# Distributed Systems 1

CUCS Course 4113
https://systems.cs.columbia.edu/ds1-class/

Instructor: Roxana Geambasu

# Time and Synchronization

# 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**, and will see how unbounded **network delays** (a.k.a. network asynchrony) creep up into semantics and challenge coordination.

# Outline

- Physical clocks
  - Synchronization challenges and protocols
- Logical clocks
  - Lamport clock protocol
- Examples
  - Global log for debugging
  - Mutual exclusion

# Why Is Time Important?

- Needed for synchronization and coordination.
- Examples:
  - Mutual exclusion
  - Barrier
  - 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
…
```

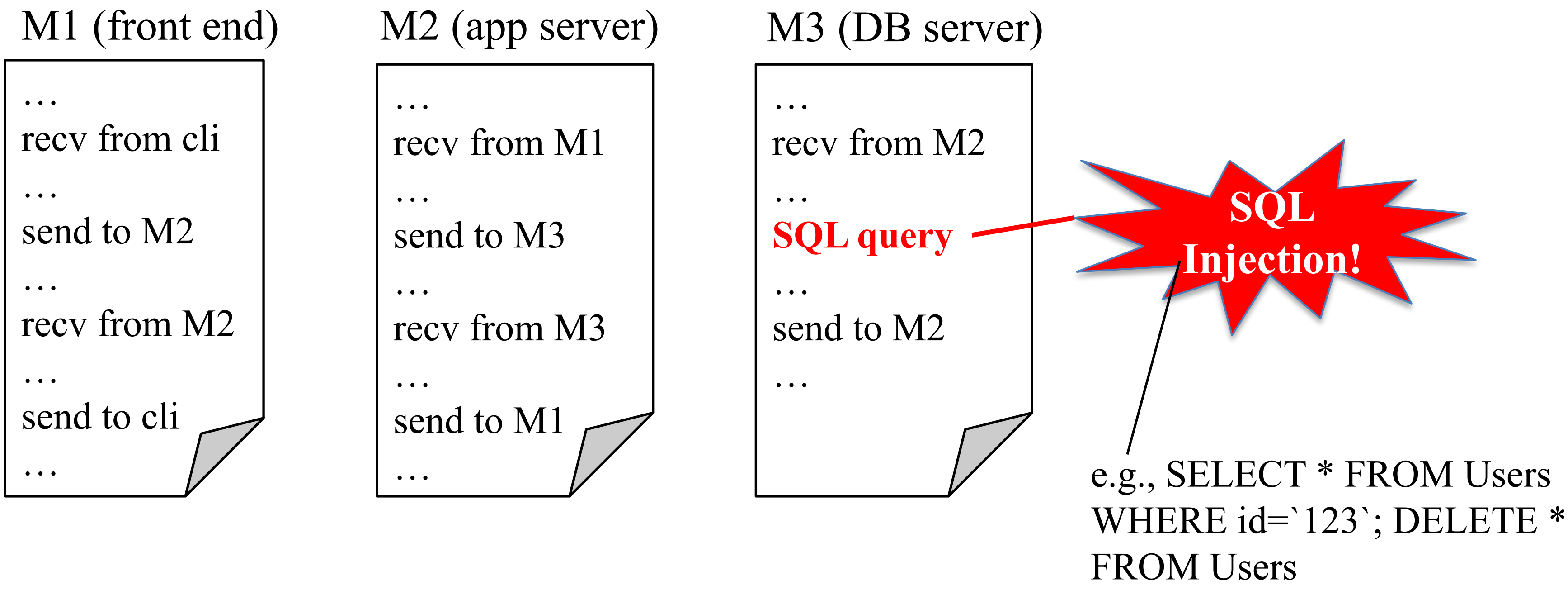
M2 (app server)

```
…
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
…

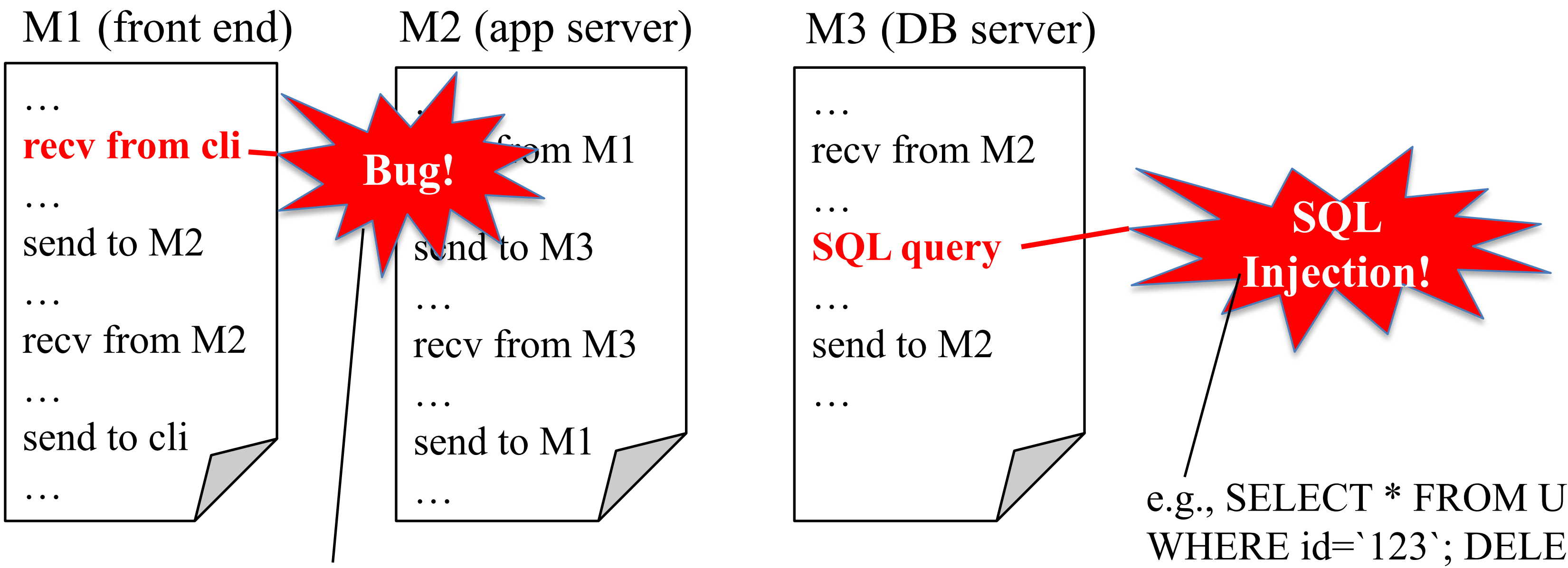
M3 (DB server)

