Logical Clocks

Operating
Systems

R. Stockton Gaines
Editor

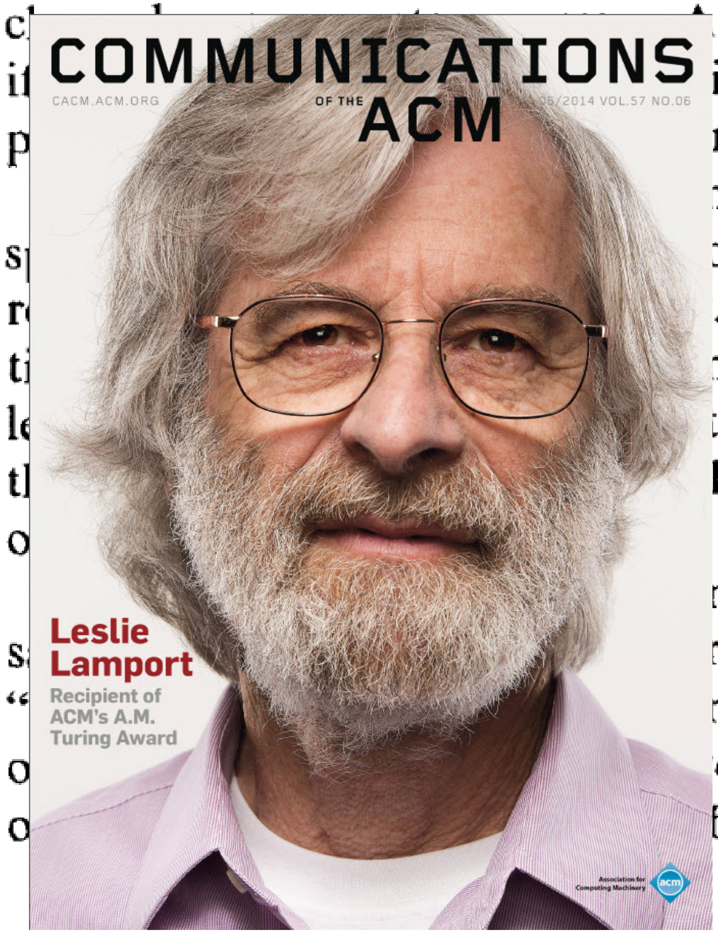
Time, Clocks, and the Ordering of Events in a Distributed System

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

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
c   system is distributed
if   is not negligible com-
p   n a single process.
      iarily with systems of
s   owever, many of our
r   . In particular, a mul-
ti   nputer involves prob-
le   ted system because of
tl   h certain events can
o

      metimes impossible to
s   red first. The relation
“   nly a partial ordering
o   e found that problems
o   fully aware of this fact



Time

- On a single process/server, we can tell which event occurred first by looking at the system clock value
- Time helps order events. Timestamp(a) < Timestamp (b) : a happened before b
- Also useful for capturing (potential) causality:
  - Timestamp(a) < Timestamp(b) : a could have potentially caused b

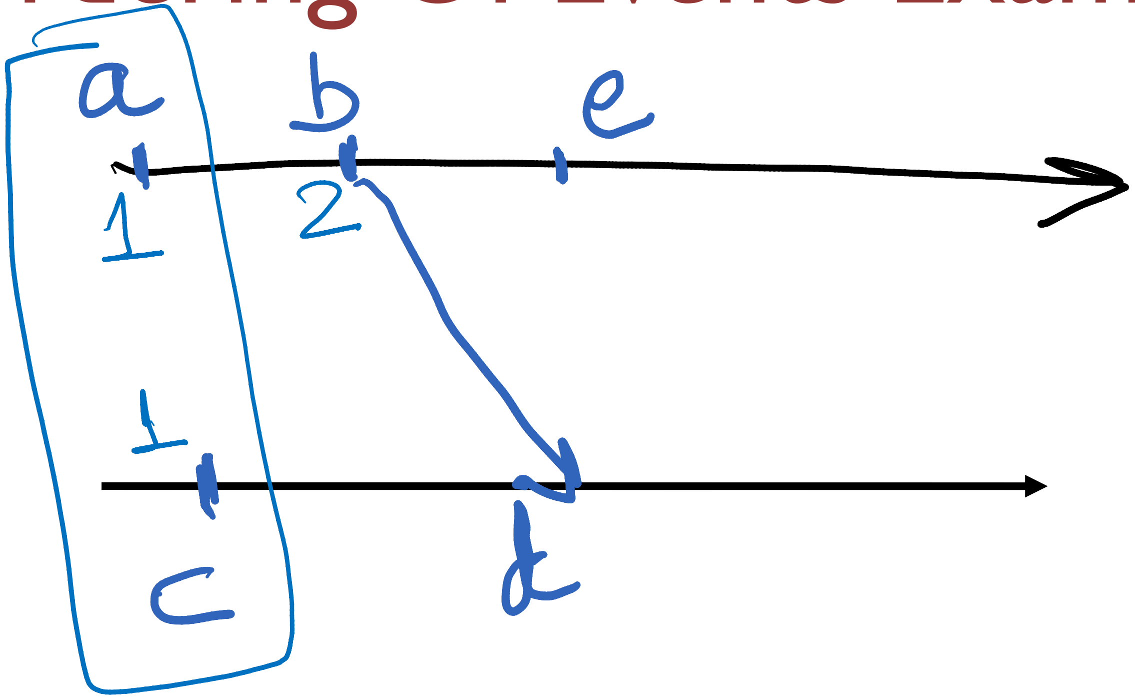
- BUT: Distributed Systems have no shared global clock
- Can’t compare timestamps across machines
- Even if we could, can we solve coordination problems without using physical time?
  - Capture the essence of what timestamps are.

