CS5412 / TIME-RELATED CONTENT (ENRICHMENT/REVIEW)

Ken Birman

Fall, 2022

RECAP: IOT SENSORS / ACTUATORS

Sensors are devices like thermostats. **Actuators** "do things", like turning on the air conditioner

An IoT device needs to live in some real-world place. Knowledge about that place is called "contextualization information".

IoT Hub has unique permissions for securely connecting to the device, and owns all subsequent firmware/software updates and communication with it.

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 10 seconds from now, Q’s action is consistent

- Like in our missile defense example

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

SENSORS HAVE BOUNDED ACCURACY

Always best to think of a sensor as reporting a bounding box

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

The Meta system taught us how to

- Use sensor intersection to (sometime) eliminate bad values, like if 2 out of 3 sensors agree but one is flakey. But if all 3 overlap we can't know which are correct, and can't eliminate any of them.
- Also how to interpret statements like "if v > x, do something". Meta has two forms of if: "if definitely", and "if possibly".

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.

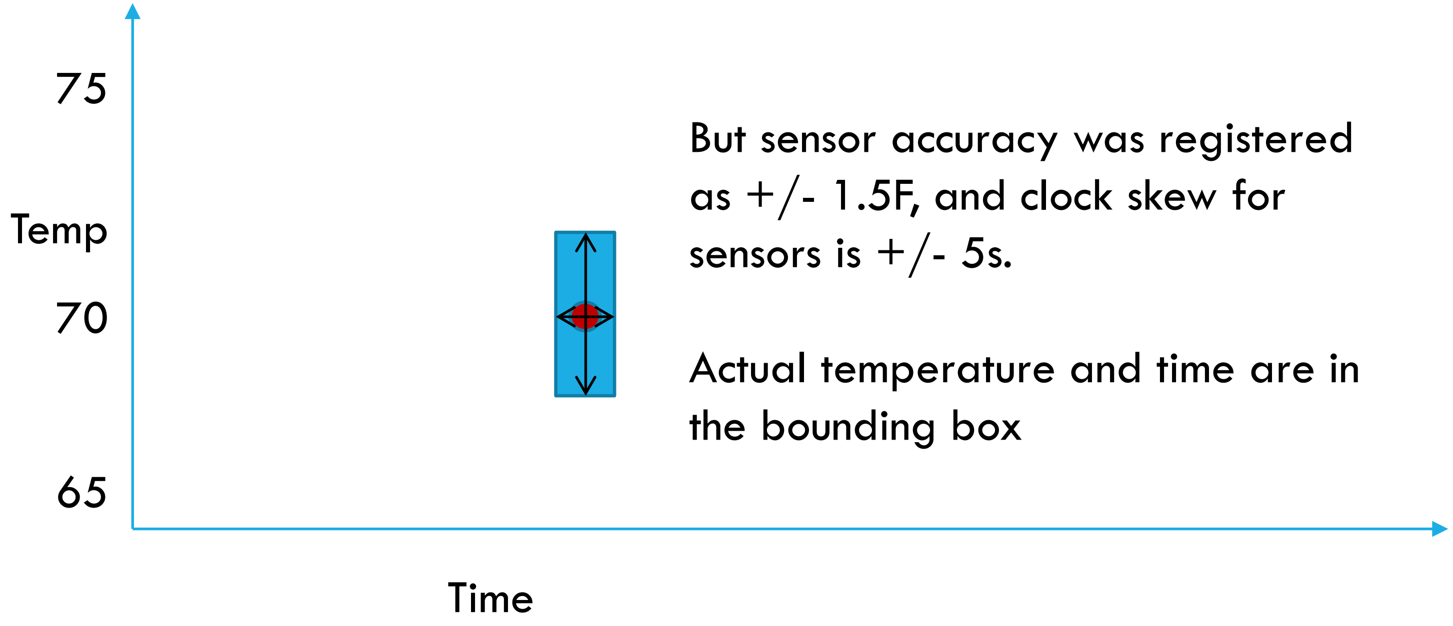
THOUGHT QUESTION

Suppose that we are managing a chemical reaction. And we use one non-faulty sensor, no need for backups and multiple-sensor-agreement

We want the reaction temperature to be *definitely* more than 100C, but also don't want it to *ever* exceed 101C, 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?