…
recv from M2
…
**SQL query**
…
send to M2
…

**SQL Injection!**

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

# Example: Distributed Debugging



M1 (front end)

…
**recv from cli**
…
send to M2
…
recv from M2
…
send to cli
…

M2 (app server)

…
from M1
…
send to M3
…
recv from M3
…
send to M1
…

M3 (DB server)

…
recv from M2
…
**SQL query**
…
send to M2
…

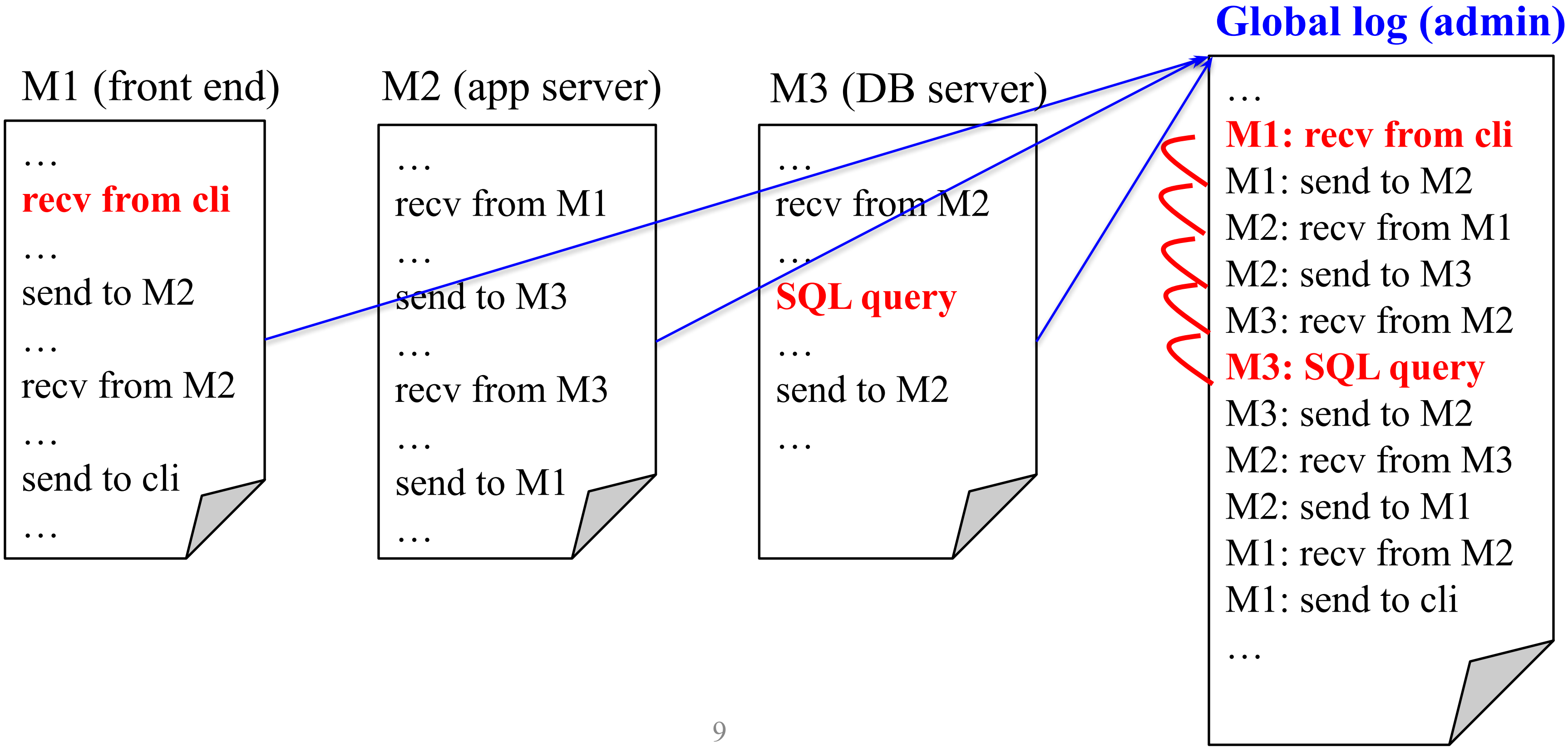
**Bug!**

e.g., improper sanitization

**SQL Injection!**

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

# 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
…

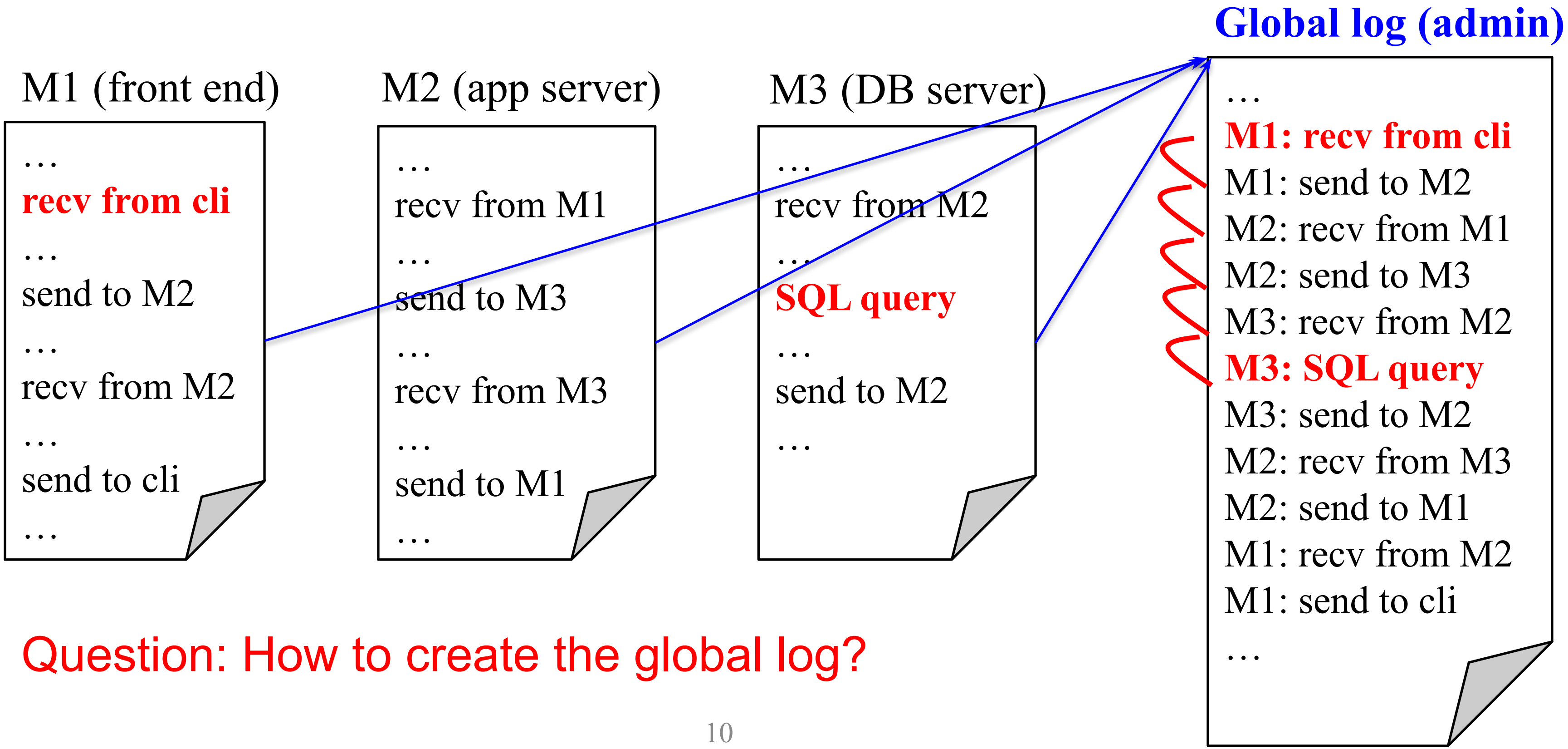
M3 (DB server)

…
recv from M2
…
**SQL query**
…
send to M2
…

**Global log (admin)**

…
**M1: recv from cli**
M1: send to M2
M2: recv from M1
M2: send to M3
M3: recv from M2
**M3: SQL query**
M3: send to M2
M2: recv from M3
M2: send to M1
M1: recv from M2
M1: send to cli
…

# 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)

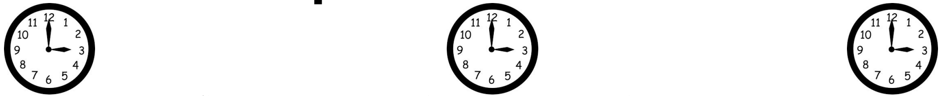
