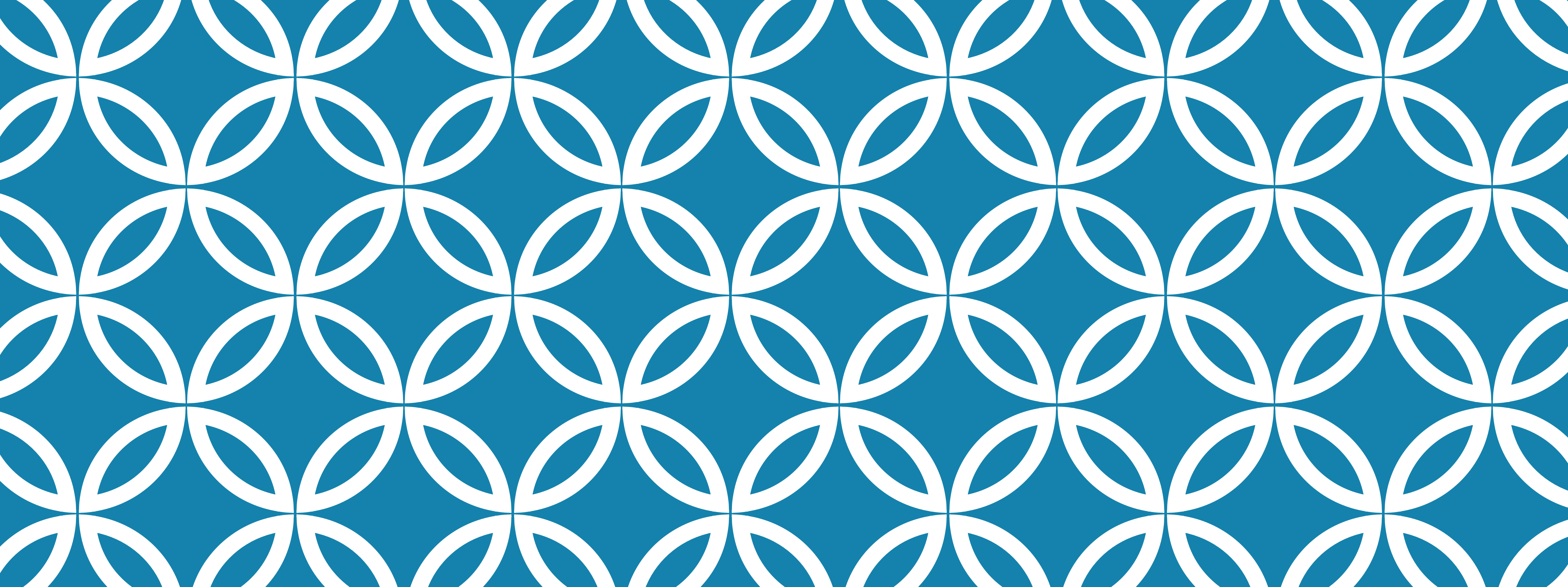


CS5412: LECTURE 8
USING TIME IN IoT APPLICATIONS

Ken Birman

Spring, 2019

CONTEXT: QUICK REMINDER

Real-time clocks have limitations on precision, accuracy and skew.

Lamport defines the happens-before relation ($\rightarrow$), implements logical clocks (and vector clocks), defines consistent cuts.

Freeze Frame File System ($FFFS_{V1}$) tracks file versions, offers a fast way to "seek in time" that lets you read data at a precise time, on a consistent cut.

$FFFS_{v2}$ will have a different API but the same capabilities: the Derecho RDMA object store has versioned objects and an efficient get(v, T) option.

ANOTHER REMINDER

Freeze Frame is basically dealing with time and consistency "both at once"

- If you try to read at time T, due to clock skew, there are many possible system states that all could satisfy the read. It tries to return data as close to T as possible.
- It has so many choices, even with this "close to T" rule, because we can do file updates 1000x faster than the clock can accurately track them.
- By constraining the responses to be along a consistent cut, FFFS ensures that a read at time $T' > T$ includes all events from time T, and also prevents mashups that might be very misleading.

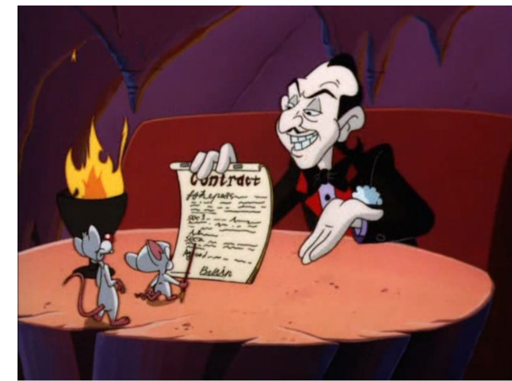
FFFS IS AWESOME. BUT NOT EVERY SYSTEM THAT USES TIME HAS BEEN A SUCCESS

In 1995, the United States launched an ambitious program to update all of US air traffic control. And controlling drones is sort of a 2018 story that sounds like a scaled up version of ATC.

In fact, the 1995 FAA project centered on an IoT computing model!

Basically, sensors + radar + on-plane GPS let us track planes. Pilots and Air Traffic Controllers interact with the system.

GUARANTEES IN IoT PLATFORMS



Let's make a deal…

Guaranteed behavior will always make it easier to create systems. *Right?*

Air traffic control systems need guarantees, and it makes sense for these to include guarantees about time: a contract between the system and the air traffic controllers who use it.

Yet you can get into surprising amounts of trouble this way!