Ordering Of Events

- Want to capture causality. If a could have potentially caused b, then a happened before b.
  - Similar to space-time diagrams in relativity (light-cone).
- Messages reflect the entire causal history upto that point.

- Happened-before relation:
  1. If a and b are two events that occur in the same process, and a comes before b, then a→b
  2. If a corresponds to sending a msg, and b is the receipt of that message, then a→b
  3. Transitive: a→b and b→c, then a→c
- This is a partial ordering of events in a system

Ordering Of Events Example



$a \to b$ $b \to d$ $b \to e$

$c \to d$

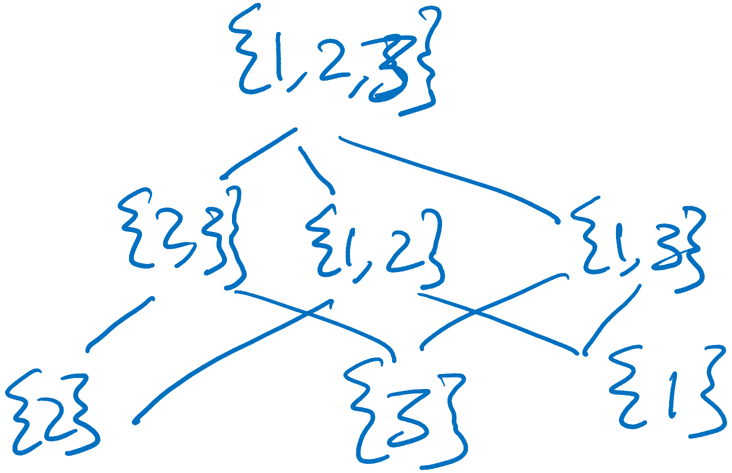
$a \parallel c$

$\neg(a \to c) \wedge$
$\neg(c \to a)$

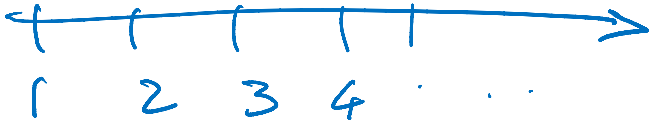
Partial order → Some events will be concurrent

Happened Before Relationship

- First fundamental result in distributed computing
- Partial order among events
  - Process-order, send-receive order, transitivity
- Important that all processes agree on the order of events



Total Order



for any i,j :

