Chapter 4
Causality, Time, and Global States II

Distributed Systems

SS 2019

Fabian Kuhn

Observable Behavior

Recall Executions / Schedules

- An exec. is an alternating sequence of configurations and events
- A schedule S is the sequence of events of an execution
  - Possibly including node inputs
- Schedule restriction for node v:

$$S|v \coloneqq \text{"sequence of events seen by } v\text{"}$$

Causal Shuffles

We say that a schedule $\boldsymbol{S'}$ is a **causal shuffle** of schedule $\boldsymbol{S}$ iff

$$\forall \boldsymbol{v} \in \boldsymbol{V}: \quad \boldsymbol{S}|\boldsymbol{v} = \boldsymbol{S'}|\boldsymbol{v}.$$

Observation: If S' is a causal shuffle of S, no node/process can distinguish between S and S'.

Causal Order

Logical clocks are based on a **causal order** of the events

- In the order, event e should occur before event e' if event e provably occurs before event e'
  - In that case, the clock value of e should be smaller than the one of e'

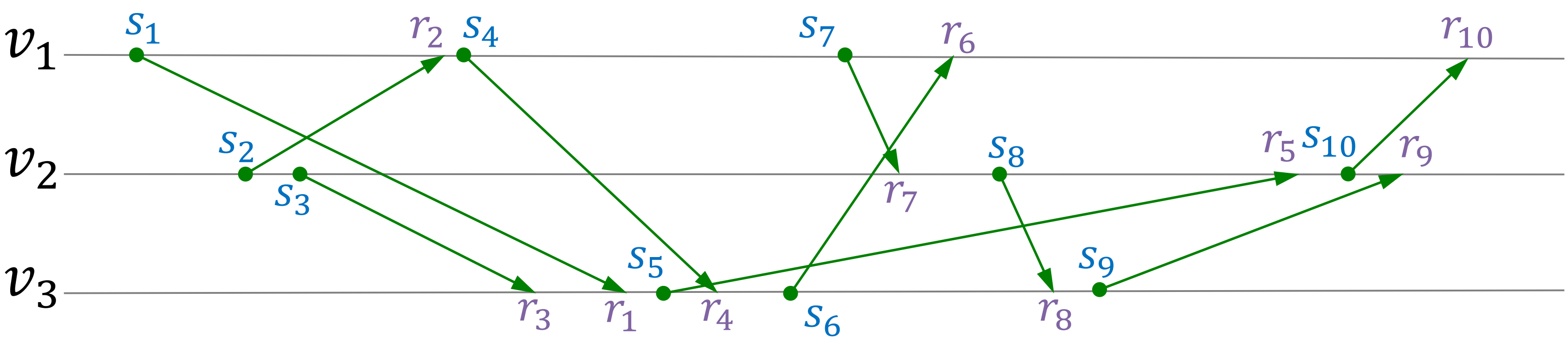
For a given schedule S:

- The distributed system cannot distinguish S from another schedule S' if and only if S' is a causal shuffle of S.
  - causal shuffle $\Longrightarrow$ no node can distinguish
  - no causal shuffle $\Longrightarrow$ some node can distinguish

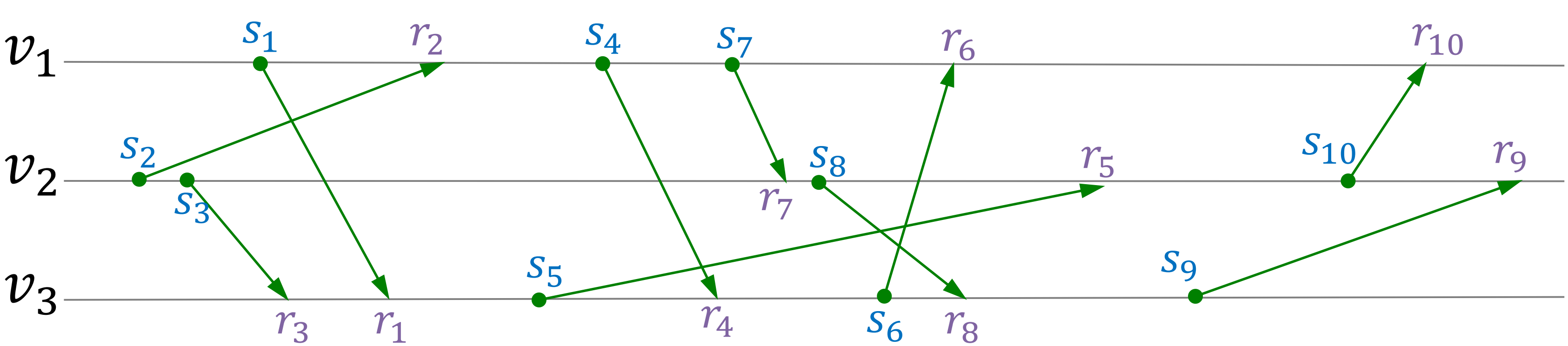
Event e provably occurs before e' if and only if e appears before e' in all causal shuffles of S