CORE REAL-TIME MECHANISM

Real-time systems often center on a publish-subscribe product called a real-time data distribution service (DDS)

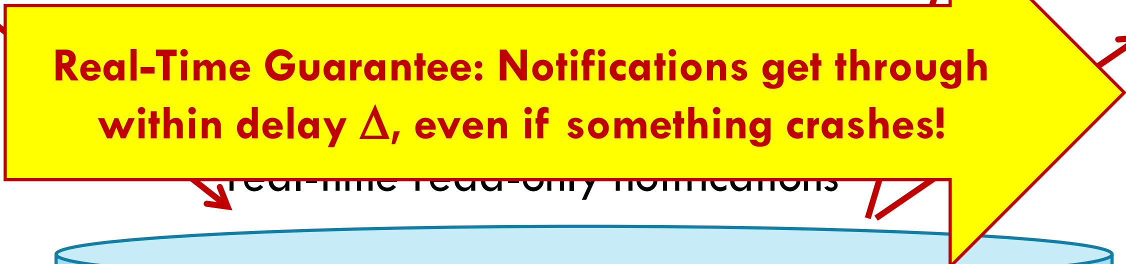
- DDS technology has become highly standardized (by the OMG)
- One can layer DDS over an object store (like in Derecho). In fact we are doing that in Derecho now, as an optional API.

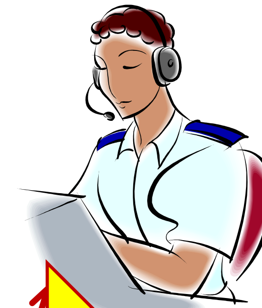
Puzzle: Unlike FFFS, DDS systems don't address consistency at all. They leave that aspect for you to "address at a higher layer."

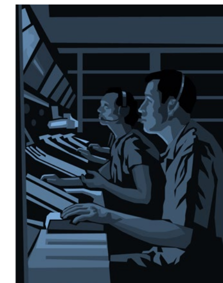
AIR TRAFFIC EXAMPLE

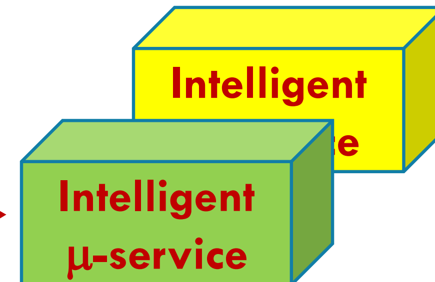
A persistent real-time DDS combines database and pub/sub functionality

<u>Owner</u> of flight plan updates it…
there can only be one owner.









SOME KEY GOALS

Every flight and every runway has a single owner ("controller").

Controllers can perhaps work in teams, in a group of computer consoles showing different aspects of a single system state ("consistency").

Each flight plan evolves through a unique series of versions. A controller edits the current version, then saves it back to create the next version.

The whole system must be fault-tolerant and rapid: actions within seconds.

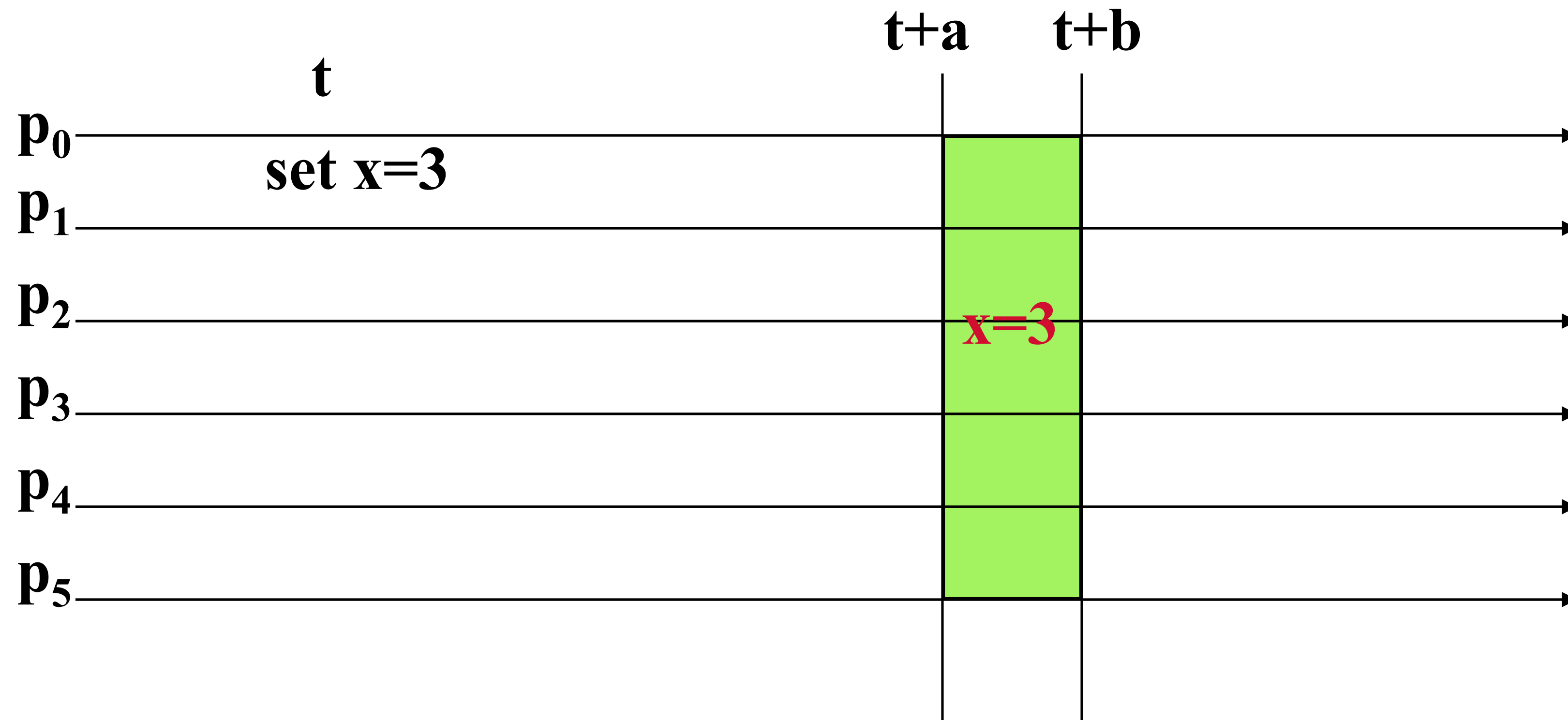
IDEAS IBM PROPOSED

A real-time key-value storage system, in which updates to variables would become visible to all processes simultaneously.

