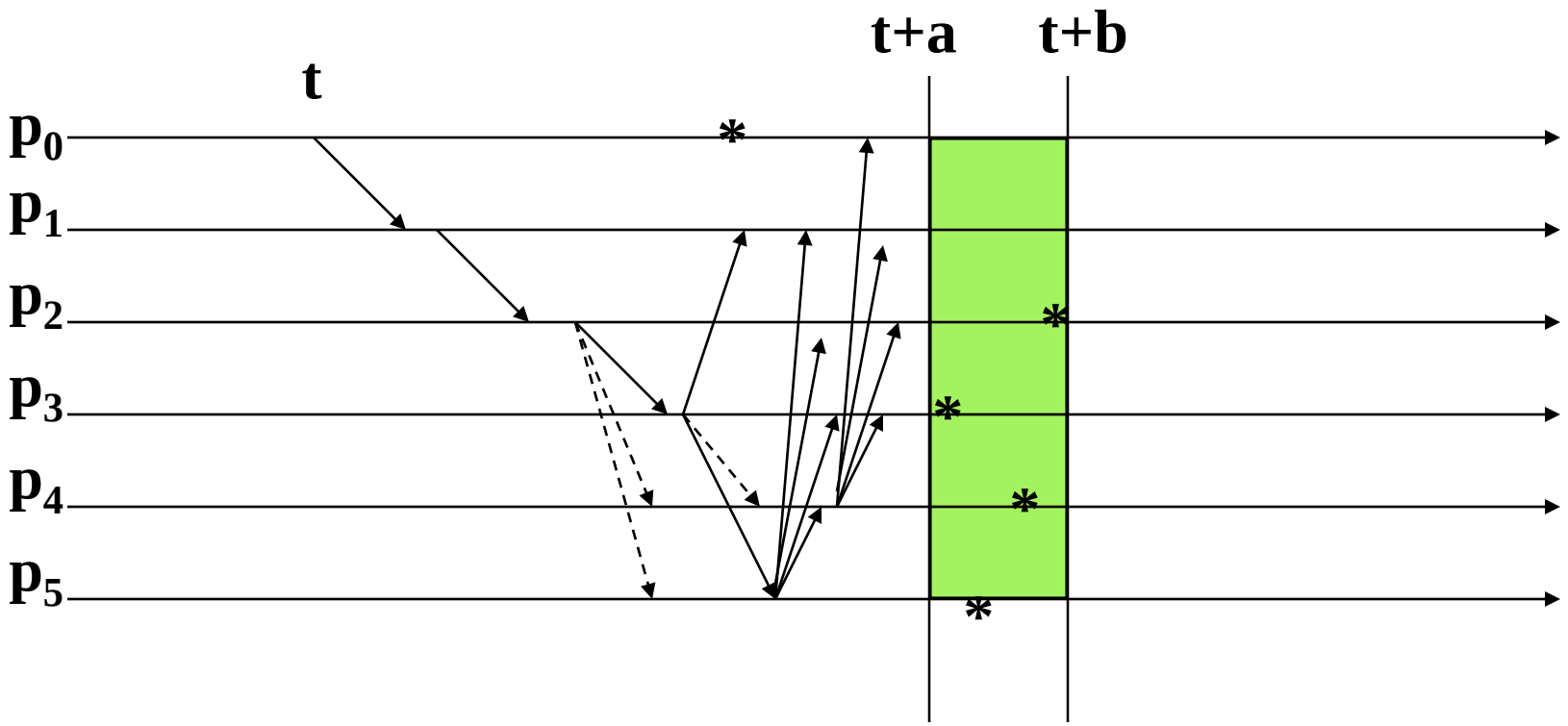
- The community that builds real-time systems favors proofs that the system is *guaranteed* to satisfy its timing bounds and objectives

- 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?

