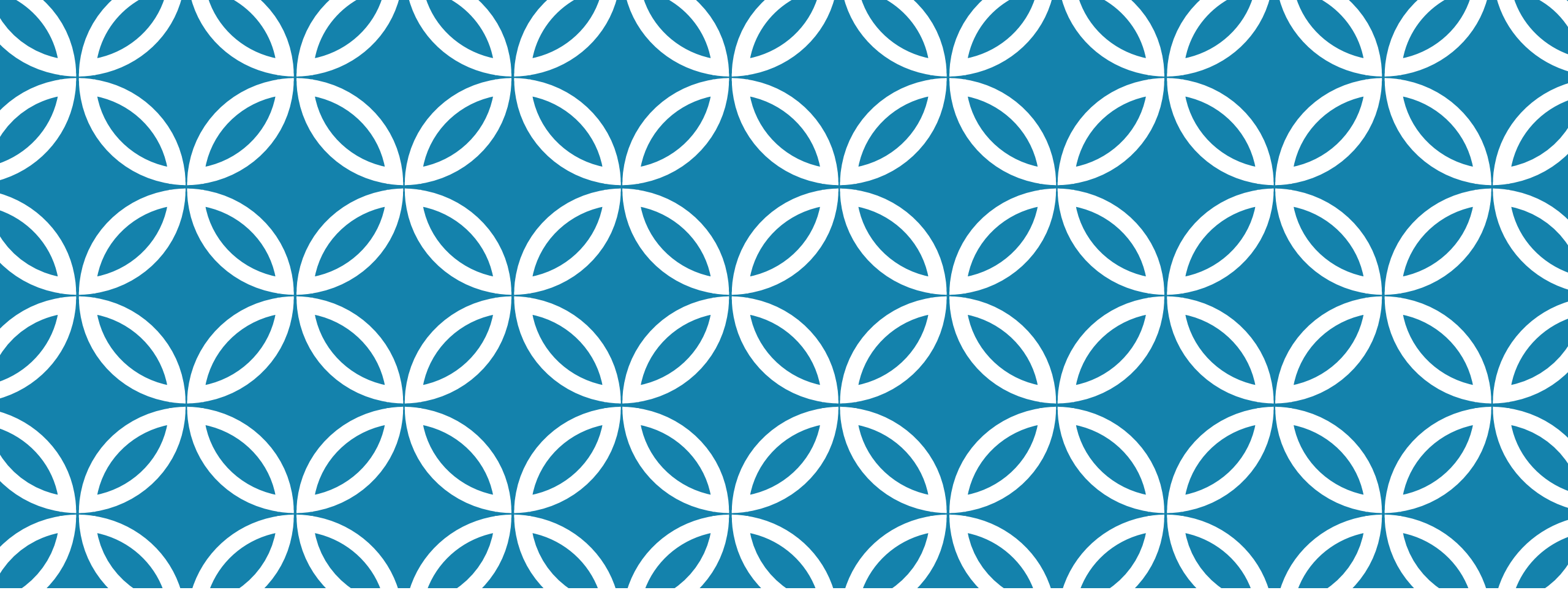


HOW SYNCHRONIZED TIME CAN YIELD INCONSISTENT BEHAVIORS

Professor Ken Birman

CS4414/5416 Lecture 13

IDEA MAP FOR TODAY'S LECTURE

Reminder: Thread Concept

Lightweight vs. Heavyweight

Thread "context"

C++ mutex objects. Atomic data types.

The monitor pattern in C++

Problems monitors solve (and problems they don't solve)

Deadlocks and Livelocks

Distributed Computing and Consistency

If you need consistency and overlook it, your whole project can fail!

Consistency Story in a Real Cloud-Styled (AI Enabled) Application

CONTEXT: ALL ABOUT TIME

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.

We heard about patterns such as atomic multicast, but obviously this is not the only choice. Some systems put time first

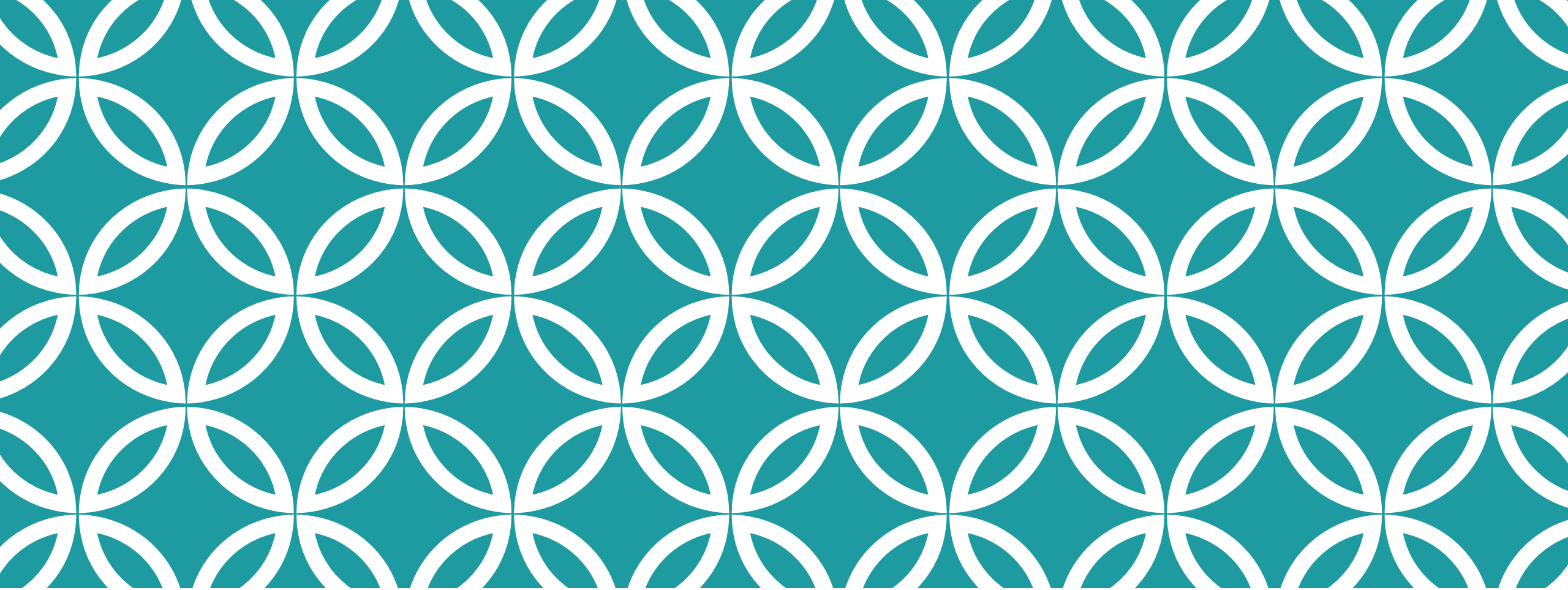
BUT WHAT SHOULD "TIME FIRST" MEAN?

As fast as possible, focusing on individual operations?

Highest possible throughput, with latency secondary or ignored?

Offer a service level delay objective (SLO)… Treat SLO misses as failures?

Tight coordination? Accuracy relative to GPS time? Low clock drift?



TIME-FIRST, TOPIC ONE

Sensors: Devices that measure something at some time

CLOCK TIME CREATES UNIQUE CHALLENGES

Clocks are never perfectly **accurate**, a term that refers to "truth"

Any clock will also **drift** over time, causing **skew** between two clocks

Accuracy relates to skew relative to a perfectly truthful clock (GPS is as close as we can get, but is pretty good!)

Precision relates to skew between pairs of correct clocks in the system.

