Clock Synchronization

Chapter 11.1-11.4, Livro do George Coulouris

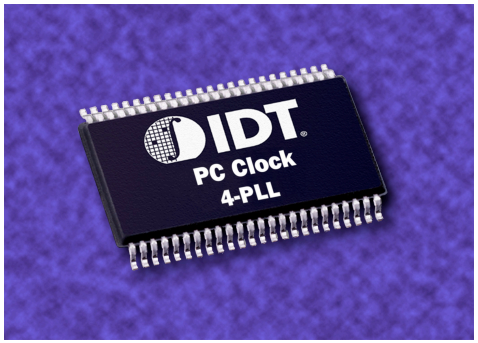
Sistemas Distribuídos

Clock Synchronization?

- Why needed?
  - to measure delays between distributed components
  - to synchronise streams, e.g. sound and video
  - to establish event ordering
    - causal ordering (did A happen before B?)
    - concurrent/overlapping execution (no causal relationship)

– for accurate timestamps to identify/authenticate
  - business transactions
  - distributed databases
  - security protocols

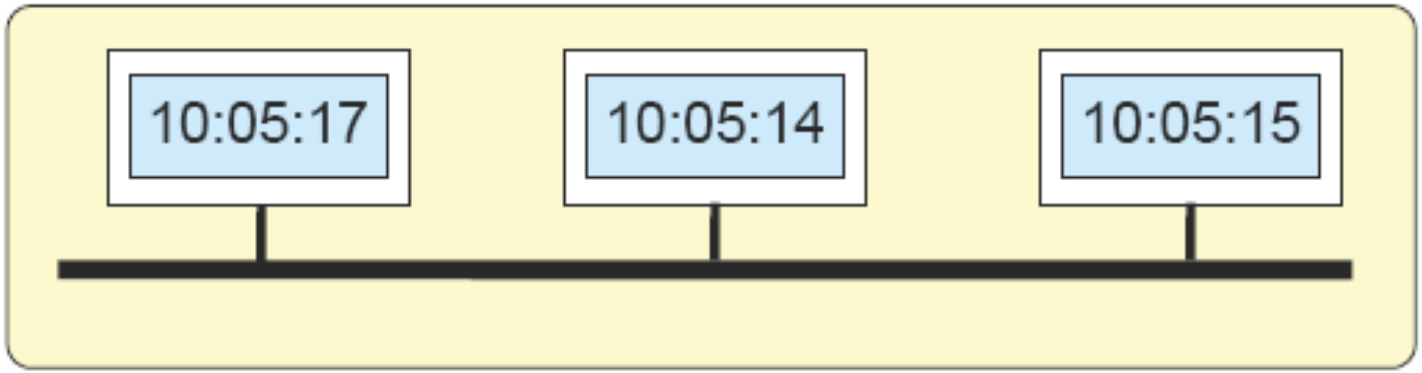
Clocks



- Internal hardware clock
  - built-in electronic device
  - counts oscillations occurring in a quartz crystal at a definite frequency
  - store the result in a counter register
  - interrupt generated at regular intervals
  - interrupt handler reads the counter register, scales it to convert to time units (seconds, nanoseconds) and updates the software clock
    - e.g. seconds elapsed since 1/01/1970

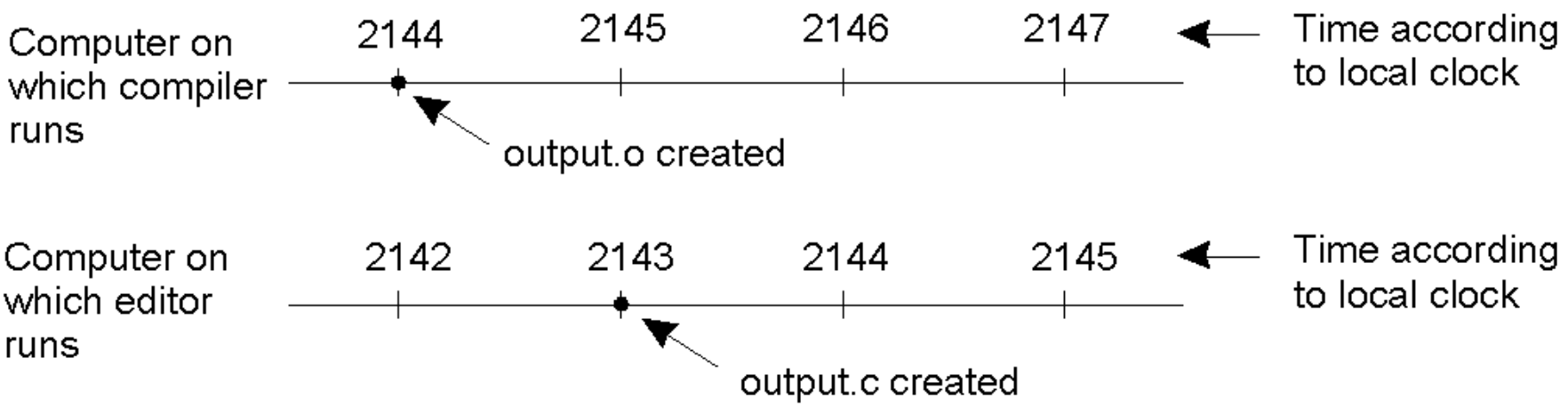
Problems with Clocks

- Frequency of oscillations
  - varies with temperature
  - different rate on different computers



- Accuracy
  - typically 1 sec in 11.6 days

Clock Synchronization



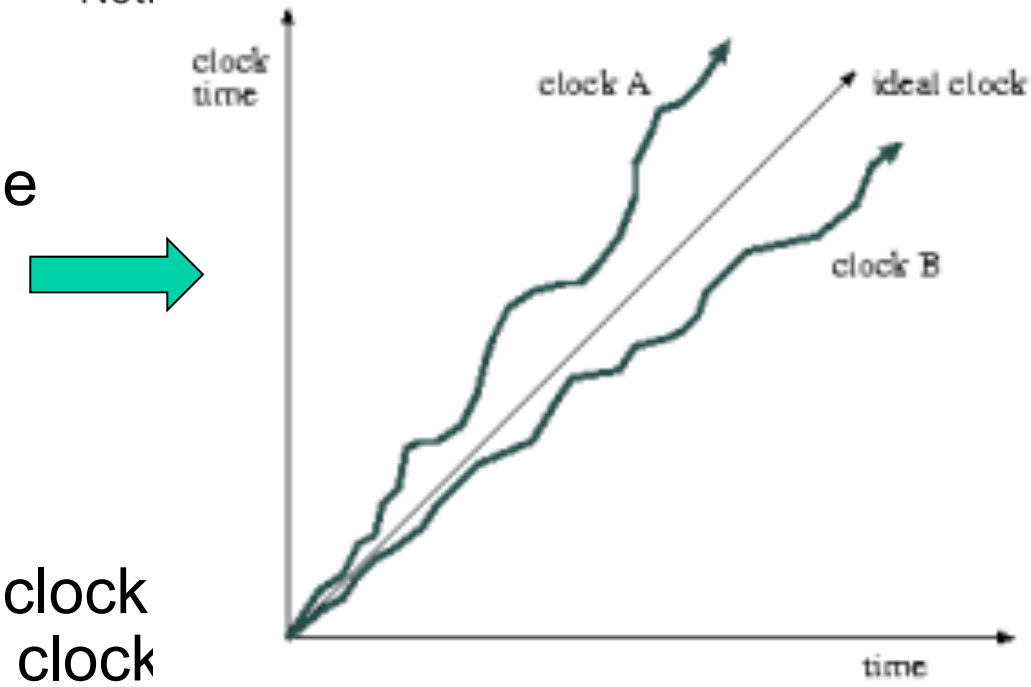
- When each machine has its own clock, an event that occurred after another event may be assigned an earlier time.

Clock Skew and Drift

Net

- **Clock skew**

  – difference between the readings of two clocks



- **Clock drift**

  – difference between a clock and a perfect reference clock per unit of time:

  • typically 1 sec in 11.6 days

Sources of Time

- **Universal Coordinated Time (UTC)**
  - based on atomic time but leap seconds inserted to keep in phase with astronomical time (Earth's orbit)
  - UTC signals broadcast every second from radio and satellite stations
    - land station accuracy 0.1-10ms due to atmospheric conditions
- Global Positioning System (GPS)
  - broadcasts UTC
- Receivers for UTC and GPS
  - available commercially
  - used to synchronise local clocks

GPS Receivers

Attach GPS to a computer: +/- 1 msec of UTC

Clock Synchronization

- **External: synchronise with UTC source of time**
  - the absolute value of difference between the clock and the source is bounded above by D at every point in the synchronisation interval
  - time accurate to within D

- **Internal: synchronise clocks with each other**
  - the absolute value of difference between the clocks is bounded above by D at every point in the synchronisation interval
  - clocks agree to within D (not necessarily accurate time)

Clock Compensation

- **Assume 2 clocks can each drift at rate R msecs/sec**
  - maximum difference 2R msecs/sec
  - must resynchronise every D/2R to agree within D
- **Clock correction**
  - get UTC and correct software clock
- **Problems!**
  - what happens if local clock is 5 secs fast and it is set right?
  - timestamped versions of files get confused
  - time must never run backwards!