…
recv from M2
…
**SQL query**
…
send to M2
…

**Global log (admin)**

…
**M1: recv from cli**
M1: send to M2
M2: recv from M1
M2: send to M3
M3: recv from M2
**M3: SQL query**
M3: send to M2
M2: recv from M3
M2: send to M1
M1: recv from M2
M1: send to cli
…

Question: How to create the global log?

# Example: Distributed Debugging



M1 (front end)

…
**t1 recv from cli**
…
t2 send to M2
…
t10 recv from M2
…
t11 send to cli
…

M2 (app server)

…
t3 recv from M1
…
t4 send to M3
…
t8 recv from M3
…
t9 send to M1
…

M3 (DB server)

…
t5 recv from M2
…
**t6 SQL query**
…
t7 send to M2
…

**Global log (admin)**

…
**t1 M1: recv from cli**
t2 M1: send to M2
t3 M2: recv from M1
t4 M2: send to M3
t5 M3: recv from M2
**t6 M3: SQL query**
t7 M3: send to M2
t8 M2: recv from M3
t9 M2: send to M1
t10 M1: recv from M2
t11 M1: send to cli
…

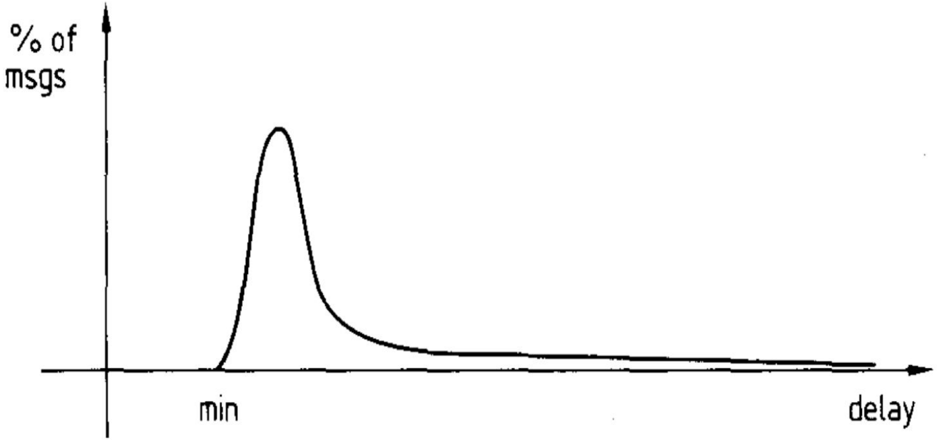
Question: How to create the global log?
Answer: Use physical clock?

