



CS4740

CLOUD COMPUTING

Time and Coordination

Prof. Chang Lou, UVA CS, Fall 2026

REMINDER (LAB 1 DUE)

- Deadline: 9/27 (this Sunday)
- If you already finished: keep the branch "lab1" intact
 - We'll collect your codes at deadline (unless you use late tokens)
- If you haven't: keep track of time and implement soon
 - Come to office hours if you need help

CONTEXT

- We looked at RPC, a key concept in DS, and saw how failures creep up into semantics and challenge coordination.
- We now look at another key concept in DS, **Time**.
 - Let's see how unbounded network delays (a.k.a. network asynchrony) complicates the very basic concept.

WHY IS TIME IMPORTANT?



(photos from my mechanical watch repair projects —
another type of debugging)

WHY IS TIME IMPORTANT?

- Needed for synchronization and coordination.
- Examples:
 - Consistency (ordering)
 - Failure detection (timeout)
 - A running (toy) example: distributed debugging based on logs

EXAMPLE: DISTRIBUTED DEBUGGING

M1 (front end)

...
recv from cli
...
send to M2
...
recv from M2
...
send to cli
...

M2 (app server)

...
recv from M1
...
send to M3
...
recv from M3
...
send to M1
...

M3 (DB server)

...
recv from M2
...
SQL query
...
send to M2
...

EXAMPLE: DISTRIBUTED DEBUGGING

M1 (front end)

```
...  
recv from cli  
...  
send to M2  
...  
recv from M2  
...  
send to cli  
...
```

M2 (app server)

```
...  
recv from M1  
...  
send to M3  
...  
recv from M3  
...  
send to M1  
...
```

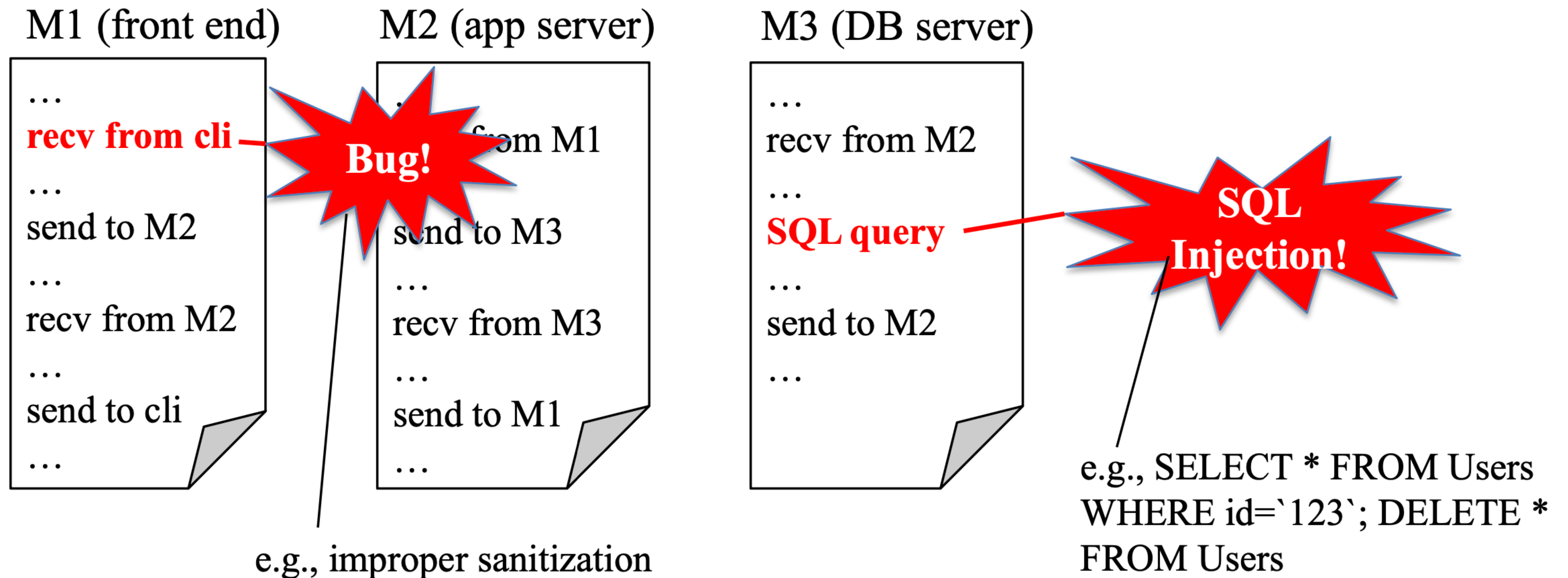
M3 (DB server)

```
...  
recv from M2  
...  
SQL query  
...  
send to M2  
...
```

