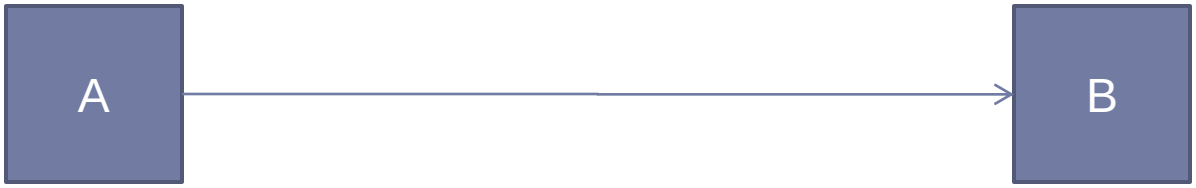
Synchronization algorithms

- Let's start assuming a **synchronous** system model.
  - *Remember*: This means known bounds for drift rate, transmission delays, and execution times.

- Say we have two processes A and B, and B wants to synchronize it's clock with that of A.
  - A sends it's time t to B.
  - B receives it after some transmission time $t_{transmit}$

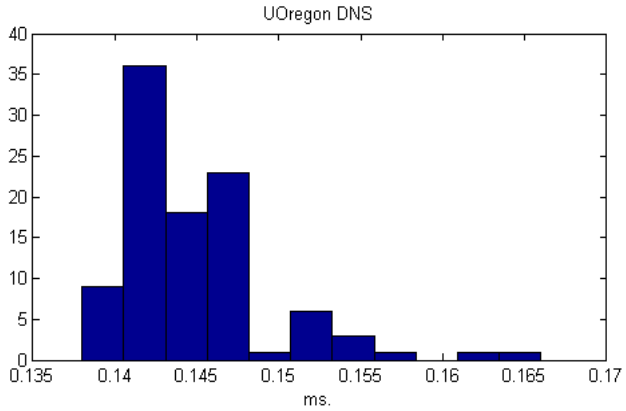


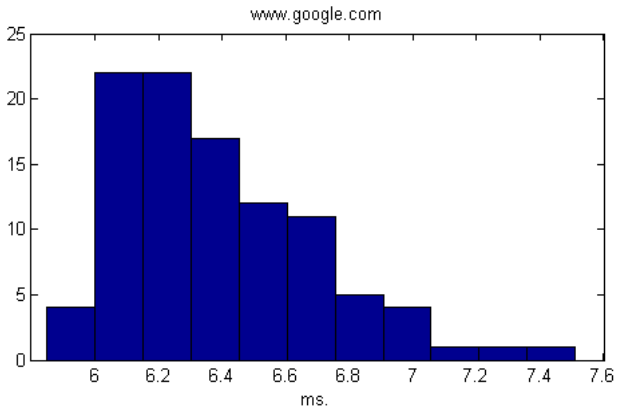
Synchronization algorithms

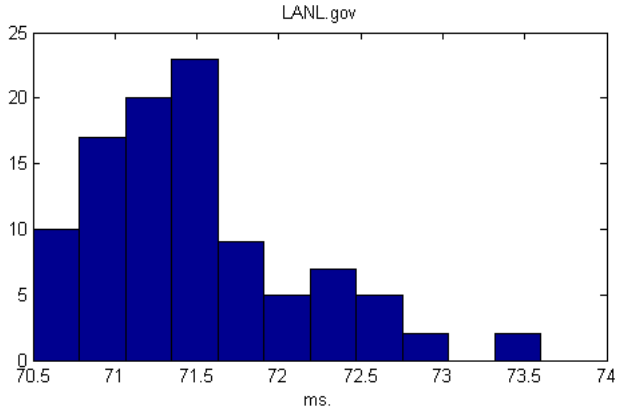
- Ideally, B would simply set it's clock to $t+t_{transmit}$.
- Why can't we do this?
  - We don't know $t_{transmit}$.
  - Why? Even though this is a synchronous system, we only know the bounds.
  - The actual transmission time can vary.
- The maximum time can be hard to determine.
- The minimum time can be estimated.
  - Measure round trip ping time over and over to characterize distribution of times.

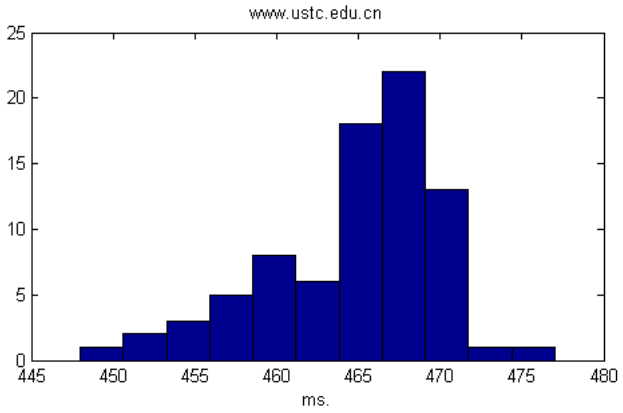
Estimating bounds

- Four example ping sequences.