Causal Shuffles / Causal Order Example

Schedule S



Some Causal Shuffle S'



Lamport's Happens-Before Relation

Definition: The **happens-before relation $\Rightarrow_S$** on a schedule S is a pairwise relation on the send/receive events of S and it contains

1. All pairs (e, e') where e precedes e' in S and e and e' are events of the same node/process.
2. All pairs (e, e') where e is a send event and e' the receive event for the same message.
3. All pairs (e, e') where there is a third event e'' such that
$$e \Rightarrow_S e'' \quad \land \quad e'' \Rightarrow_S e'$$
   – Hence, we take the **transitive closure** of the relation defined by 1. and 2.

Theorem: For a schedule S and two (send and/or receive) events e and e', the following two statements are equivalent:

a) Event e happens-before e', i.e., $\boldsymbol{e \Rightarrow_S e'}$.

b) Event e precedes e' in all causal shuffles S' of S.

Lamport Clocks

Basic Idea:

1. Each event e gets a clock value $\tau(e) \in \mathbb{N}$
2. If e and e' are events at the same node and e precedes e', then
$$\tau(e) < \tau(e')$$
3. If s_M and r_M are the send and receive events of some msg. M,
$$\tau(s_M) < \tau(r_M)$$

Observation:

- For clock values $\tau(e)$ of events e satisfying 1., 2., and 3., we have
$$\boldsymbol{e \Rightarrow_S e' \quad \longrightarrow \quad \tau(e) < \tau(e')}$$
  – because $<$ relation (on $\mathbb{N}$) is transitive
- Hence, the partial order defined by $\tau(e)$ is a superset of $\Rightarrow_S$

Lamport Clocks

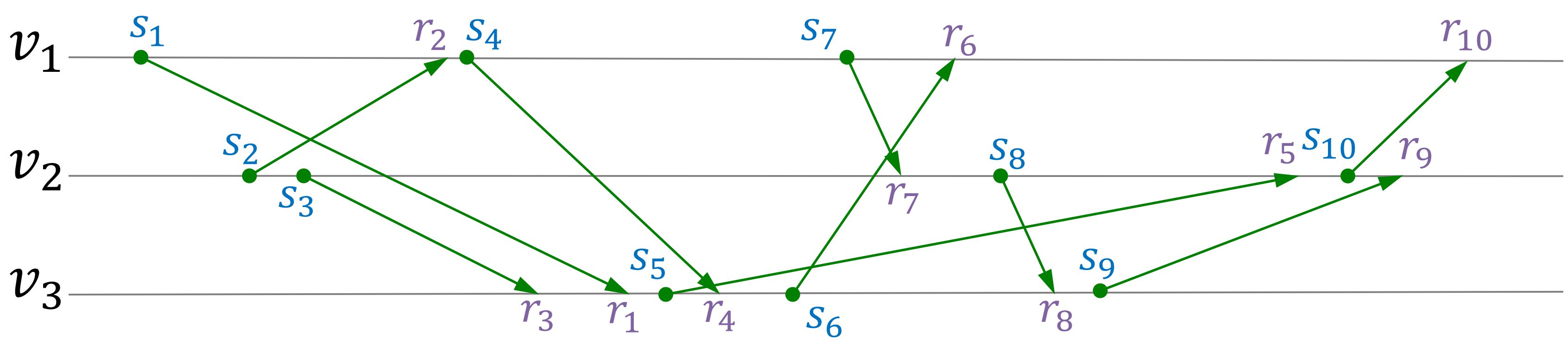
Algorithm:

- Each node u keeps a counter c_u which is initialized to 0
- For any non-receive event e at node u, node u computes
$$c_u \coloneqq c_u + 1;\ \tau(e) \coloneqq c_u$$
- For any send event s at node u, node u attaches the value of $\tau(s)$ to the message
- For any receive event r at node u (with corresponding send event s), node u computes
$$c_u \coloneqq \max\{c_u, \tau(s)\} + 1;\ \ \tau(r) \coloneqq c_u$$

Consistent Cut

Cut

Given a schedule S, a cut is a subset C of the events of S such that for all nodes $v \in V$, the events in C happening at v form a prefix of the sequence of events in $S|v$.