# Problem: Clock Synchronization Is Hard

- Machines have different physical clocks, which are never identical from a structural perspective.
  - E.g. for quartz clocks: the crystals differ inside the clocks; surrounding electro-magnetic field, temperature affect oscillators.
- Synchronizing clocks to reset their drift involves the network, whose delays can vary over time and in general, cannot be upper bounded.

# Asynchronous Networks

- In real settings, we have to model the network as **asynchronous**, meaning:
  - a lower bound, "min"
  - a "modus operandi"
  - **BUT no hard upper bound**.



- Some algorithms assume a known upper bound (a.k.a., **synchronous** network model), but this is not realistic (e.g., buggy router, queuing, attacks).

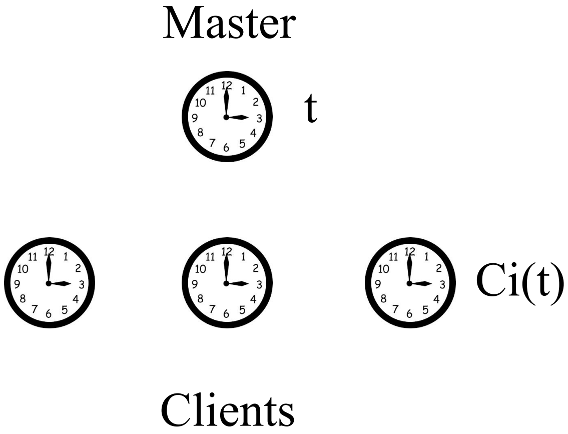
# Asynchronous Systems

- Not only networks behave asynchronously.
- **Computation** behaves similarly in real life (there is no guaranteed execution time for any operation, it all depends on how loaded the machine is for example).
- **Asynchronous systems** are those where both the network and the computation are modeled as asynchronous. Remember this concept because we'll return to it profusely in this class.

# Synchronization Protocols

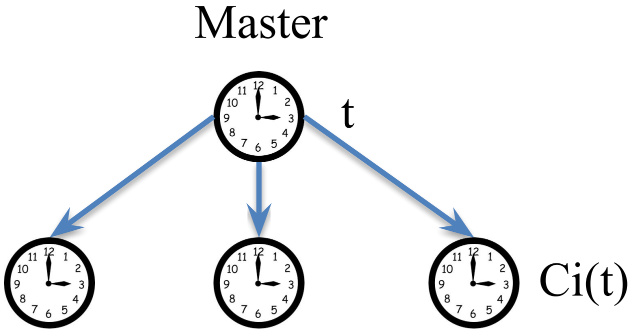
- Best-known algorithm is NTP (network time protocol, original paper [Mills-1991]).
- Synchronizes to reference clocks (Greenwich for the public Internet).
- Over WAN, synchronization is within **tens of ms**. Great to rely on for human coordination, but problematic for machine coordination.
- We'll build up to the basis for the NTP protocol.

# Notation



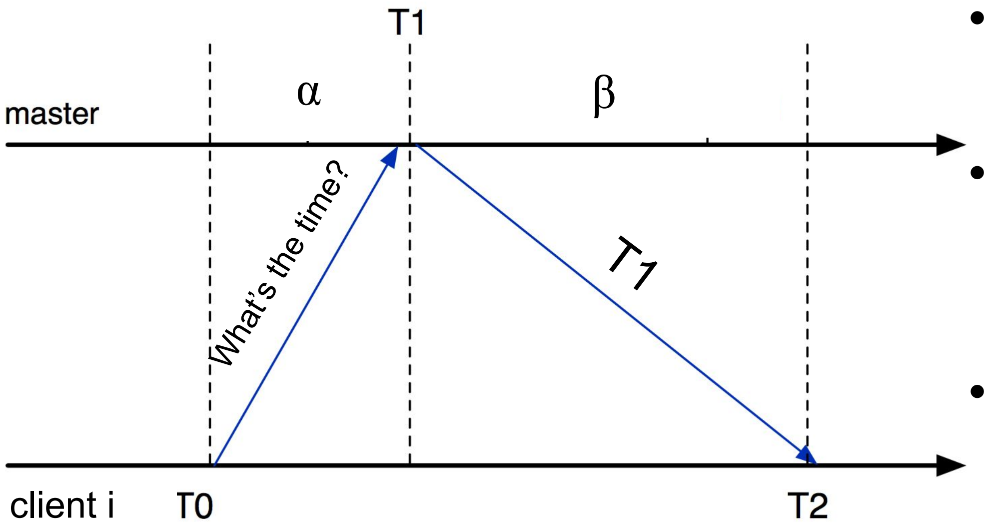
- Master clock keeps time t, which is assumed to be correct.
- Client clocks Ci that we want to synchronize to master keep clocks Ci(t).
- We want two properties:
  - Clock consistency (internal): |Ci(t)–Cj(t)|<d1 for all i, j
  - Clock accuracy (external): |Ci(t)–t|<d2 for all i
- External implies internal.

# Protocol 1: Broadcast-Based Sync



- Master broadcasts t to all.
- Client i sets its clock to (t+min) when it gets a message at some time t'. So, **Ci(t') := t+min**.
- If we assume a "max" delay, then the error between any client and the master is bounded by (max-min), which can be proven optimal.