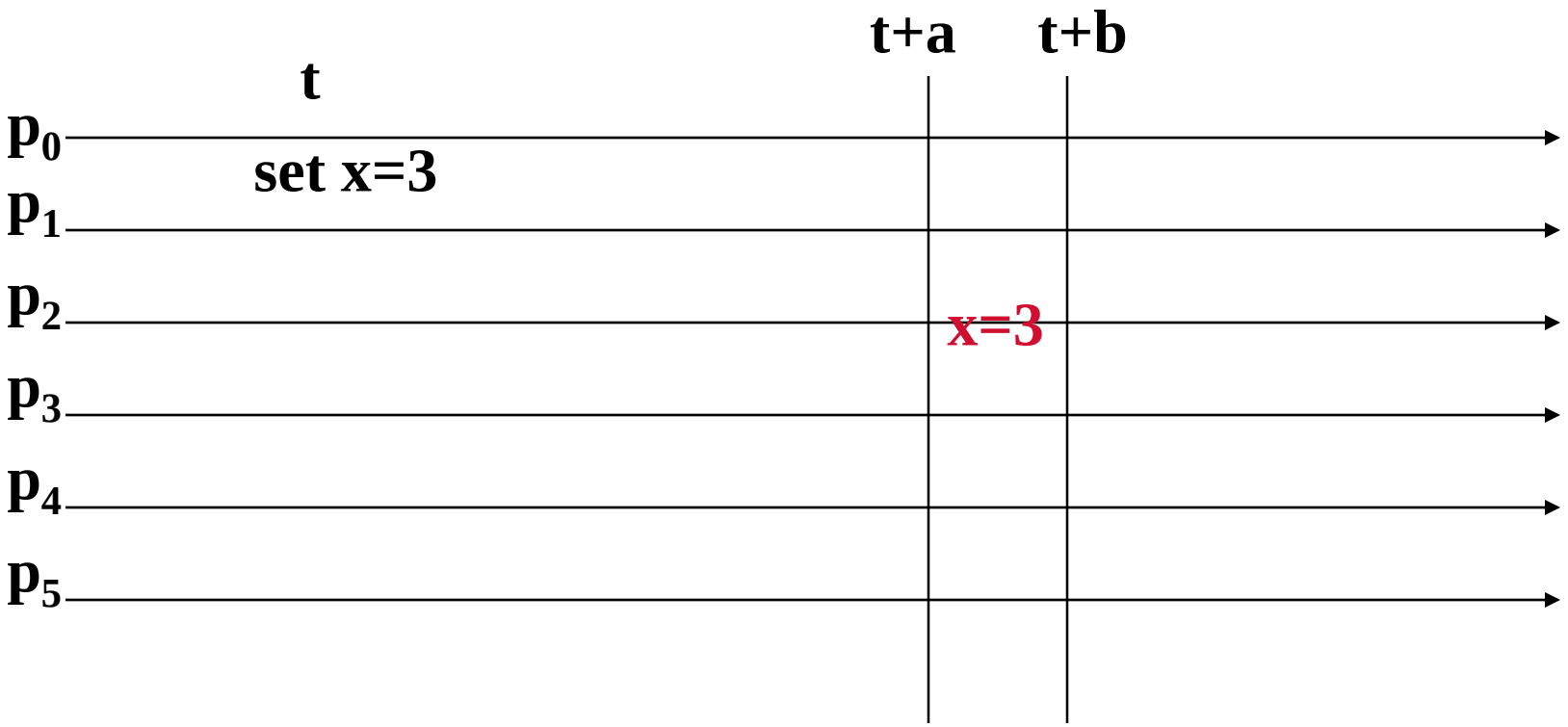
- Suppose we want to implement broadcast protocols that make direct use of temporal information
- Examples:
  - Broadcast that is delivered at same time by all correct processes (plus or minus the clock skew)
  - Distributed shared memory that is updated within a known maximum delay
  - Group of processes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