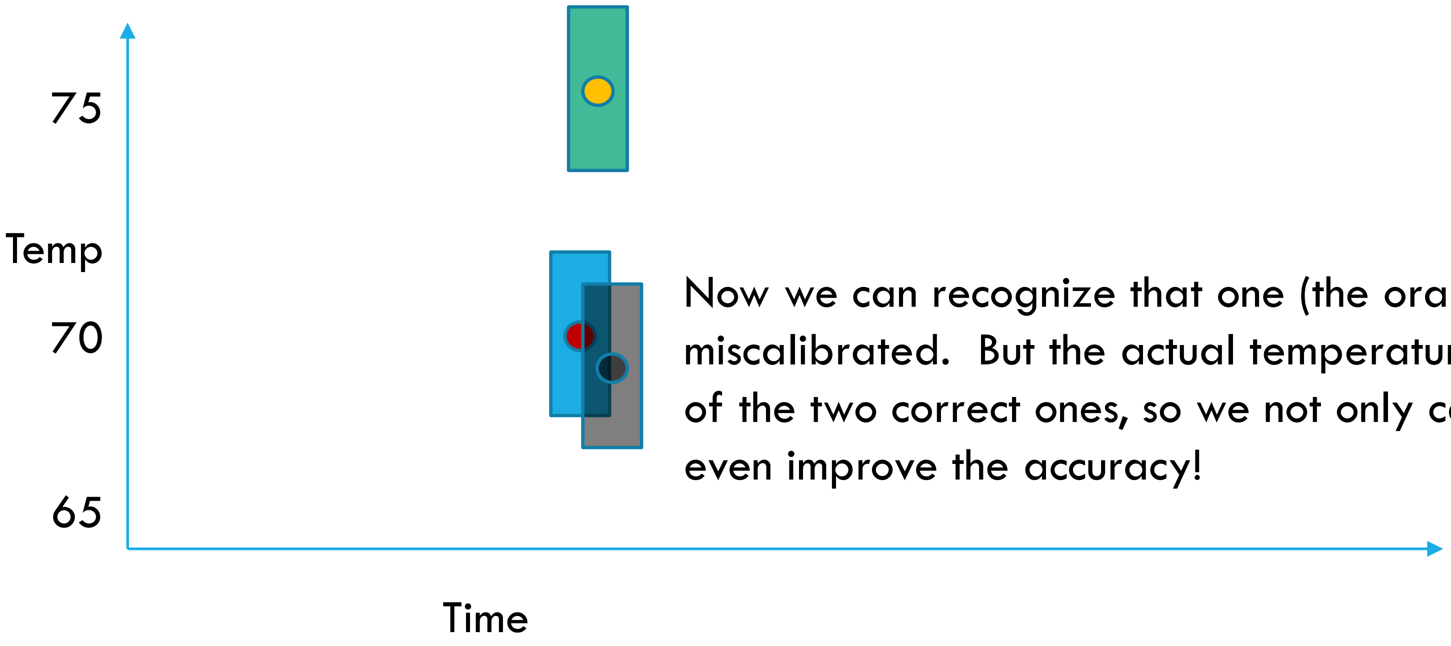
REMINDER: SENSOR “OVERLAP” CONCEPT



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

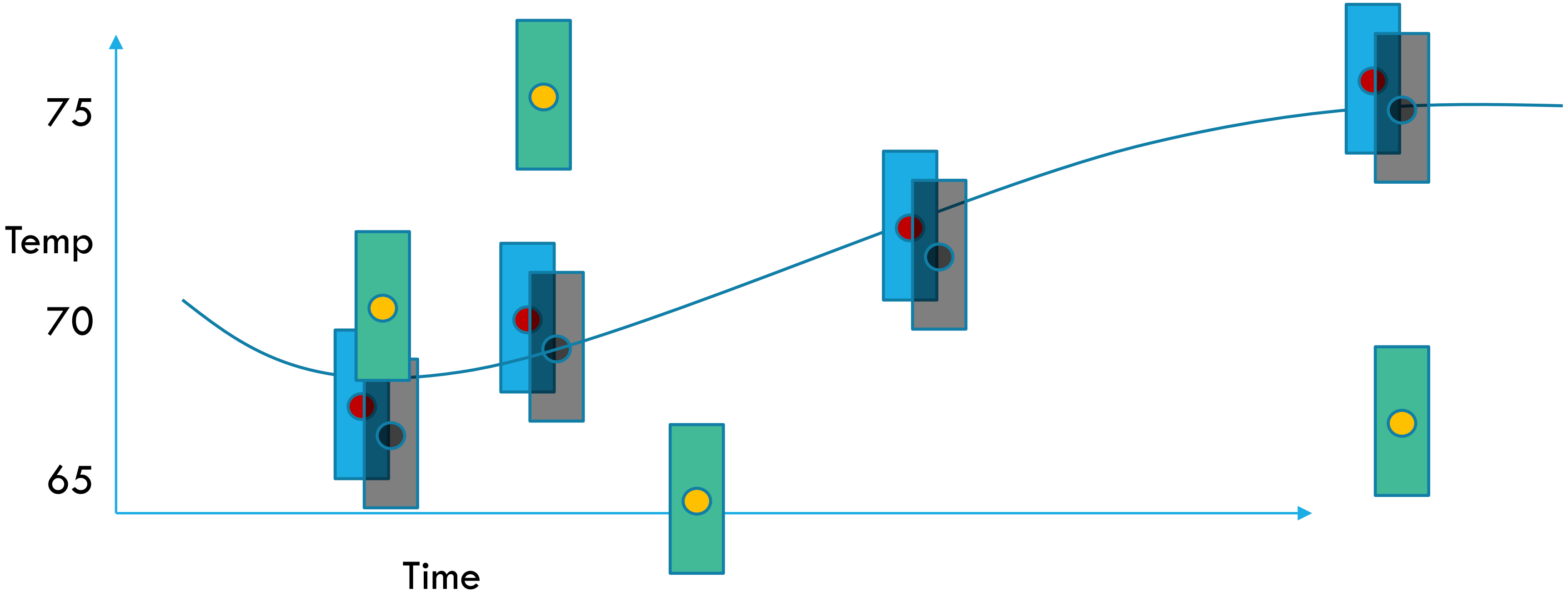
Actual temperature and time are in the bounding box

REMINDER: SENSOR "OVERLAP" CONCEPT



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!

REMINDER: SENSOR "OVERLAP" CONCEPT



Knowledge of temperature *trends* could give us a further way to improve the data! Also, by now we can see that we need to schedule service on the yellow sensor, or remove it entirely.

RULE WE USE IF WE CAN'T DETECT THAT SOME SENSOR IS FAULTY

If we know that N-F sensors were accurate, there must be some overlap region where N-F sensors overlap. The true value is in that region.

… but it could be anywhere in that overlap region.

If we have more than one candidate region, the best we can do is to take the union. The sensor value must lie in "some" overlap area with N-F overlapping sensors.

WHY NOT FIGURE OUT WHICH SENSOR IS BAD?

Sometimes a sensor has a clock that is out of sync. It reports sensible readings but at the wrong time.

Then it manages to resync. Now it is working again.

If we are two aggressive about excluding sensors, we could exclude all of them and yet the faulty ones might have recovered in the meanwhile!