A real-time "heart beat" to help devices take coordinated actions.

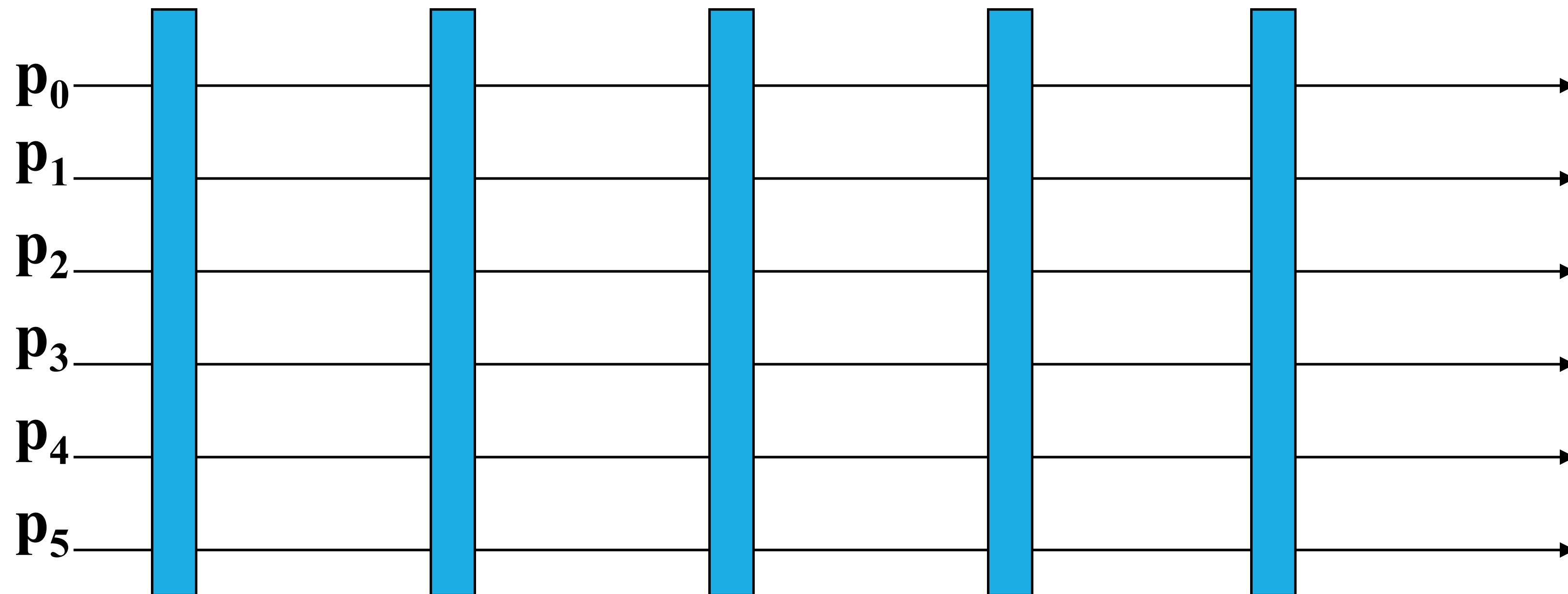
A real-time message bus (DDS) to report new events in a temporally consistent way to processes interested in those events.

A REAL-TIME DISTRIBUTED SHARED MEMORY



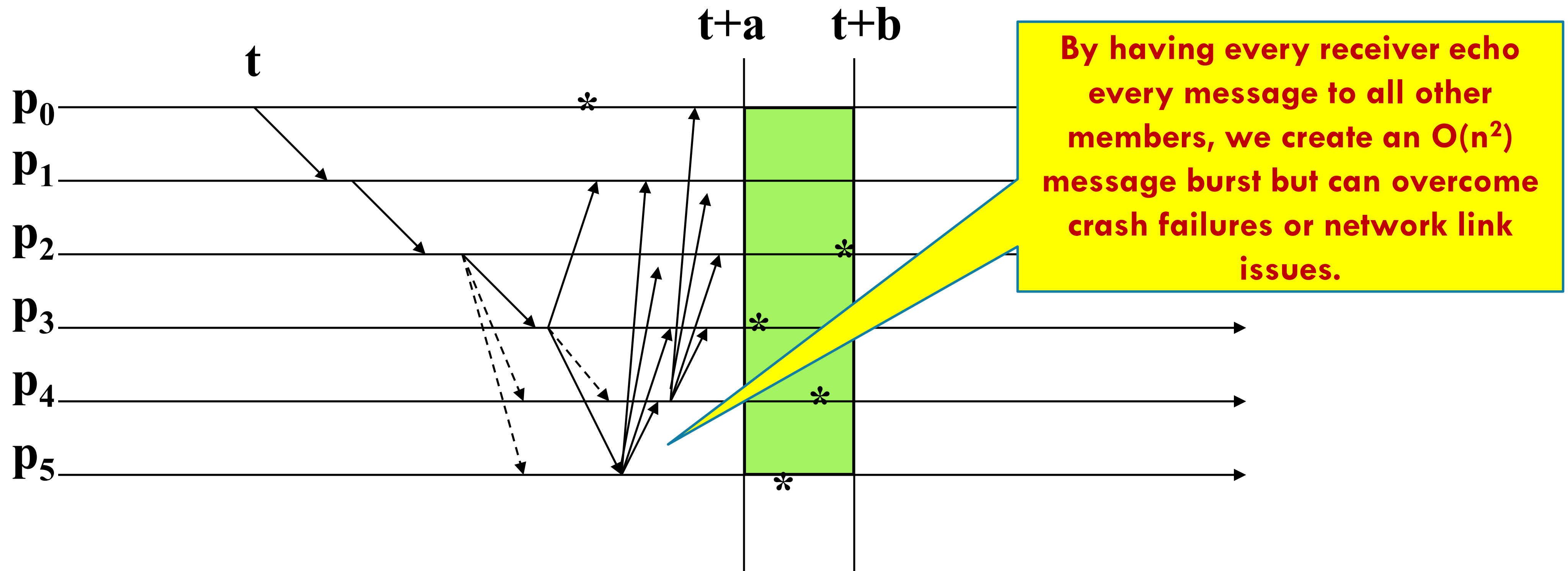
Think of “x” as a key and “3” as the new value. Here x=3 becomes visible within a time window between t+a and t+b.