A real-time broadcast



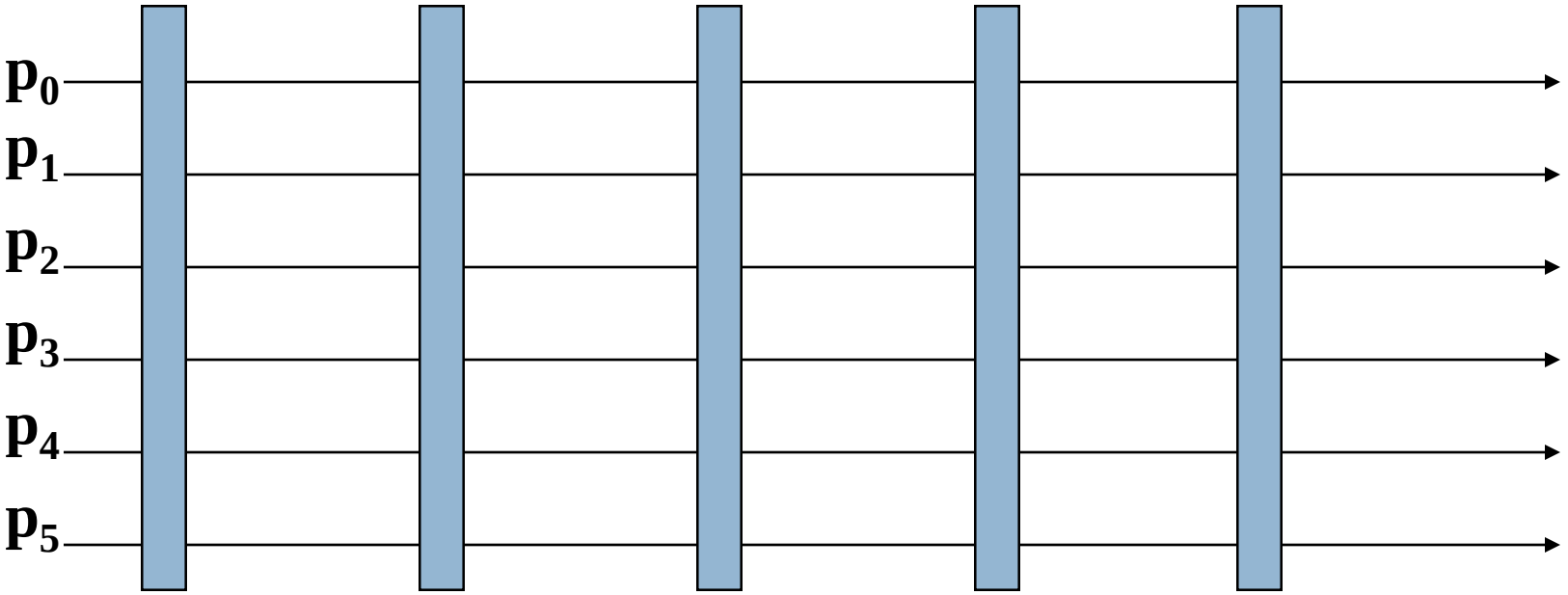
Message is sent at time *t* by p_0. Later both p_0 and p_1 fail. But message is still delivered atomically, after a bounded delay, and within a bounded interval of time (at non-faulty processes)