LAMPORT'S CAUSAL ARROW NOTATION

Lamport talks about how one event can *influence* or *cause* another event

If we write **a** $\rightarrow$ **b**, then in words we are saying "**a** happened before **b**"

- $\rightarrow$ is a mathematical notation for expressing information flow.
- Data about **a** reached **b**, and **b** might have somehow have used this data about **a.** It "depends" on a.
- **a** $\rightarrow$ **b** if (and only if) we can trace a path through the timeline of the system from the point **a** occurs to the point **b** occurs

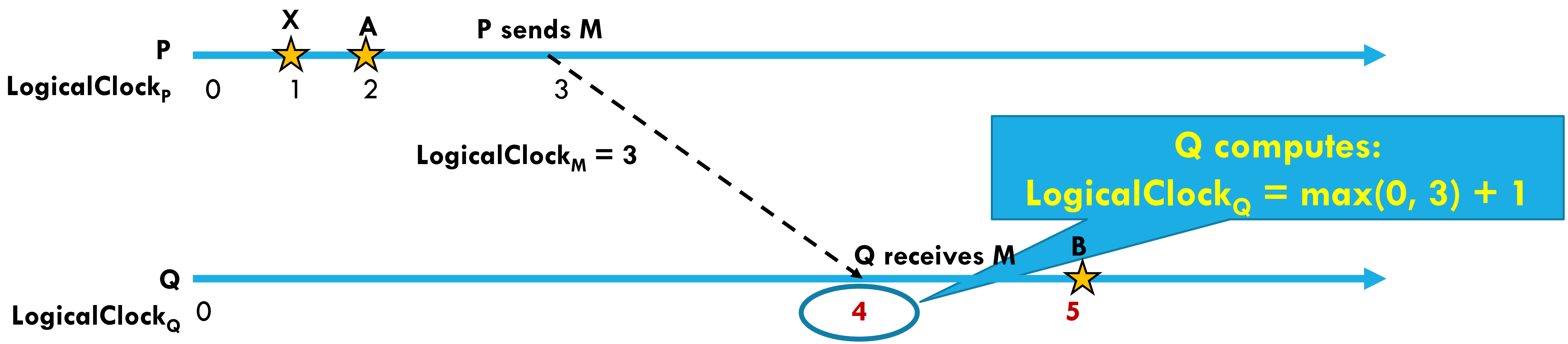
LOGICAL CLOCKS

Lamport introduced logical clocks

They are just integers that are managed using a simple rule: increment your copy when something happens. Include a copy on any message. When a message arrives, take the maximum of the local clock and the one in the message.

With logical clocks, **a** $\rightarrow$ **b** implies that LT(**a**) < LT(**b**). But not the opposite.

A SPACE-TIME DIAGRAM FOR THIS CASE



Drill down: Consistency

VECTOR CLOCKS: EXTENSION OF LOGICAL CLOCKS

A vector clock has one entry per process in the system, as an array. Only process **a** can *increment* (add one to) its own entry. But we still take the maximum, element by element, when a message arrives.

VT(**a**) < VT(**b**) if every element of VT(**a**) is less than or equal to the corresponding one in VT(**b**), and there is at least one element in VT(**a**) that is smaller than the corresponding one in VT(**b**)

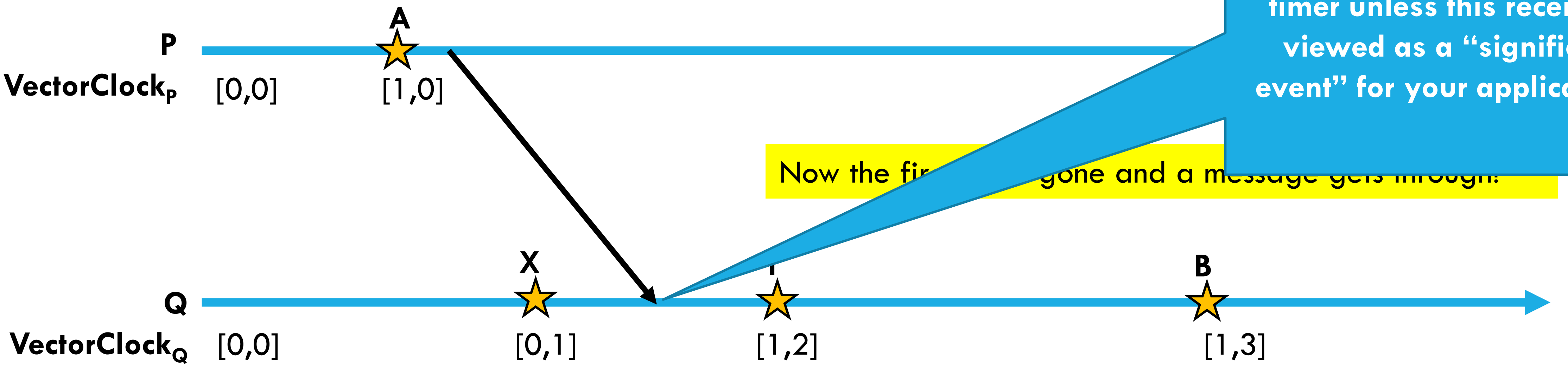
With vector clocks, **a** → **b** implies that VT(**a**) < VT(**b**).