WE OFTEN CARE MORE ABOUT PRECISION

It isn't important whether the system knows that today is Wednesday

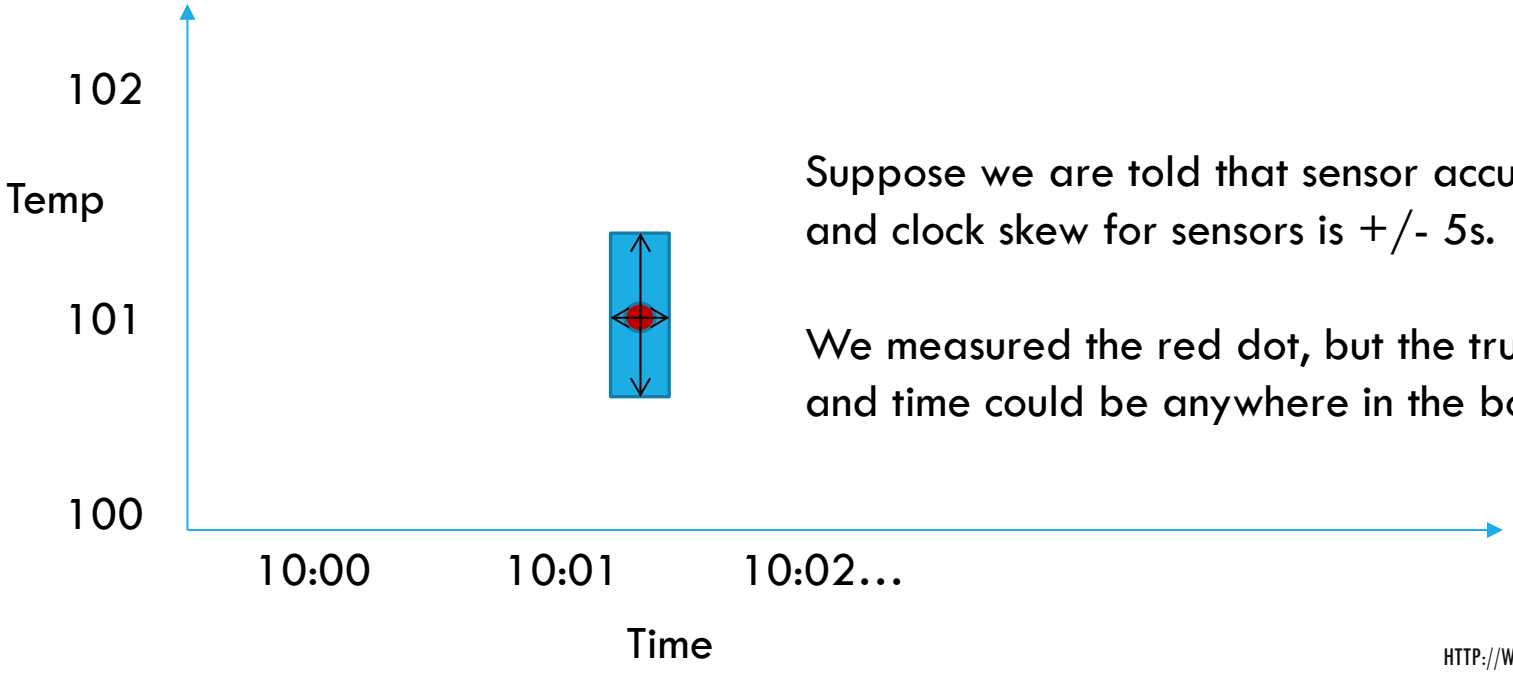
What matters more is that when process P on machine A tells process Q on machine B to take some action at some time, Q's action is consistent with what A expects. For example, "I will fill the reactor vessel at 10am. Heat slowly to 100.5C and maintain until 11pm, then turn the heat off."

This is a statement about **precision**… for this task, **accuracy** is secondary.

CLOCKS AND DEVICES HAVE BOUNDED ACCURACY

Always best to think of a time unit as reporting a bounding box

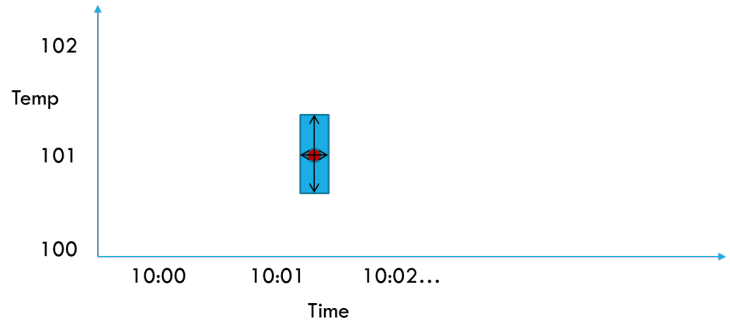
- The value is v $\pm$ ε, and was measured at time t $\pm$ δ.



Suppose we are told that sensor accuracy is +/- 1.5C, and clock skew for sensors is +/- 5s.

We measured the red dot, but the true temperature and time could be anywhere in the bounding box

POSSIBLY VERSUS DEFINITELY

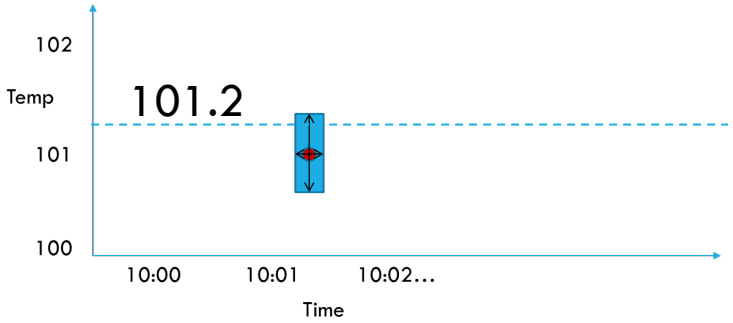


A value is “possibly” over a threshold if there is any portion of the bounding box that exceeds that threshold.

We cannot know for sure, but the potential exists that the value is over the limit.

A value is “definitely” over a threshold if the whole bounding box is over the threshold limit. There is no risk that it is under the limit.

THOUGHT QUESTION

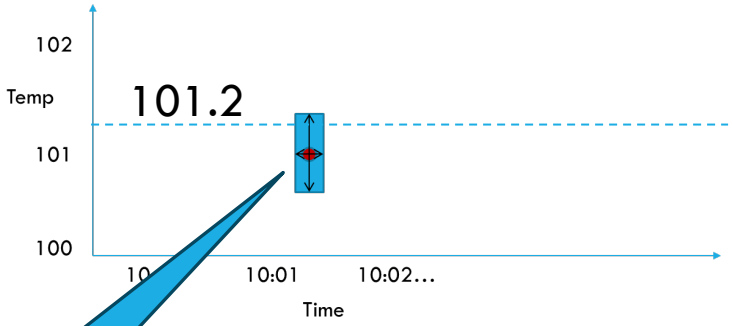


Suppose that we are managing a chemical reaction.

We want the reaction temperature to be *definitely* more than 100C, but also don't want it to *ever* exceed 101.2C, even briefly.

What do bounding boxes tell us about implementing this rule? How accurate would the sensor have to be to allow us to guarantee that we can follow it?

THOUGHT QUESTION



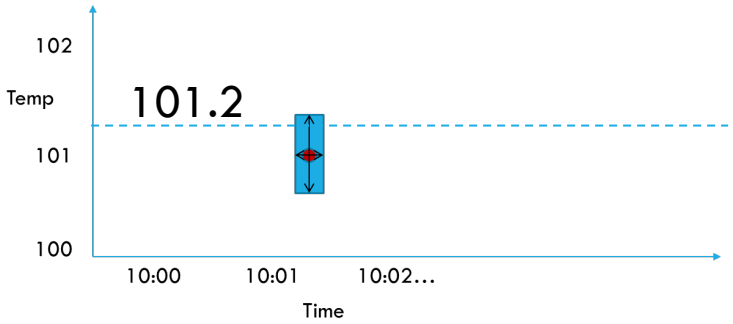
Suppose that we are managing a chemical reaction

We want the reaction temperature to be [illegible]*ely* more than 100C, but also don't want it to *ever* exceed 101.2 [illegible]en briefly.

We cannot rule out that it is over 101.2C

What do bounding boxes tell us about implementing this rule? How accurate would the sensor have to be to allow us to guarantee that we can follow it?

… AND THE ANSWER



For “definitely above”, consider the bottom of the box.