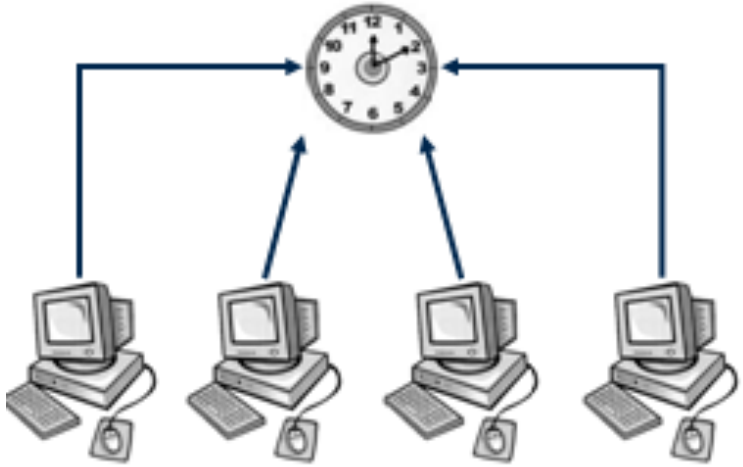
Synchronization Methods

- **Synchronous systems**
  – simpler, relies on known time bounds on system actions

- **Asynchronous systems**
  – intranets
    - Cristian's algorithm
    - Berkeley algorithm
  – Internet
    - The Network Time Protocol

Clock Synchronization

External Clock Synchronization

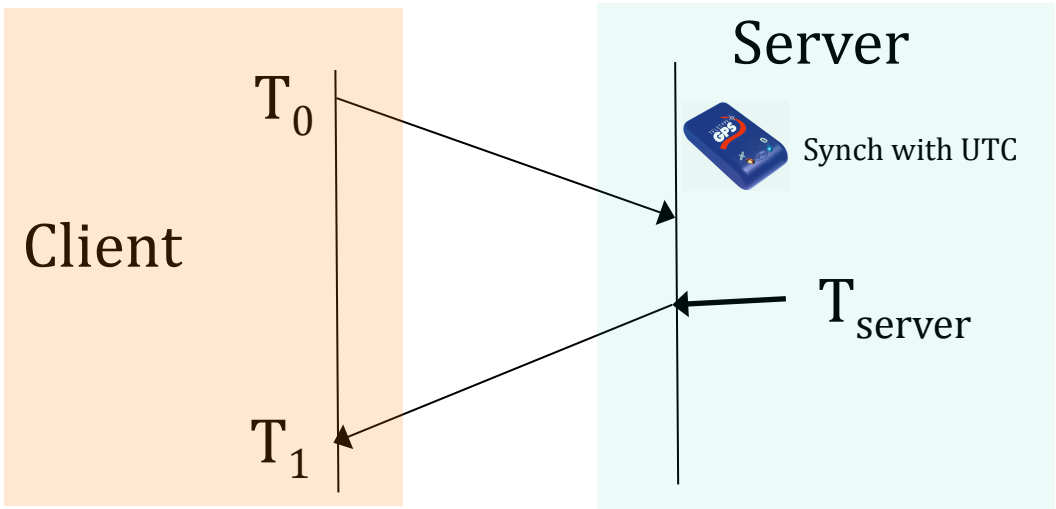


Synchronize clocks with an external time reference
Example: NTP

Internal Clock Synchronization

Synchronize clocks among themselves

Cristian's Algorithm



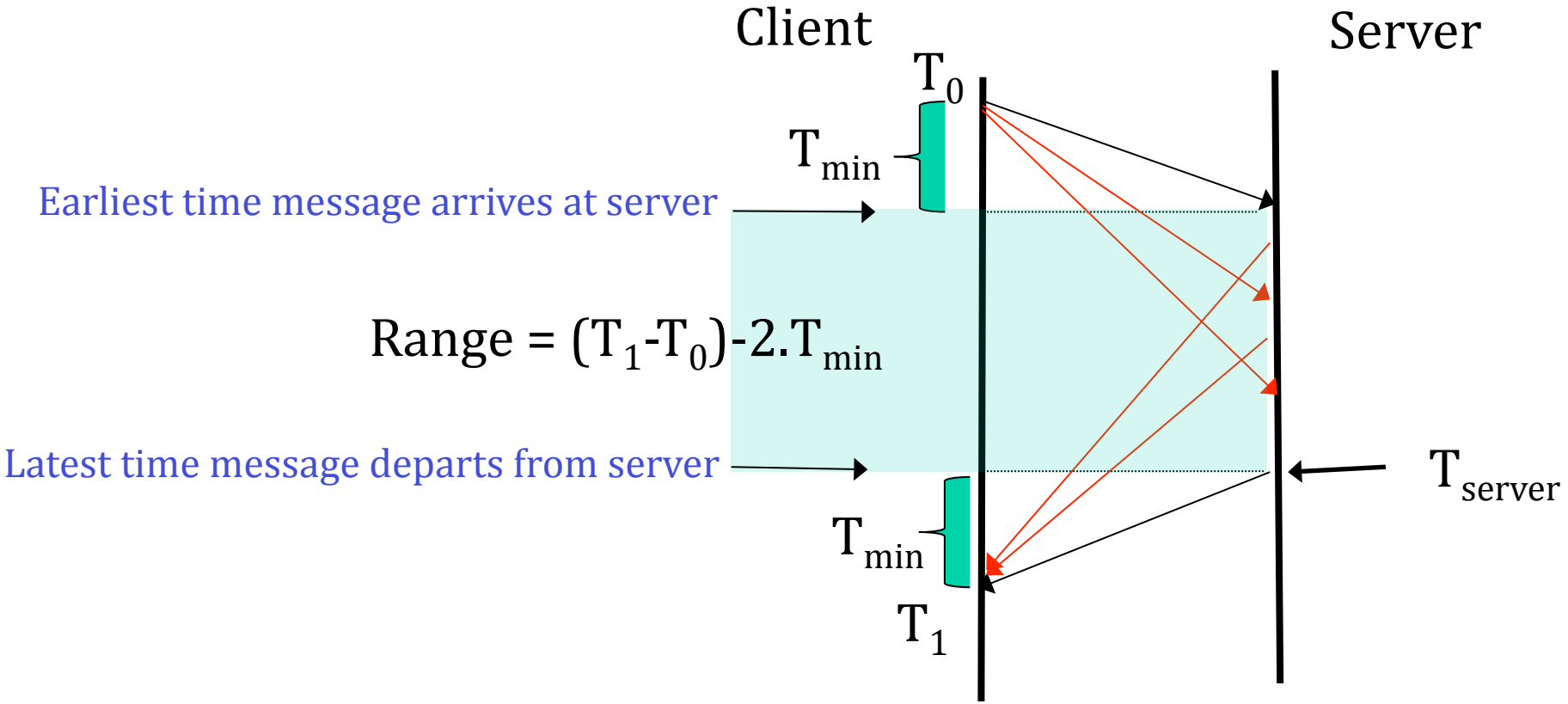
- Synch client with T_{server} + Msg latency (RTT/2):

  $T_{@\_client} = T_{server} + (T_1 - T_0)/2$

- Repeat this several times; Choose minimum latency

Error Bounds (Cristian's Algorithm)