i<j, j<i, i=j

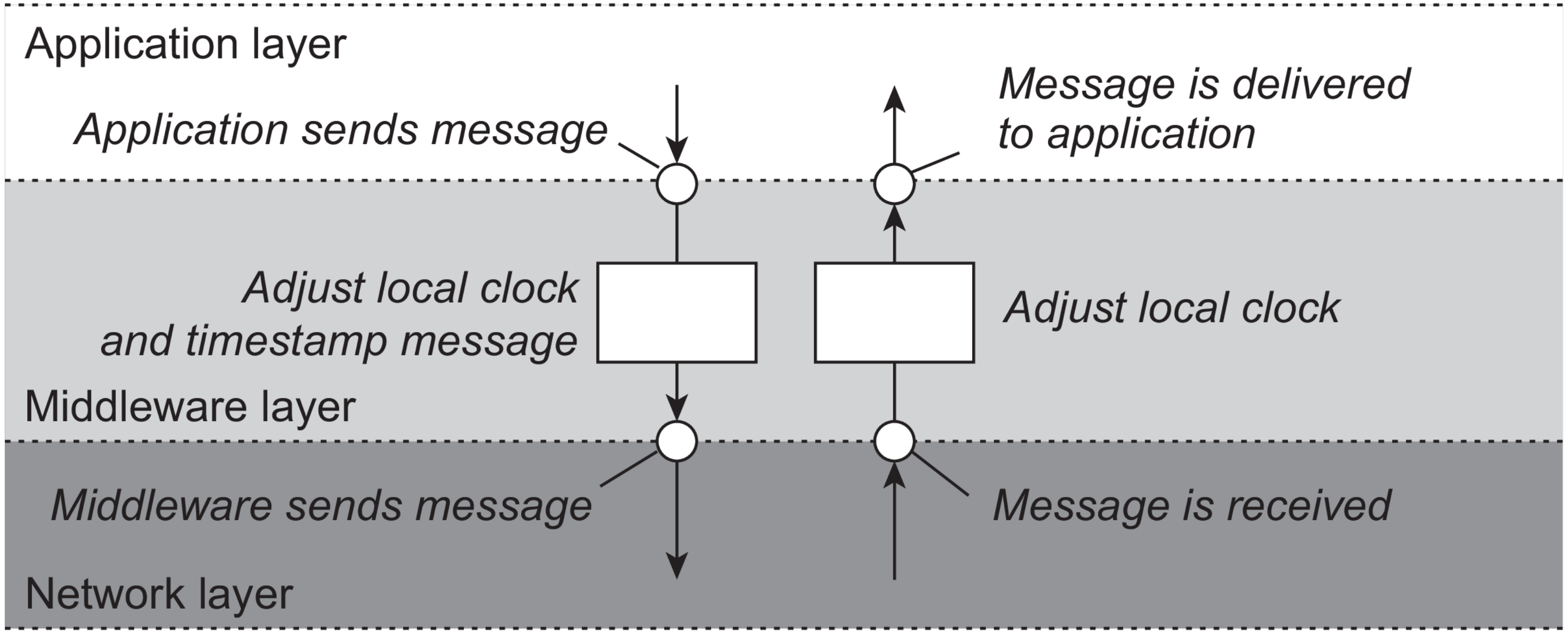
Logical Clocks

- How to maintain a global view of system behavior that is consistent with the happened-before relation?
- Approach: assign a timestamp C(e) to each event e, such that:
  - ① If a→b, then C(a) < C(b)
  - C must be monotonically increasing
- How to attach a timestamp to an event when there's no global clock?
- Maintain a consistent set of logical clocks, one per process

Lamport Clocks

- Each process maintains a local counter and adjusts this counter

1. New local event, increment
2. Send a timestamp with each message sent by   (i.e., ts(m) = sender's clock)
3. Whenever a message is received by adjusts its local counter
   = max{ ts(m), receiver clock} . Since this is a new event, increment

