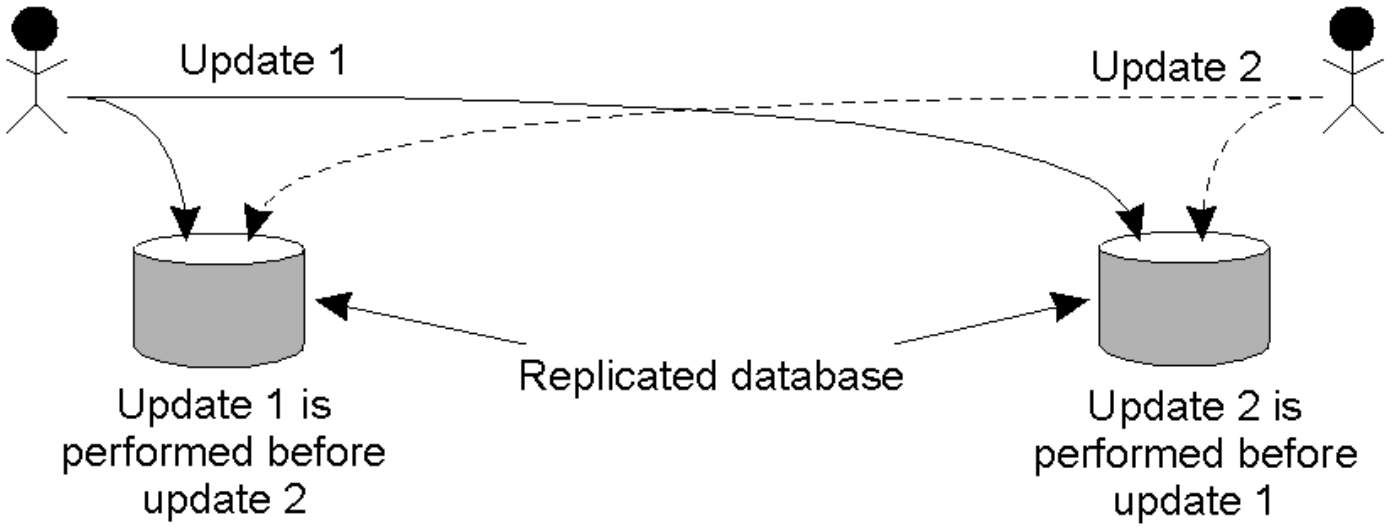
Synchronisation

What Can Go Wrong



- Updating a replicated database: Update 1 adds 100 euro to an account, Update 2 calculates and adds 1% interest to the same account.
- Clock reading might be different for different computers!
- Inconsistent state

Possible Solutions

- Include the time of every update *in* the update message
  - Clock synchronisation
- Order events in the network
  - Logical clock
- Transactions

Computer Clock

- We need to measure time accurately:
  - to know the time an event occurred at a computer
- Algorithms for clock synchronization useful for
  - concurrency control based on timestamp ordering
  - authenticity of requests e.g. in Kerberos
- Each computer in a DS has its own internal clock
- Even if clocks on all computers in a DS are set to the same time, their clocks will eventually vary quite significantly unless corrections are applied

Synchronization based on “Actual Time”.

Note: time is really easy on a uniprocessor system.

Achieving agreement on time in a DS is not trivial.

Question: is it even possible to synchronize all the clocks in a Distributed System?

With multiple computers, “clock skew” ensures that no two machines have the same value for the “current time”. But, how do we measure time?

How Do We Measure Time?

Turns out that we have only been measuring time accurately with a "global" atomic clock since *Jan. 1st, 1958* (the "beginning of time").

Refer to pages 243-246 of the textbook for all the details – it's quite a story.

Bottom Line: measuring time is not as easy as one might think it should be.