Uncertainty

- Our uncertainty is quantified as:
  - U=(max-min)
- So, if a network provides consistent timings even in the presence of load variations, uncertainty will be low.
  - LANs can look like this.
    - Those local pings varied by about 0.02 ms.
- If it doesn't, uncertainty can be high.
  - The Internet usually looks like this.
    - Google varied by about 2ms, LANL by about 3, China about 25.

Uncertainty

- t+max : Skew of u units.
- t+min : Skew of u units.
- t+((min+max)/2) : Skew of u/2 units.
  - Why?
  - min and max are a full u units from each other.
  - (min+max)/2 is in the middle, so u/2 from the endpoints.

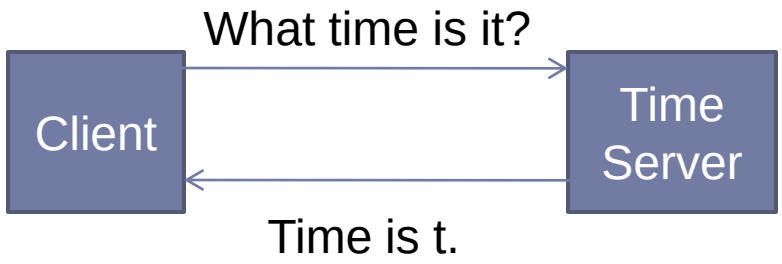
- Optimum bound on skew for N clocks:
  - u(1-1/N)
    - [Lundelius, Lynch 84]

Asynchronous systems

- The maximum is unknown. So, the best we can do:
  - $t_{transmit} = min + x, x \geq 0$
- Sampling can yield a distribution and we can estimate a reasonable max.
  - Of course, we are assuming reliability – without that, max can be infinite.

Cristian's Algorithm

- Given a time server, how do we synchronize the clock of a single node to it.
  - And what is the expected accuracy of this synchronization?



Cristian's Algorithm

- The client records the time between when it sends the request and receives the reply containing the time t of the server. Calls this the roundtrip time ($T_{roundtrip}$).
- A rough approximation of the one way time is simply $T_{roundtrip}/2$. This assumes approximately equal overhead both directions.
- So, the client can estimate the time to be:

$$t + \frac{T_{roundtrip}}{2}$$

Cristian's Algorithm

- Now, recall that we have a minimum possible message time (T_{min}) between the client and server.
- The best the time server can do is measure the time at T_{min} after the request was sent.
- The latest possible time the server can do so is T_{min} before the reply arrives at the sender.
- Why?
  - Both of these assume the fastest possible traversal of one of the paths.
  - If both paths took longer than T_{min}, then the actual time measures by the server would be somewhere in between.

Cristian's Algorithm

- So, we have a bound on the time that the receiver sees the response:

$$[t + T_{\min}, t + T_{roundtrip} - T_{\min}]$$

- A little algebra and we can see that the interval is $T_{roundtrip} - 2T_{min}$ wide.
- So, our accuracy is:

$$\pm\left(\frac{T_{roundtrip}}{2} - T_{\min}\right)$$

Cristian's Algorithm

- One can address variability by making multiple requests, and taking that which has the minimum round-trip time.
- This reduces the width of the bounds, increasing accuracy.

Berkeley Algorithm

- Useful for coordinating a set of machines on a LAN.

- All nodes send their time to a single master node.
- The master, using round trip timing for each node (like in Cristian's algorithm) computes an average time. Outliers are eliminated.
  - The average should compensate for the fact that some clocks run slow and some fast.
- The master then informs the nodes how much +/- that they need to adjust their local clocks.

NTP

- NTP is a general purpose time synchronization protocol for the Internet, not smaller networks.

- The goal:
  - A service to allow clients on the Internet to synchronize accurately with UTC.
  - Provide redundant servers to deal with potential losses of connectivity.
  - Provide a scalable protocol that allows for frequent resynchronization to deal with drift.
  - Include authentication mechanisms to ensure that the timing information originates from trusted servers.

NTP

- Based on a hierarchy of computers.
  - Messaging via UDP.
- The levels in this hierarchy are called *strata*, and the level number is the *stratum* number.
- A set of servers connected in one of these hierarchies is a *synchronization subnet*.