- If there is a minimum message latency: T_{min}



Accuracy of the result= +/- $((T_1-T_0)/2 - T_{min})$

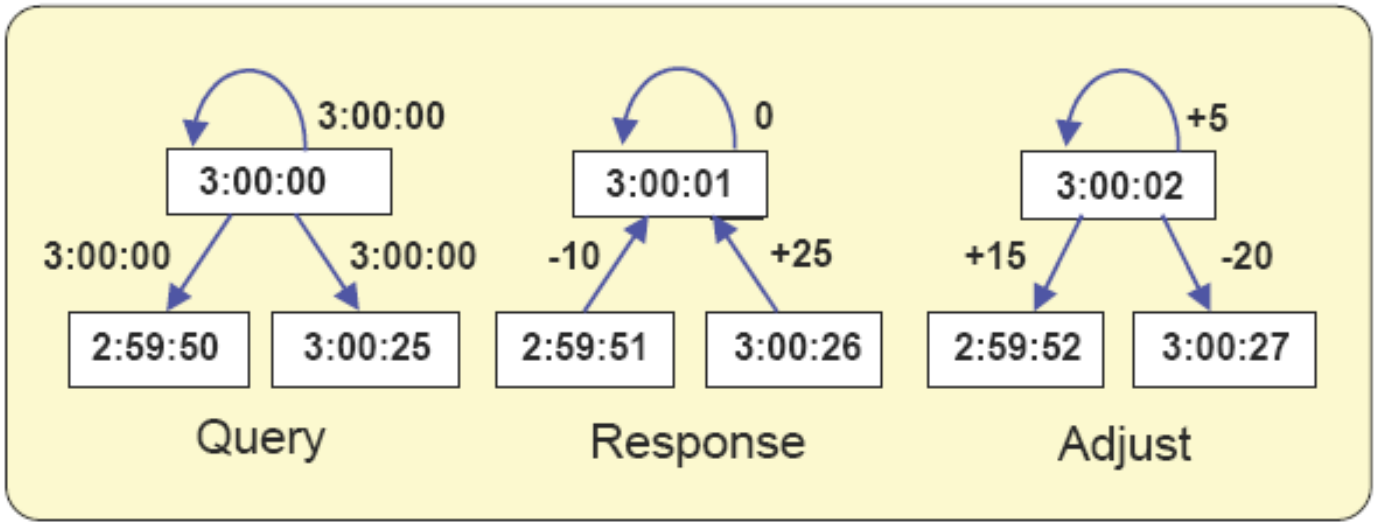
Example (Cristian's Algorithm)

- Send request at 5:08:15.100 (T0)
- Receive response at 5:08:15.900 (T1)
- Response contains 5:09:25.300 (T_{server})

- Elapsed time is (T1 -T0)
  - 5:08:15.900 -5:08:15.100 = 800 msec
- Best guess: timestamp was generated 400 msec ago
- Set time to T_{server}+ elapsed time:
  - 5:09:25.300 + 400 = 5:09.25.700
- If Tmin = 200 msec
  - Error = +/- ((400) – 200) = +/- 200

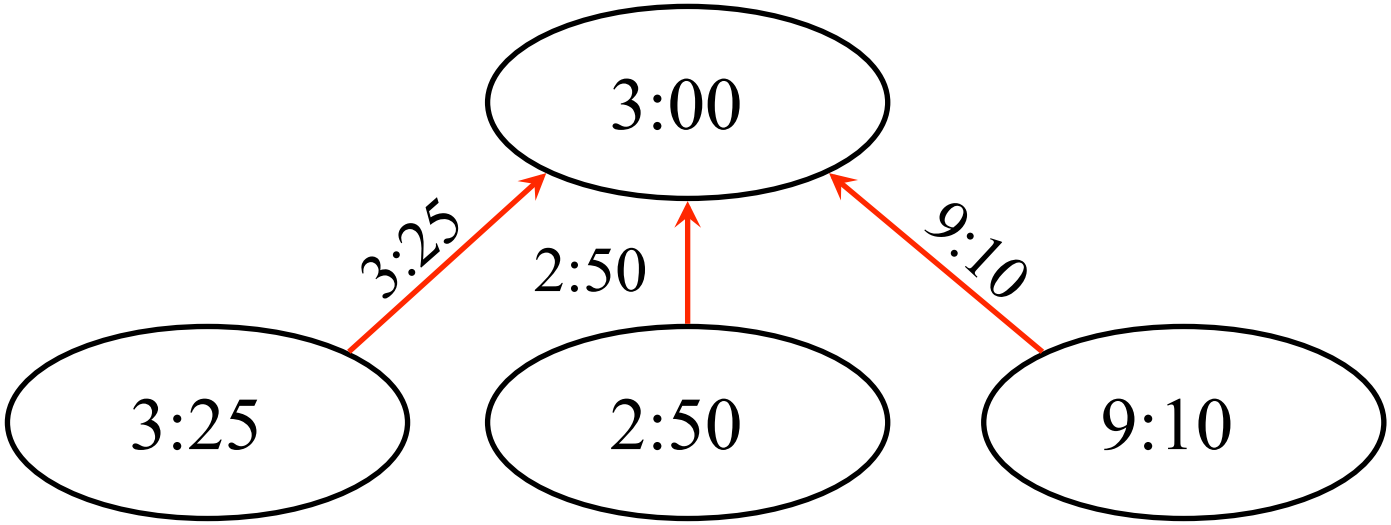
The Berkeley Algorithm

- Choose master co-ordinator which periodically polls slaves
- Master estimates slaves' local time based on round-trip
- Calculates average time of all, ignoring readings with exceptionally large propagation delay or clocks out of synch
- Sends message to each slave indicating clock adjustment



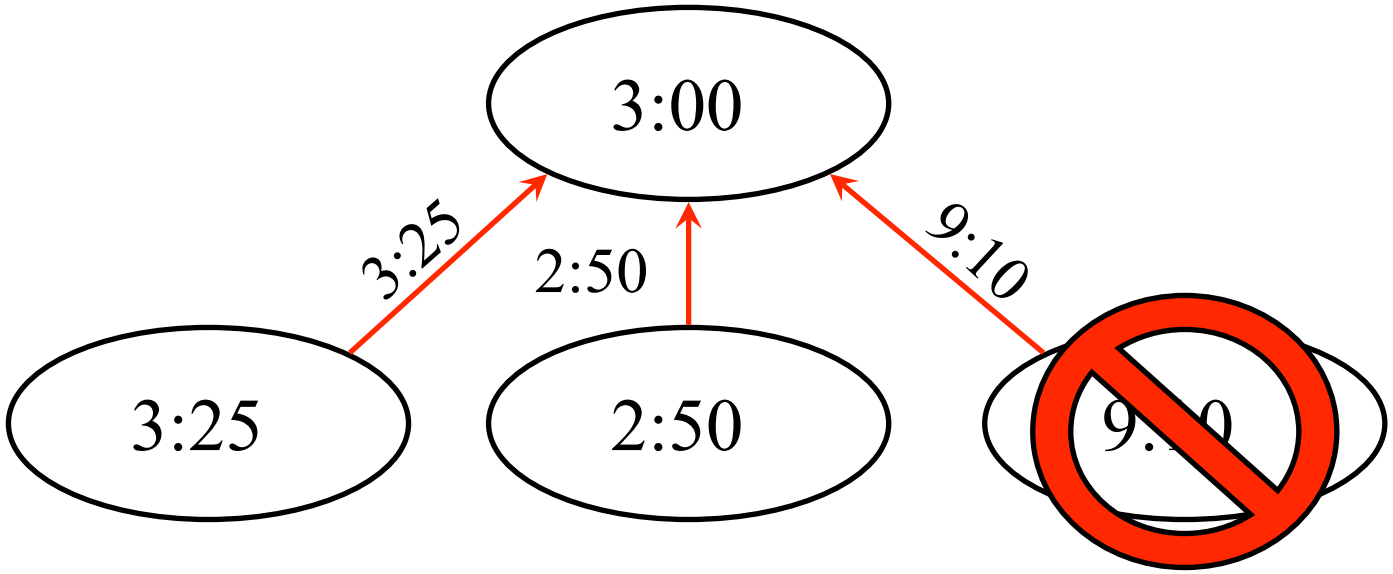
Synchronisation feasible to within 20-25 msec for 15 computers, with drift rate of 2×10^{-5} and max round trip propagation time of 10 msec.