PERIODIC PROCESS GROUP: MARZULLO



Here, a set of devices can coordinate their actions based on a heartbeat.

FAULT-TOLERANT REAL-TIME DDS



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

THE CASD PROTOCOL SUITE (NAMED FOR THE AUTHORS: CRISTIAN, AGHILI, STRONG, DOLEV)

Also known as the "Δ -T" protocols, solves all of these problems!

Goal is to implement a timed atomic broadcast tolerant of failures

- First, they looked at crash failures and message loss.
- Then they added in more severe scenarios like message corruption (the so-called "Byzantine" model).
- To make this feasible, they assumed limits on how many faults occur.

EXAMPLES OF THE ASSUMPTIONS THEY MADE

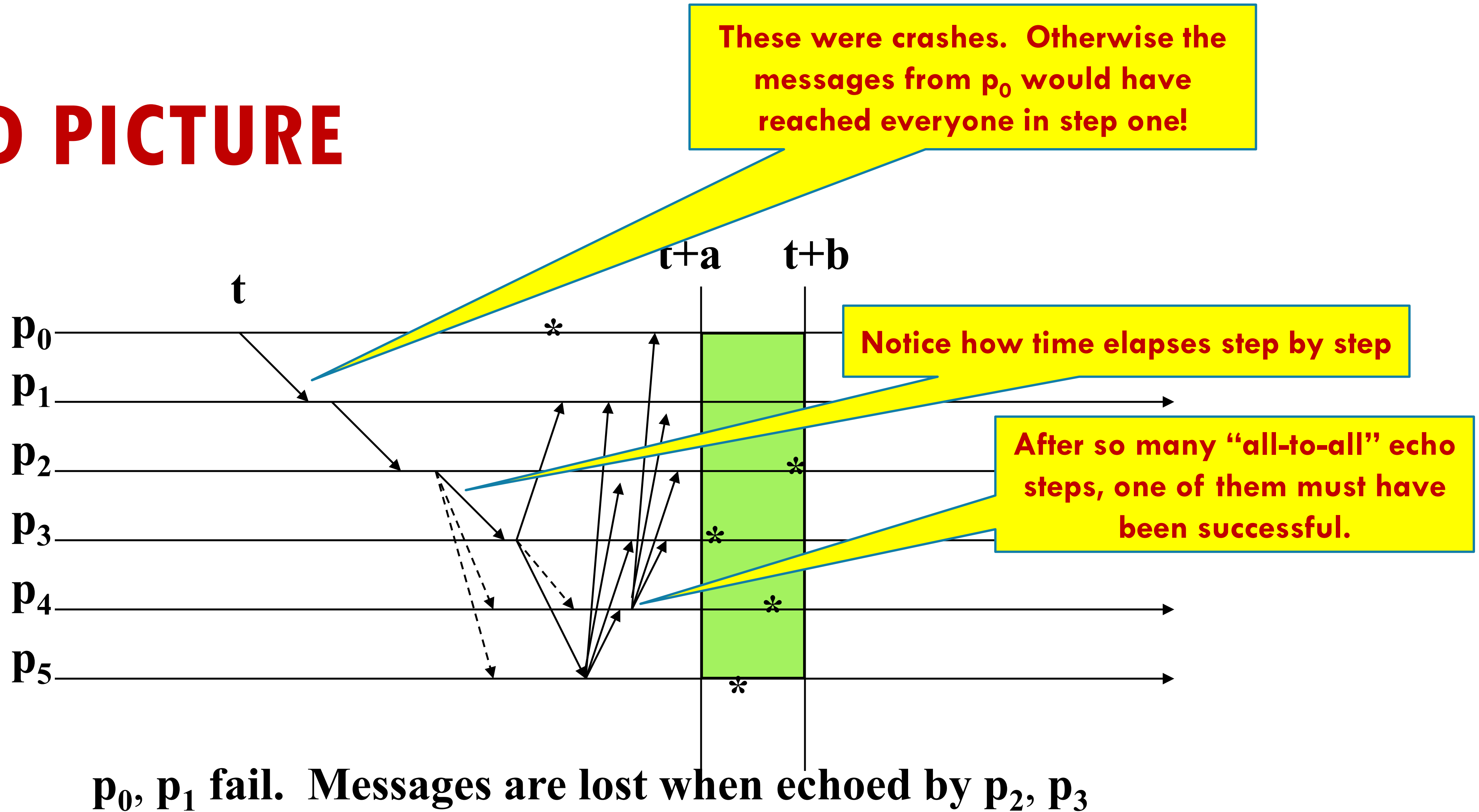
Assumes use of clock synchronization, known clock skew limits.