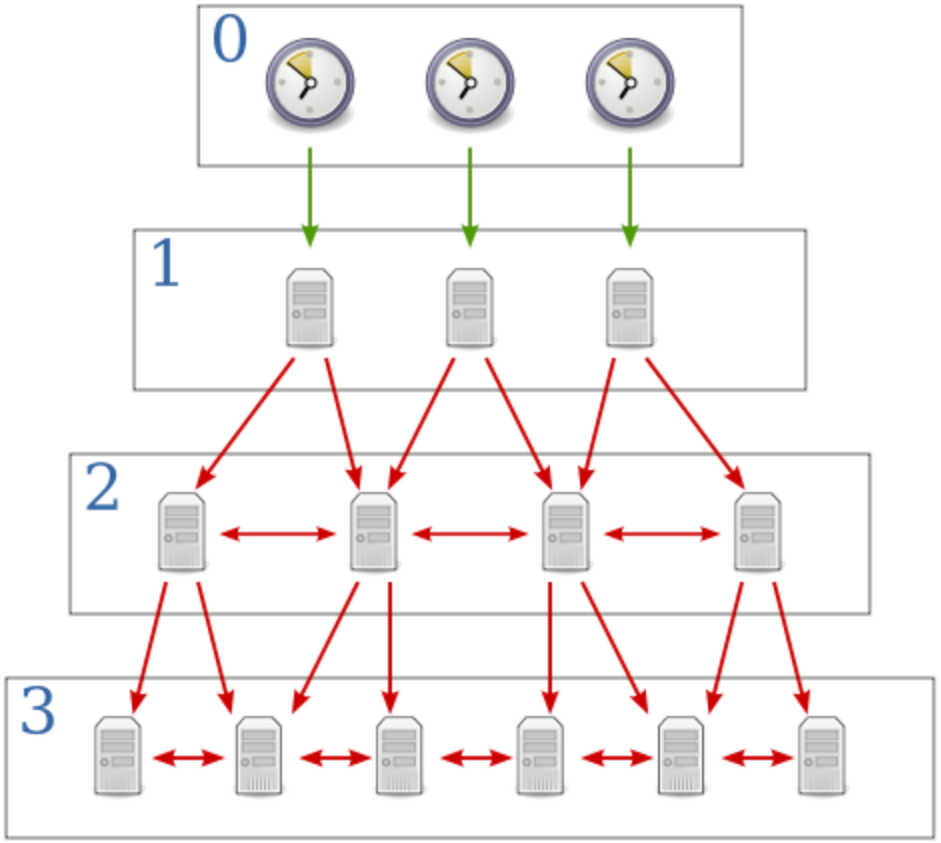


Figure source: http://en.wikipedia.org/wiki/Network_Time_Protocol
Under creative commons license.

NTP

- Machines in one layer synchronize with those above.
- Machines within a single layer can synchronize with each other.
  - Useful when the higher level servers vanish for some reason.
- As the stratum number increases, the error increases.
  - This makes sense. There is some uncertainty in level N synchronized with level N-1. These add up.

Synchronization methods

- Three modes of synchronization between NTP servers:
  - Multicast
  - Procedure-call
  - Symmetric

NTP: Multicast synchronization

- Servers periodically send time via multicast.
- Clients receive time, and set their clocks to it with some small amount of delay.
  - Good for small, local fast networks where high accuracy isn't required.
    - E.g.: A lab of computers where you want the time of day to be sufficiently correct.
  - Has significant issues.
    - One way message, so no round trip timing to help figure out latencies and corresponding delays.
    - Only works if your network supports multicast.

NTP: Procedure-call mode

- Very similar to Cristian's algorithm.
  - Clients send a request to a time server, and measure both the round trip time and the response from the server.
  - Better accuracy than multicast.
  - Also works on networks without multicast support.

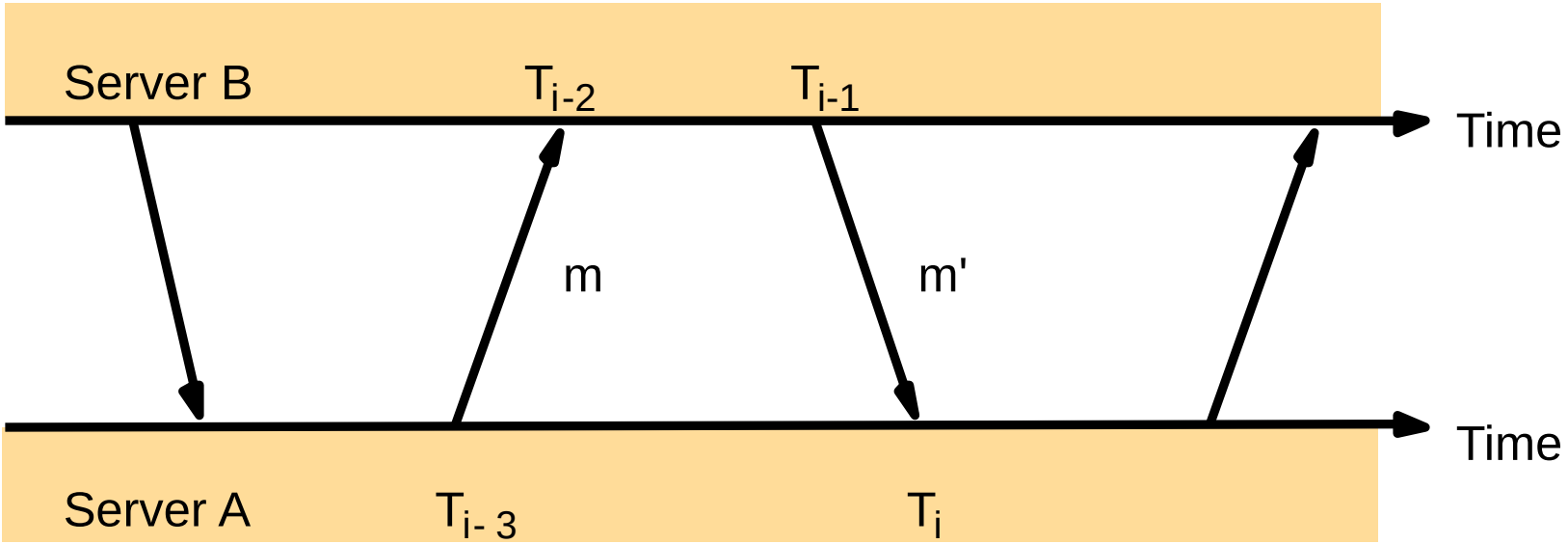
- Pairs of messages sent to estimate the time more accurately.
  - We'll go through this exchange in a couple slides.