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

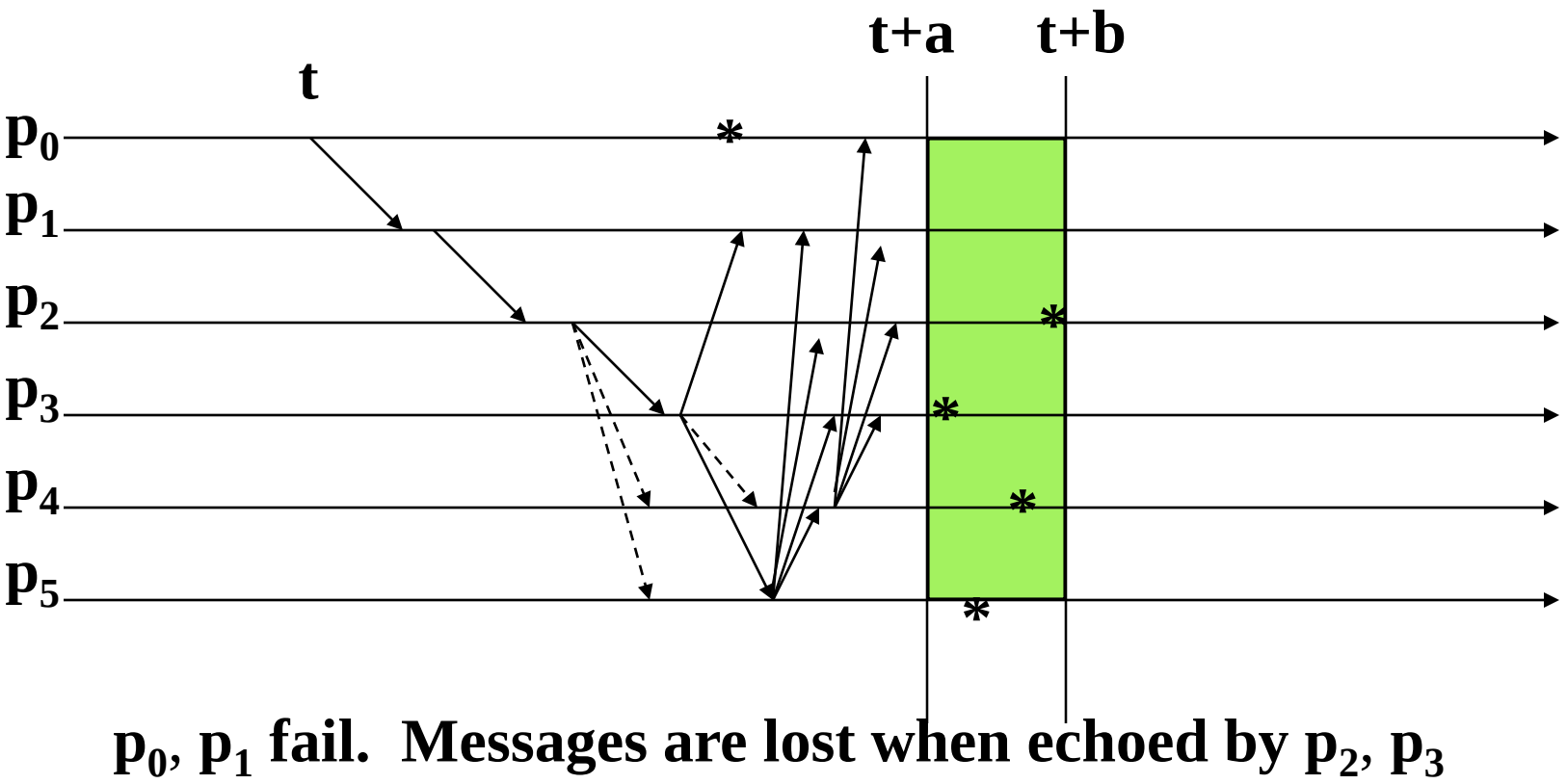
The CASD protocol suite