Example: Berkeley Algorithm



1. Request timestamps from all slaves

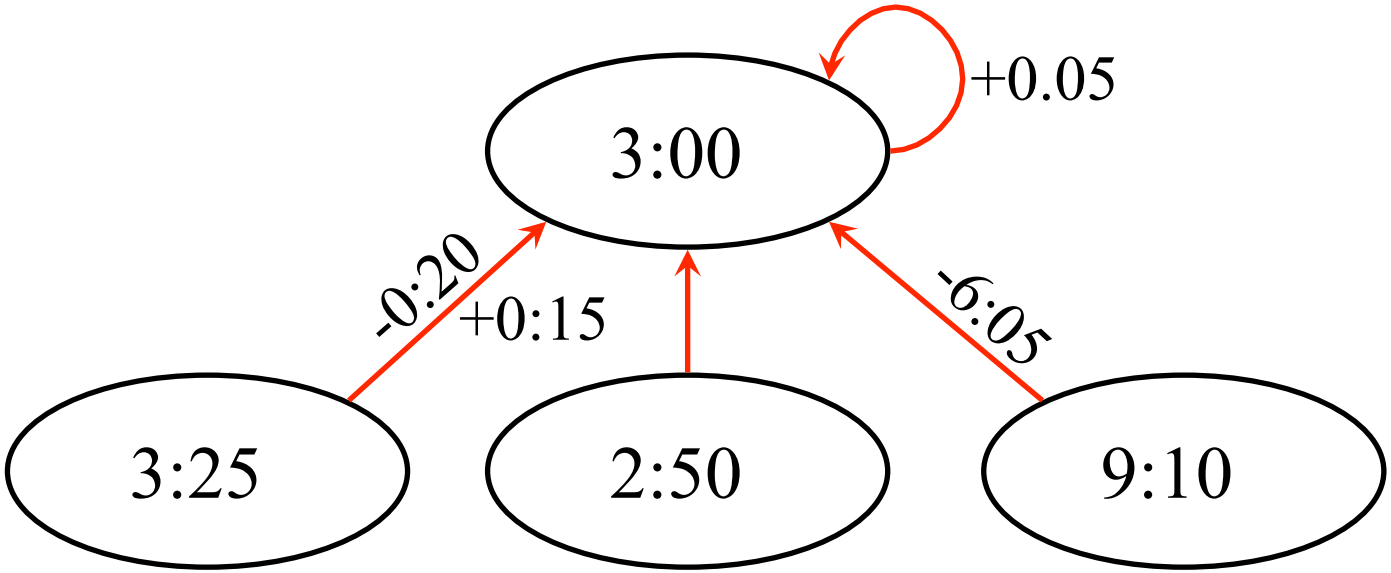
Example: Berkeley Algorithm



2. Compute fault-tolerant average:

(3:25+2:50+3:00)/3=3:05

Example: Berkeley Algorithm



3. Send offset to each client

If Master Fails, any node can take that role (through an election algorithm)

Exercício: Sincronização Relógios Berkeley Algorithm

- Considere que tem 4 máquinas (A,B,C,D) que estão a usar o algoritmo de Berkeley para sincronização de relógios. O servidor leader é a máquina A. Considere que o leader decide enviar uma mensagem de polling às 10:00:10 que é recebida nas restantes máquinas (B, C e D) nos seguintes timings, respectivamente: 9:59:30, 11:00:10, 9:58:20.
- **1**- Quais os valores de ajuste que a máquina leader envia para A, B, C e D?

-40 / (-10)

- 110 / (+60)

- **2**- O que iria acontecer se a máquina C fosse a leader? Quais os valores a enviar para A, B, C e D?
- **3**- Se tivermos uma rede de computadores totalmente sincronizada usando o algoritmo de Berkeley, acha possível que essas máquinas estejam completamente des-sincronizadas do relógio universal?

Analysis of Sync Algorithms

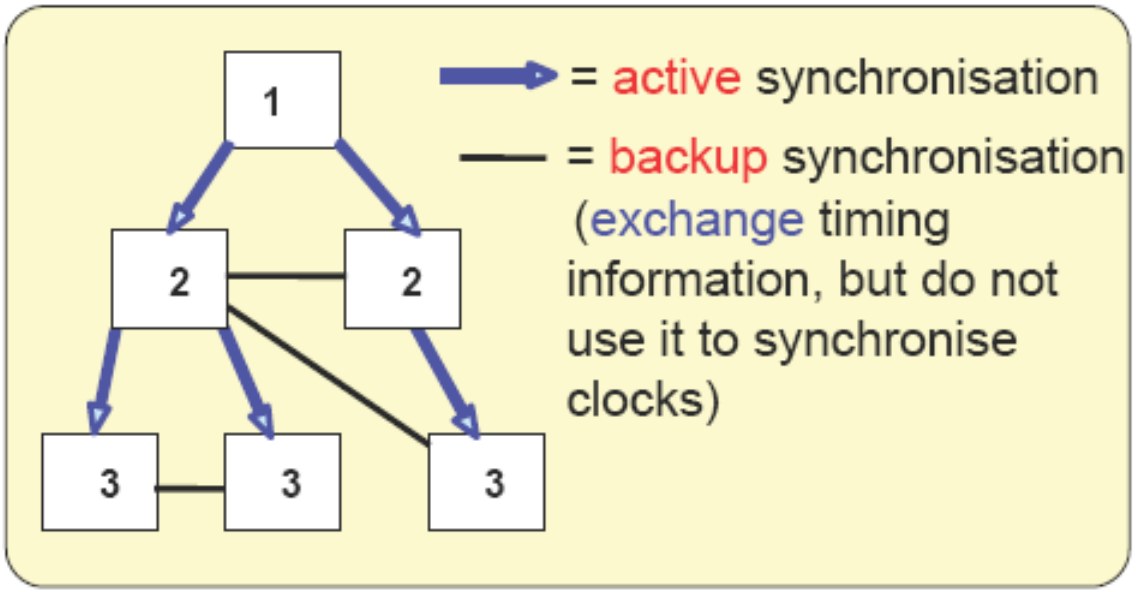
- Cristian's algorithm: N clients send and receive a message every x seconds.
- Berkeley algorithm: 3N messages every x seconds.
- Both assume a central time server or coordinator.
- More distributed algorithms exist in which each processor broadcasts its time at an agreed upon time interval and processors go through an agreement protocol to average the value and agree on it.

Network Time Protocol

- NTP: RFC 1305
- Enable clients across Internet to be accurately synchronized to UTC despite message delays
  - Provide reliable service
    - Survive lengthy losses of connectivity
    - Redundant paths
    - Redundant servers
  - Enable clients to synchronize frequently
    - offset effects of clock drift
  - Provide protection against interference
    - Authenticate source of data

Network Time Protocol (NTP)

- Multiple time servers across the Internet
- Primary servers: directly connected to UTC receivers
- Secondary servers: synchronise with primaries
- Tertiary servers: synchronise with secondary, etc
- Scales up to large numbers of servers and clients



Copes with failures of servers – e.g. if primary's UTC source fails it becomes a secondary, or if a secondary cannot reach a primary it finds another one.

Authentication used to check that time comes from trusted sources

NTP Sync Modes

- **Multicast**
  – one or more servers periodically multicast to other servers on high speed LAN
  – lower accuracy but efficient

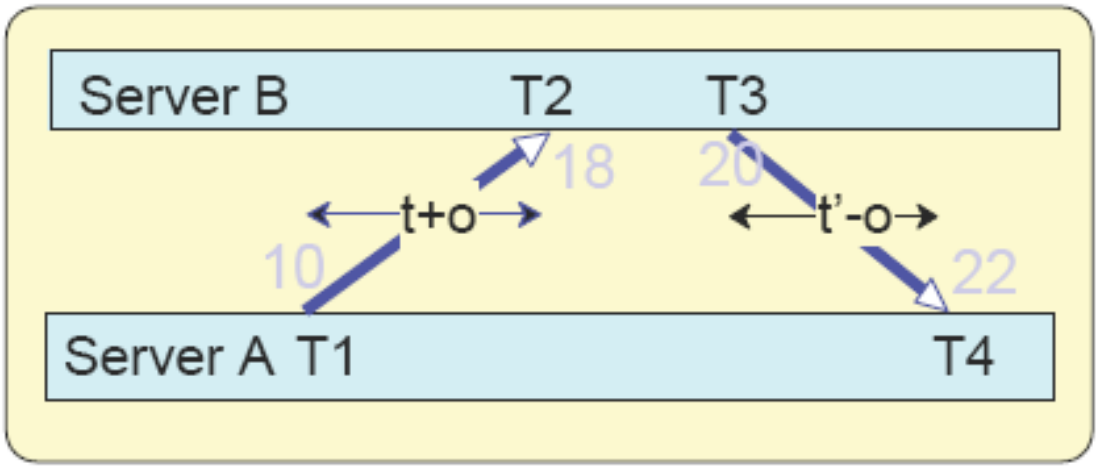
- **Procedure Call Mode**
  – similar to Cristian's algorithm: client requests time from a few other servers

- **Symmetric protocol**
  – used by master servers on LANs and layers closest to primaries
  – highest accuracy, based on pairwise synchronisation

NTP Messages

- Procedure call and symmetric mode
  - Messages exchanged in pairs
- NTP calculates:
  - **Offset** for each pair of messages
    - Estimate of offset between two clocks
  - **Delay**
    - Transmit time between two messages
  - **Filter Dispersion**
    - Estimate of error – quality of results
    - Based on accuracy of server's clock ***and*** consistency of network transit time
- Use this data to find preferred server:
  - *lower stratum & lowest total dispersion*

NTP Symmetric Protocol (SNTP)



- t = transmission delay (e.g. 5ms)
- o = clock offset of B relative to A (e.g. 3ms)
- Record local times T1 = 10, T2 = 18, T3 = 20, T4 = 22

  Let a = T2-T1= t + o, b = T4-T3 = t' - o, and assume t ≈ t'