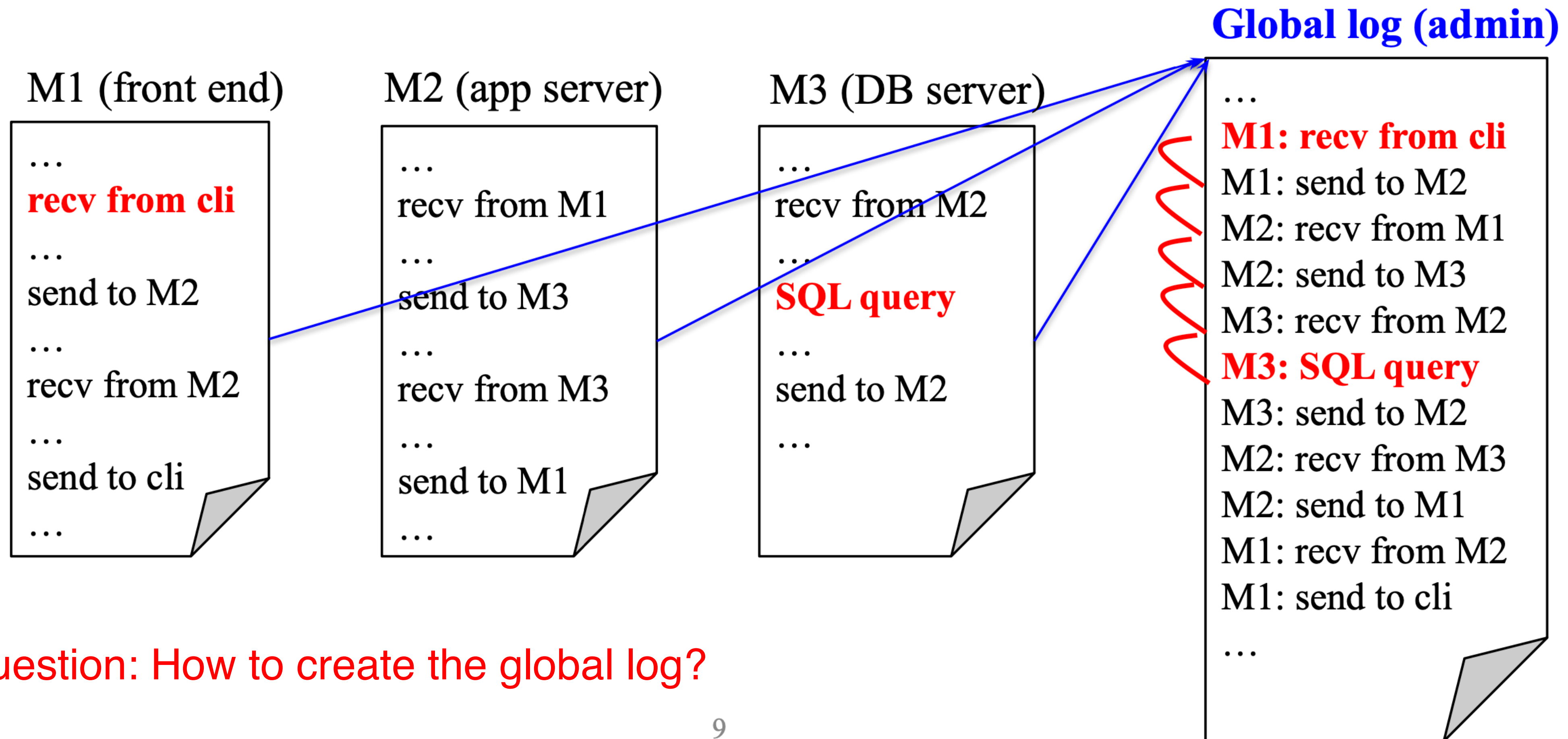


e.g., SELECT * FROM Users
WHERE id='123'; DELETE *
FROM Users

EXAMPLE: DISTRIBUTED DEBUGGING

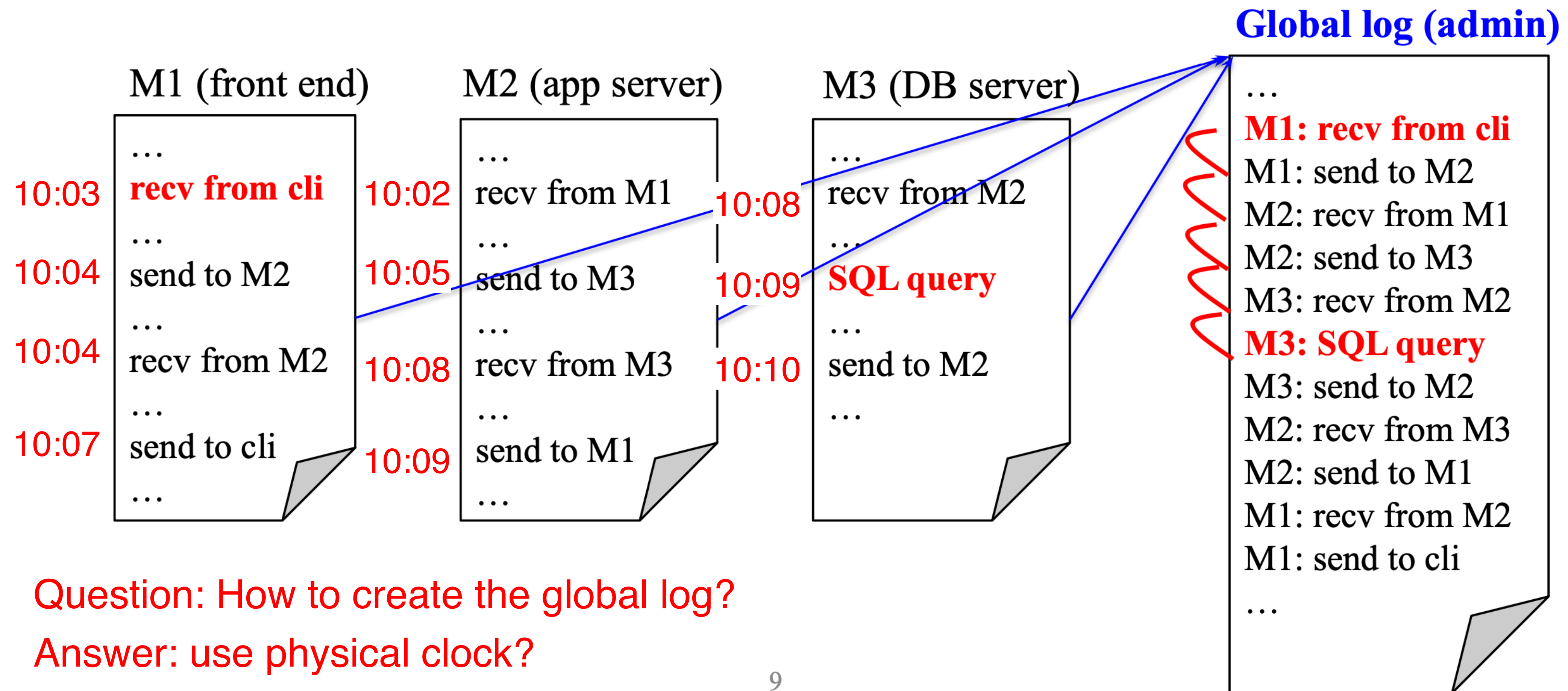


EXAMPLE: DISTRIBUTED DEBUGGING



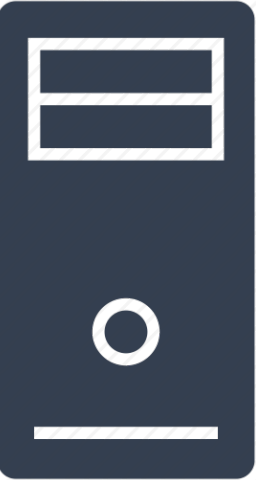
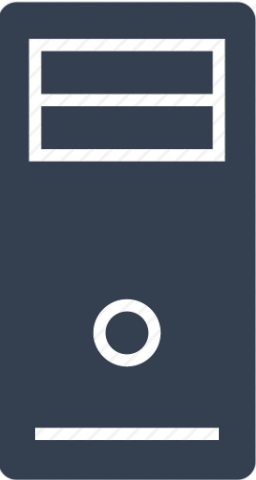
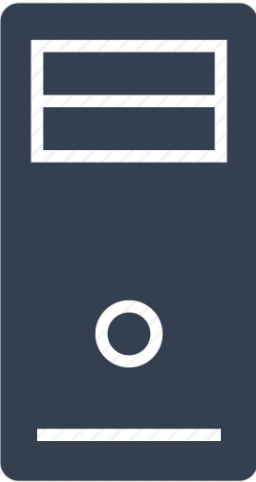
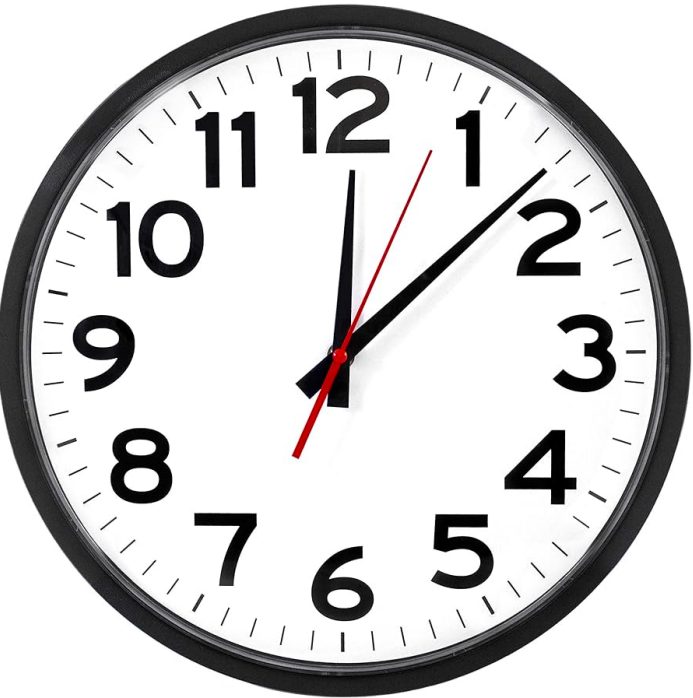
Question: How to create the global log?

EXAMPLE: DISTRIBUTED DEBUGGING

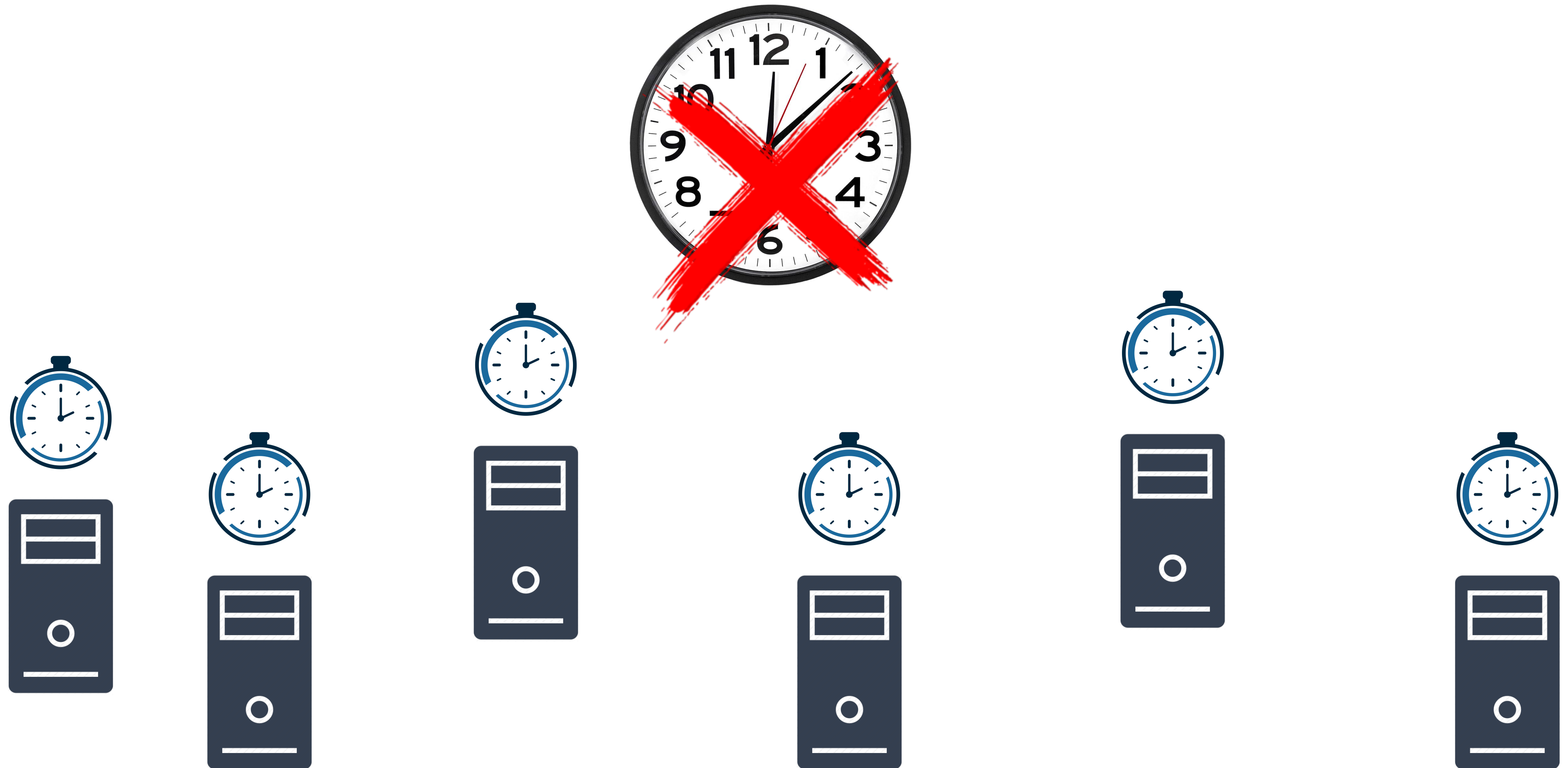


PROBLEM: CLOCK SYNCHRONIZATION IS HARD

- Quartz oscillator sensitive to temperature, age, vibration, radiation
 - Accuracy ~one part per million: one second of clock drift over 12 days
 - This means we need to synchronize them periodically
- The network is:
 - Asynchronous: arbitrary message delays
 - Best-effort: messages don't always arrive



THERE IS NO GLOBAL TIME!