- 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

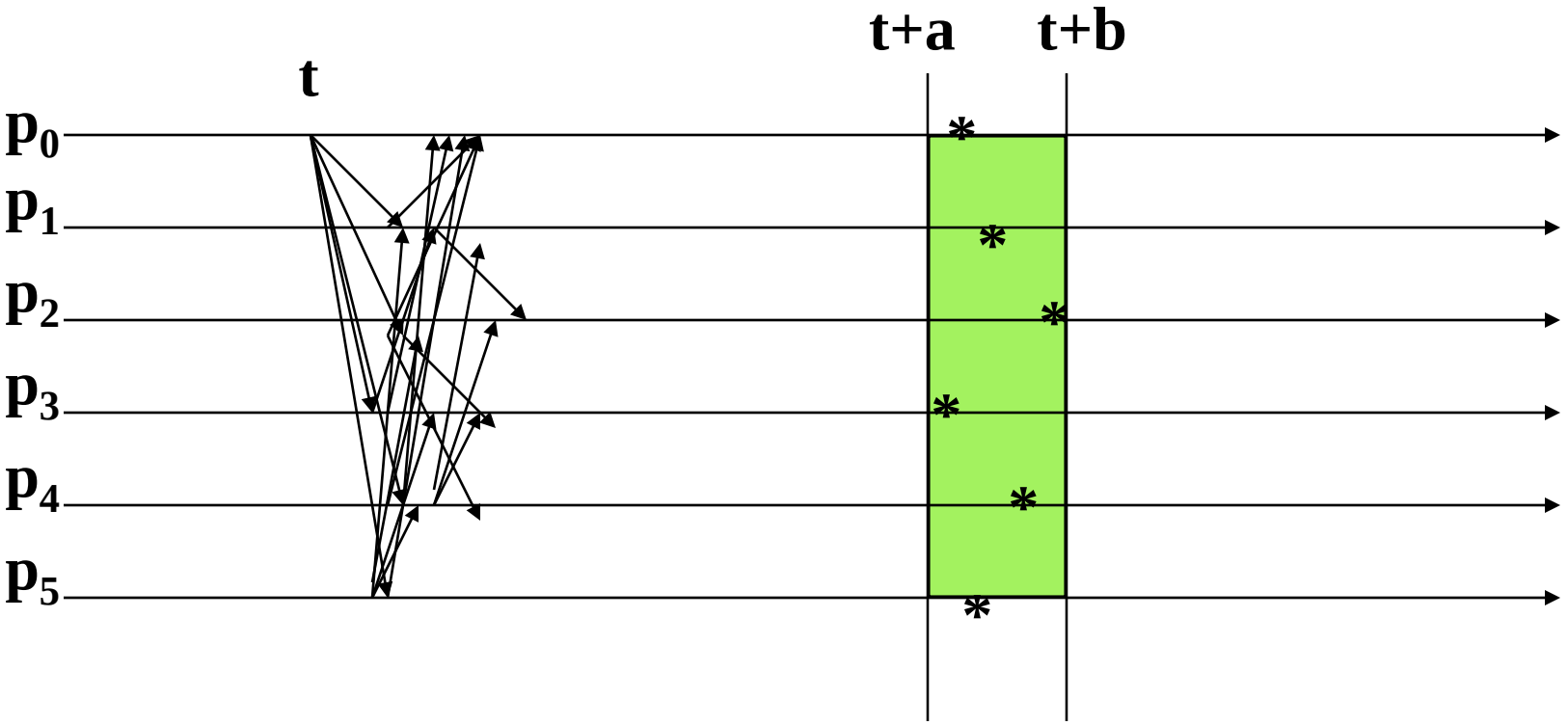
Idea of CASD

- 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