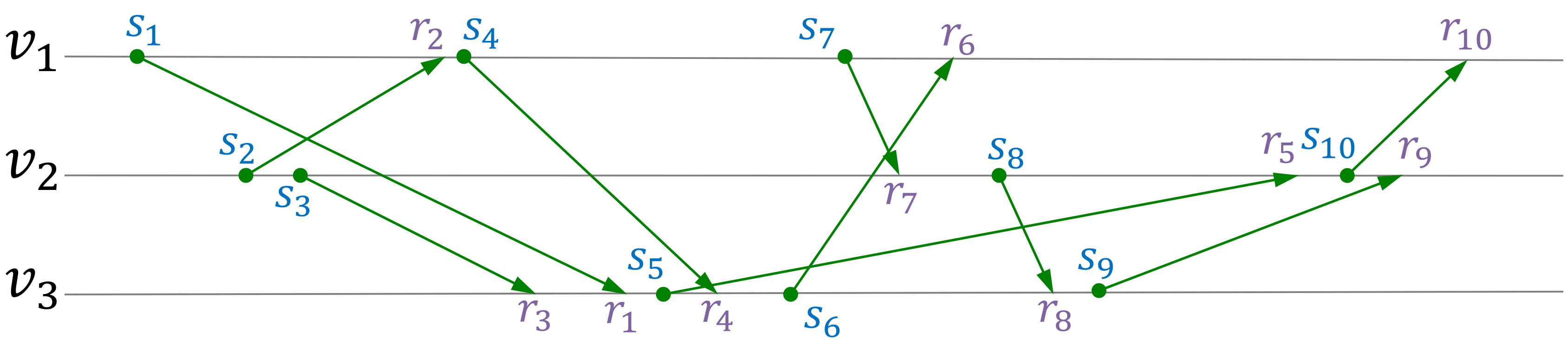


Consistent Cut

Consistent Cut

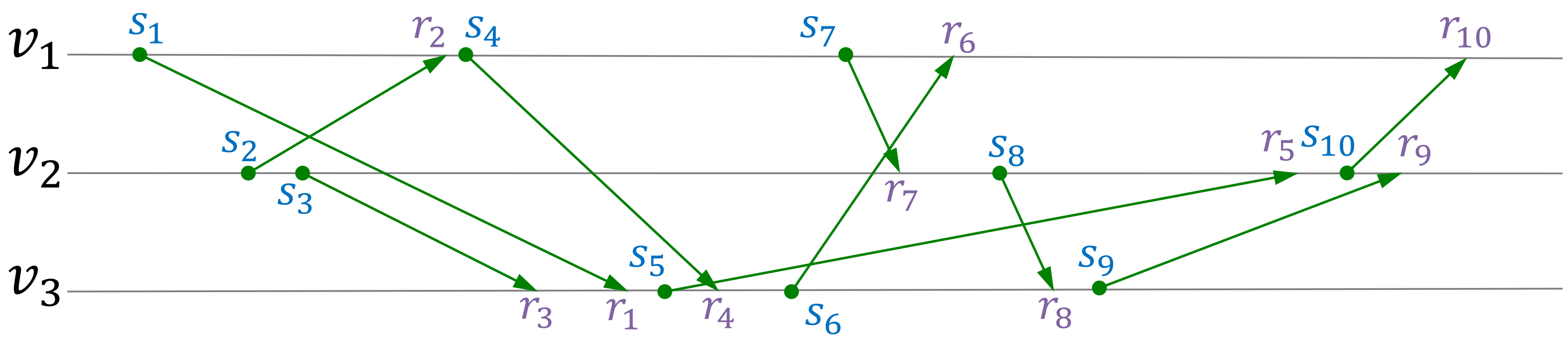
Given a schedule S, a consistent cut C is cut such that for all events $e \in C$ and all events f in S, it holds that