AGENDA

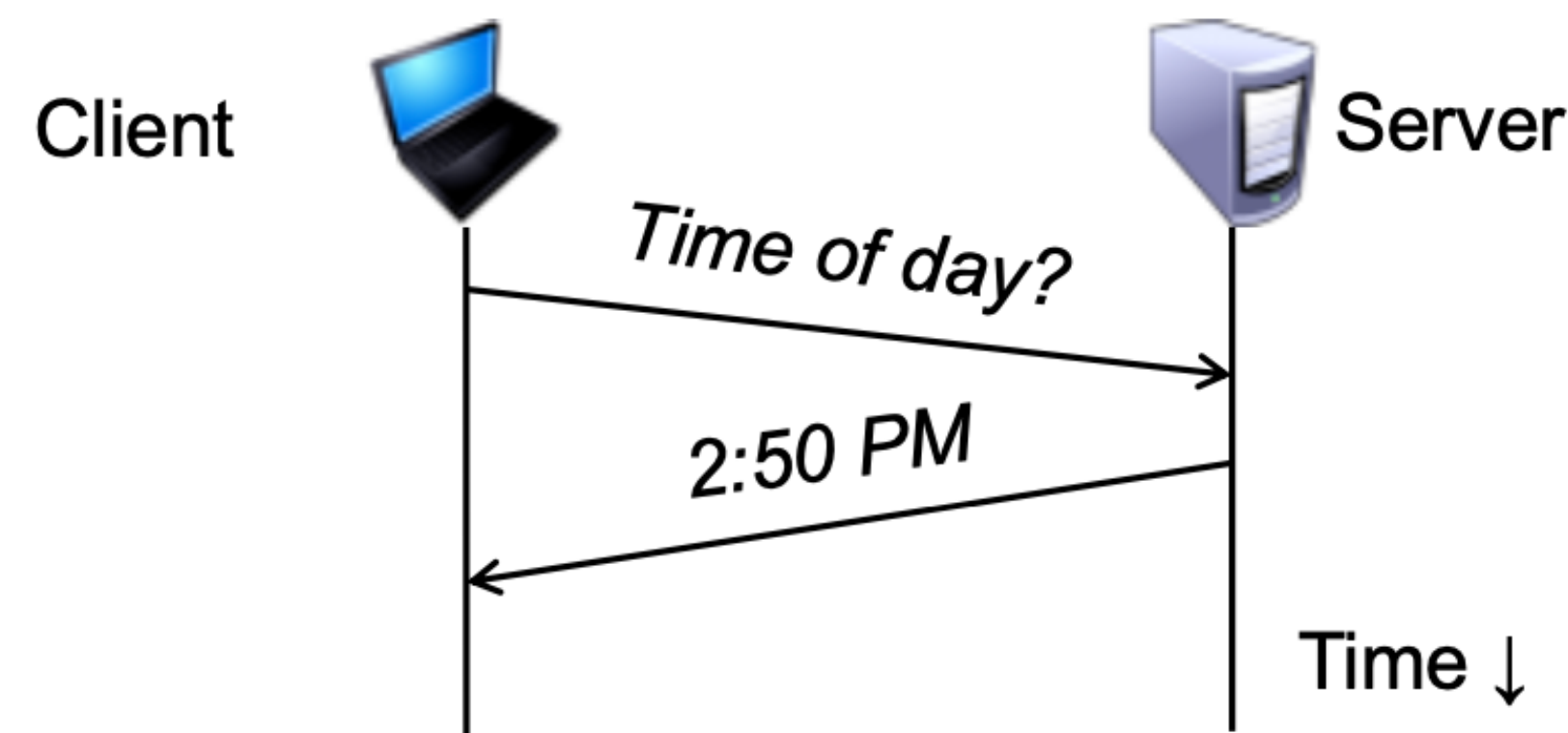
- **Physical clocks**
 - Synchronization challenges and protocols
- Logical clocks
 - Lamport clock protocol

JUST USE COORDINATED UNIVERSAL TIME?

- UTC is broadcast from radio stations on land and satellite (e.g., the Global Positioning System)
 - Computers with receivers can synchronize their clocks with these timing signals
- Signals from land-based stations are accurate to about 0.1–10 milliseconds
- Signals from GPS are accurate to about one microsecond
 - *Why don't we put GPS receivers on all our computers?*
 - *Answer: coverage and cost issues*

SYNCHRONIZATION TO A TIME SERVER

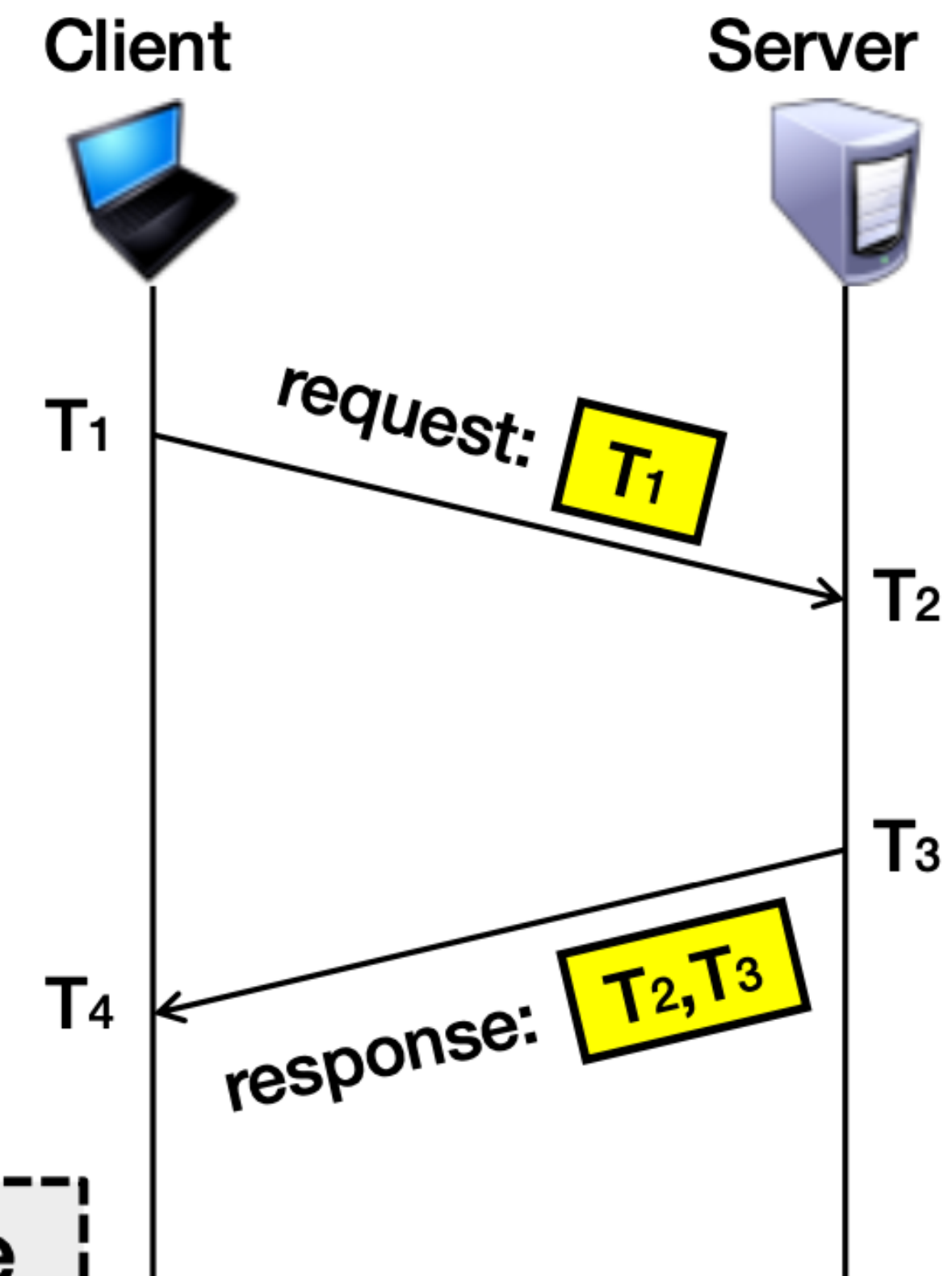
- How about installing for just one machine per cluster?
- Suppose a server with an accurate clock (e.g., GPS-receiver)
 - Could simply issue an RPC to obtain the time:



- But this doesn't account for network latency
 - Message delays will have outdated server's answer

CRISTIAN'S ALGORITHM: OUTLINE

- 1. Client sends a request packet, timestamped with its local clock T_1
- 2. Server timestamps its receipt of the request T_2 with its local clock
- 3. Server sends a response packet with its local clock T_3 and T_2
- 4. Client locally timestamps its receipt of the server's response T_4



How can the client use these timestamps to synchronize its local clock to the server's local clock?