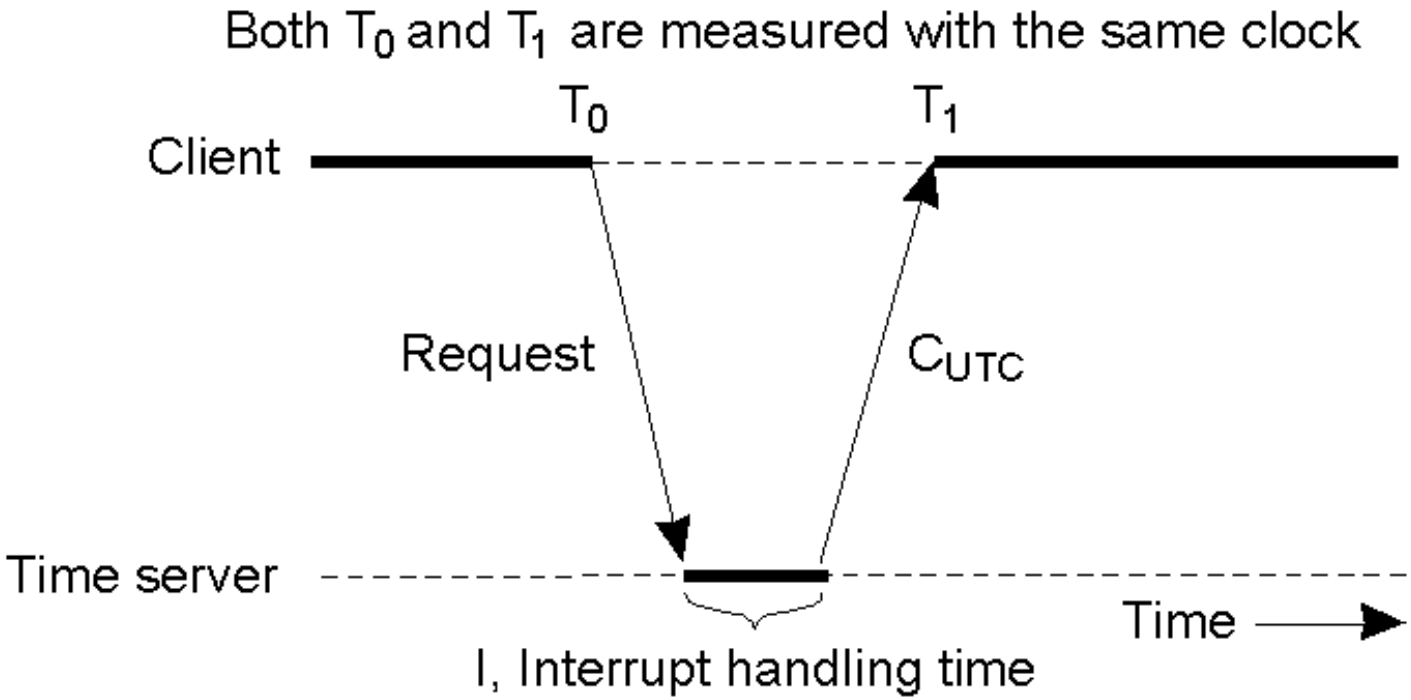
FYI: Coordinated Universal Time

- International Atomic Time is based on very accurate physical clocks (drift rate 10^{-13})
- UTC is an international standard for time keeping
- It is based on atomic time, but occasionally adjusted to astronomical time
- It is broadcast from radio stations on land and satellite (e.g. GPS)
- Computers with receivers can synchronize their clocks with these timing signals
- Signals from land-based stations are accurate to about 0.1-10 millisecond
- Signals from GPS are accurate to about 1 microsecond

Clock Synchronization

- There exists a *time server* receiving signals from a UTC source
  - Cristian's algorithm
- There is no UTC source available
  - Berkley's algorithm
- Exact time *does not matter!*
  - Lamport's algorithm