NTP: Symmetric mode

- This is used in strata with low numbers close to the root servers, where high degrees of accuracy are important.
- Symmetric mode is similar to procedure call mode, except that the servers maintain a small history of synchronization information for each server that it is synchronizing with.
  - This allows it to deal with outliers and determine which host it can trust to give the most accurate time at any given point.
  - It also allows it to switch servers that it trusts as conditions vary.

NTP: Message Exchange

- Recall, NTP is UDP based.
- In procedure-call and symmetric modes, we exchange message pairs and acquire four time readings.

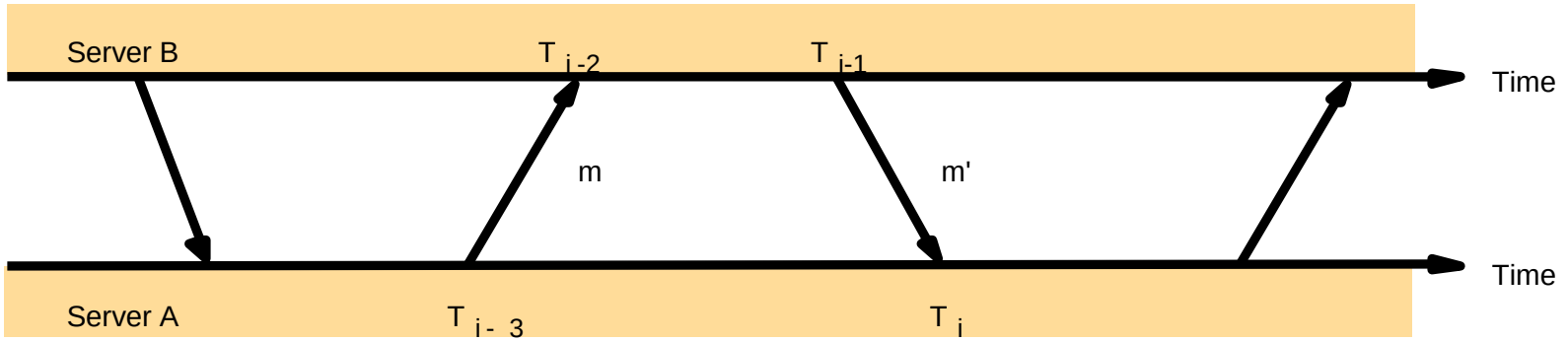


NTP: Message Exchange

- Each message exchanged has three pieces of timing information.
  - Time of send and receive of last NTP message.
    - T(i-3), T(i-2)
  - Time of send of current message.
    - T(i-1)
  - The receiver knows T(i) since it is the time when the message arrives.
- If a message is lost, that is fine, as the timing information is still valid in those received.
- Since one of the times in the message is that of the receiver, it can also easily pick out reordered messages.

NTP: Message Exchange

- Let t and t' be the *actual* transmit times of message m and m' respectively, and o be the *actual* offset of the two machines.
- We know:
  - T(i-2) = T(i-3) + t + o
  - T(i) = T(i-1) + t' + o
- NTP is interested in calculating the offset between two machines. Now, note that the total delay time d is the sum of transmission times (t+t'). A little algebra yields:
  - d = (t+t') = T(i-2)-T(i-3) + T(i) – T(i-1)



NTP: Message Exchange

- We can compute the offset o(i) of the clocks as the average difference in the times at the endpoints of both messages.
  - $o(i) = (T(i-2)-T(i-3) + T(i-1) – T(i)) / 2$
- So, the true offset is:
  - $o = o(i) + (t' – t) / 2$
- Thus the offset can be shown to be in the range:
  - $o(i) – d(i)/2 <= o <= o(i)+d(i)/2$
- o(i) estimates the offset, d(i) measures the accuracy of the estimate.