CRISTIAN'S ALGORITHM: OFFSET SAMPLE CALCULATION

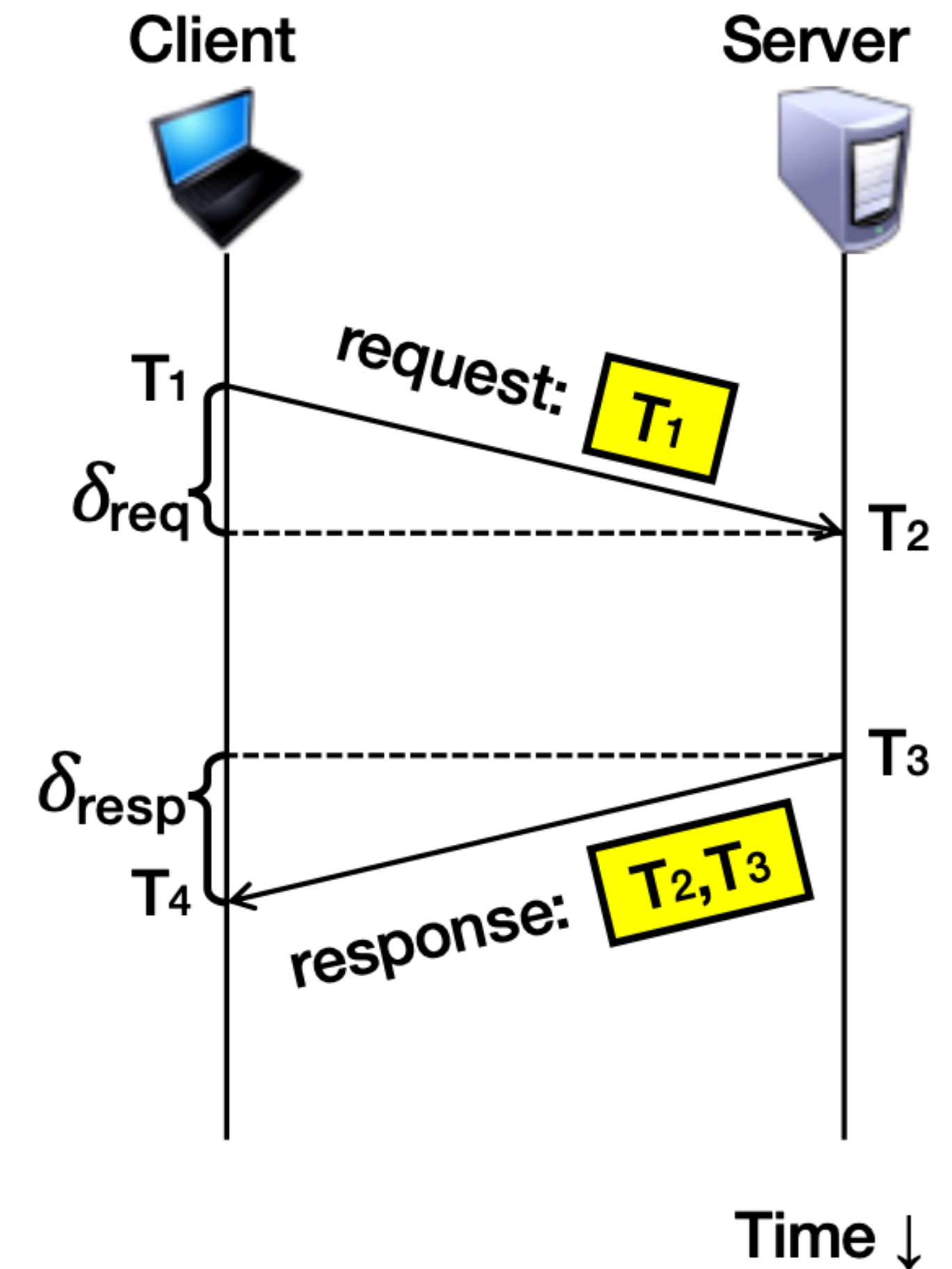
Goal: Client sets clock $\leftarrow T_3 + \delta_{\text{resp}}$

- Client samples **round trip time (δ)**
 $\delta = \delta_{\text{req}} + \delta_{\text{resp}} = (T_4 - T_1) - (T_3 - T_2)$

- But client knows δ , not δ_{resp}

Assume: $\delta_{\text{req}} \approx \delta_{\text{resp}}$

Client sets clock $\leftarrow T_3 + \frac{1}{2}\delta$



CLOCK SYNCHRONIZATION: TAKE-AWAYS

- Clocks on different systems will always behave differently
 - Disagreement between machines can result in undesirable behavior
- NTP clock synchronization
 - Rely on timestamps to estimate network delays
 - 100s μ s–ms accuracy
 - Clocks never exactly synchronized
- Often inadequate for distributed systems
 - Often need to reason about the order of events
 - Might need precision on the order of ns

AGENDA

- Physical clocks
 - Synchronization challenges and protocols
- **Logical clocks**
 - Lamport clock protocol

LOGICAL CLOCKS

- Leslie Lamport, parent of DS, observed that most coordination in distributed systems (e.g., for mutual exclusion, barriers, complete event log) doesn't require a global notion of real time!
- Most coordination only needs a global order of discrete events.
- E.g., in the distributed debugging example, you only need order between dependent events that could possibly have caused the failure.
- Achieving a global order of events is easier to guarantee than achieving zero-error real-time synchronization.
- This is why many foundational DS protocols rely on logical clocks.

HAPPEN-BEFORE PAPER (1978)

Operating
Systems

R. Stockton Gaines
Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.

General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

This work was supported by the Advanced Research Projects

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation “happened before” is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the “happened before” relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Most people would probably say that an event *a* happened before an event *b* if *a* happened at an earlier time than *b*. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the “happened before” relation without using physical clocks.

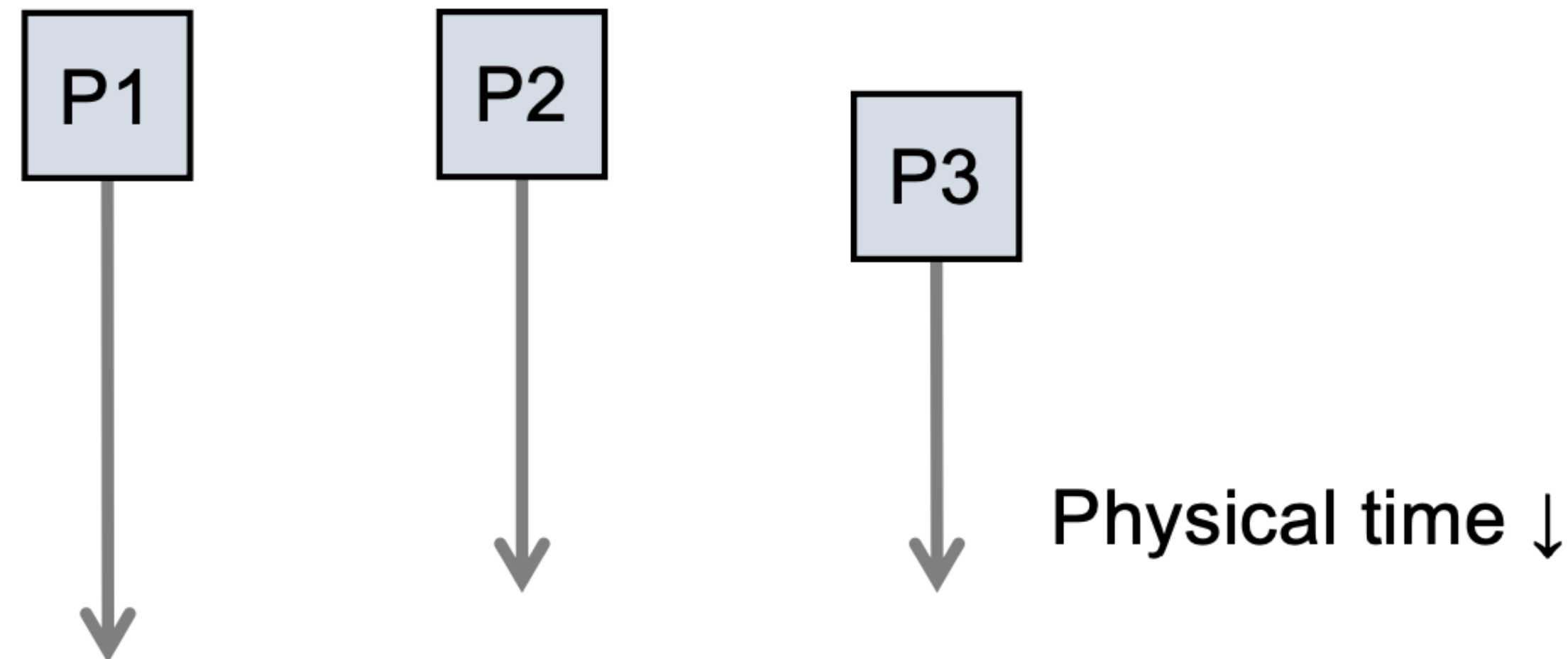
We begin by defining our system more precisely. We assume that the system is composed of a collection of

LOGICAL CLOCK REQUIREMENTS

- Lamport posited two requirements for logical clocks:
 - They must preserve **program order** (i.e., the order of events in one process needs to be preserved by the logical clock)
 - They must preserve **message order** (i.e., a message sent event always needs to precede that message's receipt event in the logical clock).
- These two requirements capture all internal **causality** between any two events in the system.