VT(**a**) < VT(**b**) implies that **a** → **b**

A SPACE-TIME DIAGRAM FOR THIS CASE

Case B: P sends a message to Q after A, and it is re

Maximum computed here. No need to increment the local timer unless this receive is viewed as a “significant event” for your application,.



The vector timestamps show that A happens before B (and also, before Y).

WHY NOT ALWAYS USE VECTOR CLOCKS?

They are kind of bulky. A system could have *many* processes

Also, if membership can evolve, we need to have a flexible vector clock representation that can evolve over time.

Often a normal logical clock, just one counter, is enough

CONSISTENT CUTS AND SNAPSHOTS

These concepts arise in timeline diagrams

- We can never predict exactly how fast a computer will run
- So "timelines" for processes can shrink or stretch

A consistent cut across a system is a set of time points, process by process, that could have occurred instantaneously

- You would just stretch some timelines (slow those processes down) and shrink others (speed them up) to "align" the time points
- Maybe they really did occur simultaneously, maybe not

CONSISTENT CUTS AND SNAPSHOTS

But one thing can't happen when you do this kind of shrink/stretch

A message can never flow backwards in time

So there are definitely sets of timepoints that cannot possibly have been simultaneous. A cut in which some message flows from the future back over the cut to the past would be an "inconsistent" cut.

SNAPSHOTS

We say that a **checkpoint** is an object that fully captures the state of some single process. Recall that we used these in state transfer, too!

A consistent snapshot is a set of checkpoints made along a consistent cut.

Sometimes we also want to include a snapshot of what was in the network at that moment (a set of messages).

The Chandy-Lamport algorithm is one of a few options for making consistent snapshots or identifying consistent cuts.