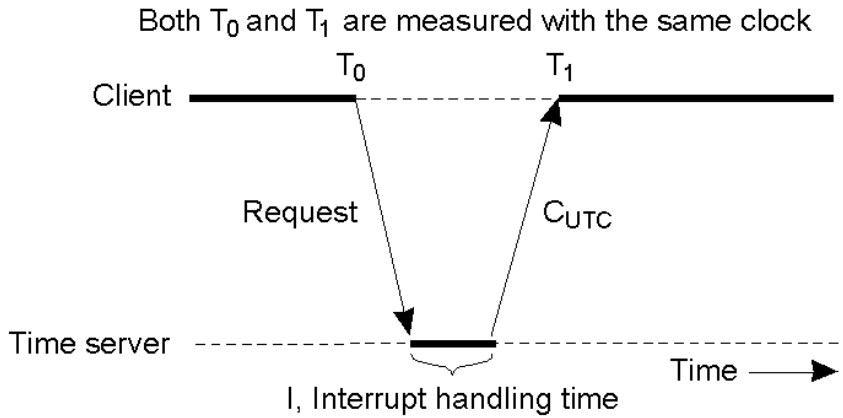
Clock Sync. Algorithm: Cristian's



1. Every computer periodically asks the "time server" for the current time
2. The server responds ASAP with the current time C_{UTC}
3. The client sets its clock to C_{UTC}

Problems

Major problem: if time from time server is less than the client – resulting in time running backwards on the client! (Which cannot happen – time *does not go backwards*).



Introduce changes gradually

Minor problem: results from the delay introduced by the network request/response: latency

Best estimate $(T_1-T_0)/2$

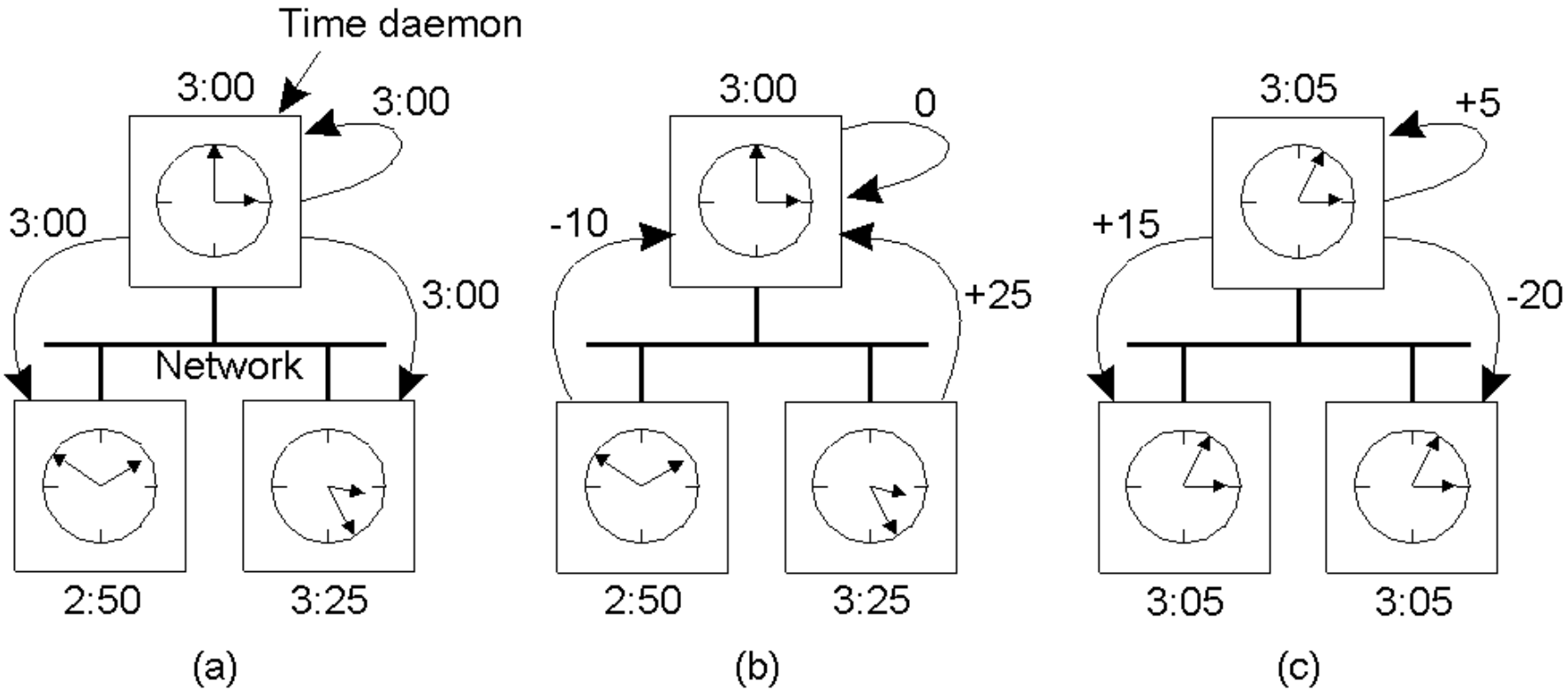
If the interrupt handling time, I, is known, $(T_1-T_0-I)/2$