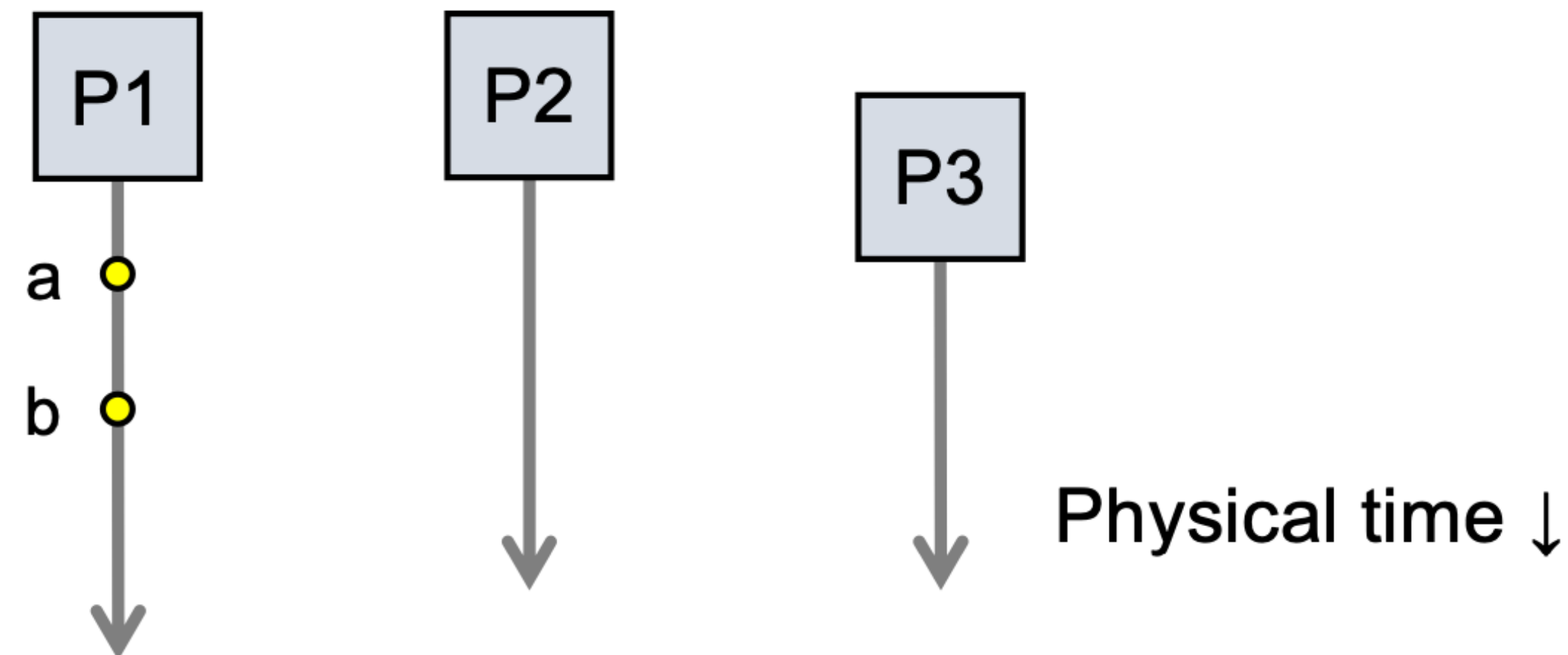
DEFINING “HAPPENS-BEFORE”

- Consider three processes: P1, P2, and P3
- Notation: Event a happens before event b ($a \rightarrow b$)



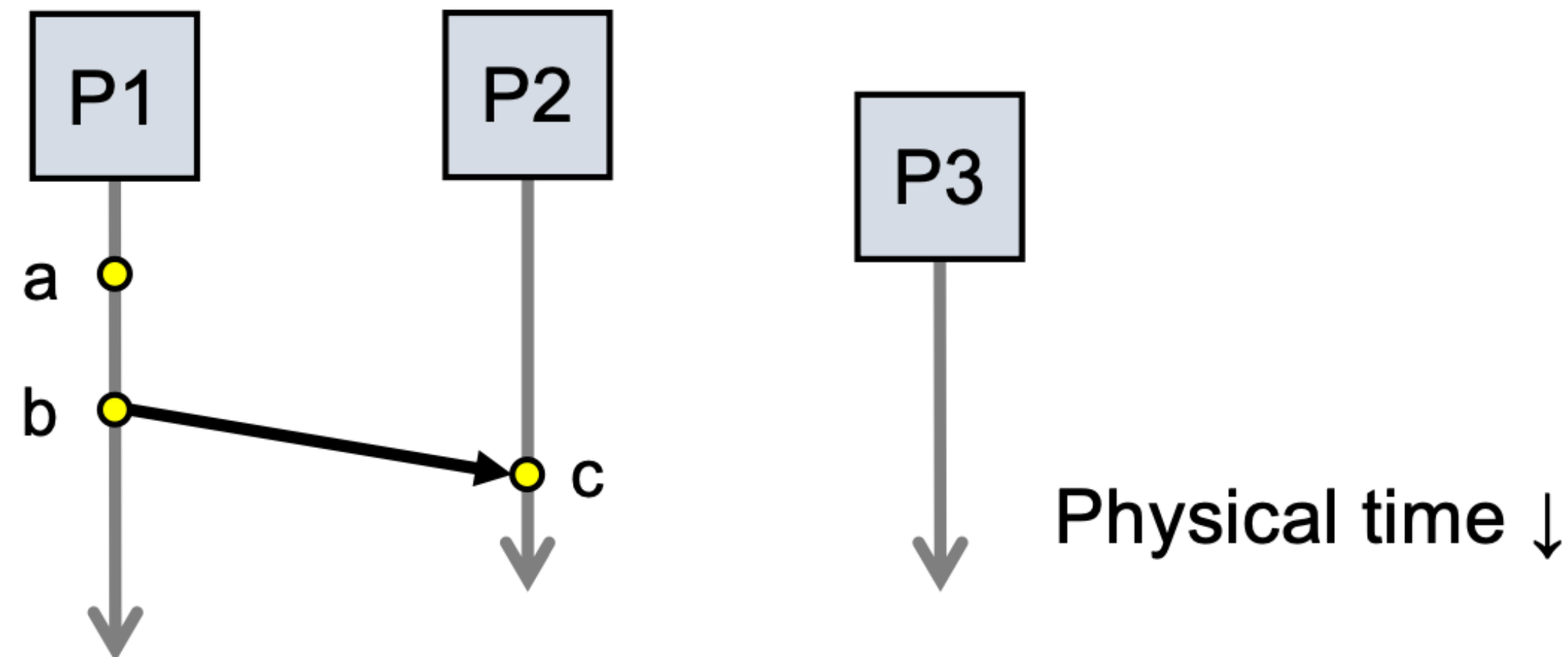
DEFINING “HAPPENS-BEFORE”