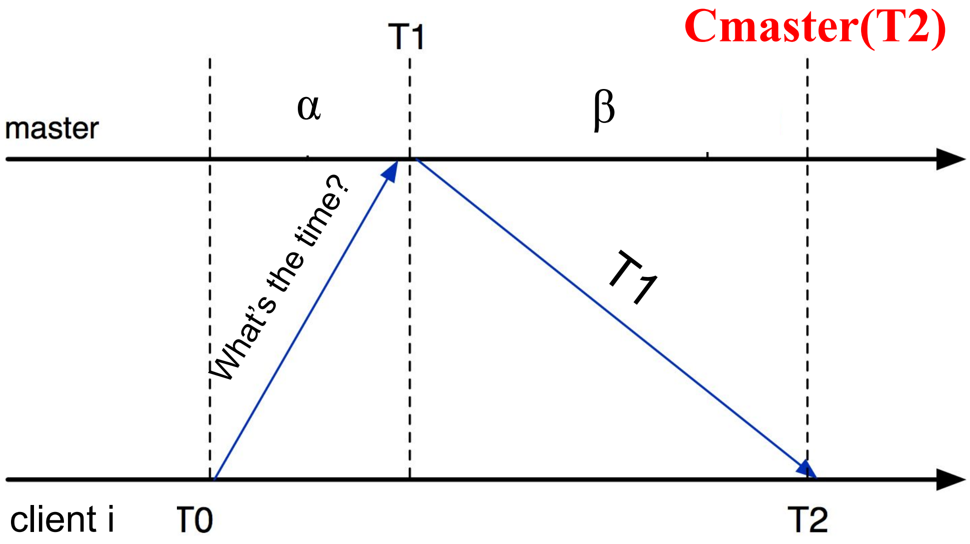
# Protocol 2: Interrogation-Based Sync



- Client i sends query to master and saves the local time, T0.
- Upon receipt of msg, master takes the local time, T1, and replies with it to client.
- Upon receipt of master reply, querier takes its local time, T2, and updates its clock.

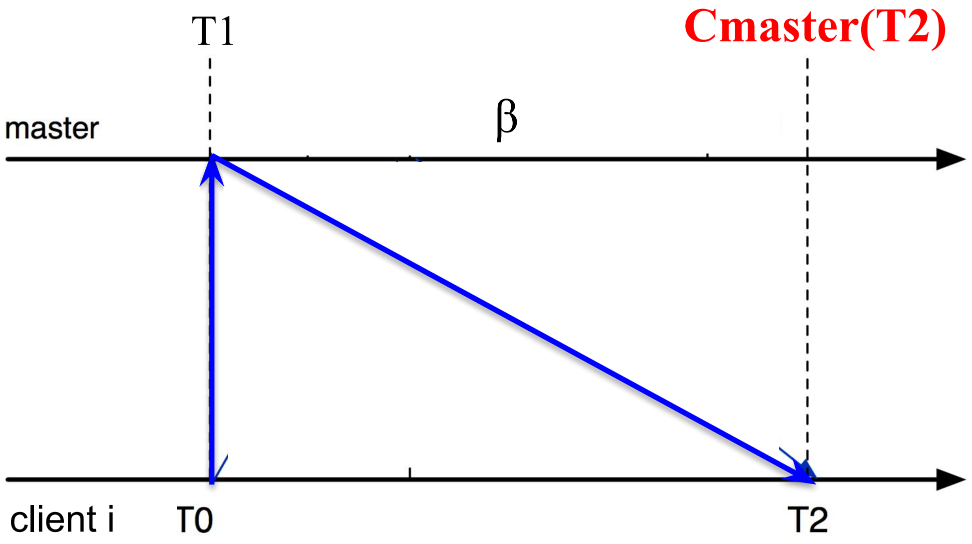
**Question: TO WHAT VALUE to ensure minimal error?**

# Protocol 2: Interrogation-Based Sync



- Client i sends query to master and saves the local time, T0.
- Upon receipt of msg, master takes the local time, T1, and replies with it to client.
- Upon receipt of master reply, querier takes its local time, T2, and updates its clock. **Question: TO WHAT VALUE to ensure minimal error?**

# Protocol 2: Interrogation-Based Sync

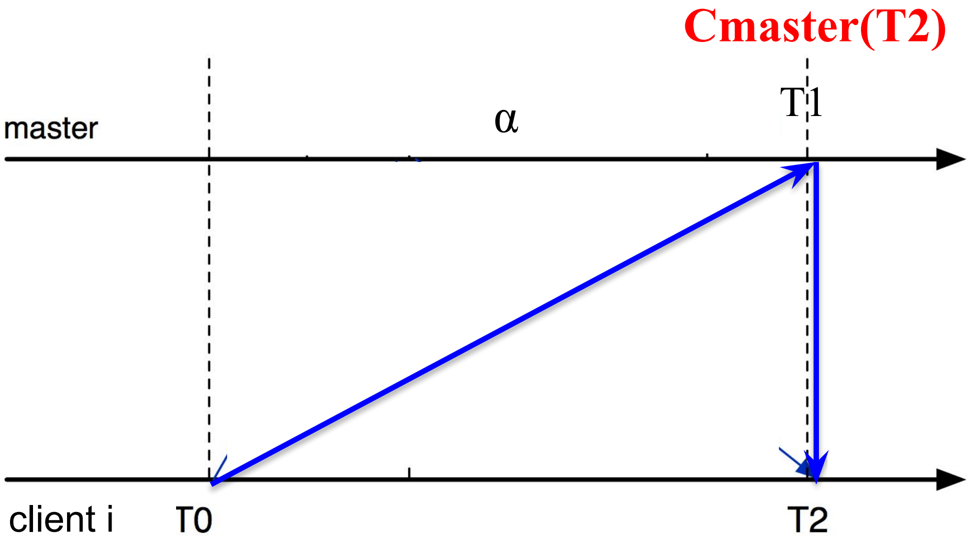


Assume min=0.