Sender timestamps message, then sends it. Could crash and not send to some receivers: they can only tolerated a few crashes of this kind.

Recipients forward the message using a flooding technique (each echos the message to others). Crashes here count towards the "crash failure" limit.

Wait until all correct processors have a copy, then deliver in unison (up to limits of the clock skew)

CASD PICTURE



IDEA OF CASD

Because we are assuming that there are known limits on number of processes that fail during protocol, number of messages lost, we can do a kind of worst-case reasoning.

Same for temporal (timing) mistakes.

The key idea is to overwhelm the failures – run down the clock.

Then schedule delivery using original time plus a delay computed from the worst-case assumptions

BASIC PROTOCOL IS VERY SIMPLE!

Every process that receives a message

1. Discards it, if the timestamp on the message is "out of bounds"
2. If this is a duplicate, no more work to do, otherwise, save a copy.
3. Echo it to every other process if it wasn't a duplicate.

Then after a determined delay, deliver the message at a time computed by taking the timestamp and adding a specific Δ (hence the protocol name).

WHERE DO THE BOUNDS COME FROM?

They do a series of "pessimistic" worst-case analyses.

For example, they say things like this:

- Suppose message M is sent at time T by some process.
- What is the earliest clock value that process Q could have when M first arrives at Q? What is the latest possible clock value?
- This would let them compute the bounds for discarding a message that "definitely was from a process with a faulty clock" in step 1.

THE ANALYSIS IS SURPRISINGLY DIFFICULT!

A message can jump from process to process so by the time it reaches Q, it may actually have gone through several hops.

Each hop contributes delay, plus at each hop some process ran through the same decision logic, and apparently, decided to forward the message.

If Q and R are correct, we need a proof that they will ultimately both deliver M, or both reject M, and this is hard to pull off!

WHAT DOES IT MEAN TO BE "CORRECT"?

In many settings, it is obvious when a process is faulty!

But with CASD a process is correct if it behaves according to a set of assumptions that cover timing and other aspects (for example, messages have digital signatures, and a process that damages a message is incorrect).

So there when we say "If Q and R are correct", this is a fairly elaborate set of conditions on their behavior.