VALUE OF CONSISTENT CUTS AND SNAPSHOTS

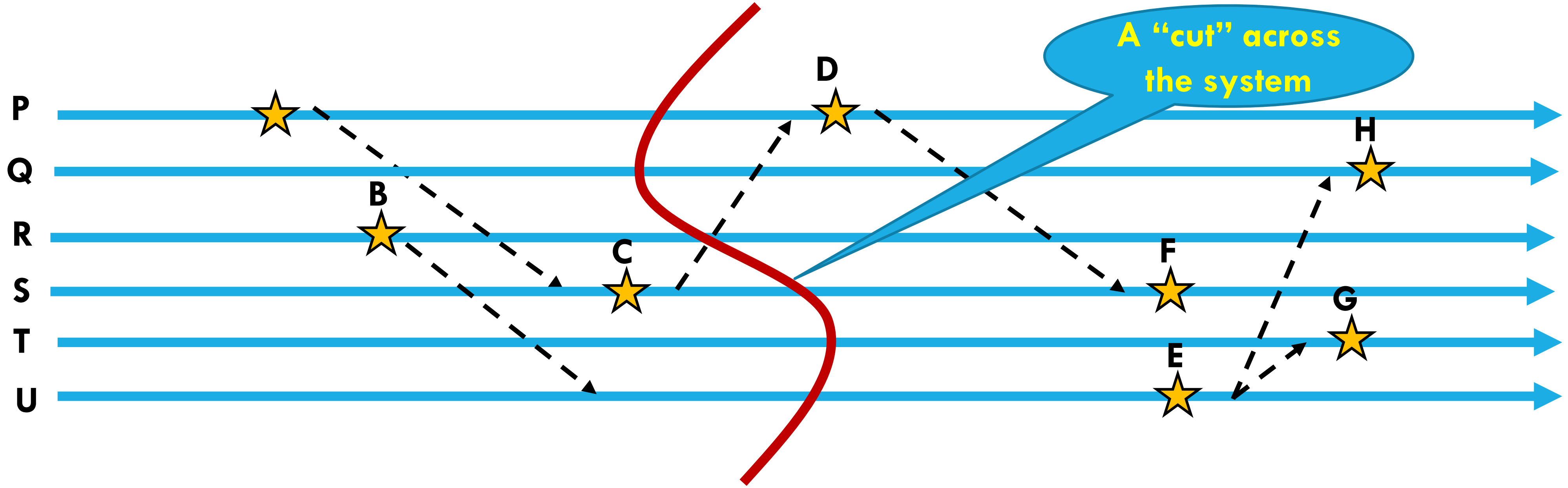
There are many examples of situations where seeing a system state inconsistently can cause errors

- A distributed reference counting scheme for garbage collection could count incorrectly and delete objects that still have references to them
- A deadlock detector might think there was a cycle, but it isn't real
- An ML algorithm might try to run on a system state that could never have arisen in real life, and conclude something totally false