NTP: Servers

- As mentioned earlier, the servers maintain some amount of state about their peers in order to choose the best peer and minimize error.
- The values <o(i), d(i)> are stored for each peer and maintained for a history of eight exchanges.
- The NTP servers compute something known as the filter dispersion to quantify the reliability of a server.
  - This is a statistical measure of the reliability of the data.
  - Choosing the minimum dispersion peer is best.
    - The description of this in RFC1305 is quite interesting actually. See pages 35 and 36 in the RFC document.
      - http://www.eecis.udel.edu/~mills/database/rfc/rfc1305/rfc1305b.pdf

Logical time and logical clocks

- Lamport observed that since you can't perfectly synchronize a set of distributed clocks, you can't rely on physical time to determine the actual order of two events within a distributed system.

- Logical time and clocks are a way to address this. Often we care about determining the *order* of events, not necessarily the precise physical time at which they occurred.
  - Lamport paper is research paper #1 to read.

Brief intermission

- I posted research paper #1.
- Lamport's classic 1978 paper.
- As discussed in lecture 1 (see first block of slides for lecture 1), you are expected to read the paper and write a two page summary discussing the content of the paper.
  - This is practice in reading and distilling information from research papers for your term paper.
  - Plus, the papers dive deeper into important topics than the book does.

Logical time

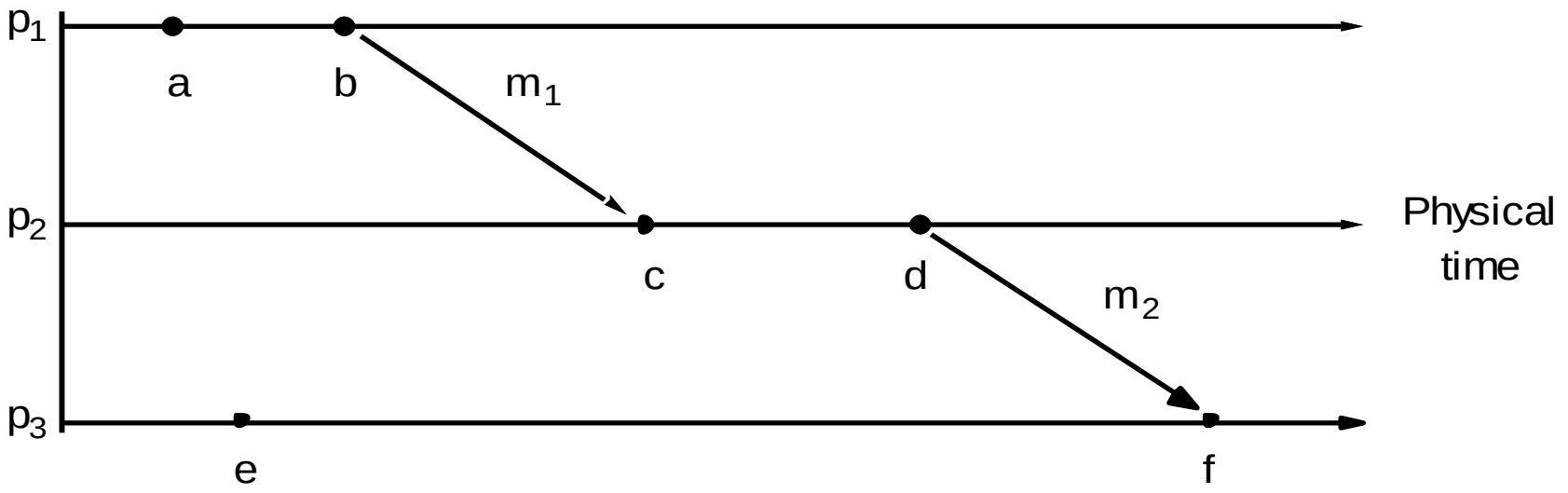
- Logical time is related to physical causality.
- *Causality*: A cause yields an effect, and the effect must occur after the cause. This imposes a partial ordering on events in the physical world.
- In our case, we care about reasoning about events occurring on distributed computers.
- We base the idea on two concepts:
  - The order that we say two events within a process occurred is the order observed by the process itself.
  - When a message is sent from process A to process B, the send event occurs before the receive event.

Lamport's "happened before" relation

- Lamport calls this ordering the "happened before" relation.
- The definition is composed of a few simple rules. We denote "happened before" as the arrow →.
  - **HB1**: If a process observes local events e → e', then we can conclude that e → e' globally.
  - **HB2**: For any message m, send(m) → receive(m).
  - **HB3**: If e, e', and e'' are events such that e → e' and e' → e'', then we can conclude e → e''.