Use series of measurements

Berkley Algorithm

- An algorithm for internal synchronization of a group of computers
- A *master* polls to collect clock values from the others (*slaves*)
- The master uses round trip times to estimate the slaves' clock values
- It takes an average
- It sends the required adjustment to the slaves (better than sending the time which depends on the round trip time)
- If master fails, can elect a new master to take over

The Berkeley Clock Sync. Algorithm



Clocks that are running fast are slowed down. Clocks running slow jump forward.

Berkley Algorithm in Work (1)

Computers	Clock Reading
A (daemon)	3:00
B (left)	2:50
C (right)	3:25

Computers	Ahead/Behind
A (daemon)	0:00
B (left)	-0:10
C (right)	+0:25

Berkley Algorithm in Work (2)

Average time: 3:05

Computers	Needed Adjustment
A (daemon)	+0:05
B (left)	+0:15
C (right)	-0:20

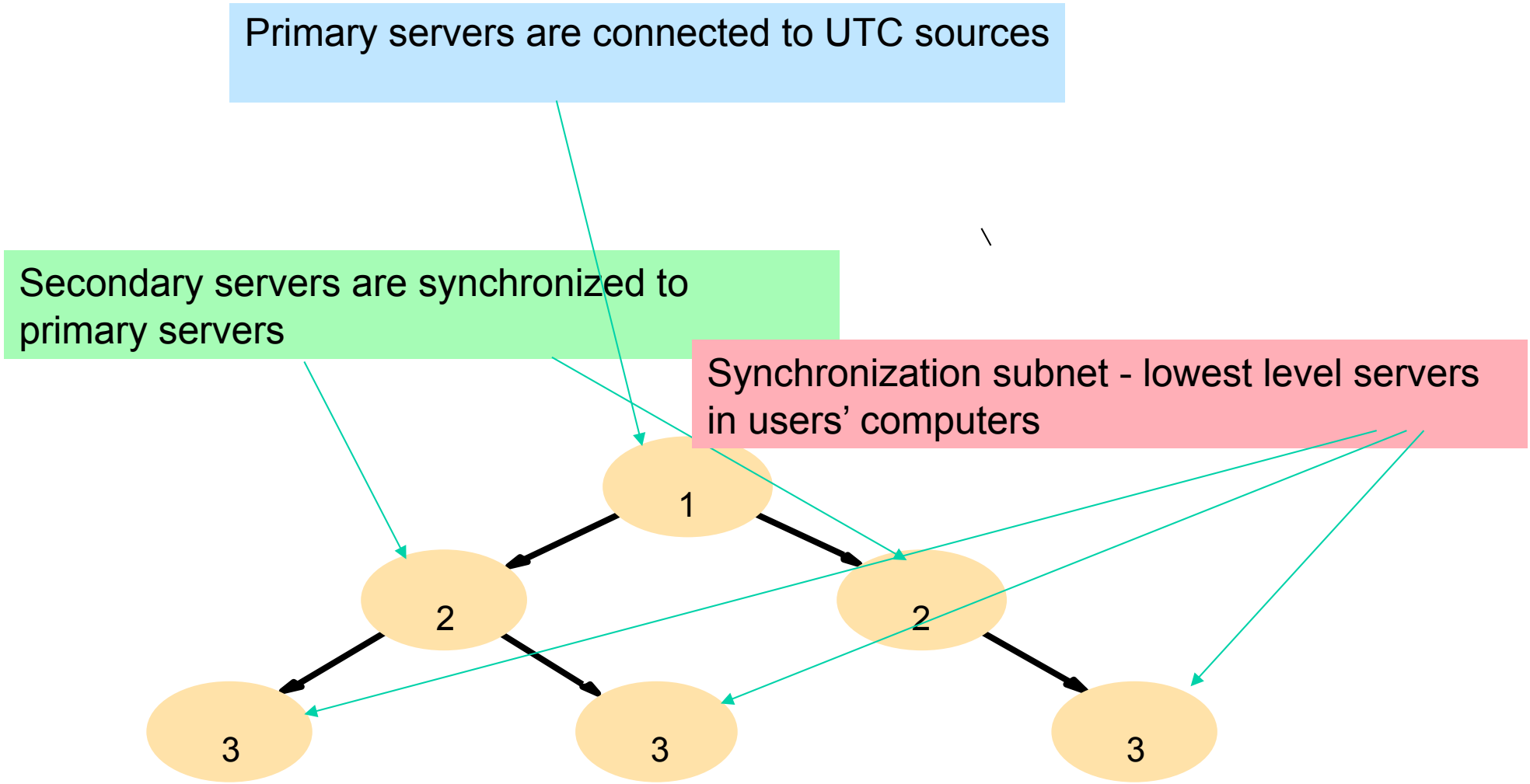
Other Clock Sync. Algorithms

Both Cristian's and the Berkeley Algorithm are centralised algorithms.

Decentralised algorithms also exist, and the Internet's Network Time Protocol (NTP) is the best known and most widely implemented.

NTP can synchronize clocks to within a 1—50 msec accuracy.

Network Time Protocol (NTP)



Logical Clocks

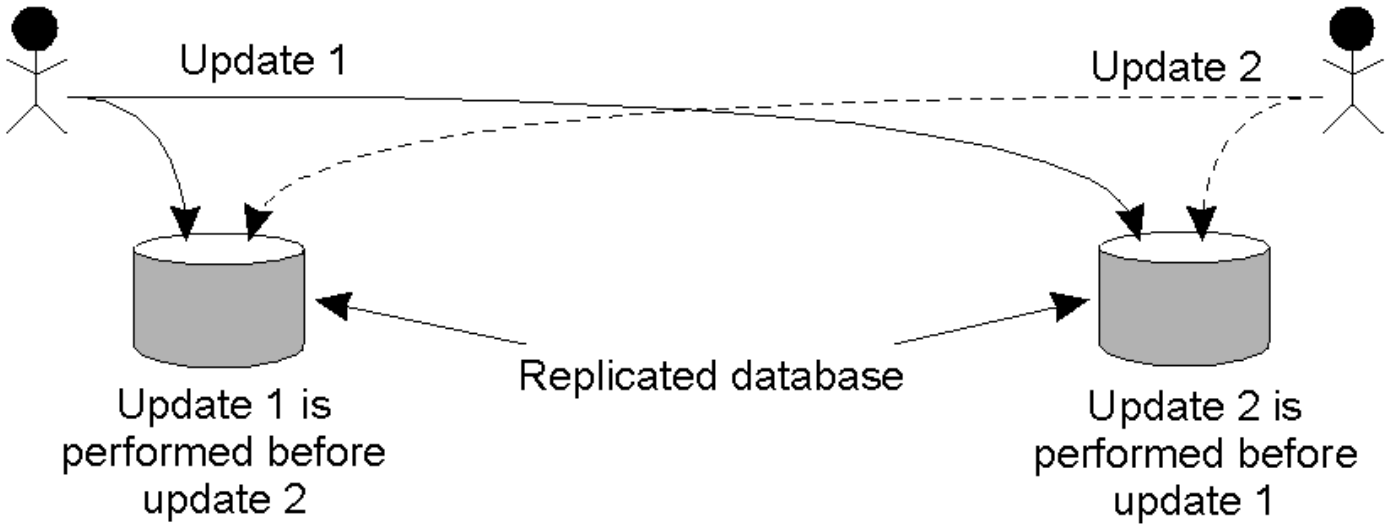
Synchronization based on “relative time”.