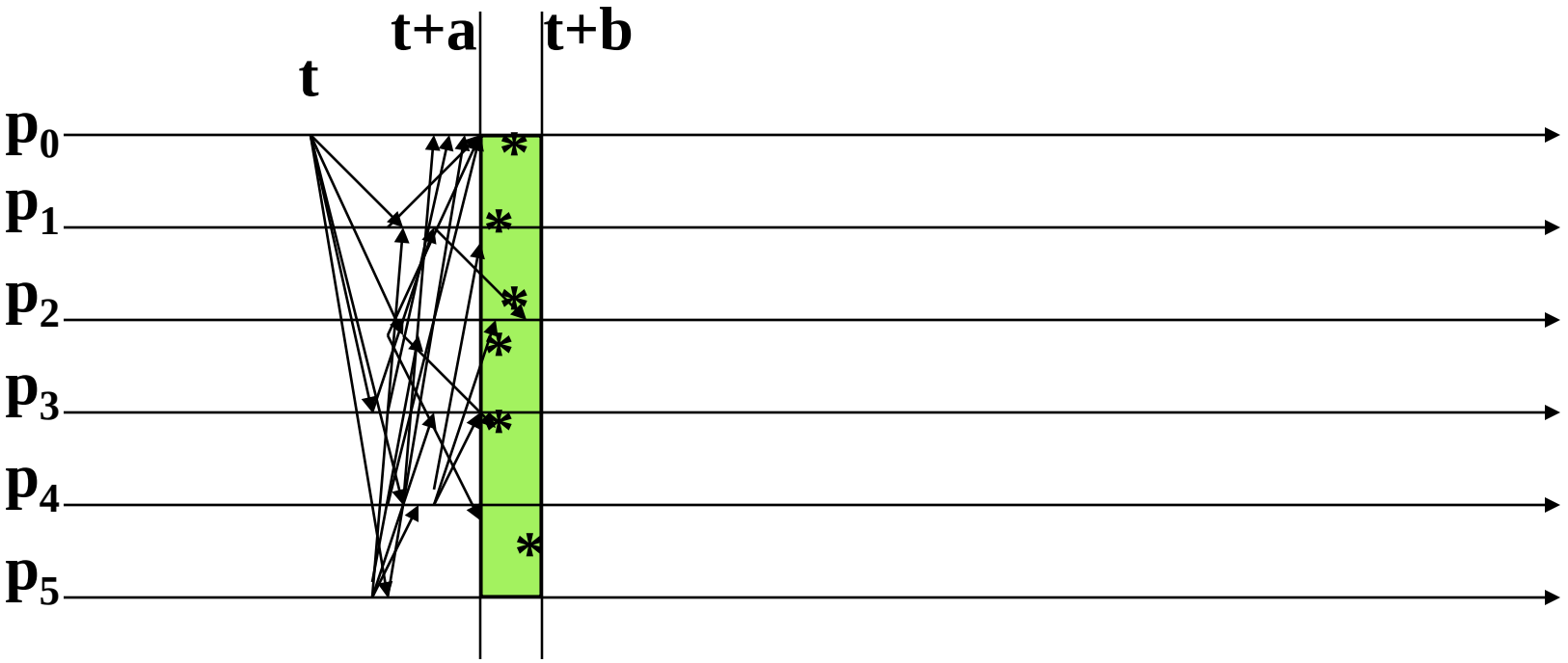
The problems with CASD

- 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?