- 1. If same process and a occurs before b, then $a \rightarrow b$



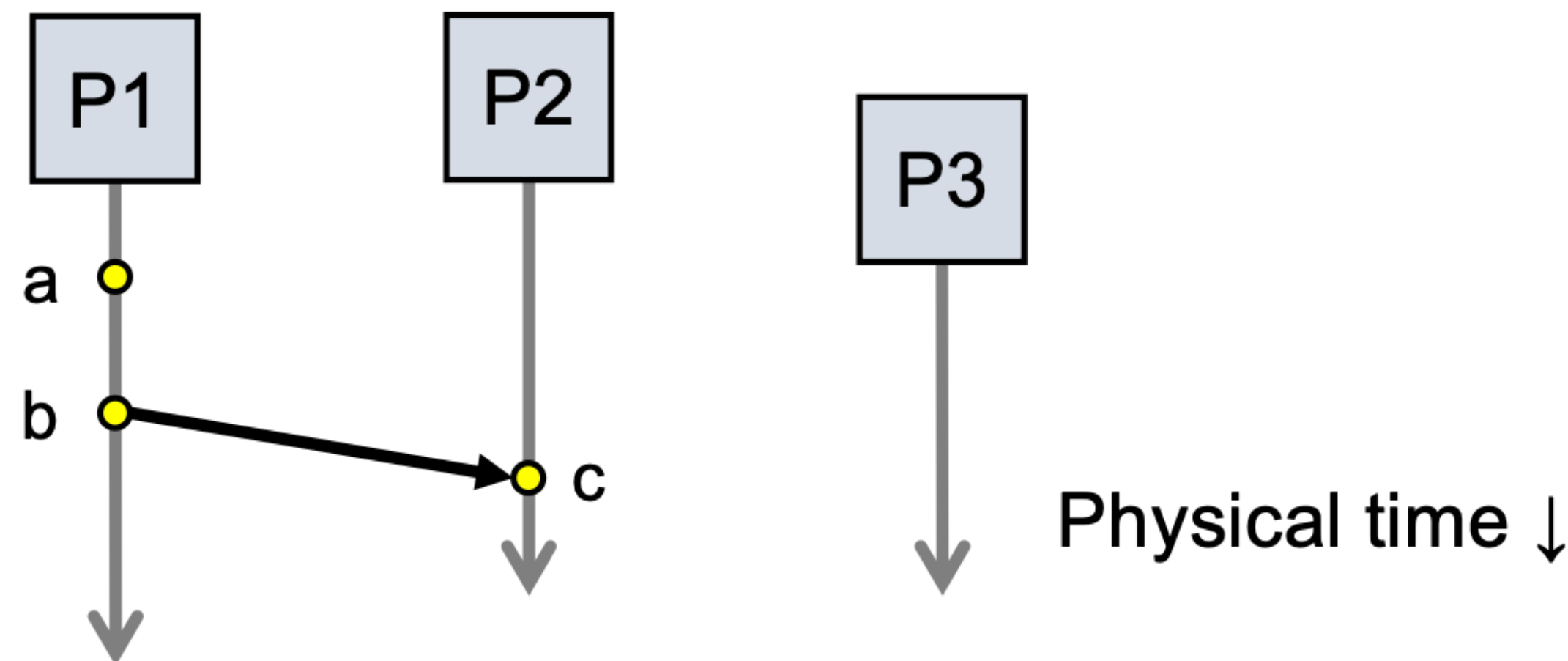
DEFINING “HAPPENS-BEFORE”

- 1. If same process and a occurs before b, then $a \rightarrow b$
- 2. If c is a message receipt of b, then $b \rightarrow c$



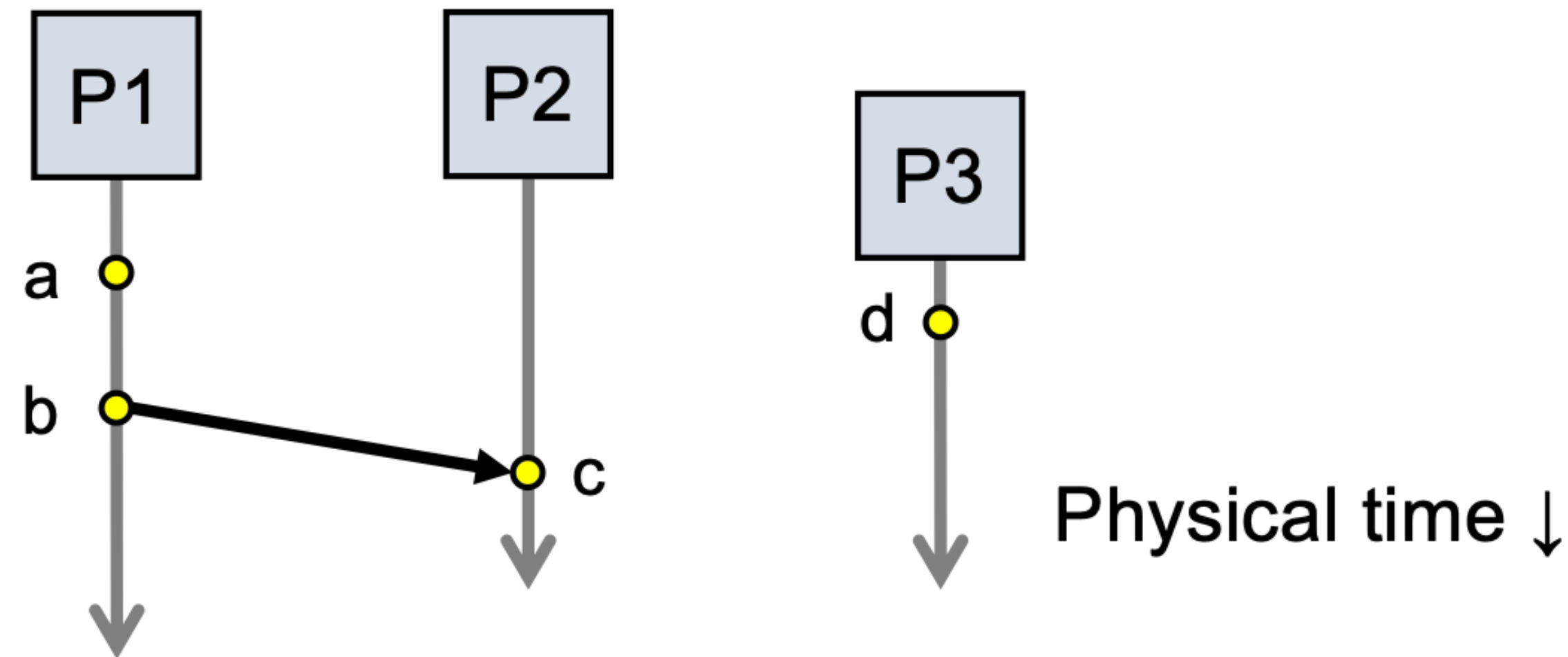
DEFINING “HAPPENS-BEFORE”

- 1. If same process and a occurs before b, then $a \rightarrow b$
- 2. If c is a message receipt of b, then $b \rightarrow c$
- 3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$



DEFINING “HAPPENS-BEFORE”

- Not all events are related by a
- a, d not related by a so concurrent, written as $a \parallel d$



LOGICAL CLOCK SYNCHRONIZATION PROTOCOL

- Lamport clock protocol [Lamport-1978].
- Setup:
 - Process = individual node in a distributed system
 - Processes communicate by messages (e.g., RPCs)
 - Events can be **messages** or system-specific **events** (e.g., write to file, read from file, whatever makes sense for the specific distributed system).
 - View each process in the distributed system as a **state machine**: has some initial state, events cause it to move from one state to another.

LAMPORT CLOCK PROTOCOL

- Each process P_i maintains a local counter, C_i
- Each process P_i increments C_i between any two successive events
- Each process piggybacks timestamp T_m on a message it sends out, where T_m is value of C_i at the time of sending m
- Upon receiving m at process P_j :
 - P_j sets its counter C_j to $\max(C_j, T_m+1)$
 - The receipt of m is a separate event that then separately advances C (i.e., $C++$)

Node P_i 's state machine:

On local event:

- C_i++

On message send:

- Piggyback C_i to msg.
- C_i++

On message(T_m) receive:

- $C_i = \max(C_i, T_m+1)$
- C_i++

GETTING A GLOBAL ORDER

- The preceding protocol gives a partial order of all causally dependent events.
- Often we need a global order on which all processes agree.
- To obtain that, use logical clock to set the order. Use process IDs as the tie breaker.
 - E.g.: use (Logical timestamp).(process ID) as your timestamp.

DISTRIBUTED DEBUGGING EXAMPLE

C₁ M1 (front end)

0	0.1 op11 rcv cli
1	1.1 op12 ...
2	?? op13 snd M2
?	?? op14 ...
?	?? op15 rcv M2
?	?? op16 ...
?	?? op17 snd cli
?	...

C₂ M2 (app server)

0	?? op21 rcv M1
?	?? op22 ...
?	?? op23 ...
?	?? op24 snd M3
?	?? op25 ...
?	?? op26 ...
?	?? op27 ...
?	?? op28 ...
?	?? op26 rcv M3
?	?? op27 ...
?	?? op28 snd M1
?	...

C₃ M3 (DB)

0	0.3 op31 ...
?	?? op32 ...
?	?? op33 ...
?	?? op34 rcv M2
?	?? op35 SQL
?	?? op36 ...
?	?? op37 snd M2
?	...

Global Log

TODO: Timestamp the ops in each machine's log using logical clocks, then assemble the global log by merge-sorting them.

(assume C_i=0 initially)

Breakout Activity!

ACTIVITY (10 MINUTES)

- Assign logical timestamps to operations in each log, then sort the operations by timestamp in global log. A few entries have already been filled in as examples.
- Hint: As you go through the operations, keep track of the logical clock value at each machine, C1-3. Use the Lamport clock protocol to update the clocks (the algorithm is pasted on the right).
- Hint: It may be useful to first draw happens-before arrows between message sends and their receipts so you know when clock synchronization happens.
- Hint: Use a totally ordered clock: timestamp is $C_i.i$.