$$\boldsymbol{f \Rightarrow_S e \quad \longrightarrow \quad f \in C}$$

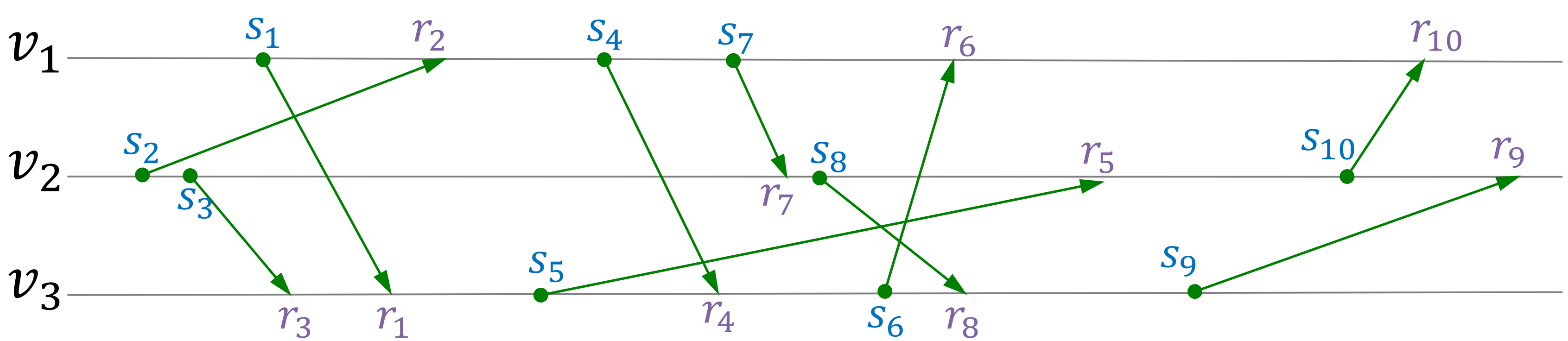


Consistent Cut

Schedule S



Some Causal Shuffle S'



Consistent Cuts

Claim: Given a schedule S, a cut C is a consistent cut if and only if for each message M with send event s_M and receive event r_M, if $r_M \in C$, then it also holds that $s_M \in C$.

Consistent Snapshot

Consistent Snapshot = Global Snapshot = Consistent Global State

- A consistent snapshot is a global system state which is consistent with all local views.

Global System State (for schedule S)

- A vector of intermediate states (in S) of all nodes and a description of the messages currently in transit
  - Remark: If nodes keep logs of messages sent and received, the local states contain the information about messages in transit.

Consistent Snapshot

- A global system state which is an intermediate global state for some causal shuffle of S (consistent with all local views)

Consistent Snapshot

Claim: A global system state is a consistent snapshot if and only if it corresponds to the node states of some consistent cut C.

Computing a Consistent Snapshot

Using Logical Clocks

- Assume that each event e has a clock value $\tau(e)$ such that for two events e, e',

$$e \Rightarrow_S e' \quad \longrightarrow \quad \tau(e) < \tau(e')$$

- Given τ, define $C(\tau)$ as the set of events e with $\tau(e) \leq \tau_0$

Claim: $\forall \tau \geq 0$: $C(\tau)$ is a consistent cut.

Remark: Not always clear how to choose τ_0

- τ_0 large: not clear how long it takes until snapshot is computed
- τ_0 small: snapshot is "less up-to-date"

Chandy-Lamport Snapshot Algorithm

Goals: Compute a consistent snapshot in a running system

Assumptions:
- Does not require logical clocks
- Channels are assumed to have FIFO property
- No failures
- Network is (strongly) connected
- Any node can issue a new snapshot

Remark: The FIFO property can always be guaranteed
- sender locally numbers messages on each outgoing channel
- messages with smaller numbers have to be processed before messages with larger numbers
- works as long as messages are not lost

Chandy-Lamport Snapshot Algorithm

Overview:

- Assume that node s initiates the snapshot computation
- The times for recording the state at different nodes is determined by sending around *marker* messages
- When receiving the first *marker* message, a node records its state and sends *marker* messages to all (outgoing) neighbors
- On each incoming channel, the set of messages which are received between recording the state and receiving the *marker* message (on this channel) are in transit in the snapshot.
- After receiving a *marker* message on all incoming channels, a nodes has finished its part of the snapshot computation

Chandy-Lamport Snapshot Algorithm

Initially: Node s records its state

When node u receives a *marker* message from node v:

if u has not recorded its state then
    u records its state
    set of msg. in transit from v to u is empty
    u starts recording messages on all other incoming channels
else
    the set of msg. in transit from v to u is the set of recorded msg.
    since starting to record msg. on the channel

(Immediately) after node u records its state:

Node u sends *marker* msg. on all outgoing channels

– before sending any other message on those channels

Chandy-Lamport Snapshot Algorithm

Theorem: The Chandy-Lamport algorithm computes a consistent cut and it correctly computes the messages in transit over this cut.