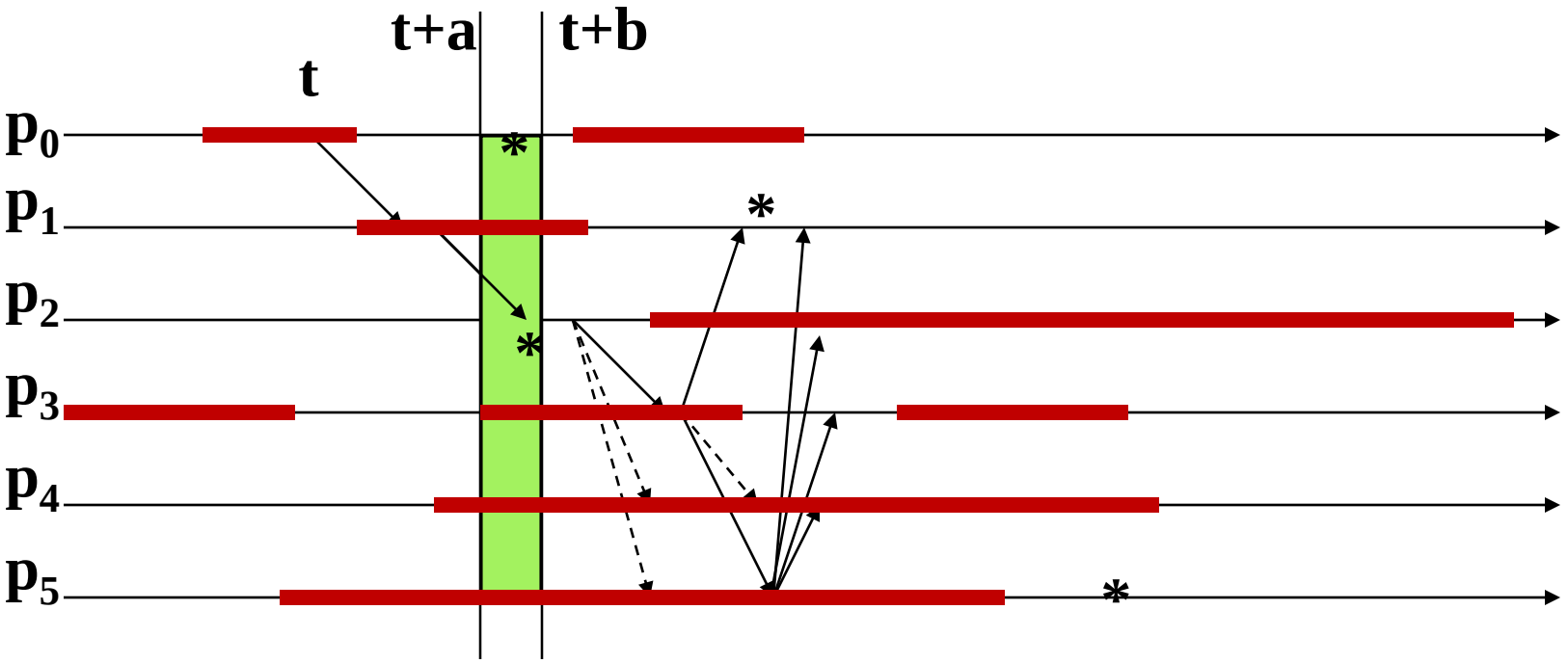
CASD in a more typical run



… 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 Vsync OrderedSend or Paxos
- When run "quickly" the CASD 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
  - Ideally you would want this to be very rare, but…
- If run *very* quickly, CASD malfunctions so often that its behavior is totally chaotic!

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?

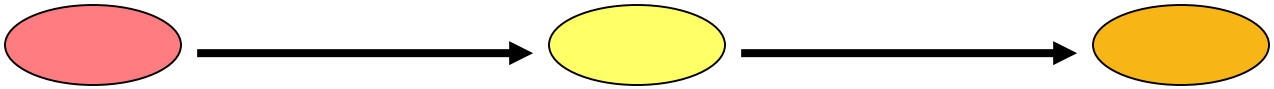
- In fact, we have seen that CASD can do just this, if the parameters are set aggressively
- Moreover, the problem is not detectable either by "technically faulty" processes or "correct" ones
- Thus, DSM can become inconsistent and we lack any obvious way to get it back into a consistent state