Example:

- Consider this example:
  - Here, the following is true: a → b, b → c, c → d, d → f. We can conclude things like a → f from our HB definition.
  - On the other hand, we cannot conclude a → e or e → a from the happened-before relation. We say a||e, which we read as “a and e are concurrent”.



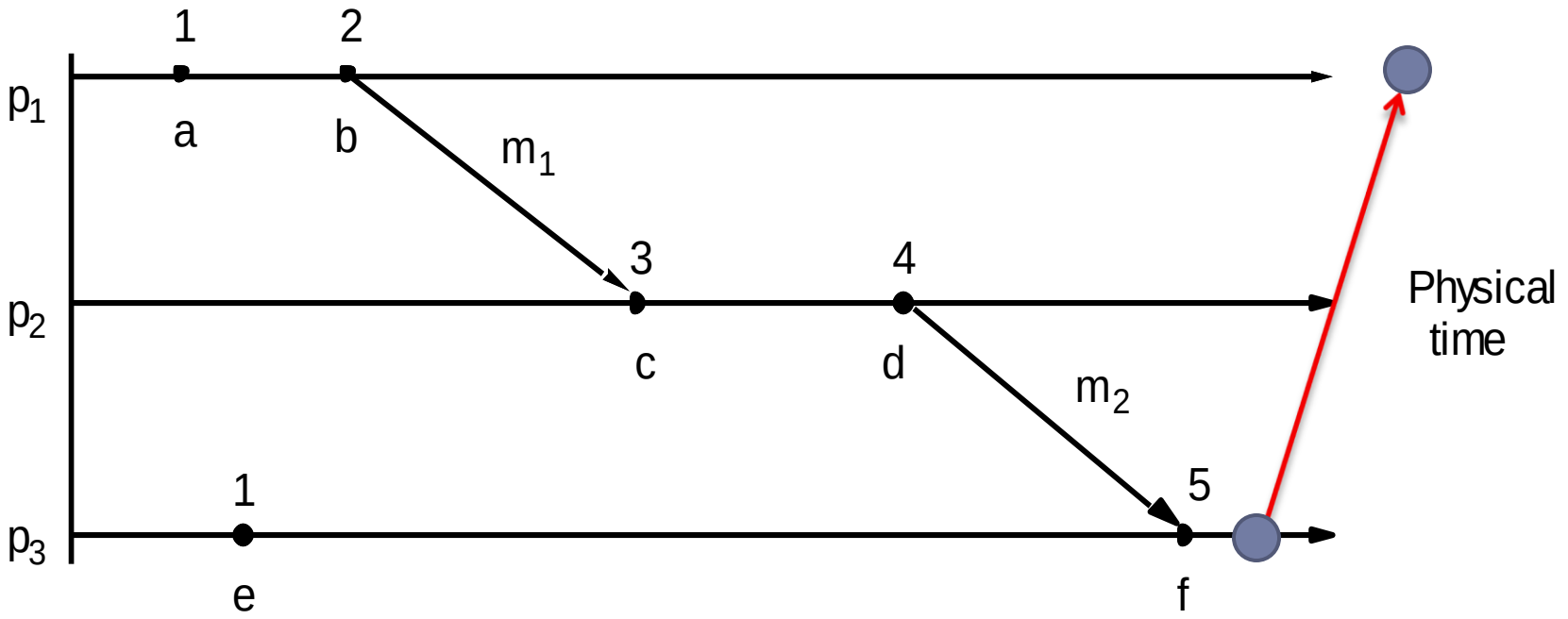
Logical clocks

- So, given this notion of logical time based on the happened-before relation, we can build a clock that allows us to determine the happened-before ordering.
  - "Lamport logical clock"
- It's very easy actually. Every process starts off with a counter initialized to the same value. A clock value (logical time) is associated with each event. Within a process, the times are unique and monotonically increasing.
  - The association of a logical time value with an event is called *timestamping* the event.

Lamport clock rules

- Start with an initialized clock on each process with the same value to start.
- **Rule 1**: The clock is incremented immediately before an event occurs on process i.
  - L(i) = L(i) + 1.
- **Rule 2**: When a process i sends a message m to process j, it includes the logical time of process i, L(i), on the message. Upon receipt of the message (m,t), the process j assigns it's logical clock to the maximum of it's local clock and the time received t.
  - L(j) = max(L(j),t)

Lamport timestamps example



Based on the rules on the previous slide, what would be the timestamps on the endpoints of the extra message I added?

Issues

- This basic Lamport clock doesn't solve everything.
- Consider this:
  - $e \rightarrow e'$ implies $L(e) < L(e')$
  - But $L(e) < L(e')$ does NOT imply $e \rightarrow e'$.
- Vector clocks were invented to overcome this issue.