Chandy-Lamport Snapshot Algorithm

Theorem: The Chandy-Lamport algorithm computes a consistent cut and it correctly computes the messages in transit over this cut.

Applications of Consistent Snapshots

Testing Stable System Properties

- A stable property is a property which once true, remains true
- More formally: a predicate P on global configurations such that if P is true for some configuration C, P also holds for all configurations which can be reached from C

Testing a stable property:

- test whether property holds for a consistent snapshot

Safety: Only evaluates to true if the property holds

- the current state is reachable from every consistent snapshot state

Liveness: If the property holds, it will eventually be detected

- initiating a snapshot (using Chandy-Lamport) leads to snapshot configuration which is reachable from the current configuration

Applications of Consistent Snapshots

Distributed Garbage Collection

- Erase objects (e.g., variables stored at some node(s)) to which no reference exists any more
- References can be at other nodes, in messages in transit, ...
- "No reference to object x" is a stable system property

Distributed Deadlock Detection

- Two processes/nodes wait for each other
- Deadlock is also a stable property

Distributed Termination Detection

- "Distributed computation has terminated" is a stable property
- Note, need also see messages in transit

Clock Synchronization

Motivation

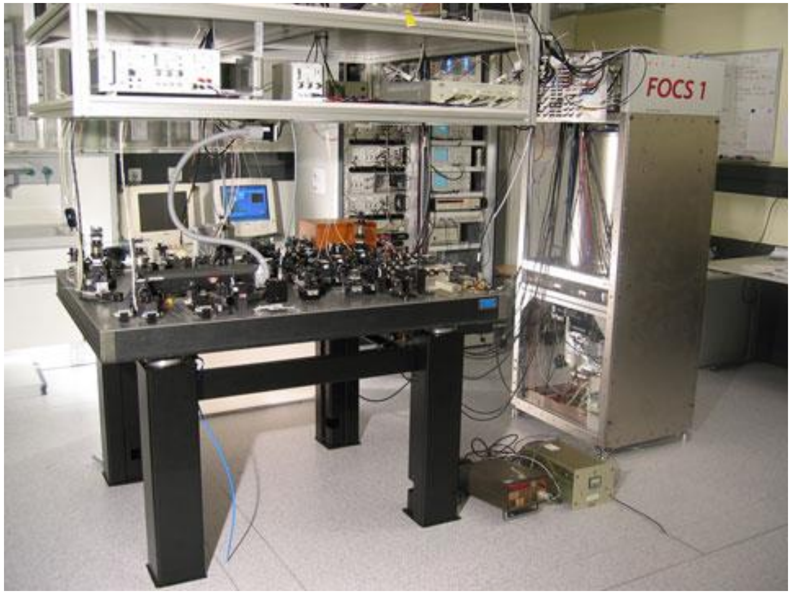
- Logical Time (“happens-before”)
  - Determine the order of events in a distributed system
  - Synchronize resources
- Physical Time
  - Timestamp events (email, sensor data, file access times etc.)
  - Synchronize audio and video streams
  - Measure signal propagation delays (Localization)
  - Wireless (TDMA, duty cycling)
  - Digital control systems (ESP, airplane autopilot etc.)

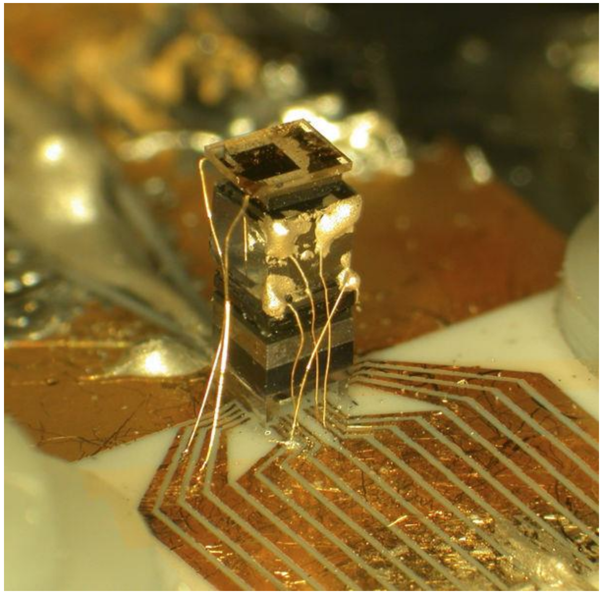
Properties of Clock Synch. Algorithms

- External vs. internal synchronization
  - External sync: Nodes synchronize with an external clock source (UTC)
  - Internal sync: Nodes synchronize to a common time
    - to a leader, to an averaged time, ...
- One-shot vs. continuous synchronization
  - Periodic synchronization required to compensate clock drift
- Online vs. offline time information
  - Offline: Can reconstruct time of an event when needed
- Global vs. local synchronization
- Accuracy vs. convergence time, Byzantine nodes, …