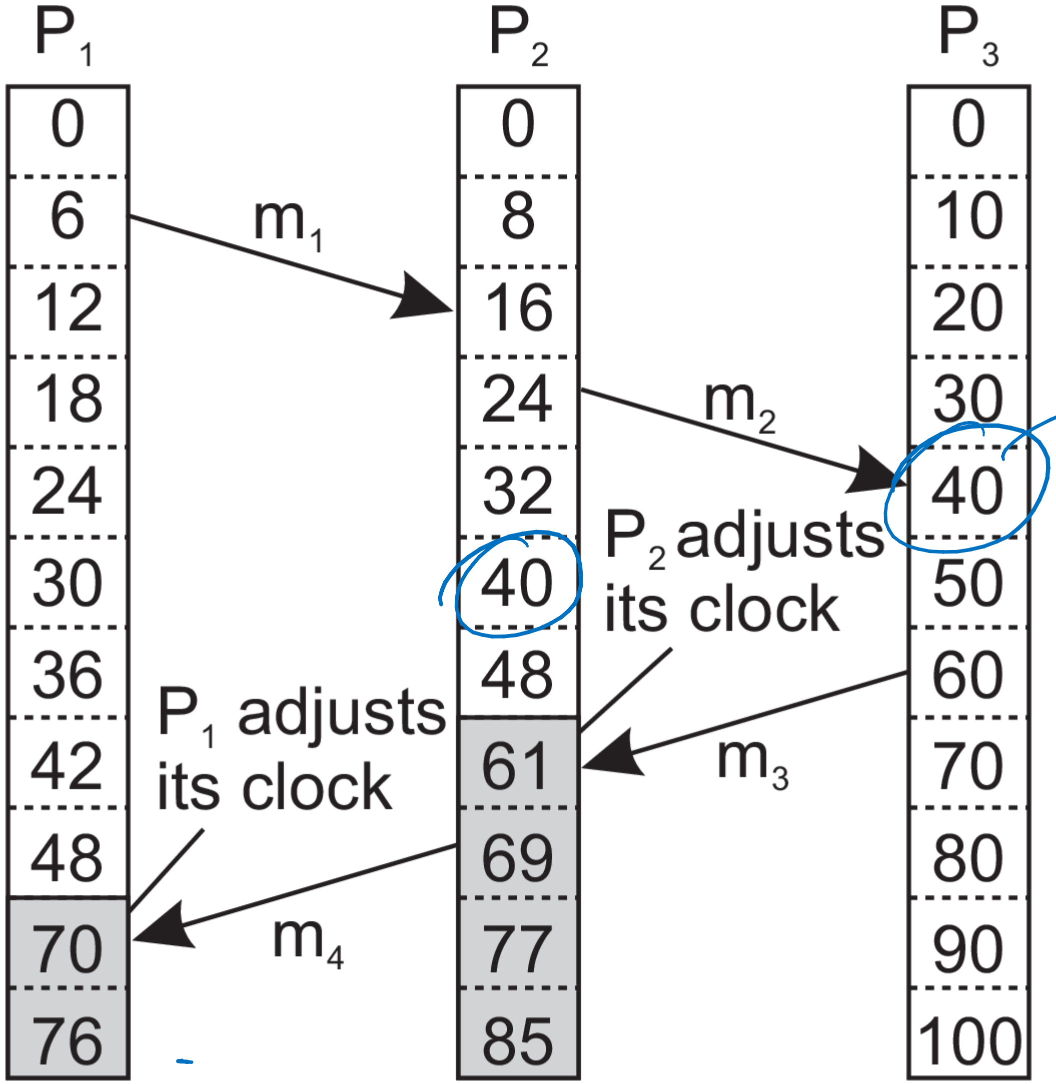


Lamport Clock Example

Actual UTC 01.00.000

Process 1: 01.01.000

Process 2: 01.02.000



40.3

40.2

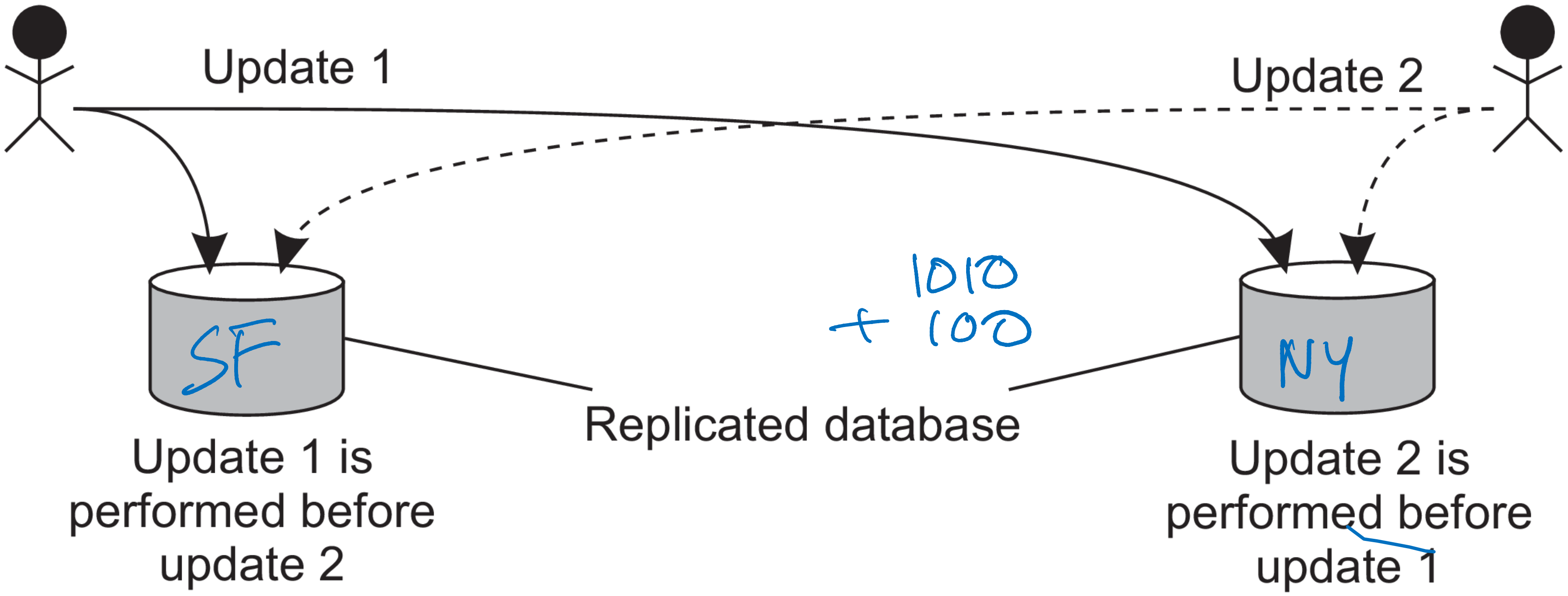
Total Ordering of Time-stamps

- It’s possible for $C_i\ (e_1) == C_j\ (e_2)$
- I.e., Lamport clock timestamps are not totally ordered
- We can break ties based on process-id
- For process P_i, the clock value becomes $C_i.i$
  - For example: 3.2 for process-id==2

Example: Total-ordered Multicast

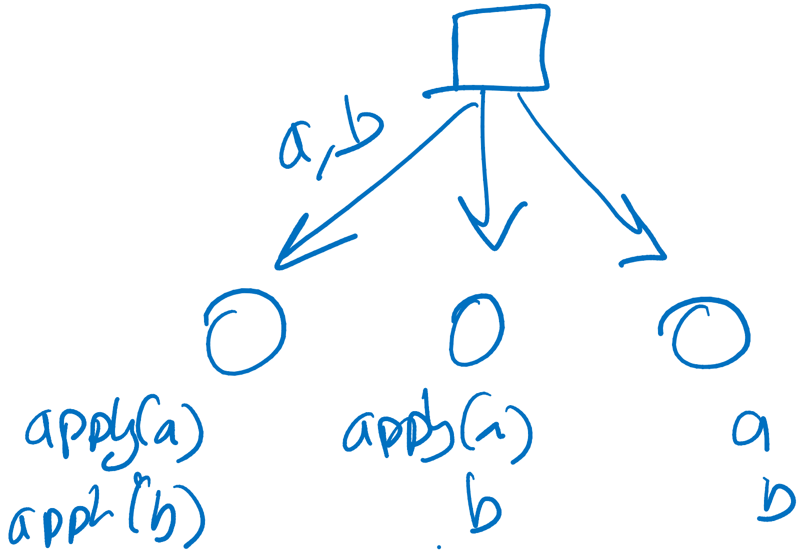
- Two replicas of database
- P1 adds $100 to an account (initial: $1000)
- P2 increments account by 1%
- Replica-1: $1111
- Replica-2: $1110

1000
+ 100



Totally Ordered Multicast

- Need to ensure that two update operations are performed in the same order by all nodes
- Actual order of operations is immaterial (add first or interest first)
- Totally ordered multicast: All messages are delivered in the same order to each receiver
- Assumption: No failures, and FIFO delivery of messages



Totally Ordered Multicast Intuition

- All messages timestamped with sender's logical time
- Sender sends to all recipients, including itself
- When a message is received, put it in a receive buffer/queue
- Requirement: All processes deliver messages in same sequence
  - Message at the head of the queue for all processes must be the same
    - 1. All processes must've received the message → Use acknowledgements
    - 2. All processes keep queue sorted based on some message property