Note that (with this mechanism) there is no requirement for “relative time” to have any relation to the “real time”.

What’s important is that the processes in the Distributed System *agree on the ordering in which certain events occur*.

Such “clocks” are referred to as *Logical Clocks*.

What Can Go Wrong-2



- Updating a replicated database
- *Even if the clock is synchronized,* due to network delays, the updates may come in different order.
- **Whoops!**

Lamport's Logical Clocks

First point: if two processes do not interact, then their clocks do not need to be synchronized – they can operate *concurrently* without fear of interferring with each other.

Second (critical) point: it does not matter that two processes share a common notion of what the "real" current time is. What does matter is that the processes have some agreement on the order in which certain events occur.

Lamport used these two observations to define the "happens-before" relation (also often referred to within the context of *Lamport's Timestamps*).

The "Happens-Before" Relation (1)

- If A and B are events in the same process, and A occurs before B, then we can state that *"A happens-before B"* is true.
- Equally, if A is the event of a message being sent by one process, and B is the event of the same message being received by another process, then *"A happens-before B"* is also true.
  - (Note that a message cannot be received before it is sent, since it takes a finite, nonzero amount of time to arrive … and, of course, time is not allowed to run backwards).
- Obviously, if *"A happens-before B"* and "*B happens-before C"*, then it follows that *"A happens-before C"*.

The "Happens-Before" Relation (2)

- Now, assume three processes are in a DS: A, B and C.
- All have their own physical clocks
- A sends a message to B
- If the time the message was sent (attached to the message) **exceeds** the time of arrival at B, things are NOT OK (as *"A happens-before B"* is not true, and this **cannot be allowed** as the receipt of a message has to occur *after* it was sent).

The "Happens-Before" Relation (3)

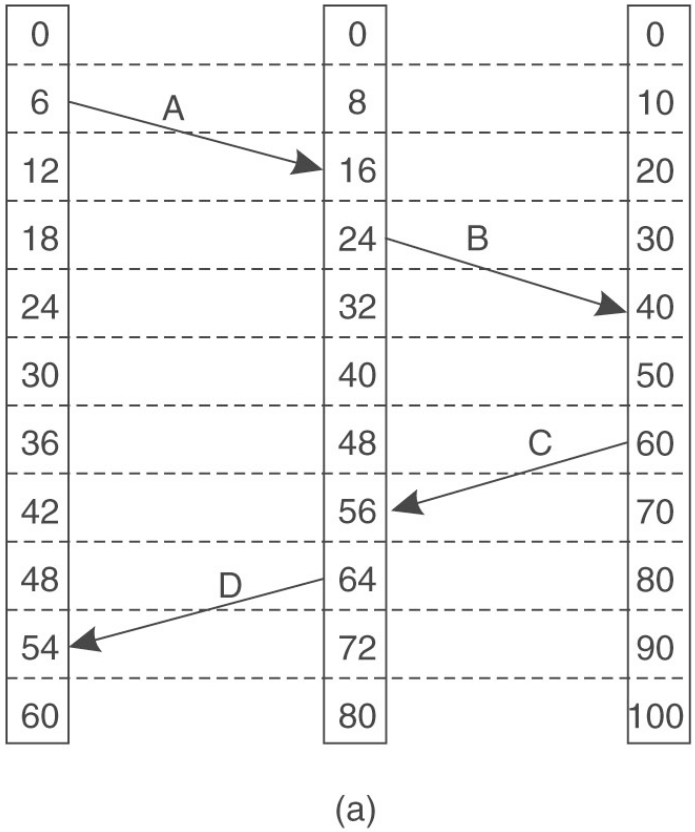
The question to ask is:

How can some event that *"happens-before"* some other event possibly have occurred at a later time??