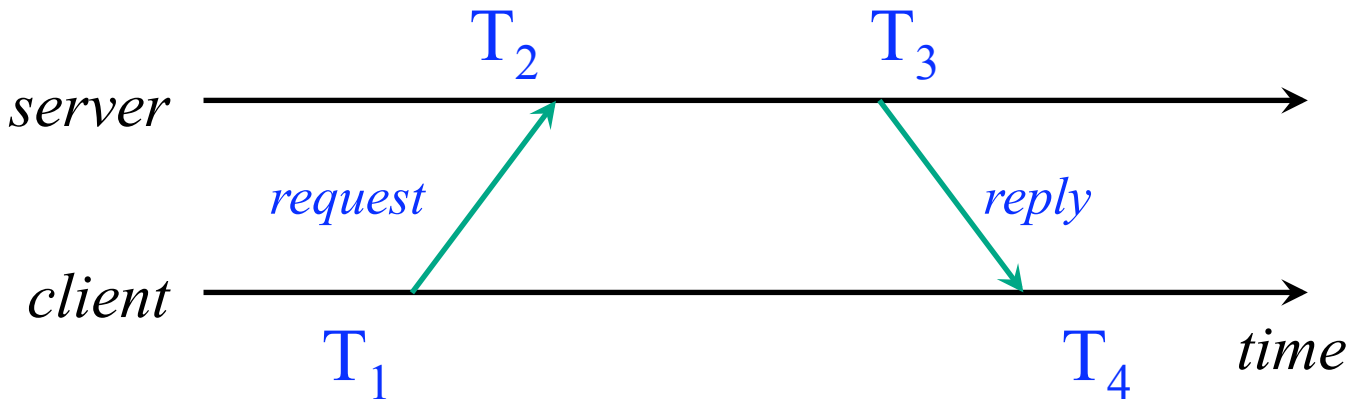
  Round trip delay = t + t' = a + b = (T2-T1)+(T4-T3) = 10

  Calculate estimate of clock offset o = (a-b)/2 = 3

NTP Symmetric Protocol

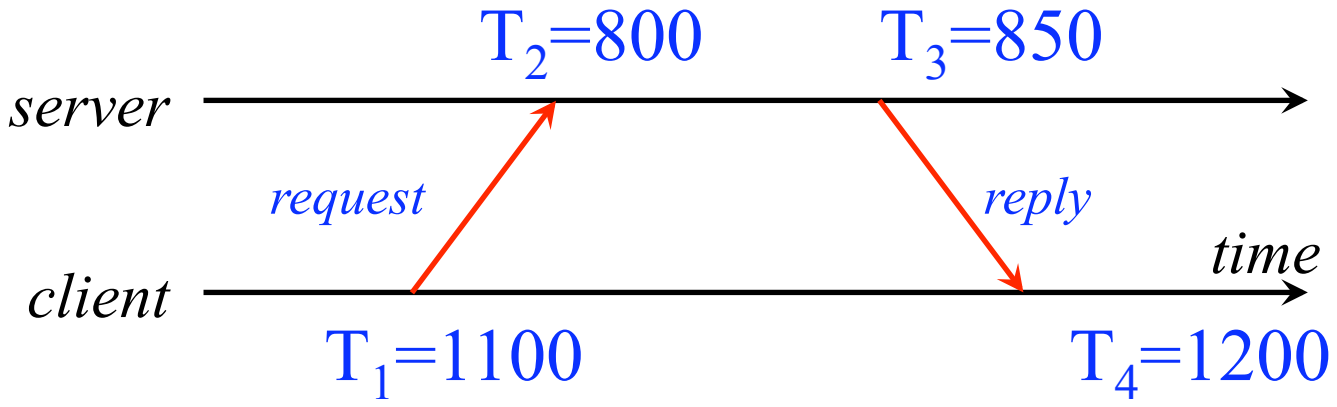
- T4 = current message receive time determined at receiver
- Every message contains
  - T3 = current message send time
  - T2 = previous receive message receive time
  - T1 = previous receive message send time
- Data filtering (obtain average values of clock offset from values of o corresponding to minimum t)
- Peer selection (exchange messages with several peers favouring those closer to primaries)
- How good is it? 20-30 primaries and 2000 secondaries can synchronise to within 30 ms

Simple NTP (SNTP)



Time offset, o= (a-b) / 2 = ((T2-T1) – (T4-T3)) / 2

SNTP Example



Offset =

((800-1100) - (1200-850))/2

=((-300) – (350))/2

= -650/2 = **-325**

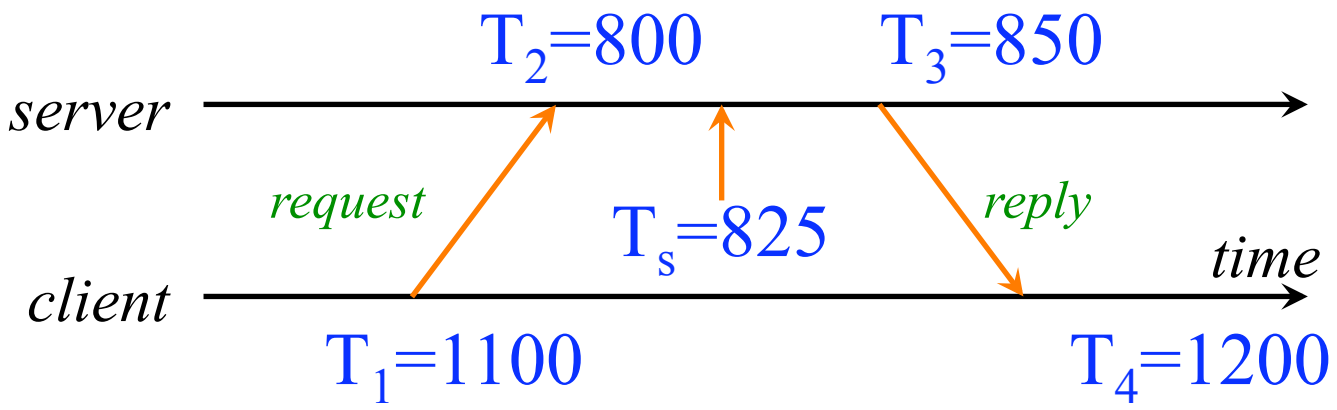
Time offset:

((T2-T1) – (T4-T3)) / 2

Set time to $T_4 + t$

= 1200 - 325 = **875**

Example with Cristian's Algorithm



Offset = (1200 - 1100)/2 = **50**

Set time to T_s + *offset*

= 825 + 50 = **875 .....**

Cristian's algorithm & SNTP

Set clock from server

But account for network delays

Error: uncertainty due to network/processor latency

Exercício de Exame

- Considere que está a usar o protocolo NTP para sincronizar duas máquinas (A e B). A máquina B recebe uma mensagem de A no tempo 10 e com um timestamp 20. Depois, a máquina A recebe uma mensagem de B no tempo 26 e com um timestamp 12.

- - Qual o RTT (*Round-Trip-Time*) da comunicação?
- - Qual o offset entre os relógios de A e B?

Logical Time

Logical Time

- For many purposes it is sufficient to agree on the same time (e.g. internal consistency) which need not be UTC time

- Can deduce causal event ordering

  a → b (a occurs before b)

- Logical time denotes causal relationships

Event Ordering

- Define a → b (a occurs before b) if

  – a and b are events in the same process and a occurs before b, or

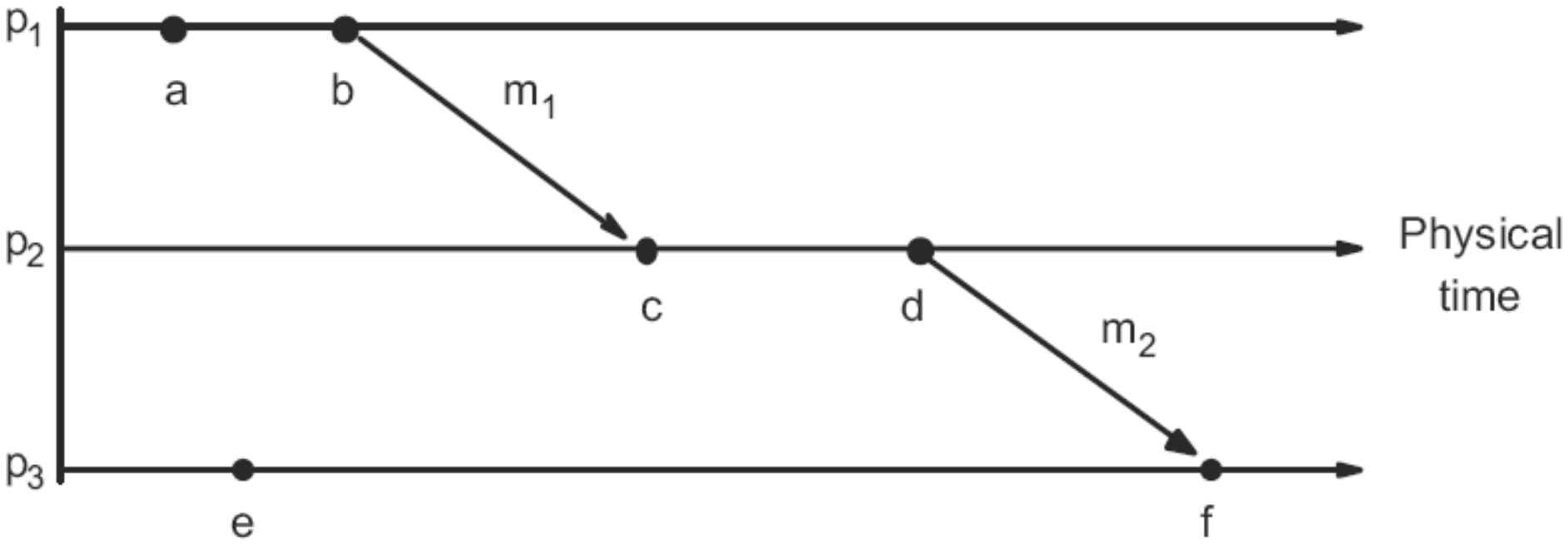
  – a is the event of message sent from process A and B is the event of message receipt by process B

- If a → b and b → c then a → c.