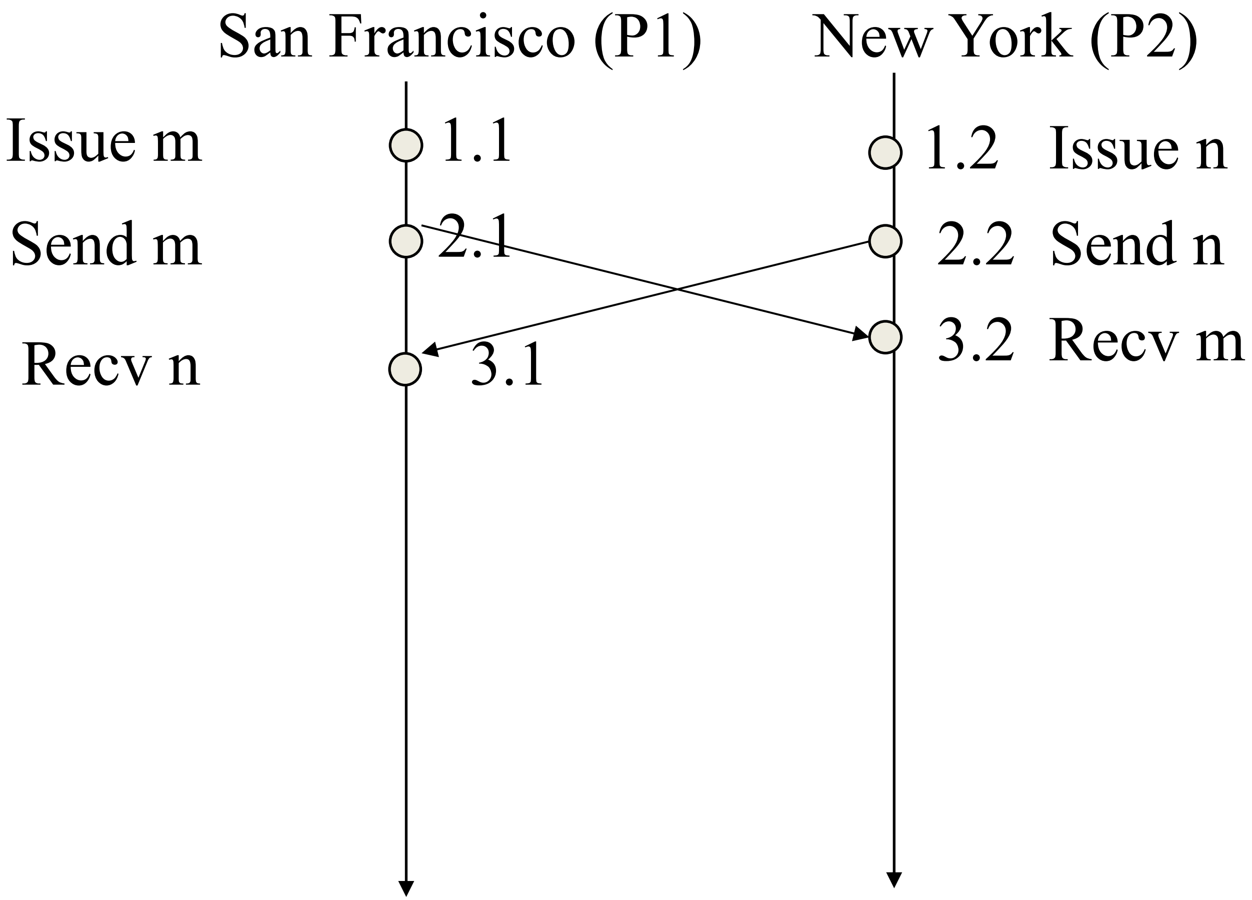
- Important FIFO message ordering assumption: Between pairs of processes, messages cannot arrive out of order.

Totally Ordered Multicast Algorithm : All acks multicast

- All messages timestamped with sender's logical time
- Sender sends to all recipients, including itself
- When a message is received:
  - 1. It is put into a local queue
  - 2. Queue is ordered based on timestamp
  - 3. The acknowledgement is multicast (with the receiver's logical time ofcourse)

- Message is delivered to application only when:
  1. It is at the head of the queue
  2. All the acknowledgement for that message have been received

Example: Totally Ordered Multicast



San Francisco (P1)

New York (P2)