Extreme Case 1: **α=0**

- Then, β=(T2-T0).
- Cmaster(T2)=T1+(T2-T0)
- In this case, client i would ideally set its time to T1+(T2-T0) for zero error.

# Protocol 2: Interrogation-Based Sync



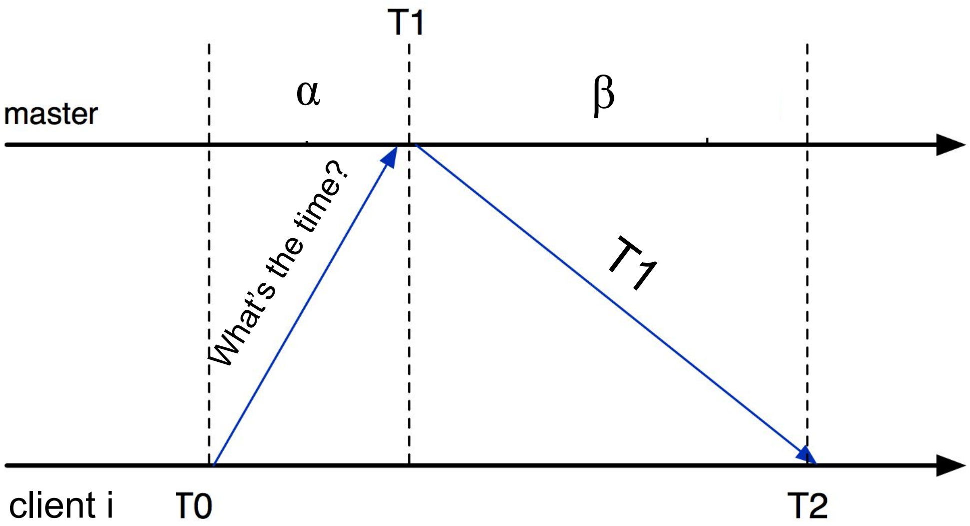
Assume min=0.

Extreme Case 2: **β=0:**

- Then, α=(T2-T0).
- Cmaster(T2)=T1.
- In this case, client i would ideally set its time to T1 for zero error.

Problem: Client cannot differentiate between Extreme Cases. So, best it can do is to set its time to the midway.

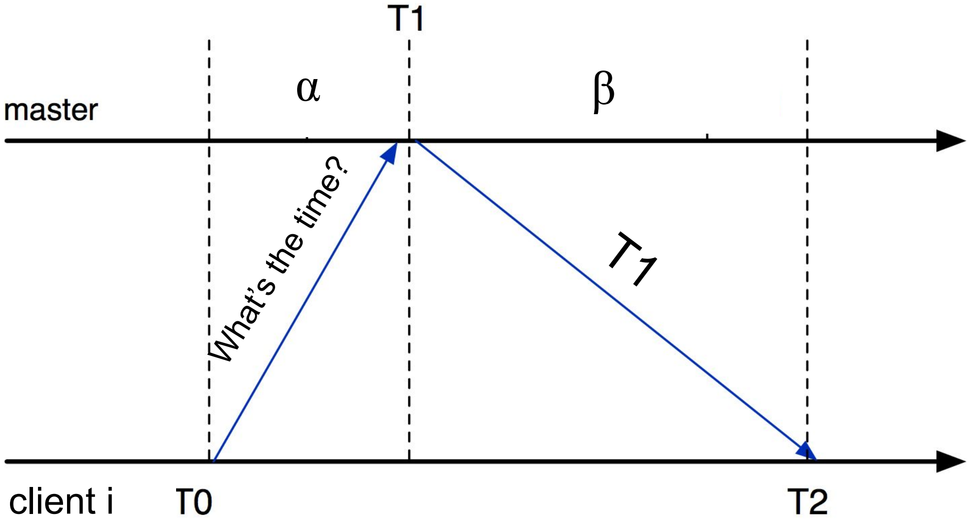
# Protocol 2: Interrogation-Based Sync



The client set its local time to the midpoint between these two extremes, which is measurable:

**Ci(T2) := T1 + (T2–T0)/2**

# Protocol 2: Interrogation-Based Sync

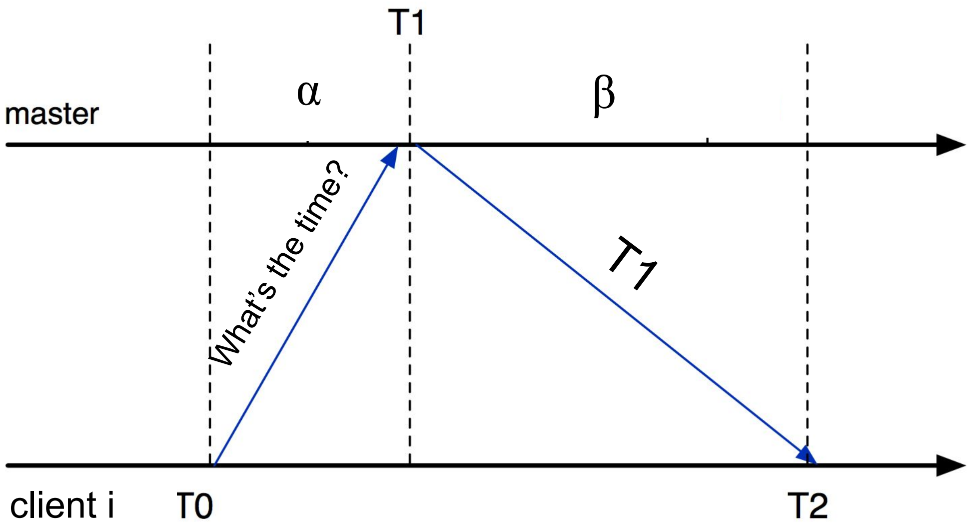


The client set its local time to the midpoint between these two extremes, which is measurable:

**Ci(T2) := T1 + (T2–T0)/2**

**What's the max error?**

# Protocol 2: Interrogation-Based Sync

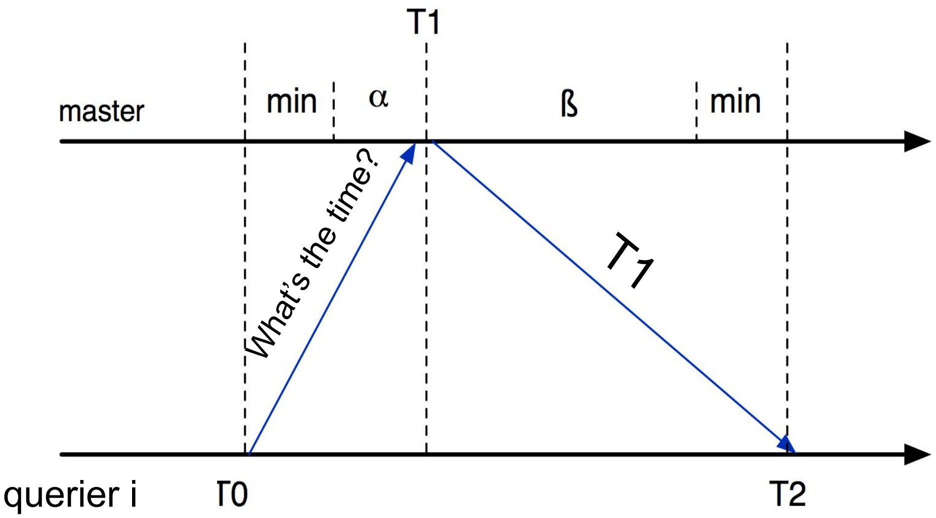


The client set its local time to the midpoint between these two extremes, which is measurable:

**Ci(T2) := T1 + (T2–T0)/2**
**max error = (T2-T0)/2**

Max error is achieved in Extreme Cases 1 and 2, when Ci is either **behind** or **ahead**.

# Protocol 2: Interrogation-Based Sync



- Preceding analysis ignores min.
- Also ignores drift of client clock during T0 to T2 period. If you include it, the equation gets a bit more complicated.
- It also ignores local delays, e.g., T2 is taken but Ci is updated later. If you include this, then things get even more complicated.
- More complications in a real protocol like NTP arise from ensuring scalability, FT for master.

# Implications

- Error diminishes as the measurement trial **RTT** approaches 2*min.
  - is a probabilistic tradeoff
  - can require measurements to be close to RTT to “accept” them and achieve rapport – increase number of trials necessary, but get tight error bounds
  - can be sloppy and take any measurement – decreases number of trials, but get worse error bounds

- **Thus, network delay impacts clock synchronization and prevents it from ever being perfect**.
  - NTP over WAN typically has an error of tens of milliseconds.
  - GPS synchronization is much tighter (low min, alpha, beta), but still non zero.

# Outline

- Physical clocks
  - Synchronization challenges and protocols
- **Logical clocks**
  - **Lamport clock protocol**
- Examples
  - Global log for debugging
  - Mutual exclusion

# 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.

# Logical Clock Requirements

Lamport posited two requirements for logical clocks:

1. They must preserve **program order** (i.e., the order of events in one process needs to be preserved by the logical clock)
2. 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.

# 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 $C_j$ (i.e., $C_j$++)

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

| $C_1$ | 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 |
| ? | … |

| $C_2$ | M2 (app server) |
|---|---|
| 0 | ?? op21 rcv M1 |
| ? | ?? op22 … |
| ? | ?? op23 ... |
| ? | ?? op24 snd M3 |
| ? | ?? op25 … |
| ? | ?? op26 … |
| ? | ?? op27 … |
| ? | ?? op28 … |
| ? | ?? op26 rcv M3 |
| ? | ?? op27 … |
| ? | ?? op28 snd M1 |
| ? | … |

| $C_3$ | 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 Ci=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, $C_{1\text{-}3}$. 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 receive:
    - $C_i = \max(C_i, T_m+1)$
    - $C_i$++