- → is partial order.

- For events such that neither a → b nor b → a we say a, b are concurrent, denoted a || b.

Example



- $a \rightarrow b, c \rightarrow d$
- $b \rightarrow c, d \rightarrow f$
- $a \parallel e$

Lamport Clocks

- Lamport defined the ***happens-before*** relation for DS.
- A → B means “A happens before B”.
- If A and B are events in the same process and A occurs before B then A → B is true.
- If A is the event of a message being sent by one process-node and B is the event of that message being received by another process, then then A → B is true. (A message must be sent before it is received).
- Happens-before is the transitive closure of 1 and 2. That is, if A→B and B→C, then A→C.
- Any other events are said to be concurrent.

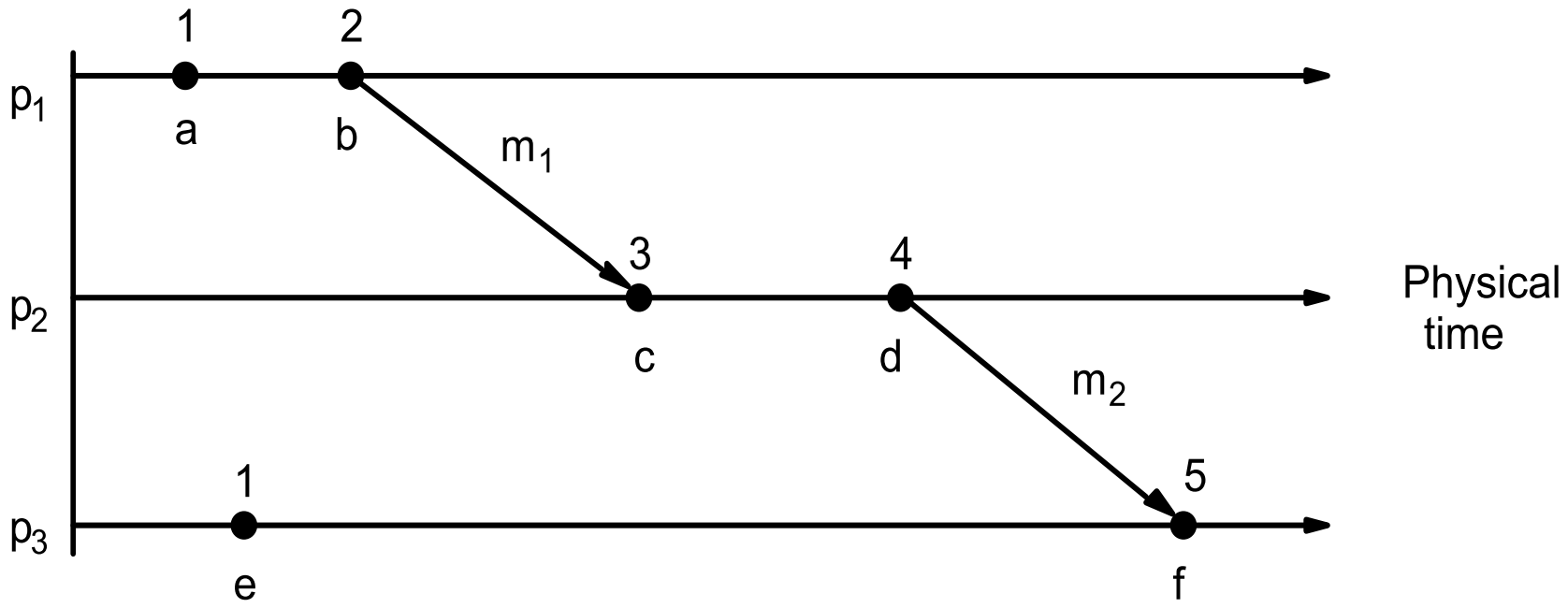
Lamport Clocks

- Desired properties:
- (1) anytime A→ B , C(A) < C(B), that is the logical clock value of the earlier event is less
- (2) the clock value C is increasing (never runs backwards)

Lamport Clocks Rules

- An event is an internal event or a message send or receive. The local clock is increased for an internal event.
- The local clock is increased by one for each message sent and the message carries that timestamp with it (piggyback).
- When a message is received, the current local clock value, C, is compared to the message timestamp, T.
  - If the message timestamp, T = C, then set C=C+1.
  - If T > C, set the clock to T+1.
  - If T<C, set the clock to C+1.

Lamport Clocks



- b → c means that C(b) < C(c)
- d → f means that C(d) < C(f)
- However, C(i) < C(j) doesn't mean i→ j
- (ex: C(e) < C(b) but it is not true that e →b)

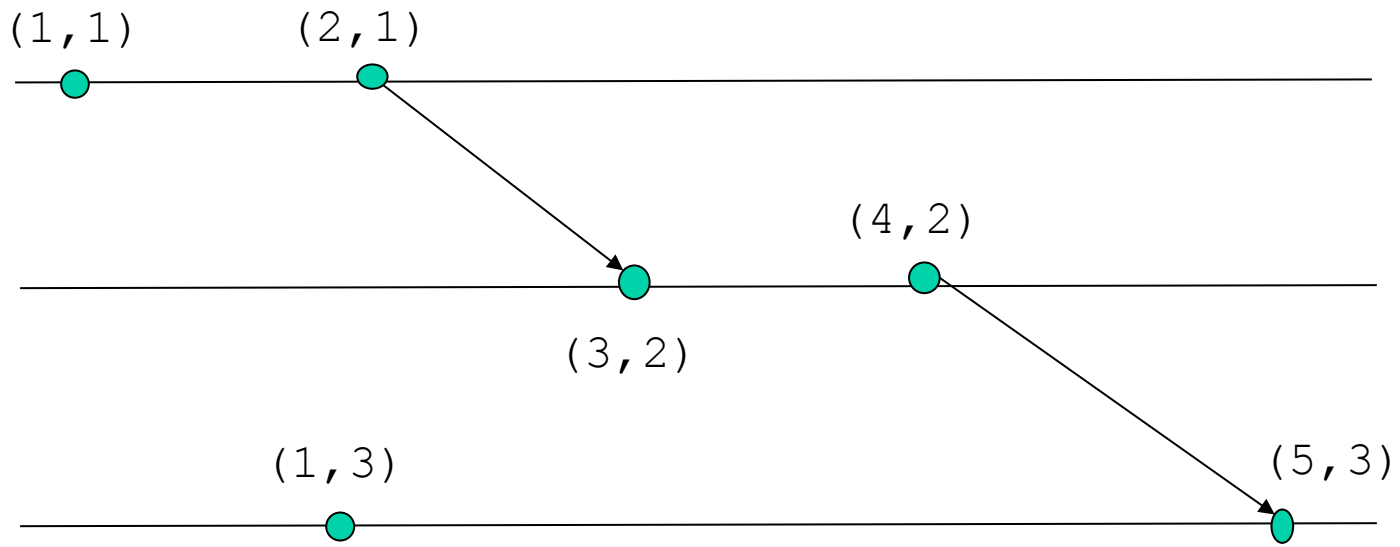
Total Order: Lamport Clocks

- If you need a total ordering, (distinguish between event 3 on P2 and event 1 on P3) use **Lamport timestamps**.