Issue m 1.1

1.2 Issue n

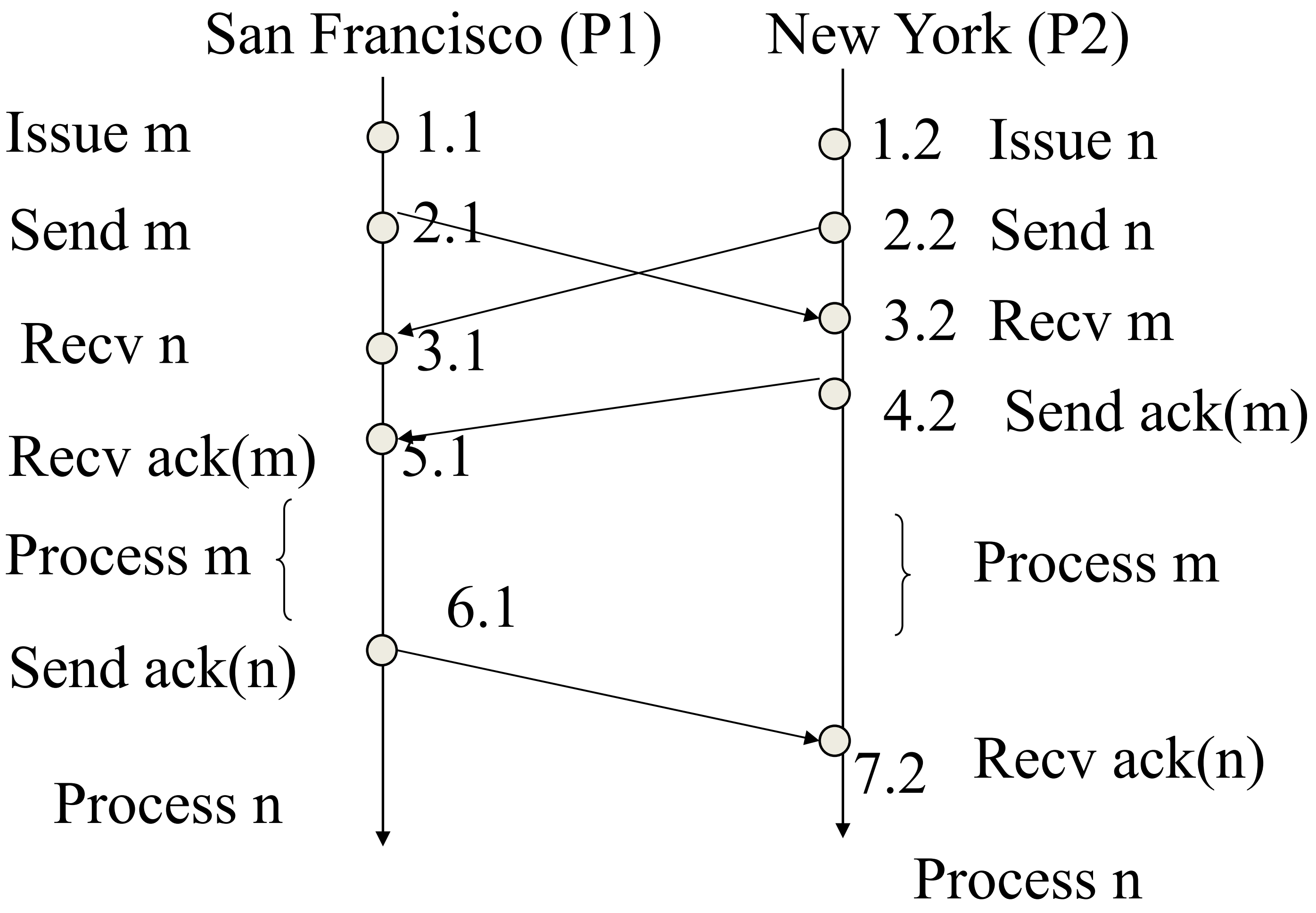
Send m 2.1

2.2 Send n

3.2 Recv m

Recv n 3.1

Example: Totally Ordered Multicast



Proof of Correctness

- Claim: All the messages will be delivered in the same order
- Proof by contradiction
- Let process A deliver i:M and B deliver j:N and wlog i<j
- B’s delivery means it has received all acks, including from A
- But has not received the original i:M message
- But this contradicts the FIFO message channel assumption
    - i:M was sent before the ack of the j:N message since i<j

Totally Ordered Multicast Algorithm : 2 Rounds

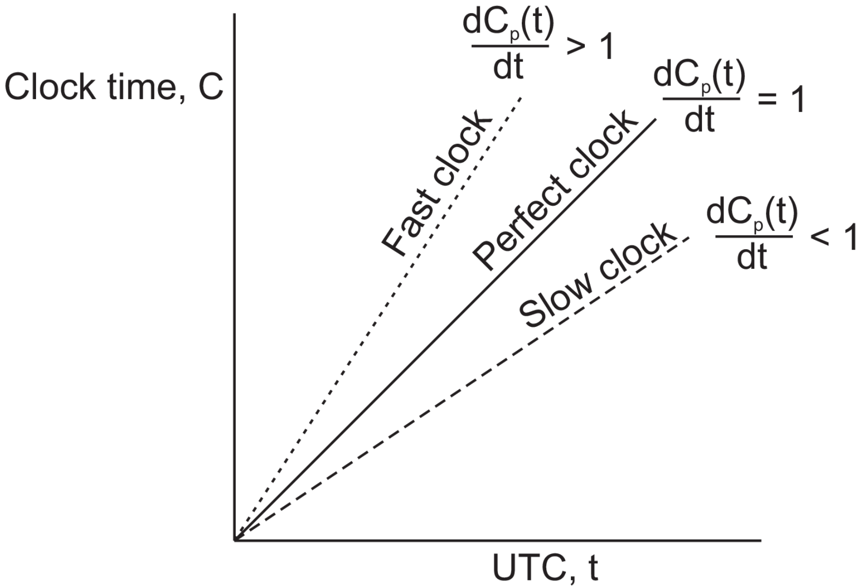
- All messages timestamped with sender's logical time
- Sender sends to all recipients, including itself
- When a message is received:
  - 1. It is put into a local queue
  - 2. Queue is ordered based on timestamp
  - 3. Send ack to original sender (no broadcast) if msg in head of queue
- Sender marks message 'ready' when it is head of queue and all acks rcvd
- Sender broadcasts second round of 'ready' messages to others.
- Message is delivered to application only when:
  1. It is at the head of the queue
  2. 'Ready' message has been received.