Unless the sensor is broken, if the bottom of the bounding box is above 100C, then the temperature is definitely above 100C

And for “definitely less than or equal to”, we want the top of the bounding box to be no higher than 101.2C

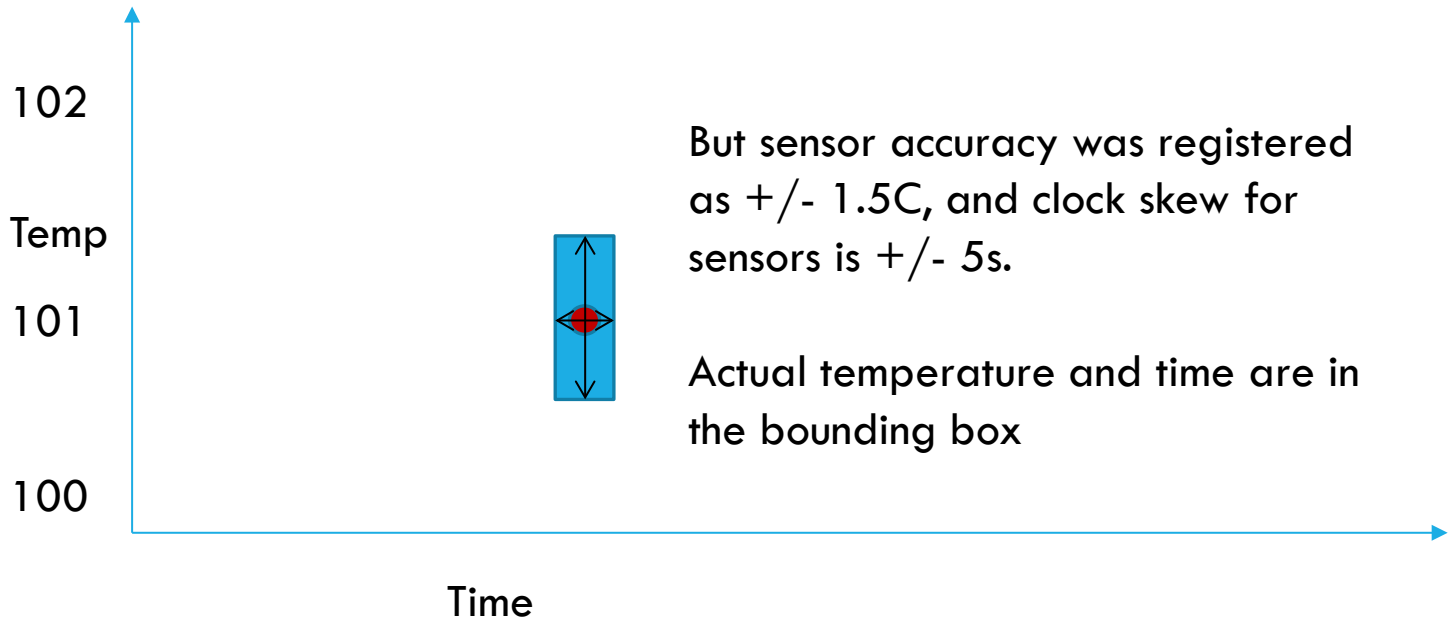
FAULTY SENSORS

Internet of things (IoT) systems often have redundant sensors but rarely try to clean up data.

Depending on what we know, we sometimes could exclude a bad sensor value and focus on the intersection of the good values.

But without enough certainly we might not know for sure which sensor is bad. Then the default is to just keep all the data and let the AI system make sense of it.

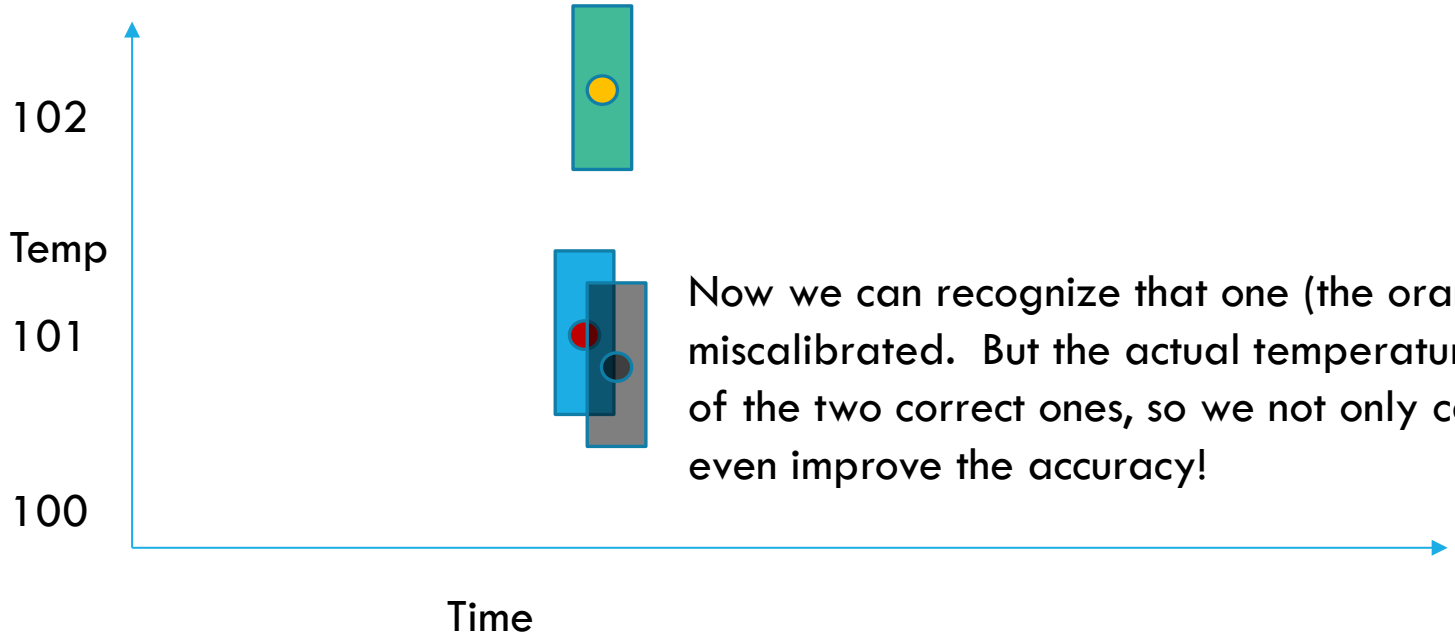
IDEALIZED PERFECT SENSOR



But sensor accuracy was registered as +/- 1.5C, and clock skew for sensors is +/- 5s.

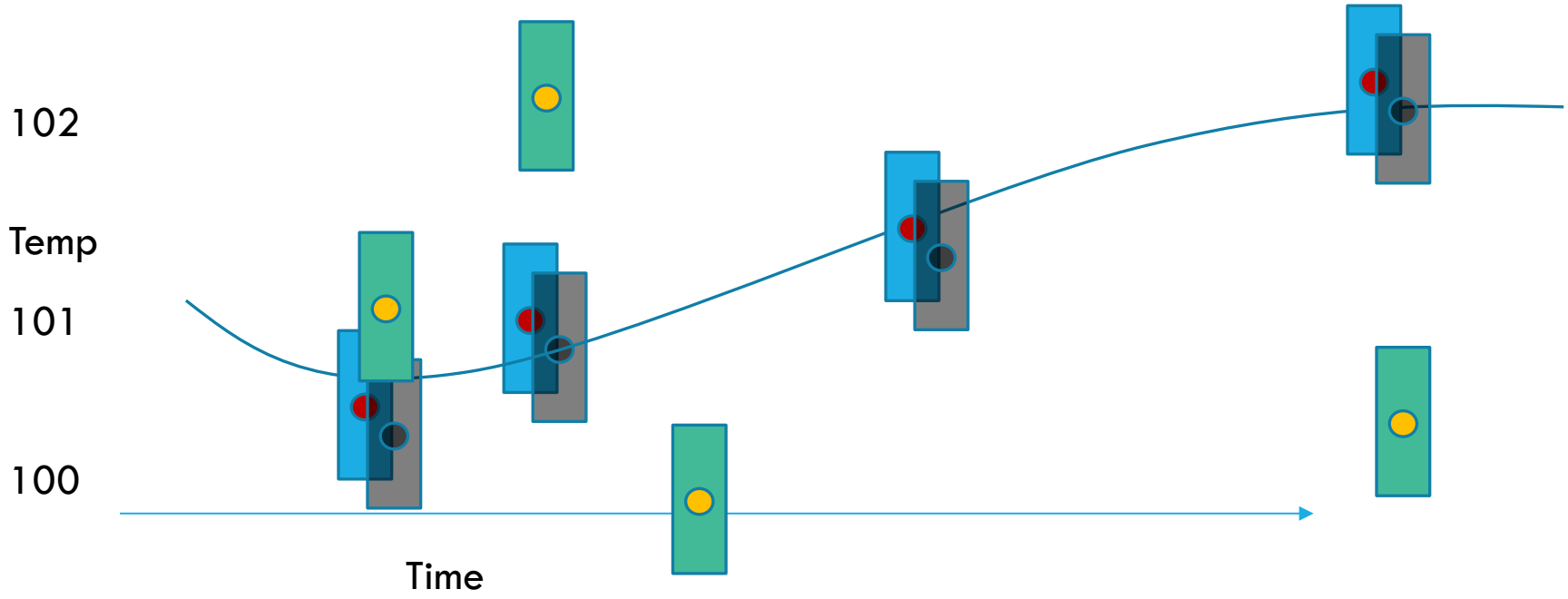
Actual temperature and time are in the bounding box