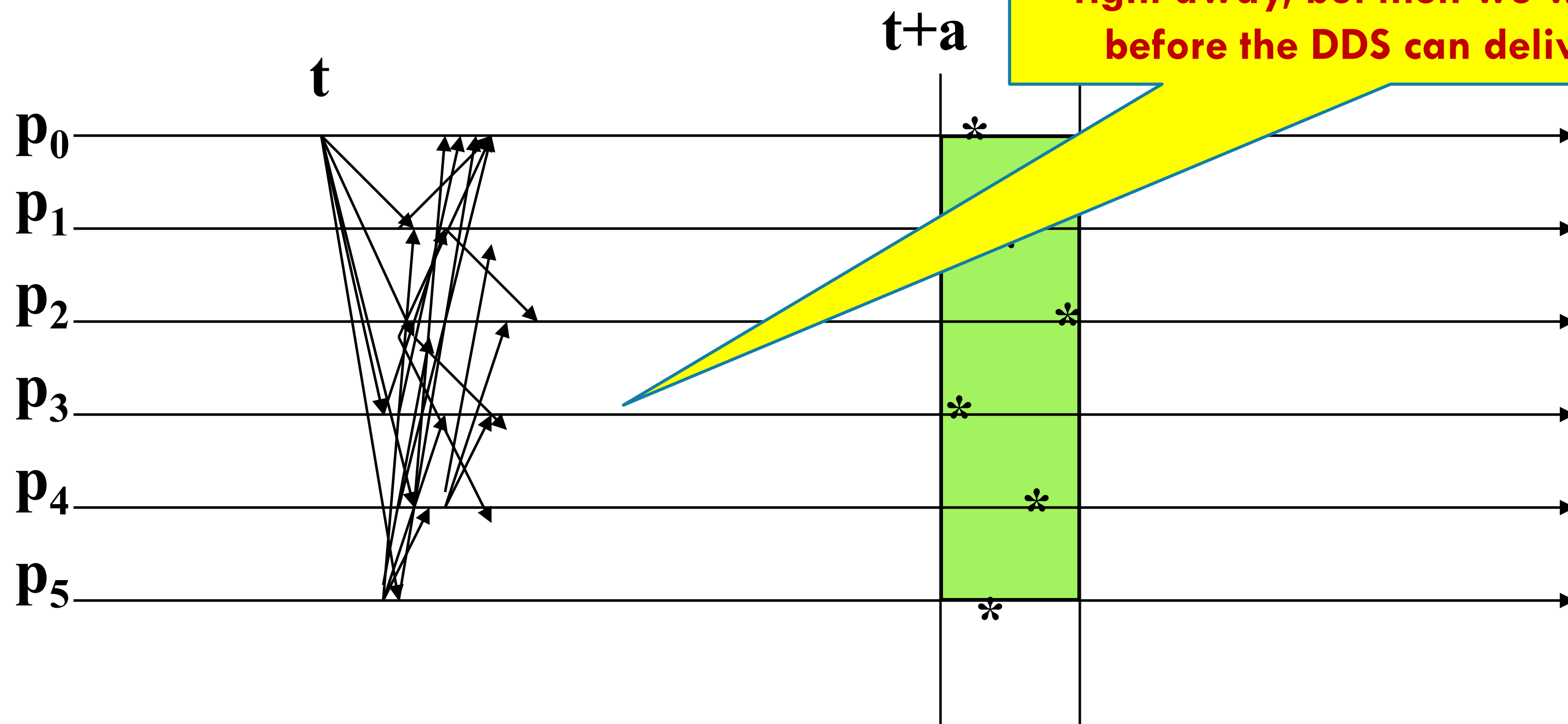
THE PROBLEMS WITH CASD

In the usual case, nothing goes wrong, hence the delay can is too conservative

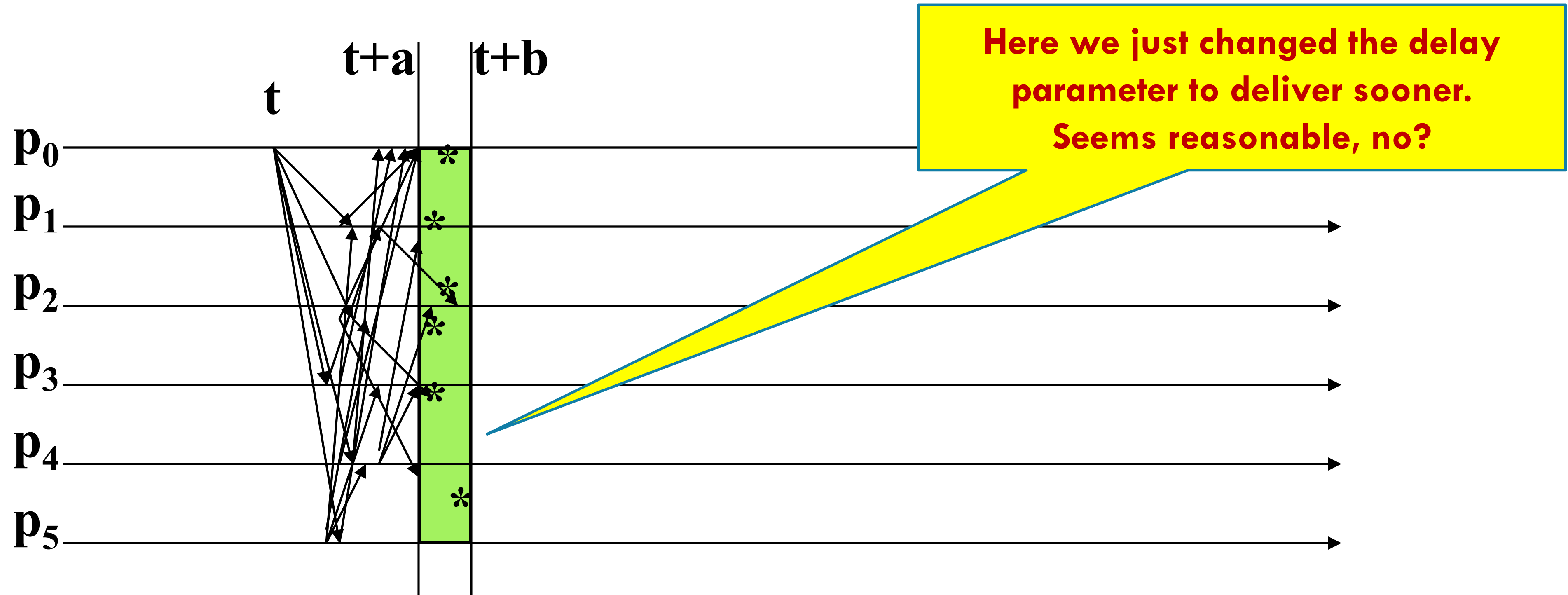
Even if things do go wrong, is it right to assume that if a worst-case message needs δms to make one hop, we should budget $n * \delta$ time for n hops?

How realistic is it to bound the number of failures expected during a run?