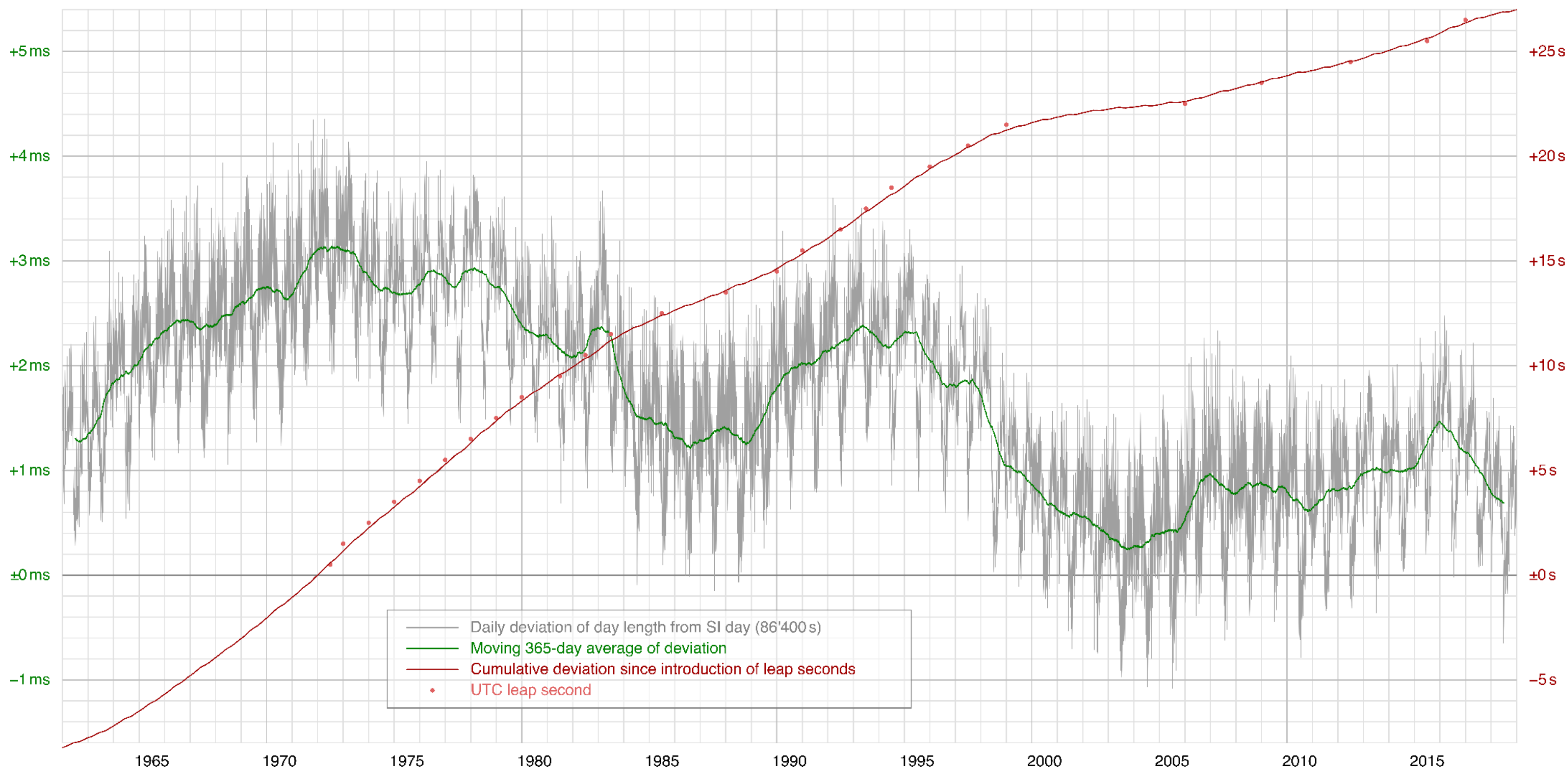
World Time (UTC)

- Atomic Clock
  - UTC: Coordinated Universal Time
  - SI definition 1s := 9192631770 oscillation cycles of the Caesium-133 atom
  - Atoms are excited to oscillate at their resonance frequency and cycles can be counted.
  - Almost no drift (about 1s in 10 Million years)
  - Getting smaller and more energy efficient!