FAULTY SENSOR SCENARIO

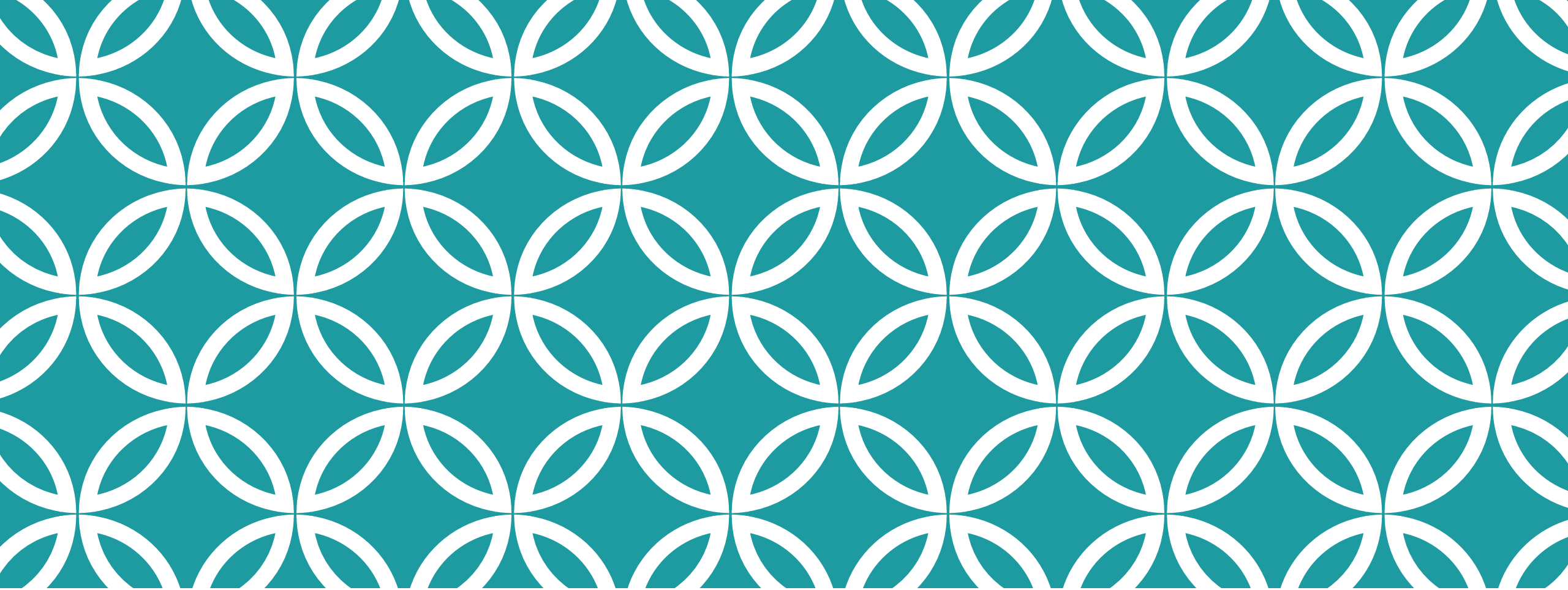


Now we can recognize that one (the orange one) is faulty or miscalibrated. But the actual temperature must be in the overlap of the two correct ones, so we not only can figure this out, we can even improve the accuracy!

FAULTY SENSOR SCENARIO



Times-series data would let an AI realize the green/yellow sensor is faulty.



MORE FUN WITH TIME-FIRST SYSTEM DESIGNS

Air Traffic Control

AIR TRAFFIC ARCHITECTURE

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

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

Real-Time Guarantee: Notifications get through within delay Δ, even if something crashes!

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

Automated services

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.

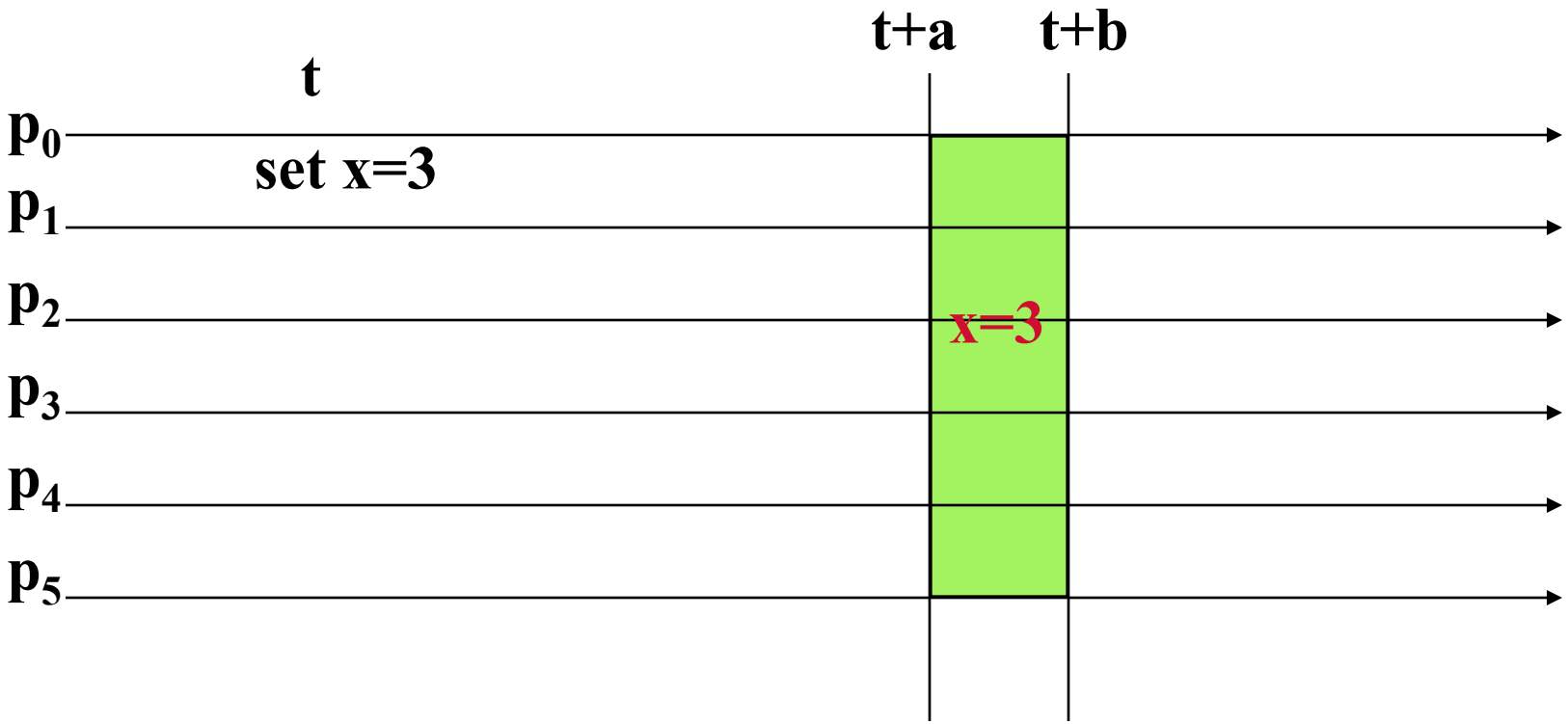
IDEAS IBM PROPOSED

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.

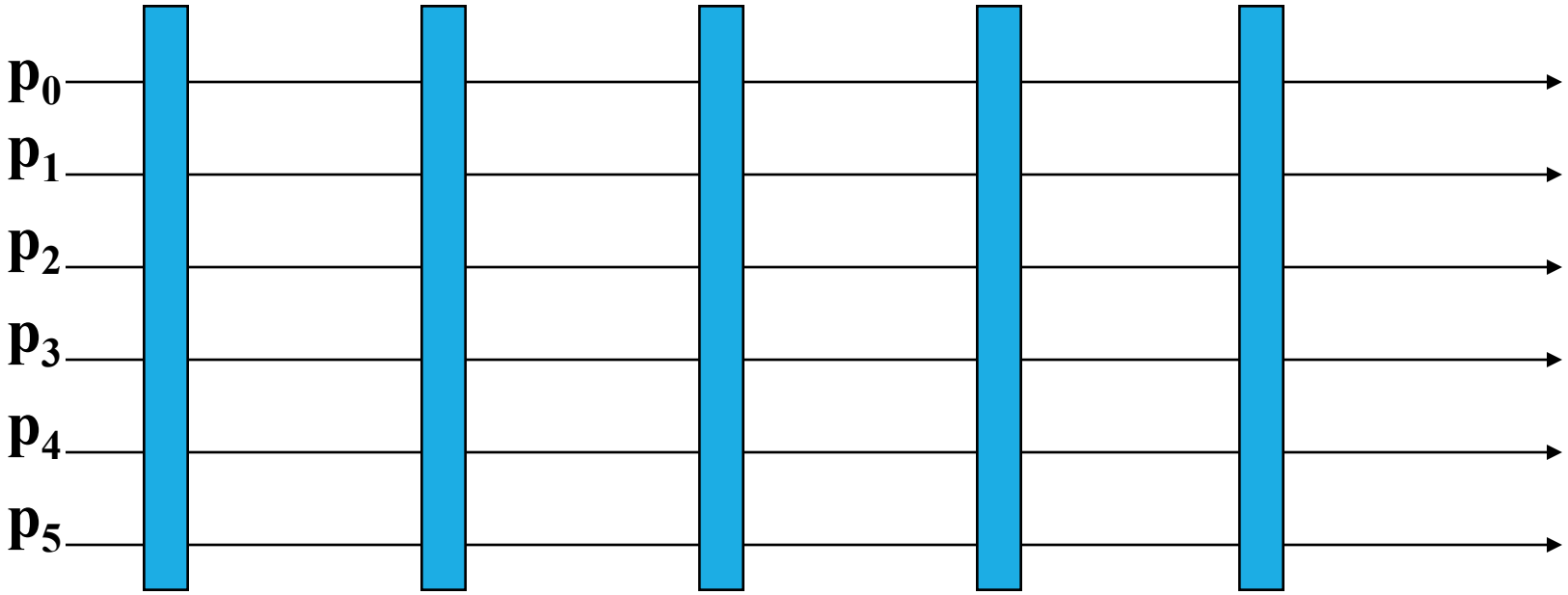
A real-time message bus (DDS) to report new events in a temporally consistent way to processes interested in those events.