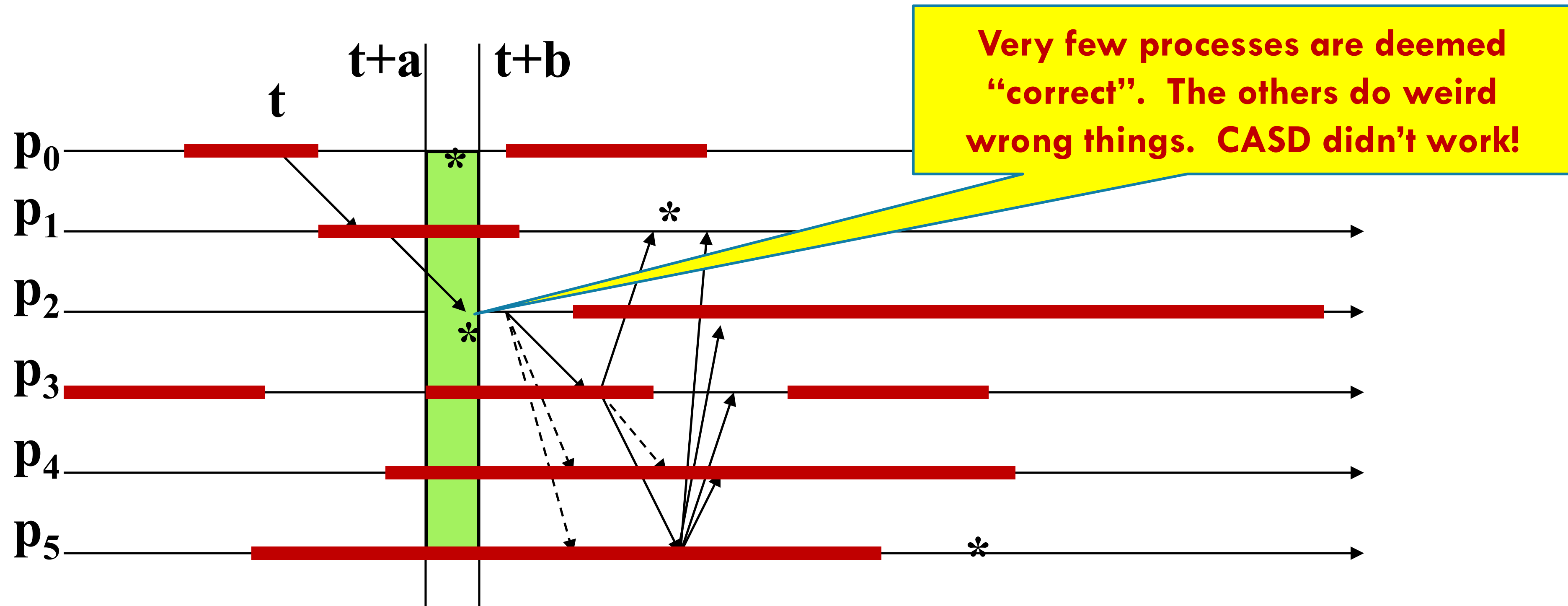
CASD IN A MORE TYPICAL RUN



… DEVELOPERS TUNED, AIMING FOR THIS



OOPS! CASD STARTS TO "MALFUNCTION"



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

WHY DOES CASD TREAT SO MANY PROCESSES AS FAULTY?

We need to think about what these assumptions meant.

Suppose CASD assumes that a message sent from P to Q will always arrive within delay δ, but then we set the limit, δ, to a very small value.

If δ=100ms, we would never have seen problems. But with δ=1ms, perhaps 10% of messages show up "late". This will be treated as if P or Q had failed!

MORE EXAMPLES

CASD has a limit on how long after a message arrives, Q can take to process it. If this limit is very low, scheduling delays make Q look faulty.

CASD has a limit on how many messages can be lost in the network.

CASD has a limit on how far the clocks on the computers can drift.

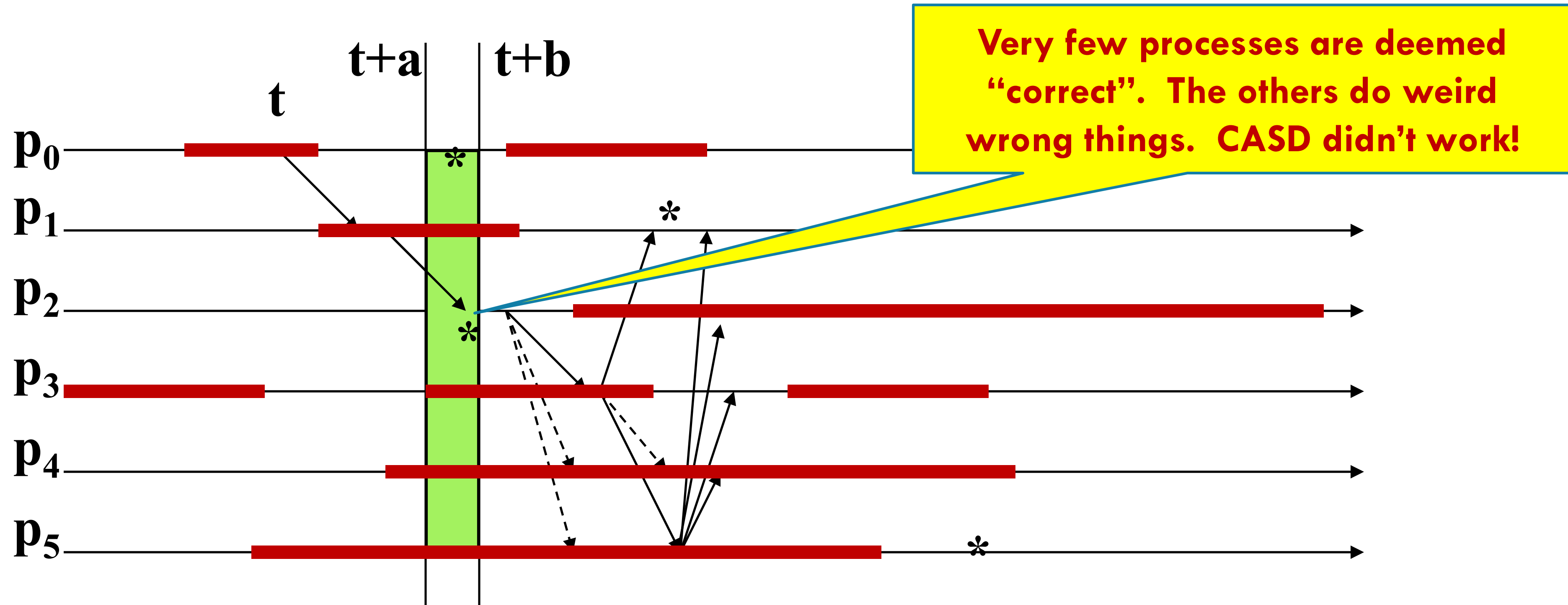
WHAT DOES FAULTY "MEAN"?

Since P and Q are still running, in what sense are they faulty?

- They won't be notified that CASD "thinks" of them as faulty.
- They don't have any local evidence they can rely on.
- In fact both think of themselves as healthy. They are normal programs.

Yet the CASD guarantees no longer apply because of these violations of the assumptions – CASD only promises atomicity and accurate timing if the model isn't violated.

CLOSER LOOK



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

CONSEQUENCE?

Only some of the drones learn the most current search plan.

Perhaps a “periodic event” triggers and only p_3 and p_5 act. Others don’t do anything, and p_4 acts but 3s late.

So clearly, running CASD “aggressively” didn’t work at all! But CASD with a 20s delay (a delay for which it works well) is useless!

LACK OF CONSISTENCY

In effect, CASD is ignoring several aspects of consistency, ones FFFS treats as critical elements of the behavior

- It is inconsistent about which processes are healthy and which have failed
- It isn't worried about causal consistency or consistent cuts.

And now we are seeing that without those properties, we can't build applications over the solution!

PROGRAMMING OVER CASD

The original idea was to configure CASD to be perfectly reliable.