Using CASD in real environments

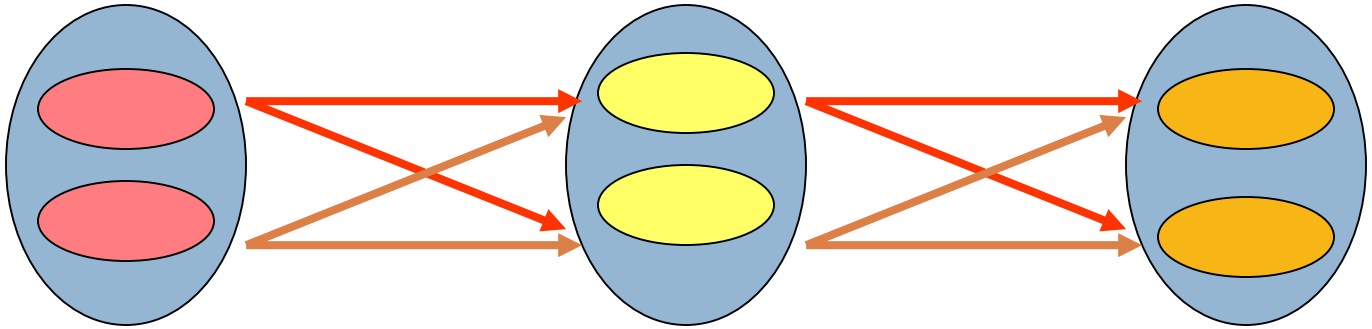
- Once we build the CASD mechanism how would we use it?
  - Could implement a shared memory
  - Or could use it to implement a real-time state machine replication scheme for processes
- US air traffic project adopted latter approach
  - But stumbled on many complexities…

Using CASD in real environments

- Pipelined computation



- Transformed computation



IBM found hard to use

- Attempted to use this approach in an air traffic control system for the US and Britain
  - But CASD properties weren't strong enough
  - They ended up giving up on the approach and just using checkpoint/restart if things crashed

- In contrast, the French ATC system was more successful and used Virtual Synchrony pretty much in the same way that IBM had hoped to use CASD

Why did Vsync succeed?

- The stronger guarantees of the multicast made it much easier to reason about the state of the system in every possible case that could arise

- So higher level code was simplified and easier to test. A good example of the real-world experimenting with weak consistency, and realizing that strong consistency is sometimes needed!

Freeze Frame File System

- This is current research here at Cornell and is also about supporting real-time applications

- Focuses on how to capture real-time data in a file system and the best options for querying that data

Read X at 10:02:59.228
Read Y at 10:02:59.227
Read Z at 10:03:12.225

Freeze-Frame FS

- Many systems send updates that have timestamps right in the update records
- For example, in the smart grid, "synchrophasor measurement devices" are used to track the power grid state, and they have GPS clocks built right in
- So if you monitor the power grid you have thousands of incoming time-stamped data streams

Freeze Frame FS

- Requires a plug-in to know how to find the GPS timestamp in the updates (else uses the server clock)
- But then when you do a write it can remember the time when the write occurred.
- Stores data in a CORFU-like memory-mapped log

Freeze Frame FS

- A file system read works like in Linux or Windows

- But FFFS also supports a special directory with a time in it and a read-only copy of the FS below it.
  - The files and data are virtual: if you read these, FFFS looks up the data applicable at the exact time requested
  - Accurate to about 1ms, and with logical consistency too (the same kind as from Chandy/Lamport!)
  - So you can build applications that explore data in time…

The concept of "real-time"

- We've seen that there really isn't any foolproof way to build time-based applications at cloud scale
- For real-time data or computation replication:
  - CASD illustrated how an all-or-nothing way of treating time can be too slow, and hard to "fix"
  - In fact Vsync isn't a real-time solution yet was fast enough to be used successfully in the French ATC
- Freeze Frame File System focuses on time in storage
  - The challenge becomes easier because we aren't as focused on taking instant actions

Asynchronous computing

- One clear message is that in the cloud, we do better with designs that are asynchronous
  - Recall that this word is about a pipelined style of computing, where updates are sent to the storage system to be registered, but we don't "wait" for them to be finished.
  - Asynchrony decouples the moving parts, allows the front-end of a system to move fast, while the back-end can do clever things like tracking time very accurately
- Synchronous solutions are too costly and slow.

Life with technology is about tradeoffs

- A bit like CAP, we apparently can have
  - Systems that are fast, but don't understand time
  - Systems that have strong temporal guarantees, but that aren't fast
- In some situations we can have speed and time (like FFFS) but this is partly because FFFS retrieves saved data after the fact.
  - Attempting to provide real-time guarantees "online", as a system runs, is what made CASD stumble….