- Lamport timestamp of event A at node i is (C, i)

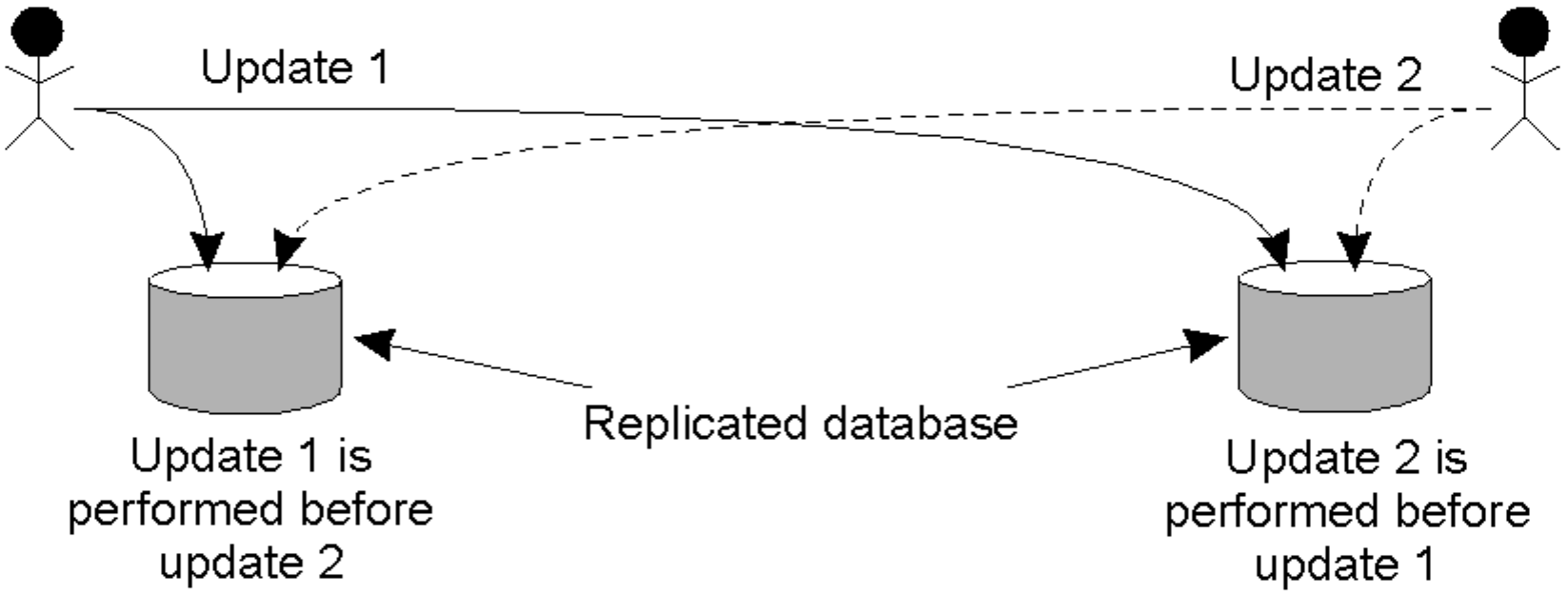
- For any 2 timestamps T1=(C(A),I) and T2=(C(B),J)
  - If C(A) > C(B) then T1 > T2.
  - If C(A) < C(B) then T1 < T2.
  - If C(A) = C(B) then consider node numbers.
    - If I>J then T1 > T2.
    - If I<J then T1 < T2.
    - If I=J then the two events occurred at the same node, so since their clock C is the same, they must be the same event.

Total Order: Lamport Timestamps



- The order will be (1,1), (1,3), (2,1), (3,2) etc

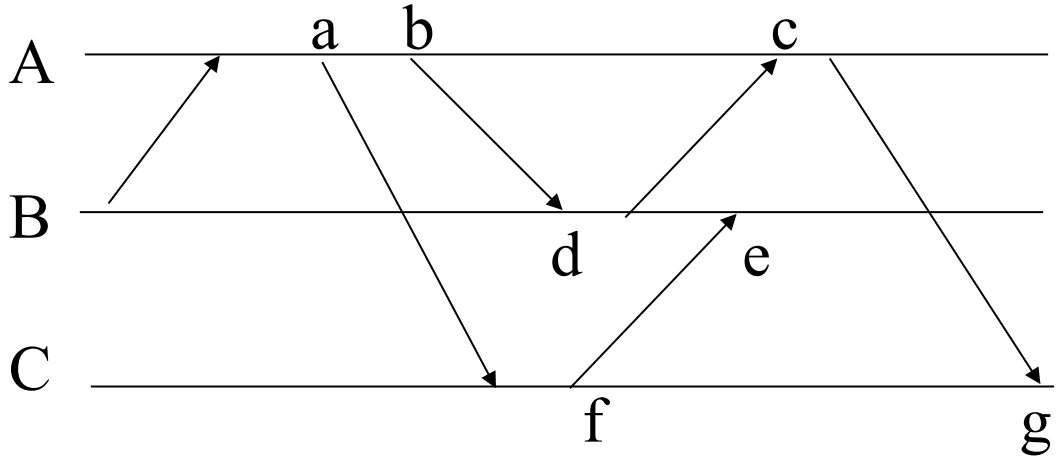
Why Total Order?



Database updates need to be performed in the same order at all sites of a replicated database.

- Process *P*1 adds $100 to an account (initial value: $1000)
- Process *P*2 increments account by 1%
- Replica #1 will end up with $1111, while replica #2 ends up with $1110

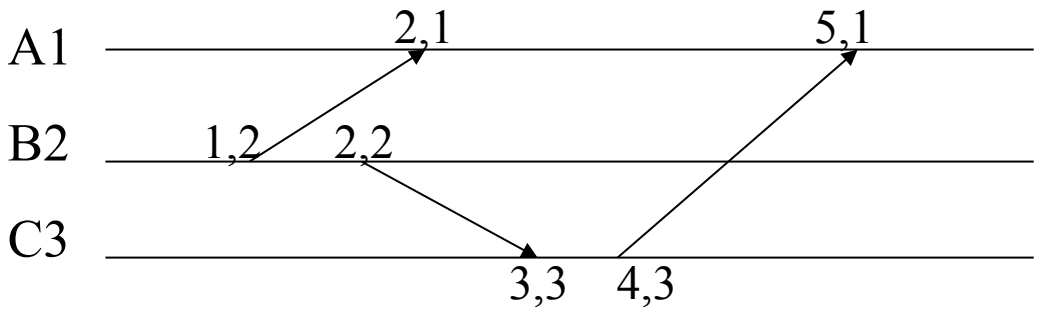
Exercise: Lamport Clocks



- Assuming the only events are message send and receive, what are the clock values at events a-g?

Limitation of Lamport Clocks

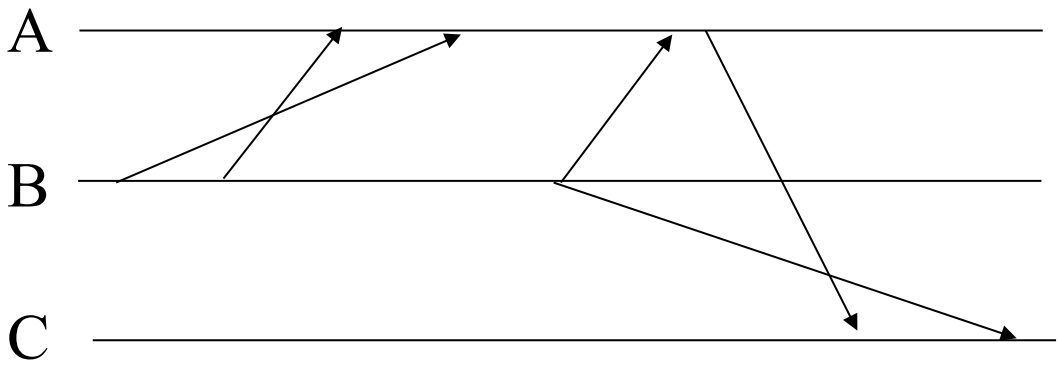
- Total order Lamport clocks gives us the property if A → B then C(A) < C(B).
- But it doesn't give us the property if C(A) < C(B) then A→B.
- If C(A) < C(B), A and B may be concurrent or incomparable, but never B→A.



Lamport timestamp of 2,1 < 3,3 but the events are unrelated

Limitation

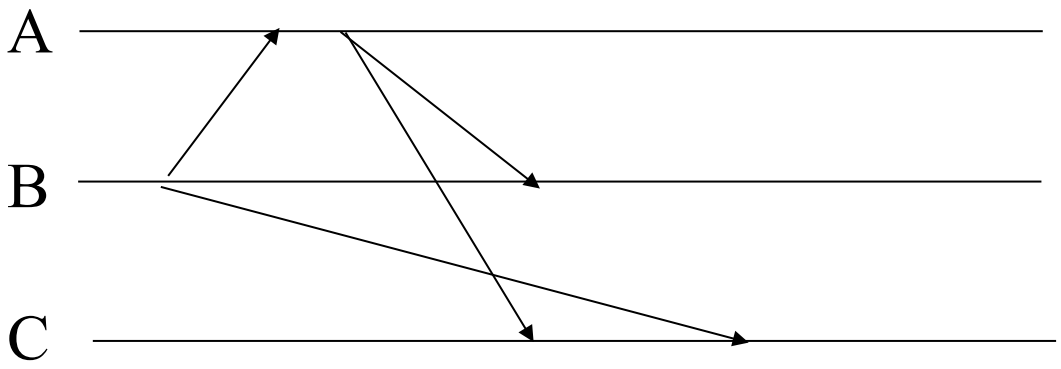
- Also, Lamport timestamps do not detect causality violations. Causality violations are caused by long communications delays in one channel that are not present in other channels or a non-FIFO channel.