Atomic Clocks vs. Length of a Day



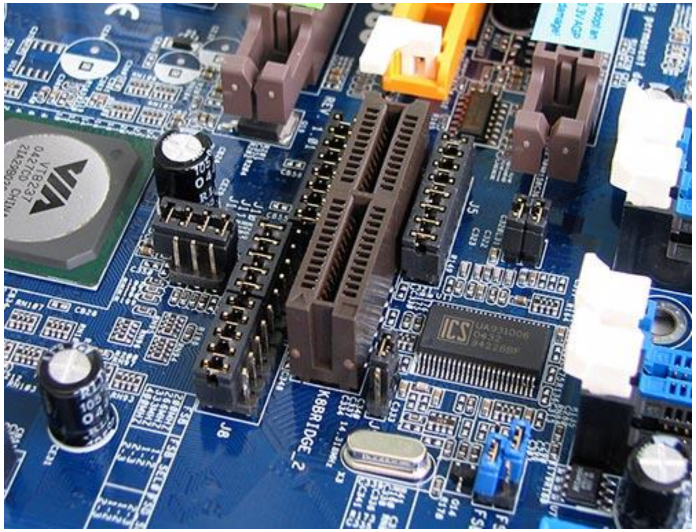
Access to UTC

- Radio Clock Signal
  - Clock signal from a reference source (atomic clock) is transmitted over a long wave radio signal
  - DCF77 station near Frankfurt, Germany transmits at 77.5 kHz with a transmission range of up to 2000 km
  - Accuracy limited by the propagation delay of the signal, Frankfurt-Freiburg is about 0.8 ms
  - Special antenna/receiver hardware required

- GPS (Global Positioning System)
  - Satellites continuously transmit own position and time code
  - Special antenna/receiver hardware required
  - Positioning in space and time!

Clock Devices in Computers

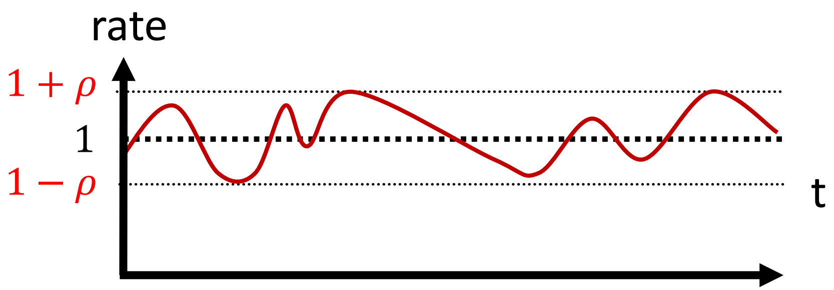
- Real Time Clock (IBM PC)
  - Battery backed up
  - 32.768 kHz oscillator + Counter
  - Get value via interrupt system



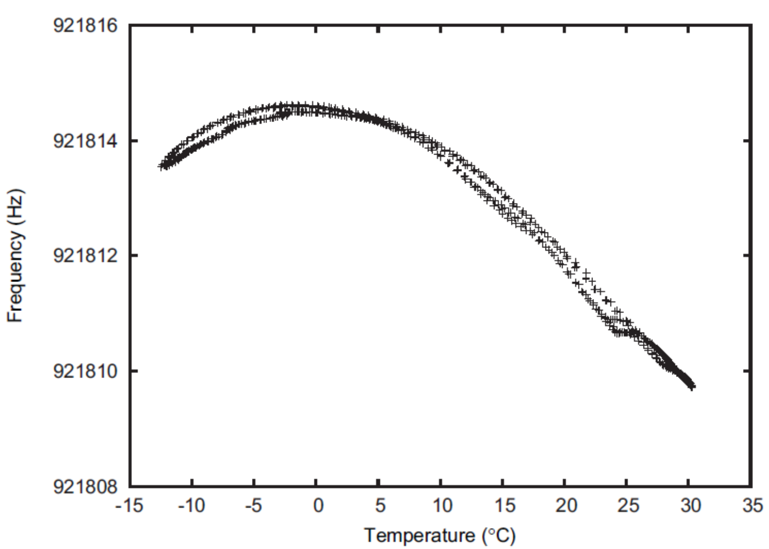
- HPET (High Precision Event Timer)
  - Oscillator: 10 Mhz … 100 Mhz
  - Up to 10 ns resolution!
  - Schedule threads
  - Smooth media playback
  - Usually inside Southbridge

Clock Drift

- Clock drift: deviation from the nominal rate dependent on power supply, temperature, etc.