A REAL-TIME DISTRIBUTED SHARED MEMORY



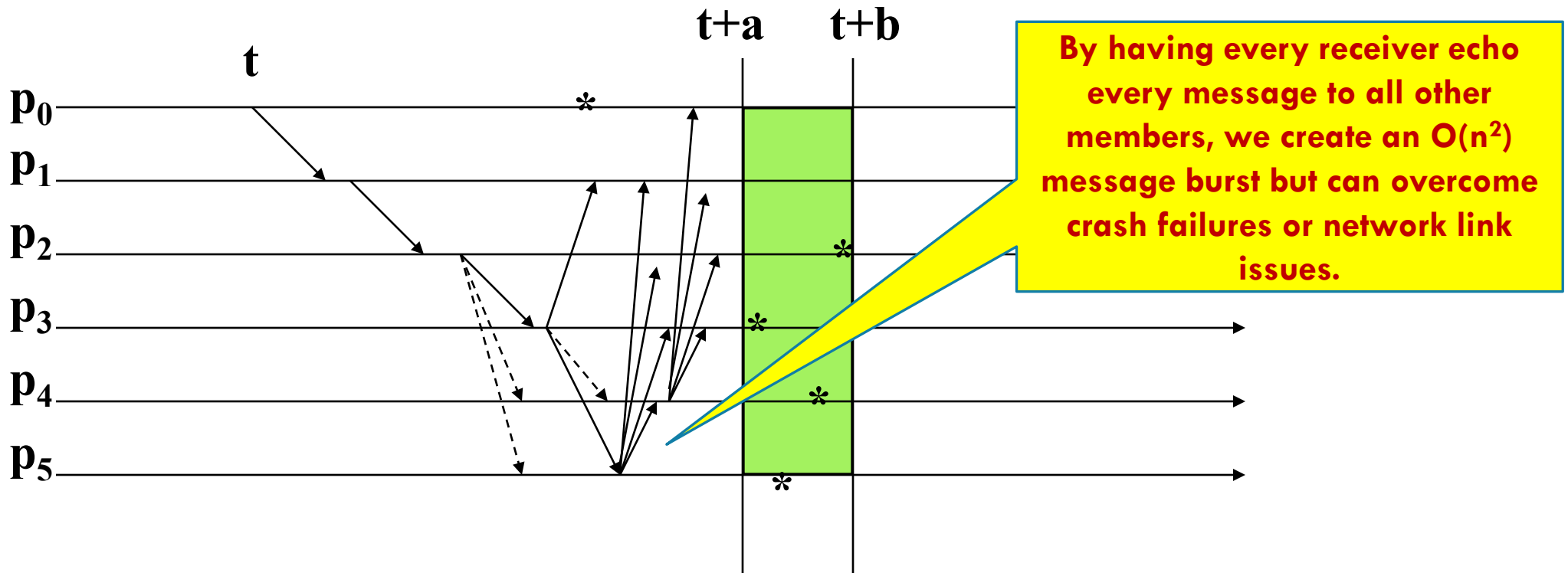
Think of "x" as a key and "3" as the new value. Here x=3 becomes visible within a time window between t+a and t+b.

PERIODIC PROCESS GROUP: SKEEN



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)

The paper introduces the Δ-T Atomic Broadcast

Goal is to implement a timed multicast that is 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

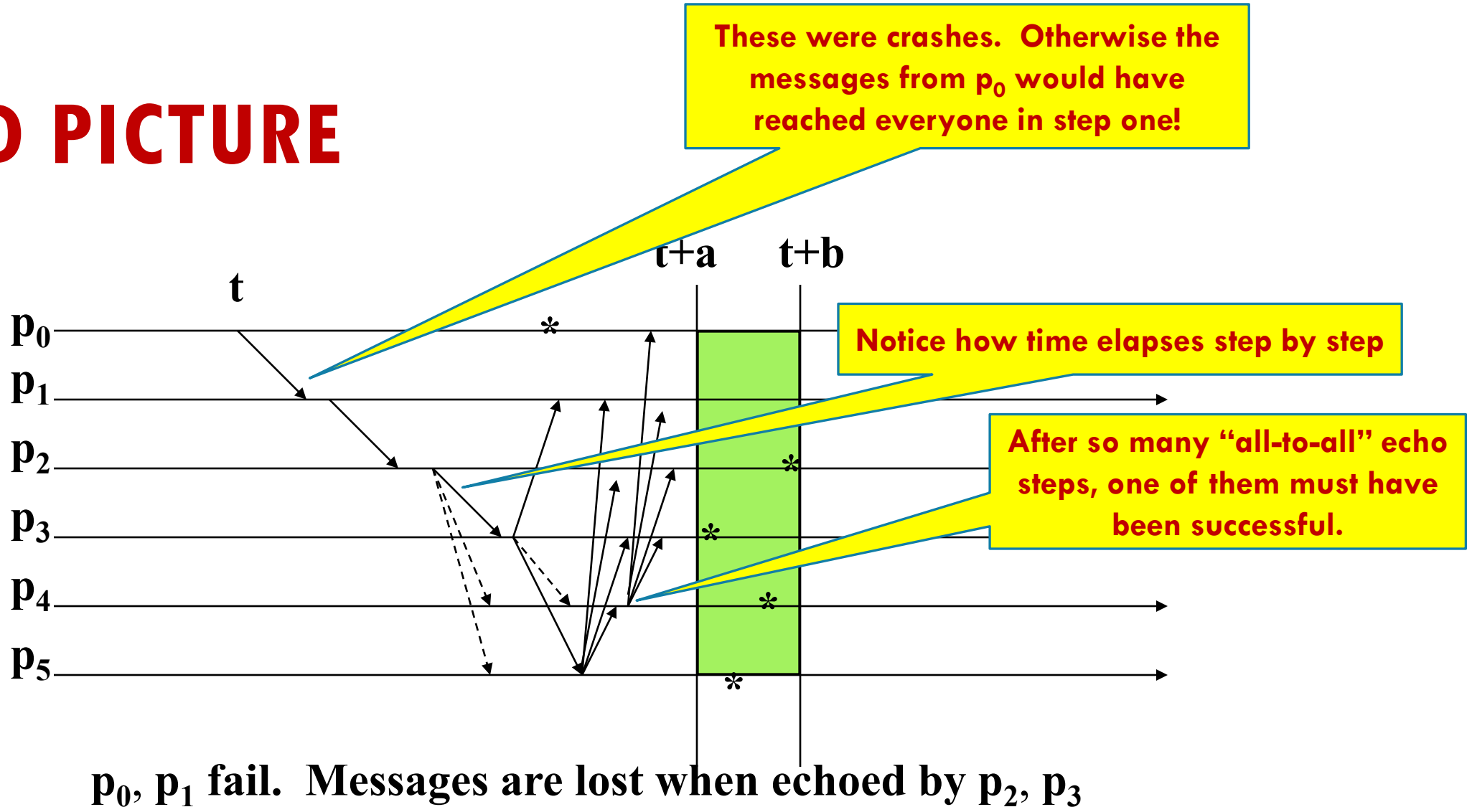
Assumes use of clock synchronization, known clock skew limits.