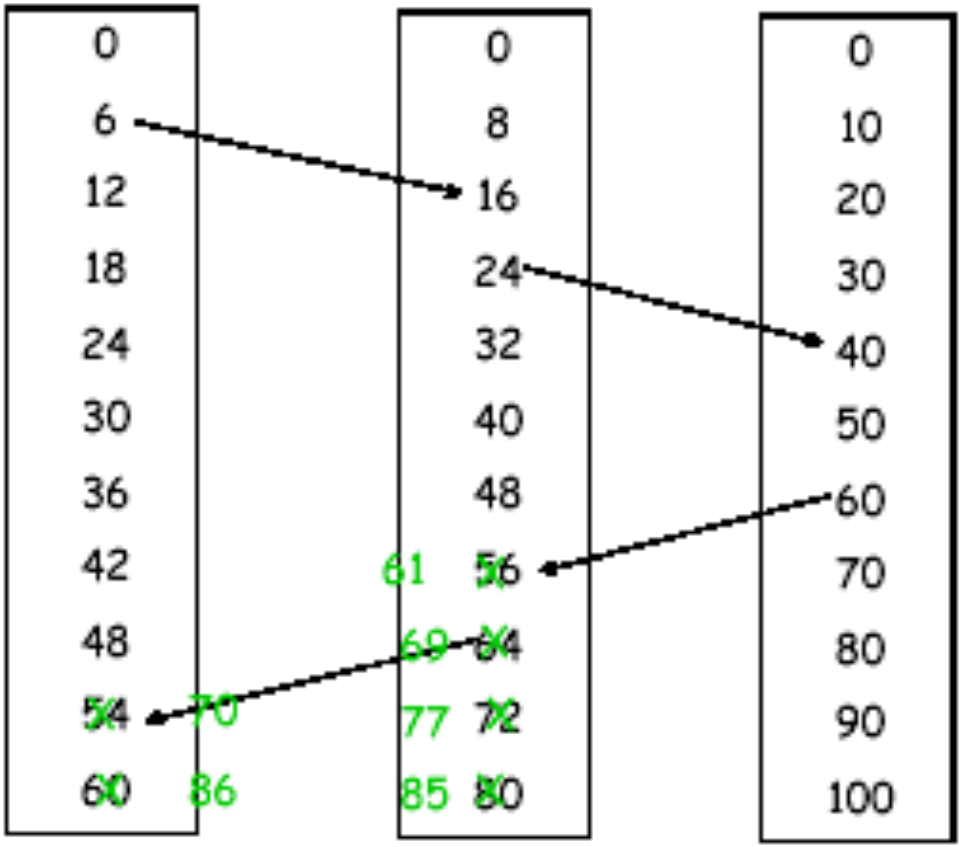
The answer is: it can't!

So, Lamport's solution is to have *the receiving process adjust its clock forward to one more than the sending timestamp value*. This allows the *"happens-before"* relation to hold, and also keeps all the clocks running in a synchronised state. The clocks are all kept in sync *relative to each other*.

Example of Lamport's Timestamps



(a)



Lamport's algorithm corrects the clock

Lamport's Timestamps: Overview

- It is hard to synchronise time across two systems
- Every message is timestamped
- If the local clock disagree, update the local clock (and always move it forward!)

- If two systems do not exchange messages, they do not need a common clock
  - If do not communicate, no ordering
  - Is it possible to achieve total ordering of events in the system?

Applications of the Lamport's Timestamps

- Total ordering of events in the system
  "Totally Ordered Multicasting"
  - To solve our problem with the replicated database
- Vector Timestamps (to capture *causality*)
  - Bulletin Board Systems, Chat rooms
- Global state
  Local state of each process + messages in transit
  - Termination detection

Consult the book (if you want).