A and C will never know messages were out of order

Causality Violation

- Causality violation example: A gets a message from B that was sent to all nodes. A responds by sending an answer to all nodes. C gets A's answer to B before it receives B's original message.
- How can B tell that this message is out of order?
  - Assume one send event for a set of messages



Causality: Solution

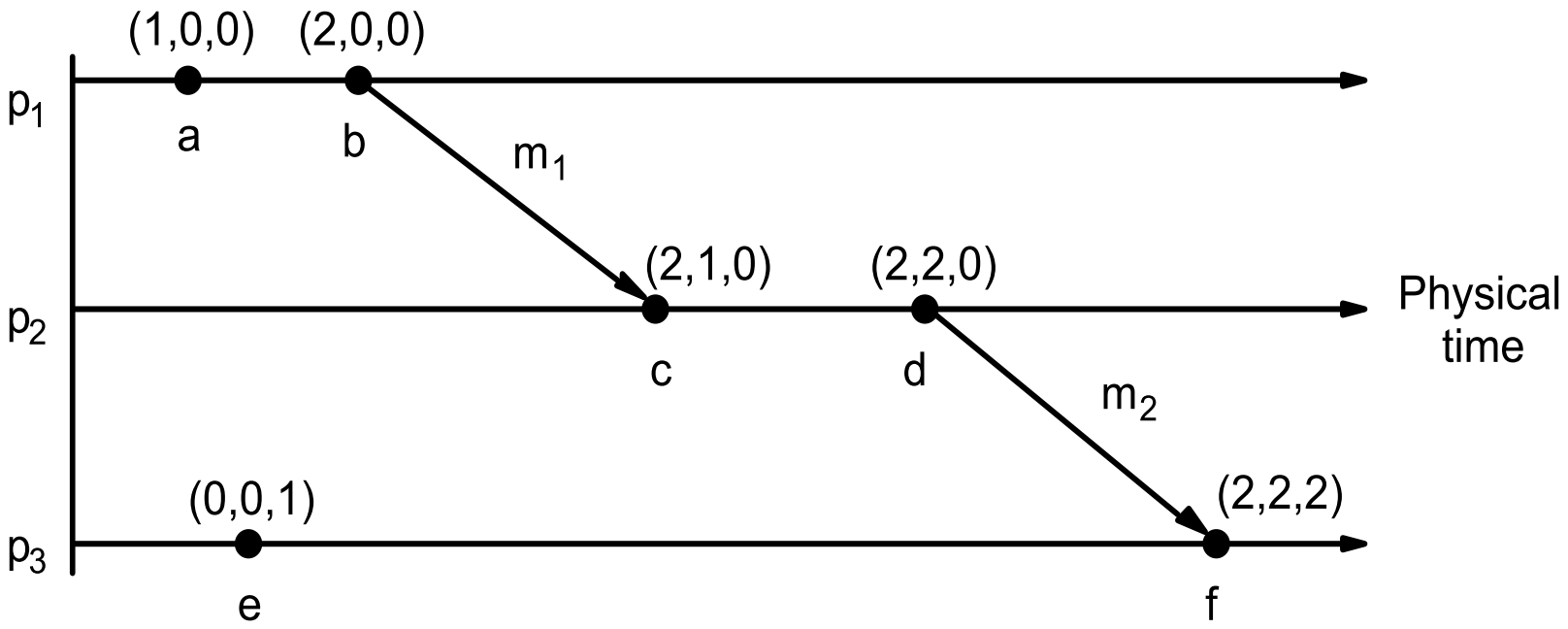
- The solution is vector timestamps: Each node maintains an array of counters.
- If there are N nodes, the array has N integers VT(N). VT(I) = C, the local clock, if I is the designation of the local node.
- In general, VT(X) is the latest info the node has on what X's local clock is.
- Gives us the property e → f iff VT(e) < VT(f)

Vector Timestamps

- Each site has a local clock incremented at each event.
- The vector clock timestamp is **piggybacked** on **each** message sent.

- **RULES:**
  - Local clock is incremented for a local event and for a send event. The message carries the vector time stamp.
  - When a message is received, the local clock is incremented by one. Each other component of the vector is increased to the received vector timestamp component if the current value is less. That is, the maximum of the two components is the new value.

Vector Clocks



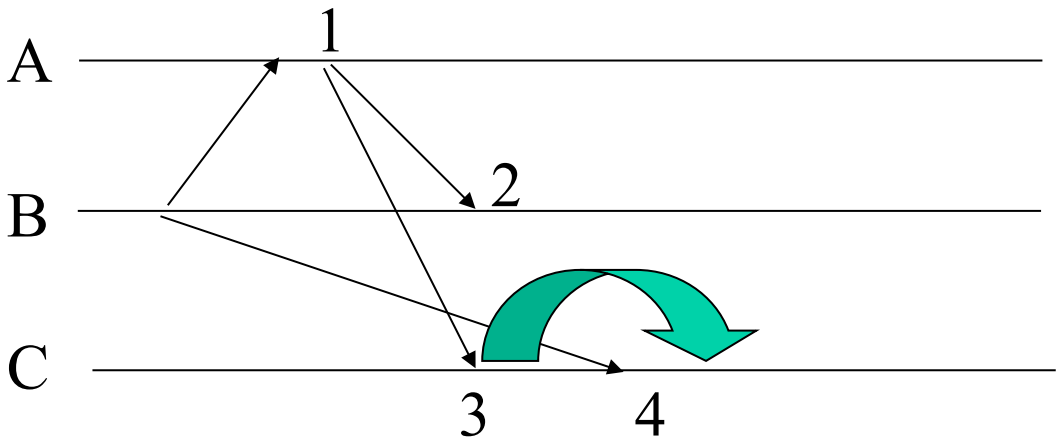
VT(a) < VT(b), hence a→ b

neither VT(x) < VT(y), nor VT(y) < VT(x), hence x || y

Vector Clock Comparison

- VT1 > VT2 if for each component j, VT1[j] >= VT2[j], and for some component k, VT1[k] > VT2[k]
- VT1 = VT2 if for each j, VT1[j] = VT2[j]
- Otherwise, VT1 and VT2 are incomparable and the events they represent are concurrent

Assume that only sends e recvs are the counting events

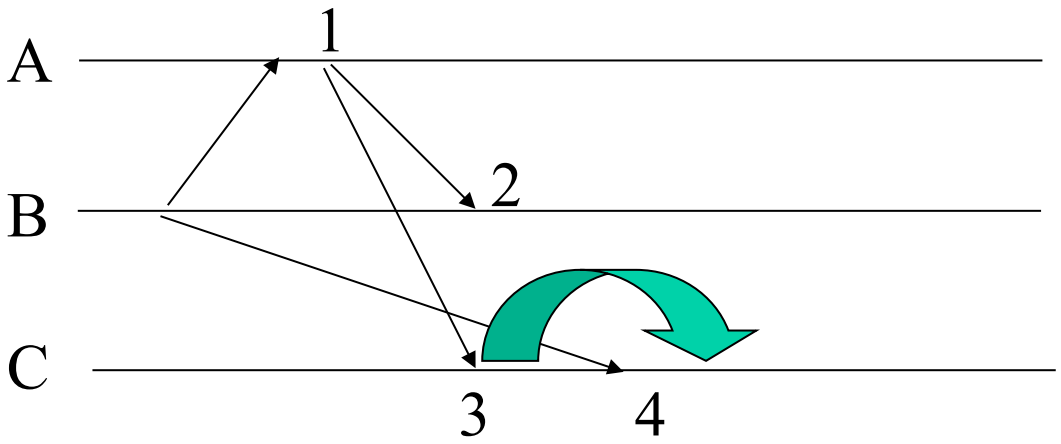


Clock at point
1= (2,1,0)
2= (2,2,0)
3= (2,1,1)
4= (2,1,2)

??

Vector Clock Comparison

Assume that only send_message() is the counting event



Clock at point
1= (0,1,0)
2= (1,1,0)
3= (1,1,0)
4= (0,1,0)

??

Vector Clock Exercise

- Assuming the counting events are send and receive:
- What is the vector clock at events a-f?
- Which events are concurrent?