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



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

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:

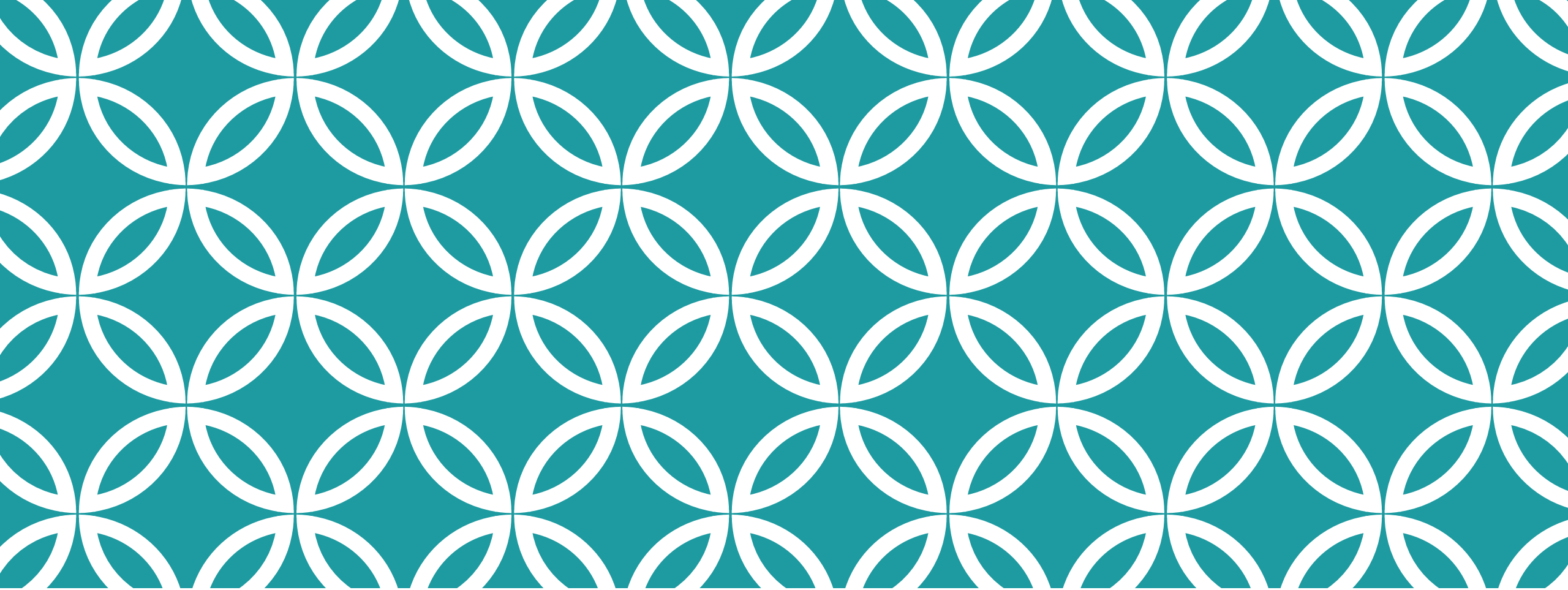
- Suppose message M is sent at time T by some process.
- What is the earliest clock value that process Q could have when M first arrives at Q? What is the latest possible clock value?
- This would let them compute the bounds for discarding a message that "definitely was from a process with a faulty clock"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



SO… IS CASD THE IDEAL PROTOCOL FOR ATC SYSTEMS?

Air Traffic Control

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

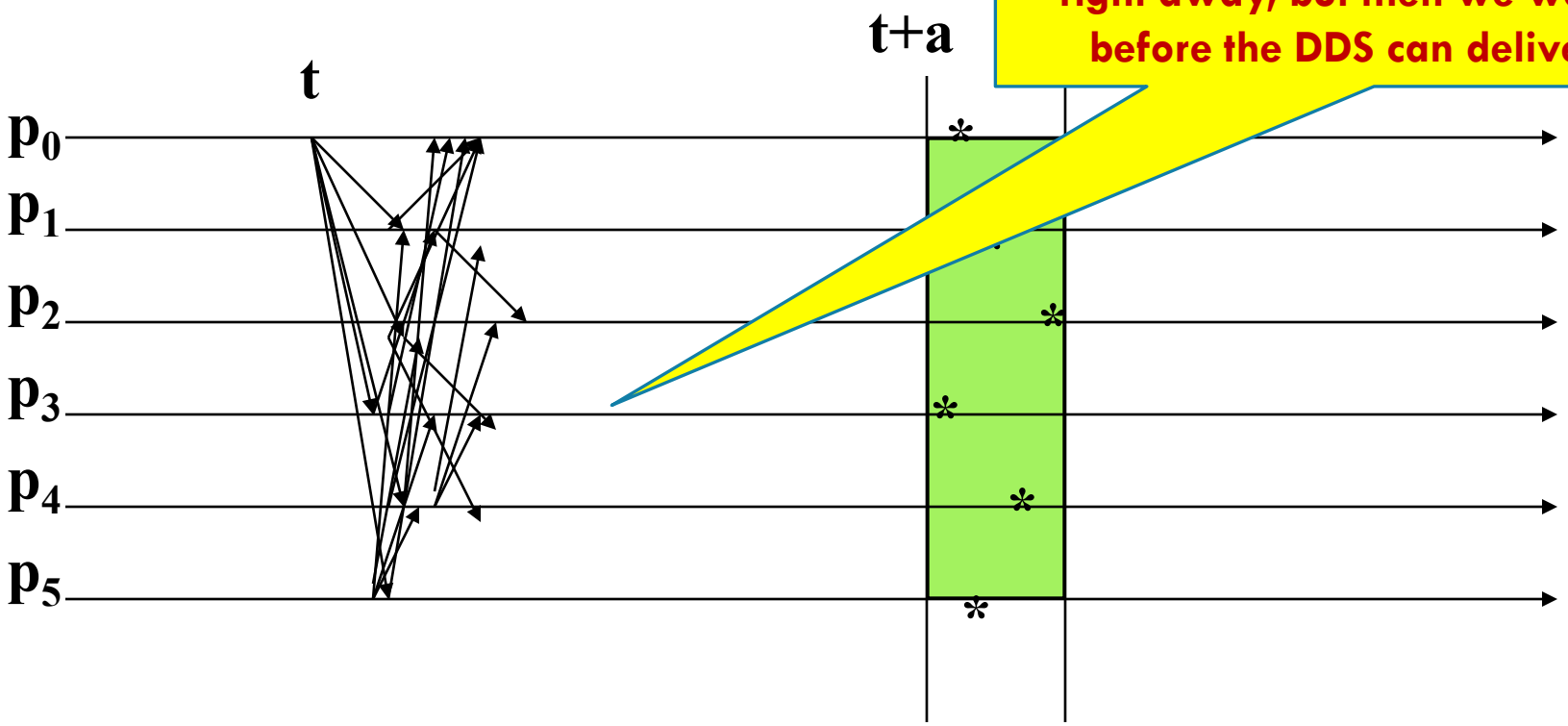
THE PROBLEM WITH CASD

In the usual case, nothing goes wrong, hence the delay 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 * δ time for n hops?

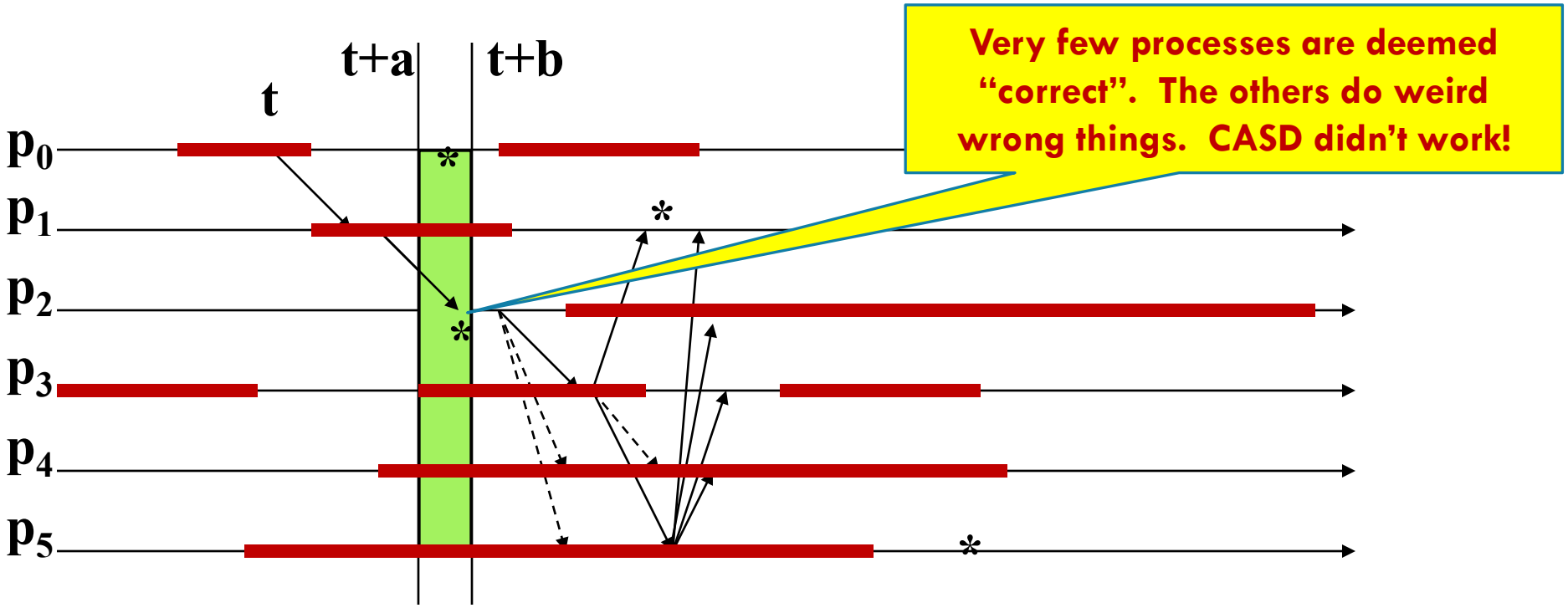
How realistic is it to bound the number of failures expected during a run?