State Machine Replication

- Totally ordered multicast enables state machine replication
- Servers can be thought of deterministic state machines that change state based on the messages they receive.
- Replicating state machines has many benefits: fault-tolerance, performance..
  - If a server crashes, then contact other replicas
- With totally ordered multicast, all servers can execute the same operations in the same order
- Totally-ordered multicast is the “holy grail” of distributed computing
  - Yet we could somehow achieve it? Or did we??

Time in a conventional OS

- Processes can get "system time" via systemcalls
  - Such as gettimeofday() in UNIX
- Time-stamps can be used for ordering and coordination
- Example: make uses file modified time to decide what actions to run
- If the OS is the only time-source, then all processes observe the same time
- **Is it possible to synchronize all clocks in a distributed system?**



Physical Clocks and UTC

- Local time-keeping: crystal oscillator inside CPUs triggers timer interrupt in the OS
- Multiple CPUs 💻 difference in clock values
- Temperature (and other factors) also affect frequency 💻 **clock skew**
- Standardization: Universal Coordinated Time (UTC)
- Based on atomic clocks
- UTC “time” broadcast via radios, satellites, and through phone-numbers
    - NIST +1-808-335-4363
- Accuracy is around 0.5ms

Clock Synchronization

- Precision: Keep deviation between two clocks on any two machines within a specified bound

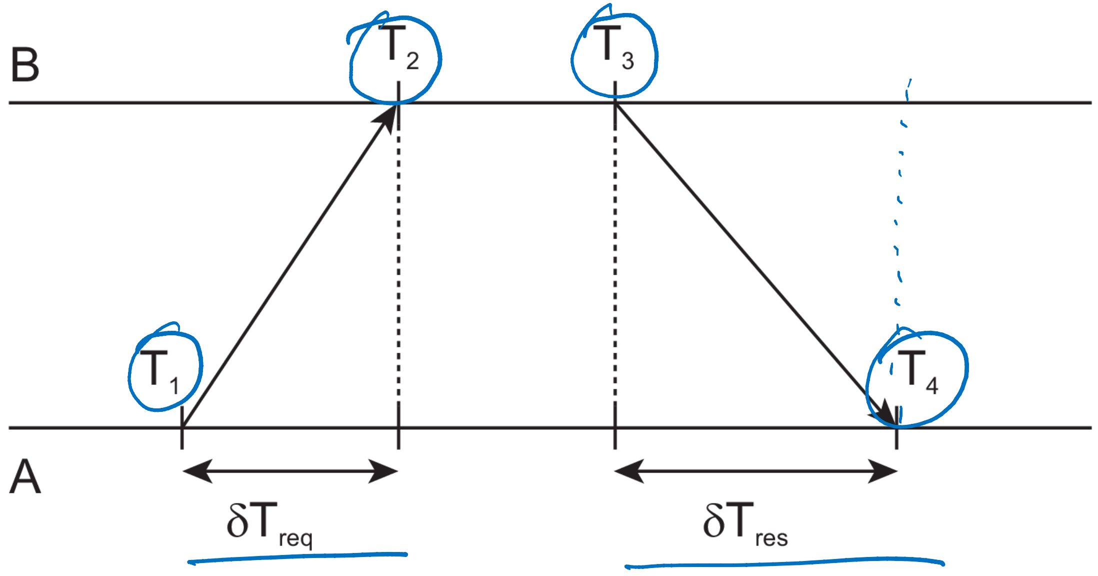
$$\forall t \quad \forall p,q : |C_p(t) - C_q(t)| \leq \pi$$

- Accuracy: The difference between all clocks and UTC time is less than \alpha

$$|C_p(t) - t| \leq \alpha$$

- Internal synchronization: Keep clocks precise
- External synchronization: Keep clocks accurate

NTP: Network Time Protocol



A adjusts its time by θ.

$$\theta = T_3 - T_4 + \delta$$

$$\delta = \frac{\delta T_{req} + \delta T_{resp}}{2}$$

$$\theta = T_3 - T_4 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

- Server B is a time-server with an accurate clock (say, atomic)
- Server A wishes to get it's clock synchronized with B's clock
  - By periodically polling B
- A asks B for the time
- But messages face network delays!
- Collect 8 pairs of (\theta,\delta) and choose \theta with lowest \delta
- Never set a clock backwards!
- Adjust rate of clock-ticks to slow-down or catch-up to A's time
- Hierarchy of NTP servers

Keeping Time Without UTC

- NTP allows for external synchronization with UTC time
- Sometimes, internal consistency suffices
- Time server polls all other machines, computes average time, and broadcasts it