Consistent cuts and snapshots avoid these buggy behaviors

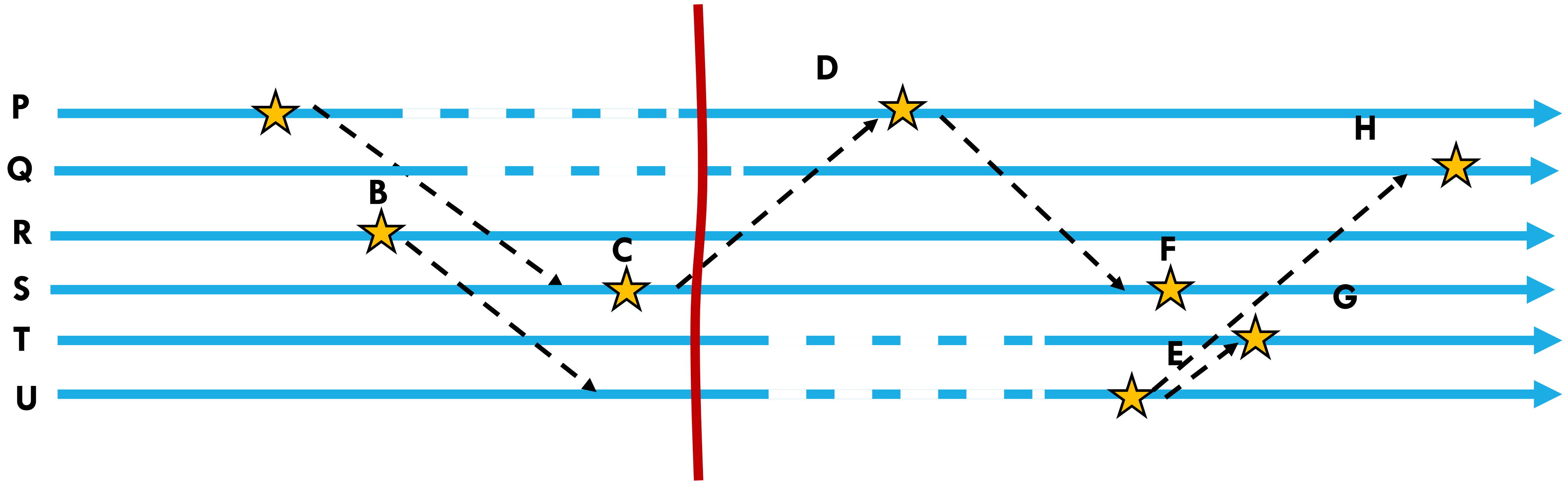
CONSISTENT CUTS AND SNAPSHOTS

Recall: Lamport looks at "pictures" of such a system, like these



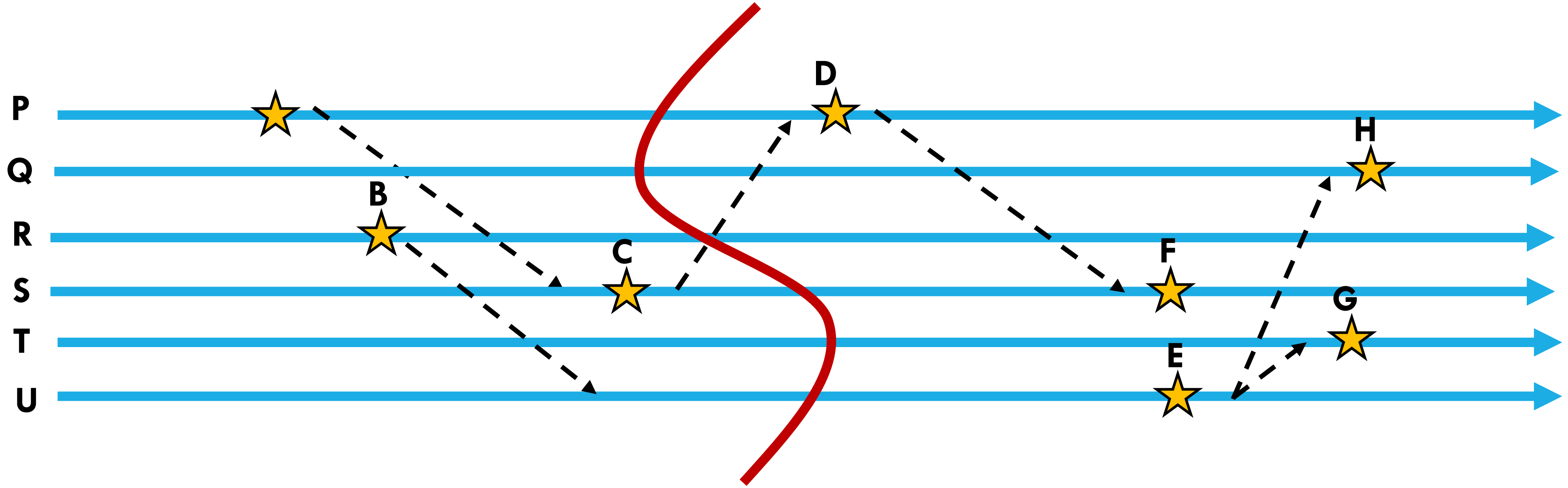
STRETCHING AND SHRINKING TIMELINES

Faster execution pulls events to the left… Slower would push to the right



CONSISTENT CUTS AND SNAPSHOTS

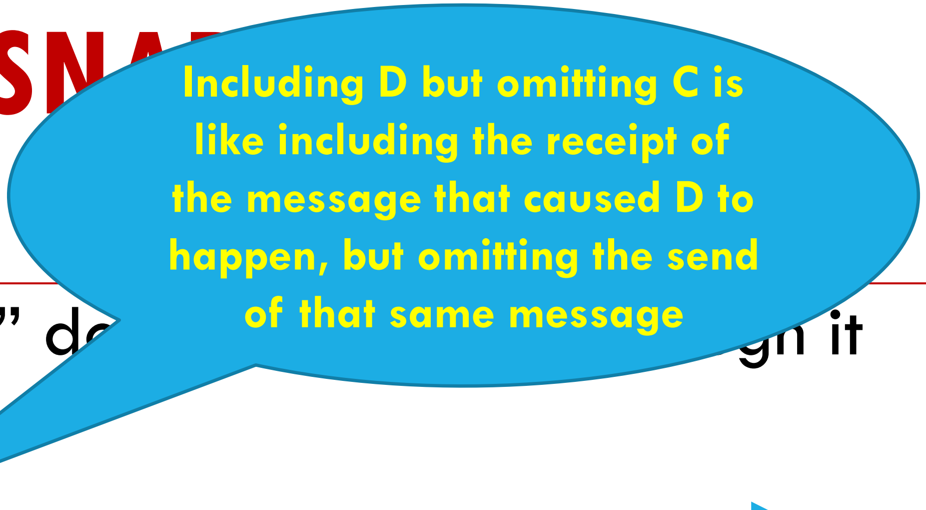
A cut is consistent if no "message arrows" go backwards through it