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.

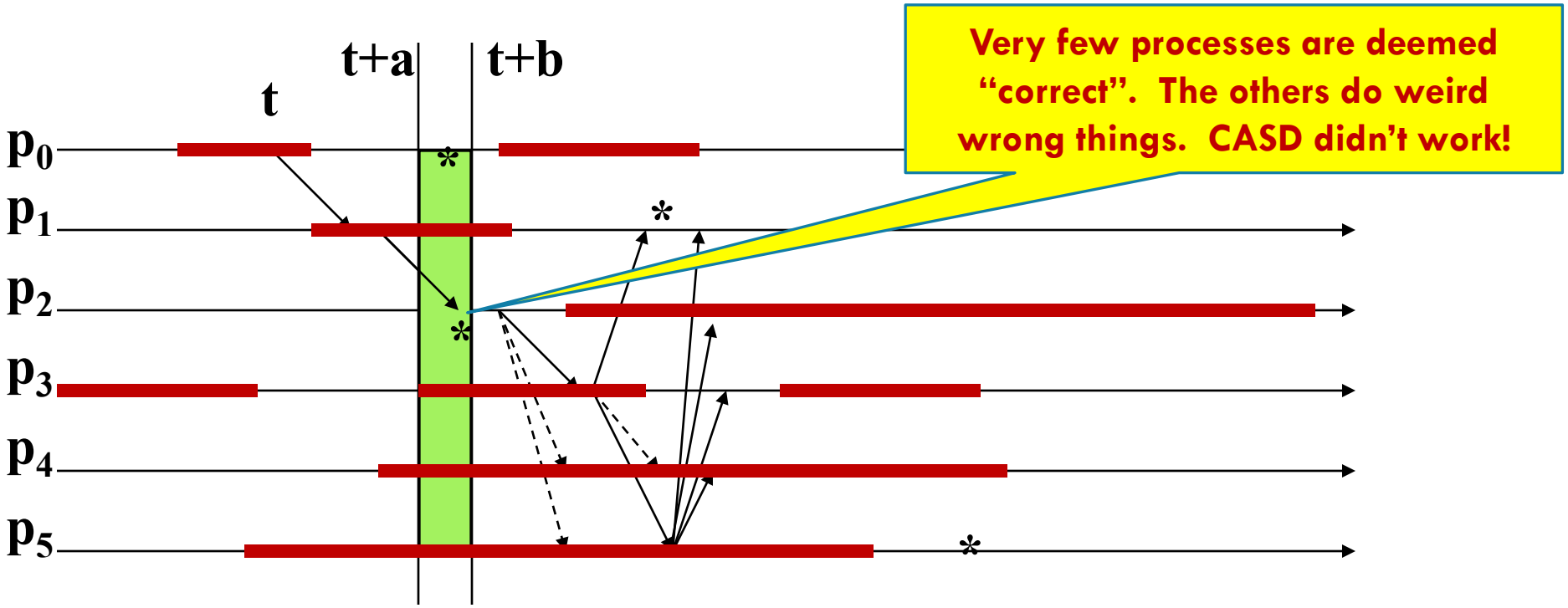
WHAT DOES FAULTY "MEAN"?

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



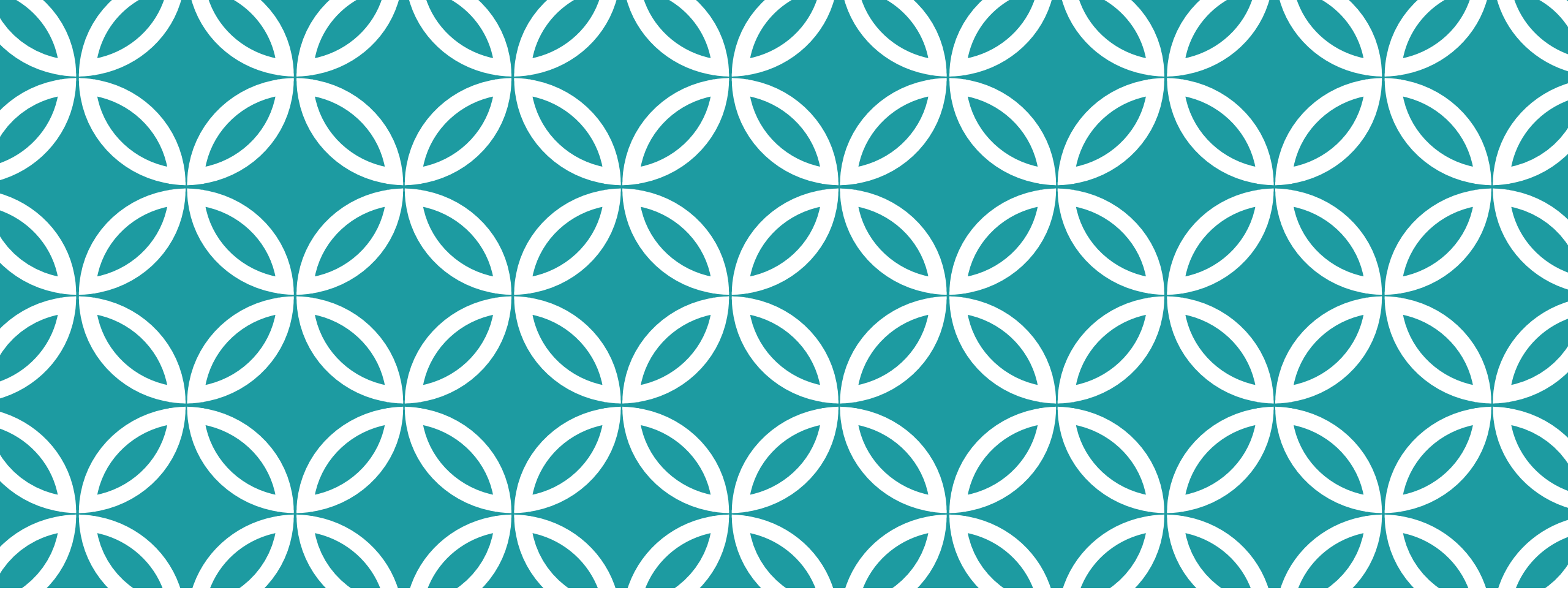
all processes look "incorrect" (red) from time to time

CONSEQUENCE?

Only some airport management units and ATC controllers see each update

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!



HOW DO CASD LIMITATIONS PLAY OUT IN A REAL ATC SETTING?

Air Traffic Control

BACK IN 1994 OR SO, IBM PROPOSED TO BUILD THE NEW FAA SYSTEM

They wanted to use ideas like periodicity and ΔT-Shared memory and the CASD ΔT atomic multicast as the lowest level infrastructure.

On this they would build basic services for tracking airplanes and runway activity, coordinating handoffs from controller to controller, fault-tolerance, etc. The tools are like "library methods" and these services would use the libraries as their building blocks.

Then on top of that they could add what we would call AI microservices today.

IMAGINE BUILDING AN ATC SUPPORT APPLICATION USING CASD

Certain information is being replicated, such as the flight plans of airplanes or the updated state of our power grid.