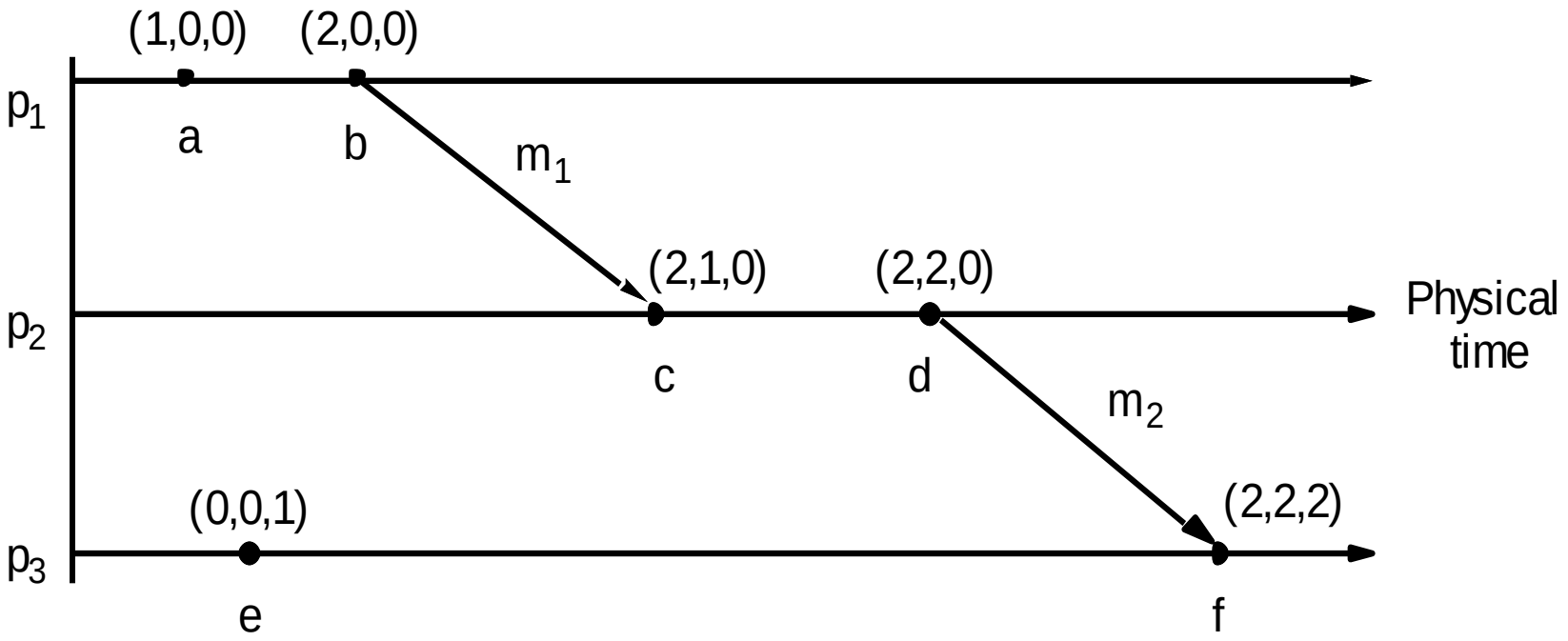
Vector clocks

- Same basic idea as the Lamport timestamps, except instead of a single integer, we store a vector of integers.
- We start off with the elements all being zero for the vectors.
  - V(i,j) = 0 for all i,j = 1,2,…,N.
- Right before an event occurs and is timestamped, the process i sets V(i,i) to V(i,i) + 1.
- The vector is sent along with messages when they are sent, just like in the Lamport scheme.
- When a message is received on process j, we create a new vector by merging V(j,:) with the timestamp vector on the message.
  - V(j ,i) = max(V(j,k), t(k)) for k = 1,2,…,N.

Vector clocks

- What does this mean?
  - V(i,i) represents the number of events that process i has timestamped.
  - V(i,j) where i != j is the number of events that have occurred at process j that process i has potentially been affected by.
- Comparing the timestamps follows some simple rules:
  - V = V' iff V(j) = V'(j) for j = 1,2,…,N
  - V <= V' iff V(j) <= V'(j) for j = 1,2,…,N
  - V < V iff V <= V' and V != V'
- Concurrent events are those where neither $V_a <= V_b$ nor $V_b <= V_a$ are true.

Vector clock example



Global states

- This is all about determining if some property about a distributed system is true as it executes.
- The term “global” means the property holds for the entire system, not just a node or subset of nodes.
- Examples:
  - Distributed garbage collection.
  - Dead lock detection.
  - Distributed debugging.

- The difficulty stems from the lack of a globally known time. If we knew that, we could just pick a time and have everyone dump their state at that time to make a snapshot of the global state.

Global state formalism

- The book presents a formalism for reasoning about global states and whether or not they are consistent.
- We start with the notion of an event history.