Node P_i 's state machine:

On local event:

- C_i++

On message send:

- Piggyback C_i to msg.
- C_i++

On message(T_m) receive:

- $C_i = \max(C_i, T_m + 1)$
- C_i++

STUDENT WORKSHEET

C₁ M1 (front end)

0	0.1 op11 rcv cli
1	1.1 op12 ...
2	?? op13 snd M2
?	?? op14 ...
?	?? op15 rcv M2
?	?? op16 ...
?	?? op17 snd cli
?	...

C₂ M2 (app server)

0	3.2 op21 rcv M1
?	?? op22 ...
?	?? op23 ...
?	?? op24 snd M3
?	?? op25 ...
?	?? op26 ...
?	?? op27 ...
?	?? op28 ...
?	?? op26 rcv M3
?	?? op27 ...
?	?? op28 snd M1
?	...

C₃ M3 (DB)

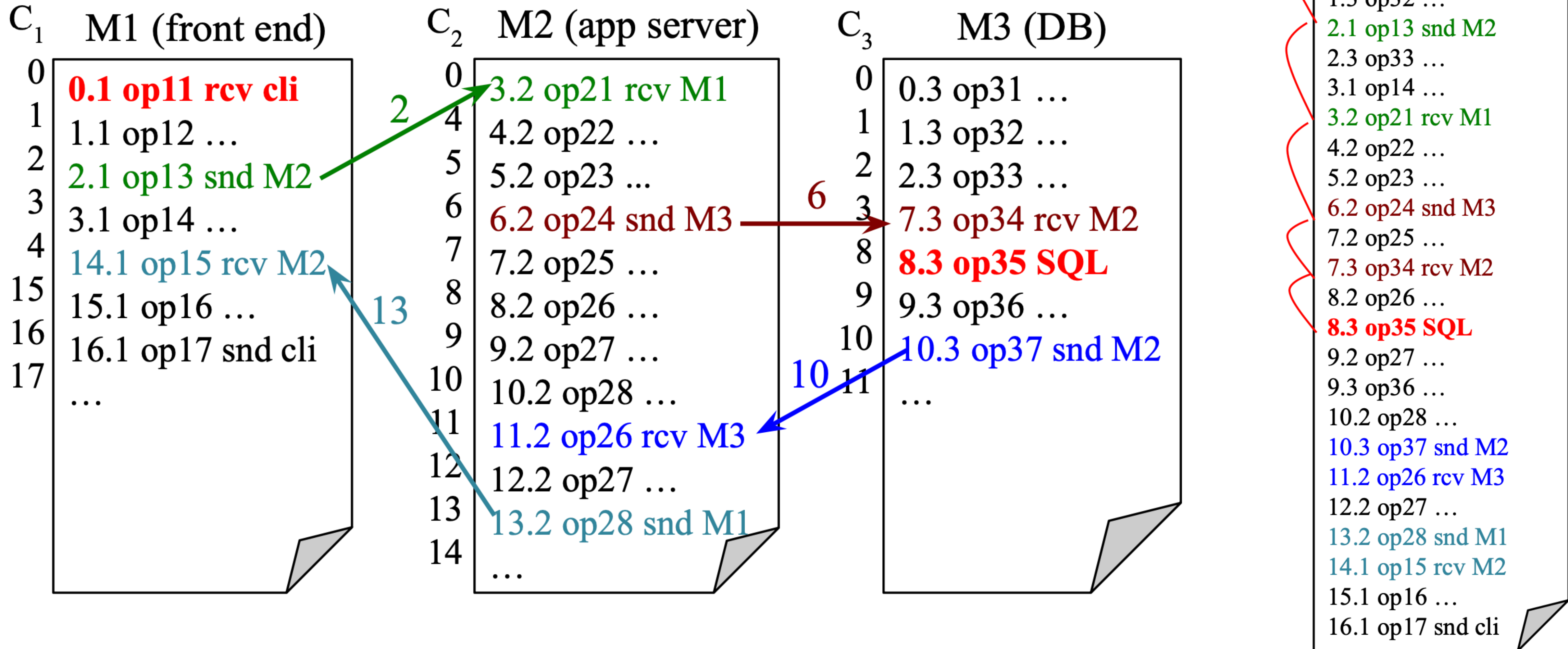
0	0.3 op31 ...
?	?? op32 ...
?	?? op33 ...
?	?? op34 rcv M2
?	?? op35 SQL
?	?? op36 ...
?	?? op37 snd M2
?	...

0.1 op11 rcv cli

... enter all events in
order of their logical
timestamp

Node Pi's state machine:
On local event:
- Ci++
On message send:
- Piggyback Ci to msg.
- Ci++
On message(Tm) receive:
- Ci = max(Ci, Tm+1)
- Ci++

SOLUTION



PLUSES AND MINUSES OF LAMPORT CLOCKS

- Advantages

- Respect causality, which can address many coordination problems in distributed systems.

- Disadvantages

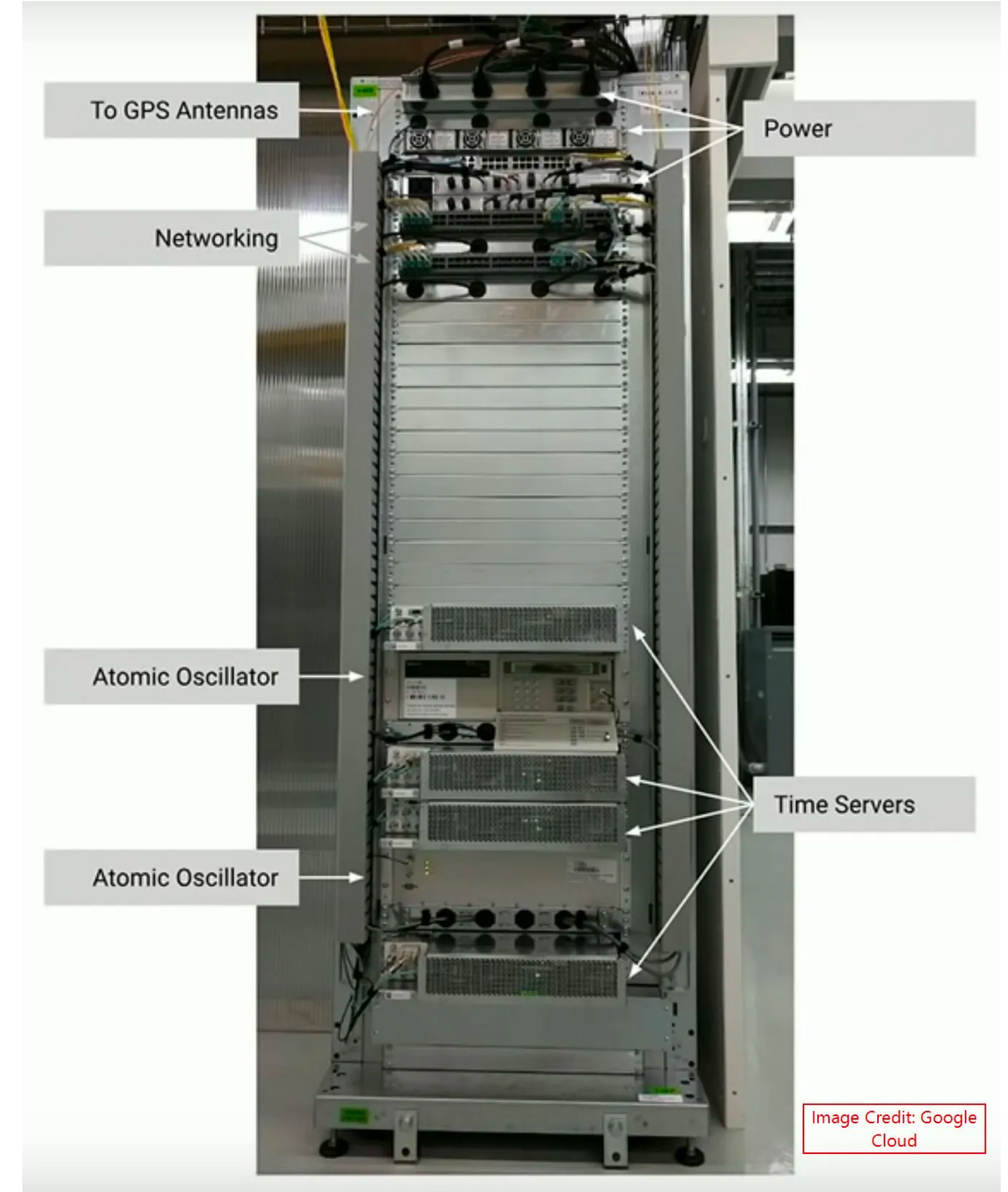
- Capturing causality is sometimes insufficient, as there can be events outside the system that have causal influence on the evolution of the system. The ordering doesn't capture these relationships.
 - Lamport clock ordering doesn't actually imply causality/influence, just potential influence. Hence, the order can be too much order, affecting performance/scalability.