You are building an application like the ATCconsole and it receives an update. The requirement is that every controller sees the same sequence of events and same background data, so they stay consistent

But due to these CASD issues, sometimes messages show up out of order, or duplicated, or are dropped. *And you have no way to know this happened.*

LACK OF CONSISTENCY

In effect, CASD is ignoring several aspects of consistency, ones atomic multicast treats as critical elements of its 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?

- They tried for two years. Eventually the NAS hired Fred Schneider to lead a study of the debacle and to recommend a fix.
- Instead, the study recommending dropping the CASD-based DDS and use a database with transactions. In the end billions were lost.
- Fred's advice was accepted in the follow-on project (which also took advantage of GPS clocks and tracking)
- Meanwhile Ken designed a solution for France using a system like Derecho. They still run it now.

WHAT WENT WRONG??

Fundamentally, the IBM effort stumbled because these particular building blocks provably did achieve their specified behavior, but we lacked a methodology for using them to implement those kinds of services.

Fault tolerance and consistency were the biggest problems. Nobody could figure out how to use those building blocks to build services guaranteed to offer safe ATC guidance for planes and runways, and to be fault tolerant

WHAT WENT WRONG??

With much weaker time bounds it might have worked – then the building blocks would have been "almost always perfect". This is how the leaders imagined it.

To get highly reliable behavior, protocols like CASD needed big ΔT. **But ATC controllers rejected the big ΔT.** If something important happens, it isn't safe to delay 30s before notifying the air traffic controller.

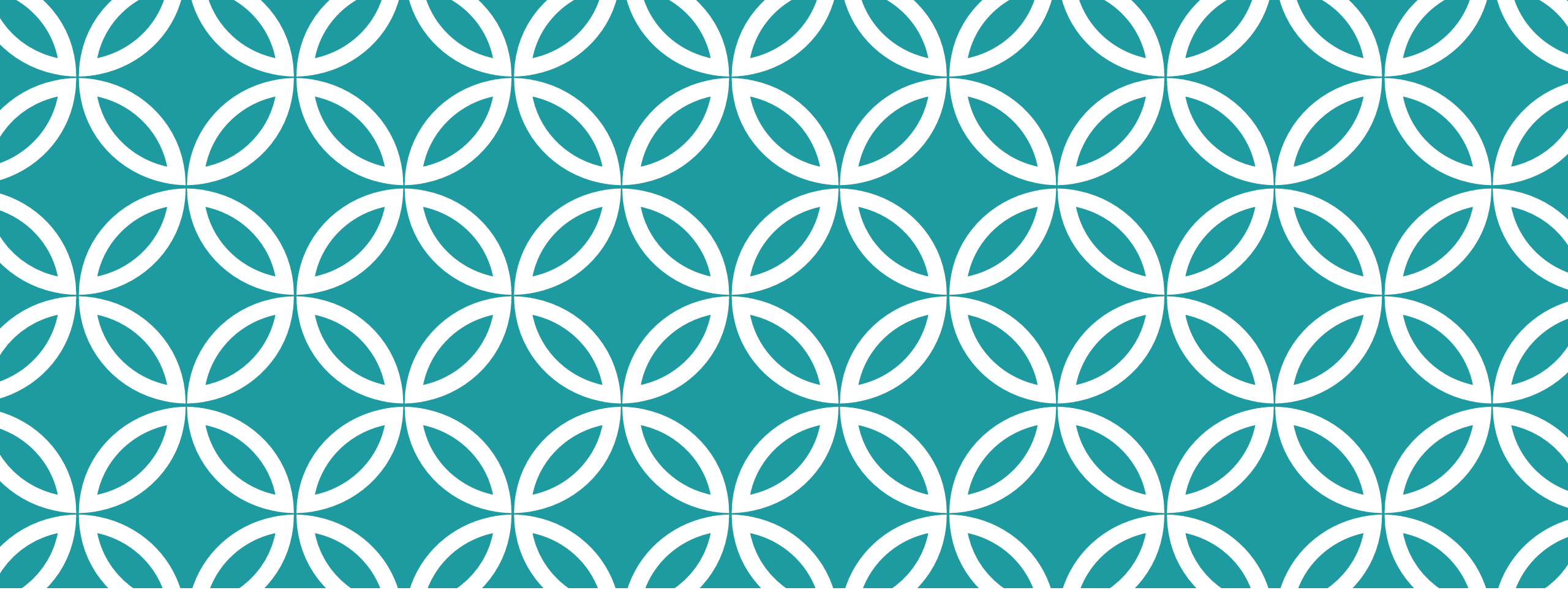
But with aggressive timing settings (small ΔT) inconsistency was too frequent: IBM's building blocks just stopped being useful.

BUILDING A CASTLE ANALOGY

It is easy to build a castle with lego building blocks.

But suppose legos just aren't an option and your team decides to construct building blocks from damp sand packed into molds shaped like legos?

These sand-lego pieces may look right,
but crumble under weight and wash away easily.

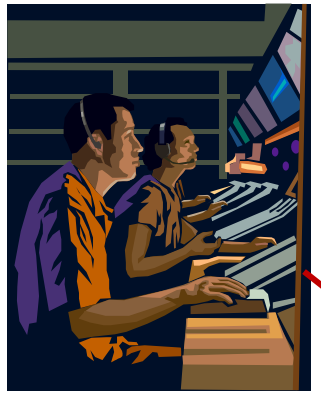


DOES THE ATC EXAMPLE HINT AT ISSUES IMPORTANT TO ML?

Machine Learning and Time

DID YOU KNOW THAT MODERN ATC USES AI?

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



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

… Other ATC controllers see
real-time read-only notifications

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

Intelligent

Intelligent μ-service

WHAT IS AN INTELLIGENT MICRO-SERVICE?

… An AI component, abrieved “μ-Service” on my slides

… that wakes up when an ATC notification message relevant to it is published by the real-time DDS

… and reviews the impacted flight plan (and other plans) looking for an opportunity to optimize airport operations or to offer benefits to airlines

IT WOULD BE LIKE OTHER AI SOLUTIONS!

It would need to be trained to understand what is beneficial and what is considered to be problematic

We would also need it to understand priorities and other "notable attributes" the airlines might share through the flight plan, like "must not arrive late" or "many customers will transit via gates in terminal B"

There could be many, specialized in different functionality

THERE COULD BE MANY OF THESE SMART ATC MICROSERVICES!

What would be AI roles for such μ-services?

- Which of my passengers might miss connections?
- Could I purchase priority landing rights? How much would it cost?
- How will my flights be impacted by the approaching weather system?
- If the luggage handlers go on strike, which of my flights should I route to other airports?

THIS IS A TYPICAL FINDING!

Many ML services and AIs that use them encounter measurements of real-world properties (like temperature), using clocks with skew and accuracy limits (like our bounding box pictures).

And many need data replication for lots or reasons, as we saw in previous lectures, such as for fault-tolerance and to do parallel ML compute.

They may operate under time pressure too: the AI may need to show data to a person (like an ATC controller) or take an action within some time limit

CONCLUSIONS?

First, it seems clear from the ATC experience that consistency-first is a better methodology. Build a consistency service and make it fast, don't build a real-time primitive and then try to add consistency properties.

Next, be very aware that modern platforms **do not guarantee these properties** unless you use infrastructure tools that offer strong assurances.

So this is an area where ML will encounter the real world, and platforms will surely need to evolve to match their requirements!

THOUGHT AND STUDY QUESTIONS

In AI systems used for real tasks should we favor consistency or performance? Derecho offers both… is this valuable?

Are real-time deadlines the most important consideration, or should we focus more on proving some other aspects of the solution?

If the AI services are big and tricky to run within the time bound, can we do things to make them perform better?

MORE THOUGHT QUESTIONS

Batching is a great way to improve throughout: the ingress process of a system collects some number BSIZE requests, then sends them through as a single operation holding a list of requests to perform.

How would batching impact the distribution of delays (latency), as you vary the BSIZE parameter? In fact this question is nuanced because there are many factors to consider and they give different answers when considered separately.

MORE THOUGHT QUESTIONS

When you are brought on board as the performance engineering lead for a new project, knowing nothing, what would you want to learn before making any recommendations?

Suppose that you are hired to work on an AI for highway management (speed limits, rules for COV lanes, etc).

How would the issues discussed today enter into your requirements and design plan for that sort of AI?

AN OFFLINE DEBATE TOPIC: SERVICE LEVEL LATENCY AND MISS RATE OBJECTIVES

Suppose that you measure some system and discover that you can tune it for very low per-request delay (latency), but that this reduces throughout. The highest throughput has poor latency performance.

When would you opt for throughput even with this consequence? When would you opt for low latency?

Can an SLO perspective offer a third possible configuration option, or is it really just another word for low latency? If it is genuinely is a third possibility, when would you opt for it?