- For process i, the event history is:

$$history(p_i) = h_i =< e_i^0, e_i^1, e_i^2, \cdots >$$

- We can take a finite prefix of this history as:

$$h_i^k =< e_i^0, e_i^1, \cdots, e_i^k >$$

Global state formalism

- We can theoretically record the state of each process immediately before each event occurs. So, the history of states corresponding with the history of events on a process are:

$$s_i^k =< s_i^0, s_i^1, \cdots >$$

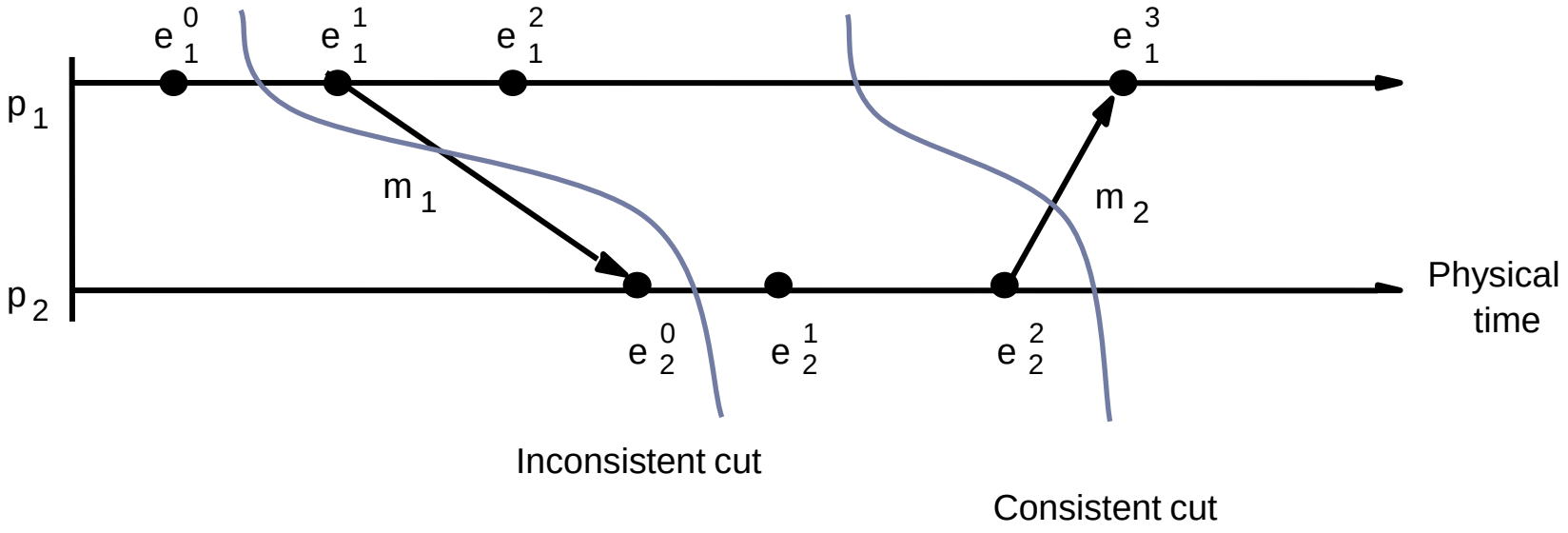
- Now, we can define a set of prefix histories of the processes. We call this a cut, where the cut is specified as the set of prefix lengths for all processes. We also call this the frontier of the cut. The state right before the cut point on each process corresponds to the global state.

$$C = h_1^{c1} \cup h_2^{c2} \cup \cdots \cup H_N^{cN}$$

(Yes, there is a case typo here.)

Global state formalism

- So, a cut simply represents a snapshot of the state of the system with respect to events across all processes.
- The question is whether or not a cut is consistent. The figure shows an example of both.



Global state formalism

- Why was that an inconsistent cut?
  - It showed the receipt of a message but not the send.
  - So, effects existed without a cause. This isn't meaningful.
- So, we care most about cuts that are consistent.
  - A consistent cut contains all the events that happened-before each event that it contains.
- A consistent global state is therefore the set of states corresponding to a consistent cut.

Global state formalism

- We can think about the progression of the program as a sequence of states.

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots$$

- The progress from one state to the next corresponds to a single event in the system and the consistent cut that contains it on the frontier of the cut.

Runs and linearizations

- A run is a total ordering of the events in the global history such that the local happens-before relationship holds.
- A linearization is an ordering where the happens-before relationship holds on the whole global history.
  - A linearization is a run. Runs are not necessarily linearizations.
- What is important about the distinction?
  - Linearizations only pass through consistent global states (they preserve the cause/effect relationship).

Global state predicates and stability

- A global state predicate is a function mapping from the set of global states of processes to a boolean value.
  - E.g.: Is an object in a distributed object system garbage?
- A stable predicate stays true once it becomes true.
  - E.g.: Object is garbage, system is deadlocked.
- An unstable predicate does not stay true.
  - E.g.: Variable values in a program change, so predicates for distributed debugging may become false after they are true as the programs run.