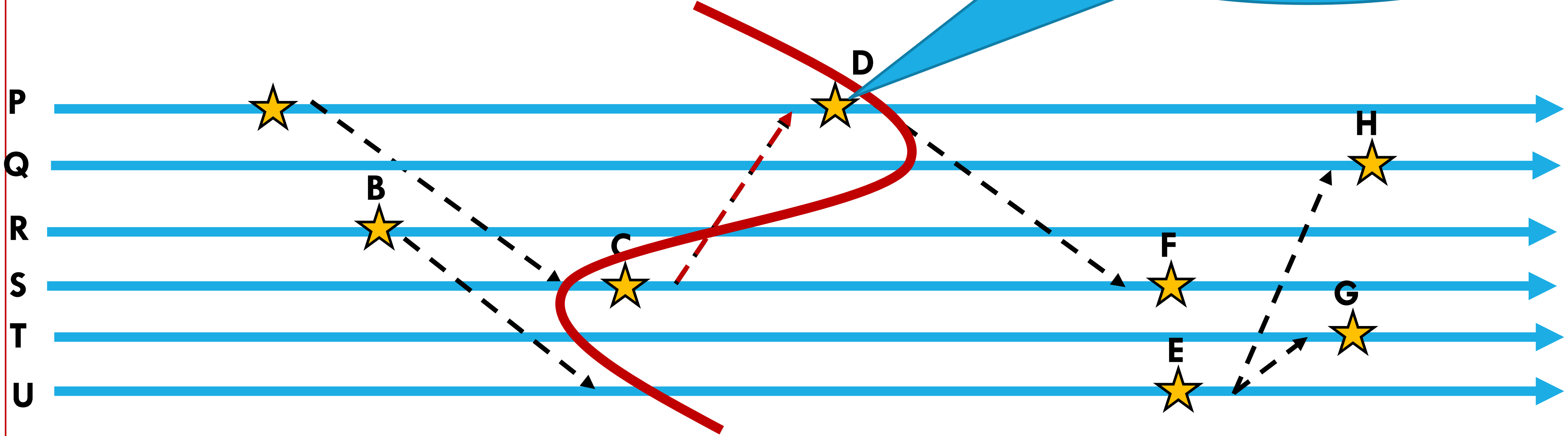
… this cut is a consistent one.

Drill down: Consistency

CONSISTENT CUTS AND SNA



A cut is **in**consistent if "message arrows" d… …gh it



… this cut is ***inconsistent***. C → D, and the cut included D, yet it omits C.

THOUGHT QUESTION

Go back to slide 22, with examples. Focus on the deadlock detector or the reference-counting garbage collector.

Now draw some time-line pictures like in slides 23-26. Can you show, in "with/without" pictures, errors that might occur for a deadlock detector, or a reference counter, if it runs on an inconsistent cut?

Do the pictures help us understand why a consistent cut won't lead to those mistakes?

THOUGHT QUESTION

Suppose that a sharded key-value store (DHT) is using state machine replication (atomic multicast or Paxos) to implement replicated updates.

Now suppose that when an update is delivered, rather than just somehow doing the update and being done, or just logging the update, some sort of fancy computation runs (like reference counting, or deadlock checking)

Would that computation be running on a consistent cut? Explain why (or why not)