AGENDA

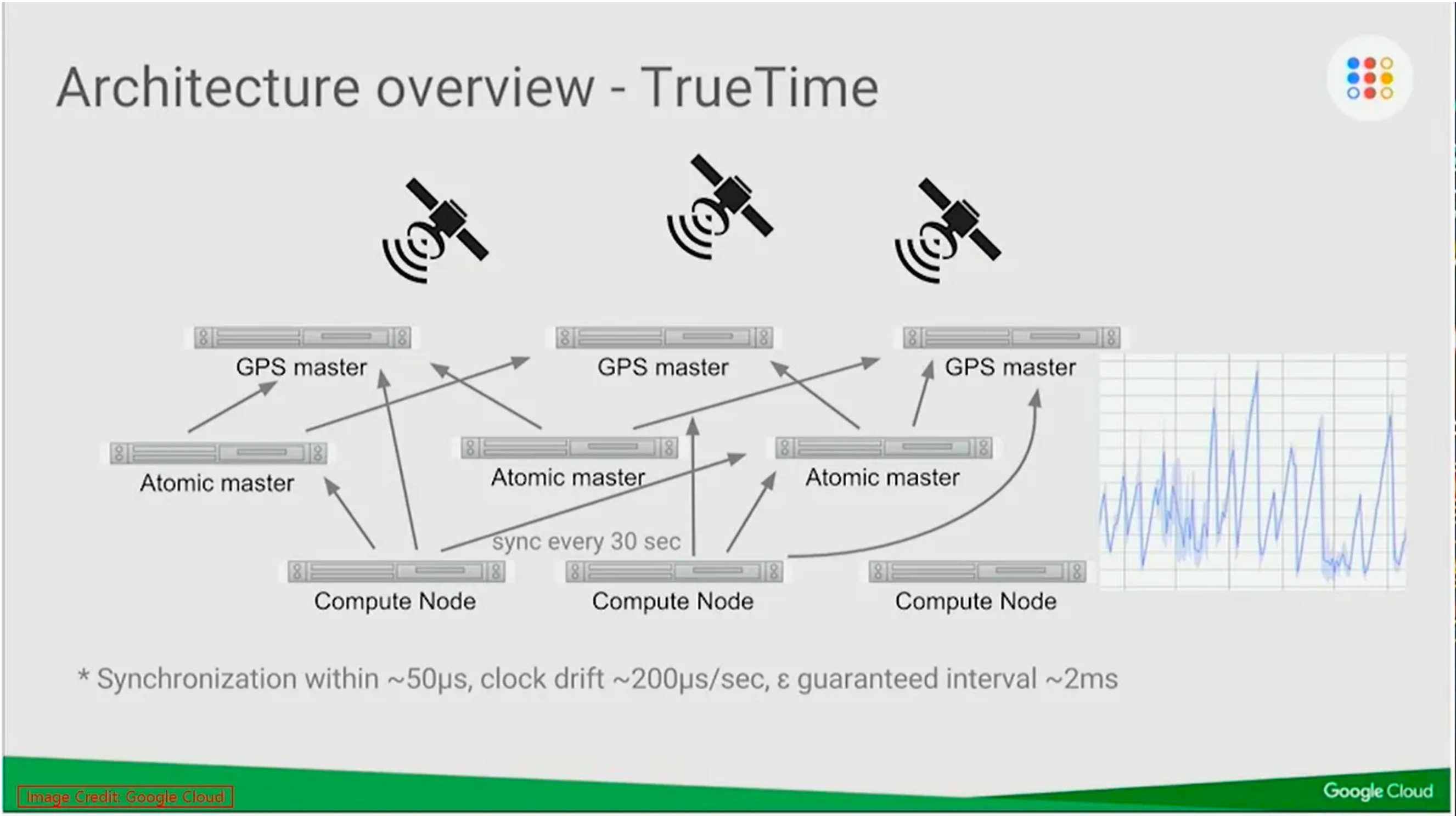
- Physical clocks
 - Synchronization challenges and protocols
- Logical clocks
 - Lamport clock protocol
- (*) **Google TrueTime**

GOOGLE TRUETIME

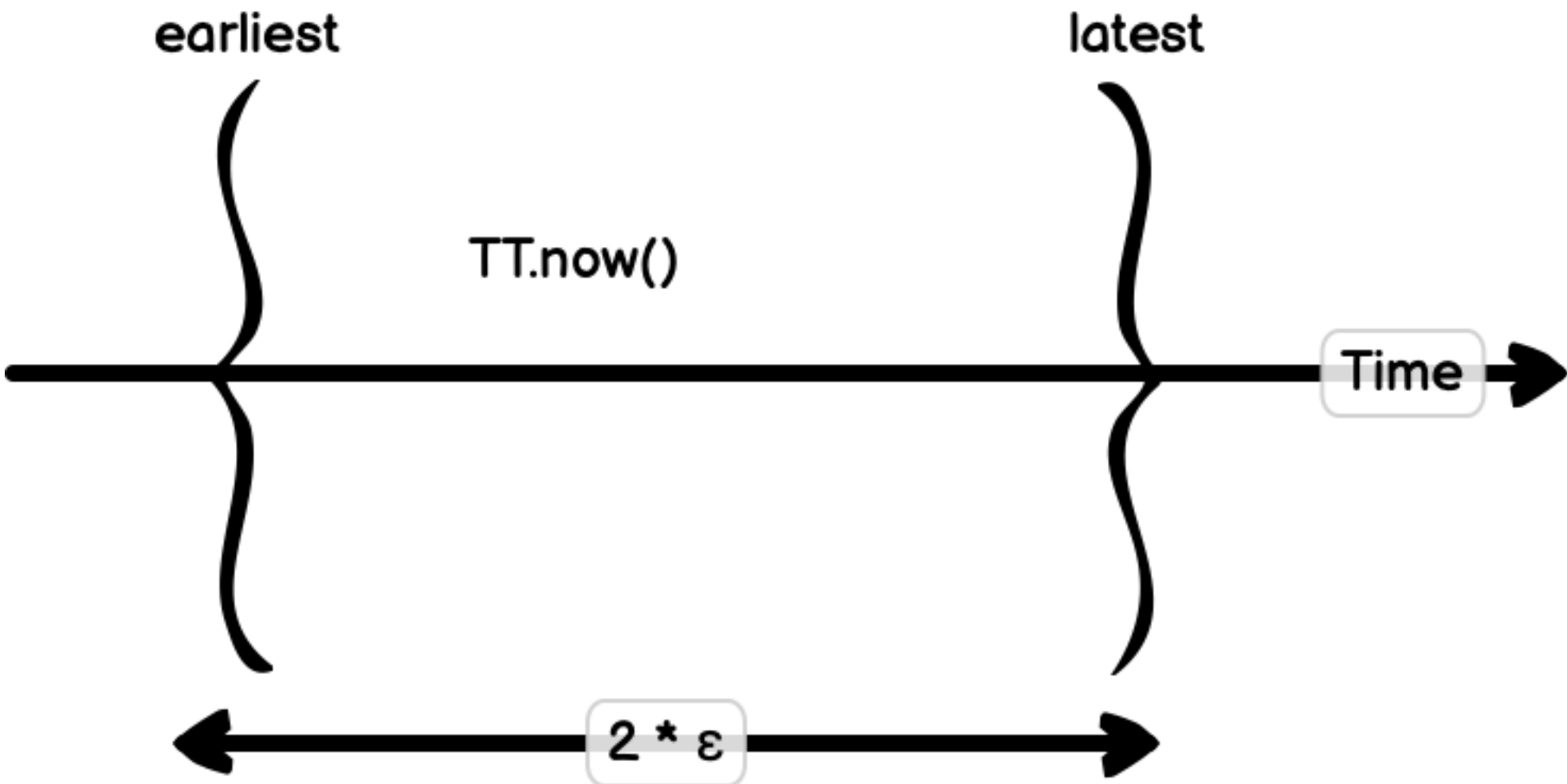
- A global synchronized clock with bounded non-zero error
 - if T2 starts to commit after T1 finishes committing, then the timestamp for T2 is greater than the timestamp for T1
- Underlying source of time: a combination of GPS receivers and atomic clocks
 - GPS Time Master: These nodes are equipped with GPS receivers which receive GPS signals include time information directly from satellites.
 - Armageddon Master: These nodes are equipped with local Atomic clocks. Atomic clocks are used as a supplement to GPS time masters in case satellite connections become unavailable.



GOOGLE TRUETIME API



Method	Returns
<i>TT.now()</i>	<i>TTinterval</i> : [<i>earliest</i> , <i>latest</i>]
<i>TT.after(t)</i>	true if <i>t</i> has definitely passed
<i>TT.before(t)</i>	true if <i>t</i> has definitely not arrived





TAKEAWAYS

- Time is crucial to distributed system coordination.
 - Disagreement between machines can result in undesirable behavior.
- Approaches:
 - Physical time: Often inadequate for distributed systems, need ns precision
 - Logical time: Lamport clocks, happens-before relation
- Next class: **Agreement**



ACKNOWLEDGEMENT

THIS COURSE IS DEVELOPED HEAVILY BASED ON COURSE MATERIALS SHARED BY PROF. INDRANIL GUPTA, PROF. ROBERT MORRIS, PROF. MICHAEL FREEDMAN, PROF. KYLE JAMIESON, PROF. WYATT LLOYD AND PROF. ROXANA GEAMBASU. MANY APPRECIATIONS FOR GENEROUSLY SHARING THEIR MATERIALS AND TEACHING INSIGHTS.