# Student Worksheet

| $C_1$ | 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 |
| ? | … |

| $C_2$ | 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 |
| ? | … |

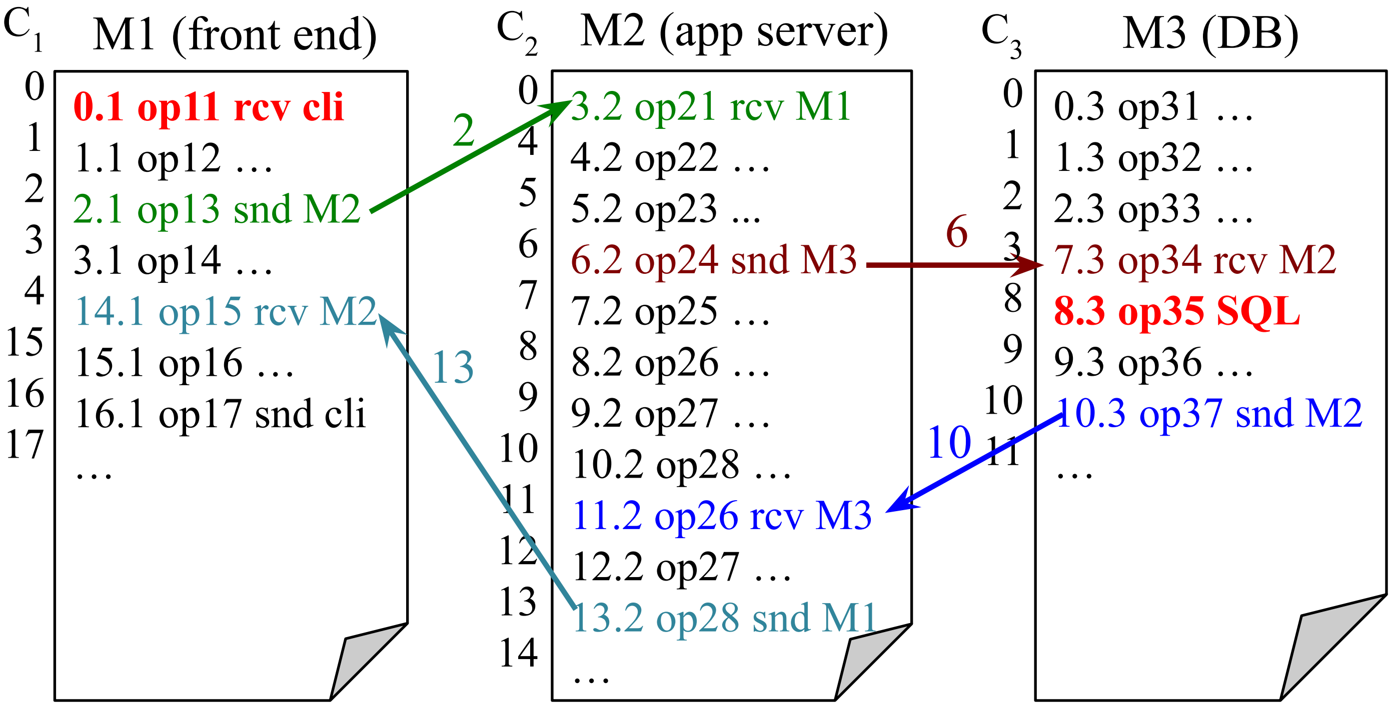
| $C_3$ | M3 (DB) |
|---|---|
| 0 | 0.3 op31 … |
| ? | ?? op32 … |
| ? | ?? op33 … |
| ? | ?? op34 rcv M2 |
| ? | ?? op35 SQL |
| ? | ?? op36 … |
| ? | ?? op37 snd M2 |
| ? | … |

## Global Log

0.1 op11 rcv cli

… enter all events in order of their logical timestamp

# Solution



$C_1$ M1 (front end)

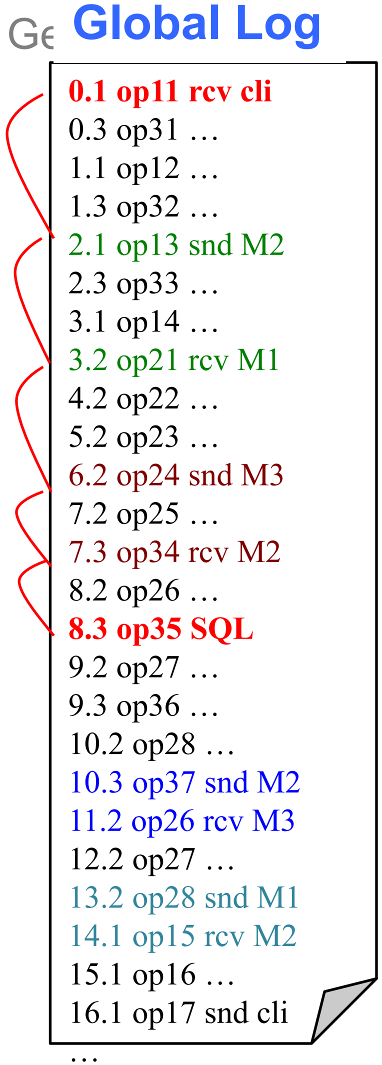
0 **0.1 op11 rcv cli**
1 1.1 op12 …
2 2.1 op13 snd M2
3 3.1 op14 …
4 14.1 op15 rcv M2
15 15.1 op16 …
16 16.1 op17 snd cli
17 …

$C_2$ M2 (app server)

0 3.2 op21 rcv M1
4 4.2 op22 …
5 5.2 op23 ...
6 6.2 op24 snd M3
7 7.2 op25 …
8 8.2 op26 …
9 9.2 op27 …
10 10.2 op28 …
11 11.2 op26 rcv M3
12 12.2 op27 …
13 13.2 op28 snd M1
14 …

$C_3$ M3 (DB)

0 0.3 op31 …
1 1.3 op32 …
2 2.3 op33 …
3 7.3 op34 rcv M2
8 **8.3 op35 SQL**
9 9.3 op36 …
10 10.3 op37 snd M2
11 …



**Global Log**

**0.1 op11 rcv cli**
0.3 op31 …
1.1 op12 …
1.3 op32 …
2.1 op13 snd M2
2.3 op33 …
3.1 op14 …
3.2 op21 rcv M1
4.2 op22 …
5.2 op23 …
6.2 op24 snd M3
7.2 op25 …
7.3 op34 rcv M2
8.2 op26 …
**8.3 op35 SQL**
9.2 op27 …
9.3 op36 …
10.2 op28 …
10.3 op37 snd M2
11.2 op26 rcv M3
12.2 op27 …
13.2 op28 snd M1
14.1 op15 rcv M2
15.1 op16 …
16.1 op17 snd cli
…

# Another Example: Mutual Exclusion

- https://columbia.github.io/ds1-class/lectures/04-clocks-mutex-example-ppt.pdf
- slides 2 and 13-23

# Pluses and Minuses of Lamport Clocks

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

- 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.

# Next Classes

- Diverge a bit from Lamport clocks, but we’ll return.
- Essentially, Lamport clocks are used in many coordination protocols, including protocols that solve consensus, a key coordination problem in DS with many instantiations.
- We next formulate the consensus and related problems and we’ll return to Lamport clocks when we discuss the solution.

# Key Papers

- [Mills-1991] David Mills. *Internet Time Synchronization: the Network Time Protocol*. In *IEEE Transactions on Communications, 1991.*
- [Lamport-1978] Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*. In *Communications of the ACM*, 1978.