But now we can see that with aggressive parameter settings, CASD becomes extremely unreliable.

How would we compensate for this risk in software?

CAN CASD BE FIXED?

Not in time for the project: The FAA gave up!

- They tried for two years. Eventually the NAS hired Fred Schneider to lead a study of the debacle.
- The study made a recommendation to drop the DDS idea and just build a database with transactions, but the FAA decided this was too big a change.
- In the end the whole project failed.

CASD ultimately illustrates the risk of wishful thinking!

VERISSIMO FOLLOW-ON WORK

In fact a subsequent paper from Europe did fix this issue:

AMp: a highly parallel atomic multicast protocol. In Symposium proceedings on Communications architectures & protocols (SIGCOMM '89). P. Veríssimo, L. Rodrigues, and M. Baptista. ACM, New York, NY, USA, 83-93.

The key to the solution centers on tracking membership and excluding faulty processes so that consistency is restored! Processes using AMp:

- They learn that they are excluded
- They won't be allowed to interact with healthy ones with prior "repair"

OTHER FEATURES OF aMP

The protocol itself is also somewhat better.

In AMp, the “uncertainty” in delay is what limits speed of the protocol, rather than “worst case delay”.

Since uncertainty is much smaller than worst case, AMp is a faster protocol. But Verissimo needed several years to develop and “polish” it.

COMPARISON: FFFS (OR OBJECT STORE) VERSUS REAL-TIME DDS

The DDS model puts time first, and in the IBM case, ignored consistency.

- Clearly a disaster!

AMp brings consistency in as a secondary guarantee.

- Seems to solve the problem, but never used on a large scale project.

FFFS puts consistency first, then speed. Treats time as a kind of "index"

- So here the real guarantee centers on consistency, not time.
- Used in a pilot project for the US Smart Power Grid (ISO NE, NYPA, NY ISO).

LESSON FOR IoT SYSTEMS?

PUT CONSISTENCY FIRST!

We need to look at these applications by starting with legitimate application requirements, then work our way down to problem statements.

Putting consistency first is a key to ending up with a working solution.

Temporal guarantees seem weaker, but this is better than offering a broken model to the user and leaving them to fix it!

MORE LESSONS

With early generation of IoT solutions, you can't just trust time, or sensor data, or platform components: you need to really learn how they work!

Otherwise your SEC monitoring tool won't realize that John is an insider doing illegal trades. Your drones will crash, and your self-driving cars will run over families of ducks.

Better take CS5412!

TODAY'S STATE OF THE ART?

We think Derecho offers a great mix of performance, fault-tolerance and consistency and could be a far better option than the ones listed before.

- Partly, this is because Derecho makes interesting protocol choices.
- And partly, this is because it leverages RDMA hardware accelerators and makes use of Optane NVMe for storage.
- With Derecho, we can guarantee "fast", but not "The system has worst-case delays under a long list of assumptions". Is this good enough?

TODAY'S STATE OF THE ART?

Similar remarks apply to Azure's storage layer, and to Cornell's $FFFS_{v1}$

In fact Derecho includes $FFFS_{v2}$: the key-value object store we talked about.

- $FFFS_{v2}$ will support a file API, so that legacy applications can run on it.
- For performance-critical tasks the key-value API is ideal, and the use of RDMA offers major speedups compared to TCP.

HOW CAN PEOPLE AVOID GIANT MISHAPS LIKE THE FAA DEBACLE?

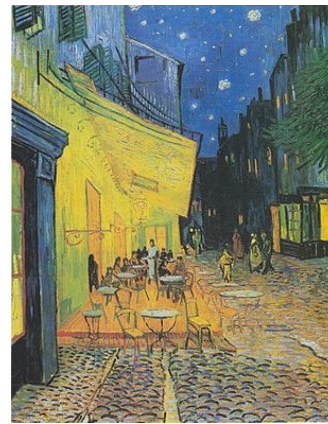
Please tell me what to do

A frustration with emerging technologies is that we do have to work through the details, because otherwise surprises can happen!

Some people much prefer stability in their work lives and are happy to learn one skill set, deeply, and then use it for years. They shouldn't jump into IoT!

But some people get a thrill out of living closer to the "bleeding edge"! For them, the uncertainty is a chance to be first to solve exciting problems.

… THE FRENCH GOT IT RIGHT!



FAA project took giant leaps all at once, and ended up making many "snap judgments".

- Every subsystem (and there were many) raised hard questions. Every team was on its own.
- Leadership acted as if "all this stuff is a bunch of no-brainers."
- In effect, the FAA assumed that the project as a whole was just a matter of making a wish-list of technical features, then paying a lot of money to IBM (and later, Loral).

The French had less money and were forced to evolve their systems through careful steps.

- The existing solution was inadequate, but they wanted to gradually replace it, not all at once.
- Leadership was really smart, required presentations & "red teaming" on every detail.
- Issues were caught during the design stage. Experiments were used to validate assumptions.

Red team: People tasked to "attack" a design, to identify flaws.

WHY DID THE FRENCH BET ON CONSISTENCY?

Their older style of air traffic control system bet on consistency. It was based on a fault-tolerant mainframe database.

- This wasn't scalable enough (Jim Gray's points!) but still, was giving a safe solution at the scale it could handle.
- So they were already used to a "consistency first" model. Rather than jump to demand "guaranteed real-time", they preferred "keep what works, but fix the scalability and performance issues."
- The technology they picked (it was a bit like Derecho, but an earlier version) had this exact mix of properties.

CONCLUSIONS?

If we look at the technology needs of IoT applications, we quickly run into genuine choices. Often it can seem as if we lack really clear answers.

Standard Azure cloud won't be the whole story for Azure IoT, but Azure IoT is an evolving platform and it builds on things that work very well.

A step by step approach that puts consistency first prevails.