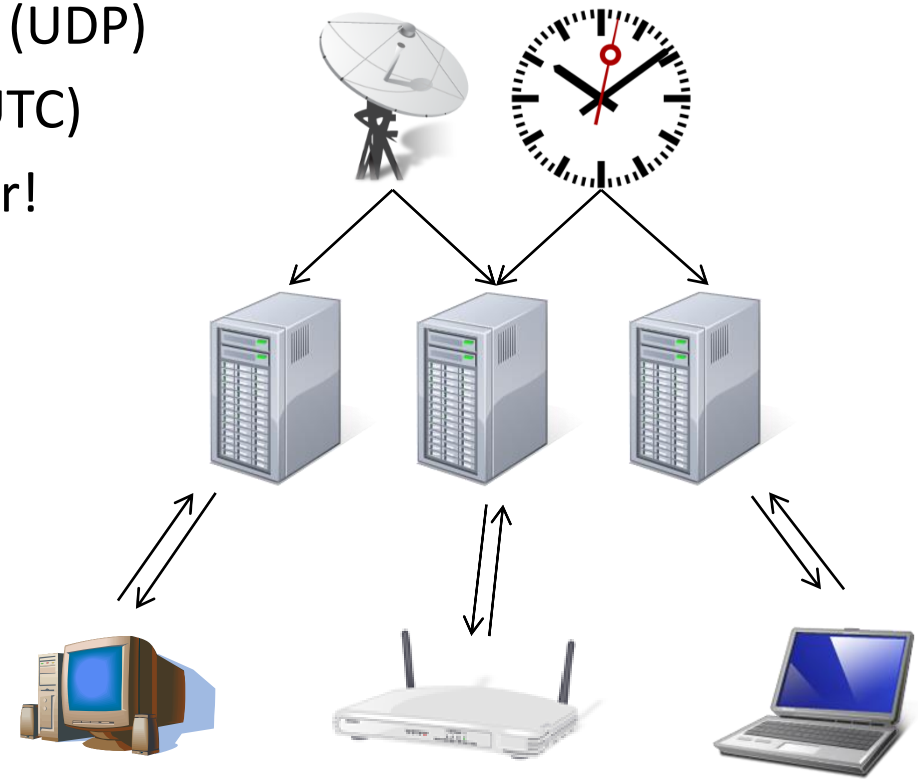
- E.g., TinyNodes have a max. drift of 30-50 ppm (parts per million)



This is a drift of up to 50µs per second or 0.18s per hour

Clock Synch. in Computer Networks

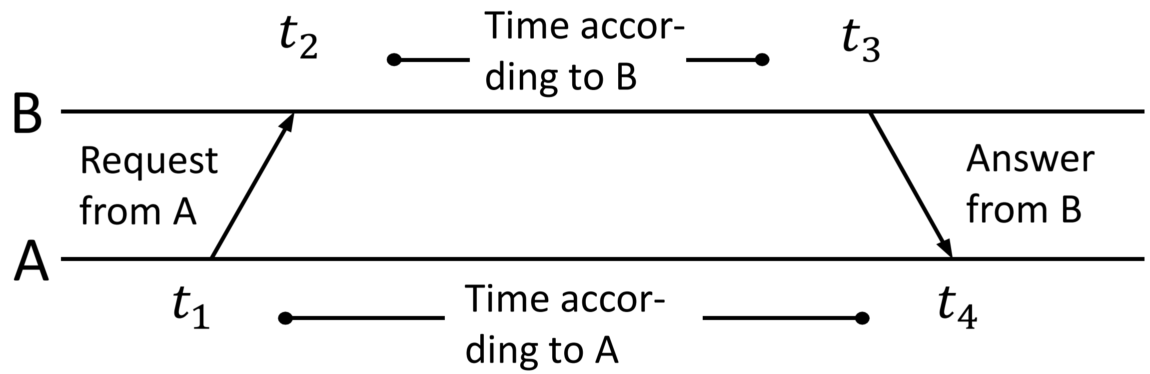
- Network Time Protocol (NTP)
- Clock sync via Internet/Network (UDP)
- Publicly available NTP Servers (UTC)
- You can also run your own server!



- Packet delay is estimated to reduce clock skew

Propagation Delay Estimation (NTP)

- Measuring the Round-Trip Time (RTT)



- Propagation delay δ and clock skew Θ can be calculated

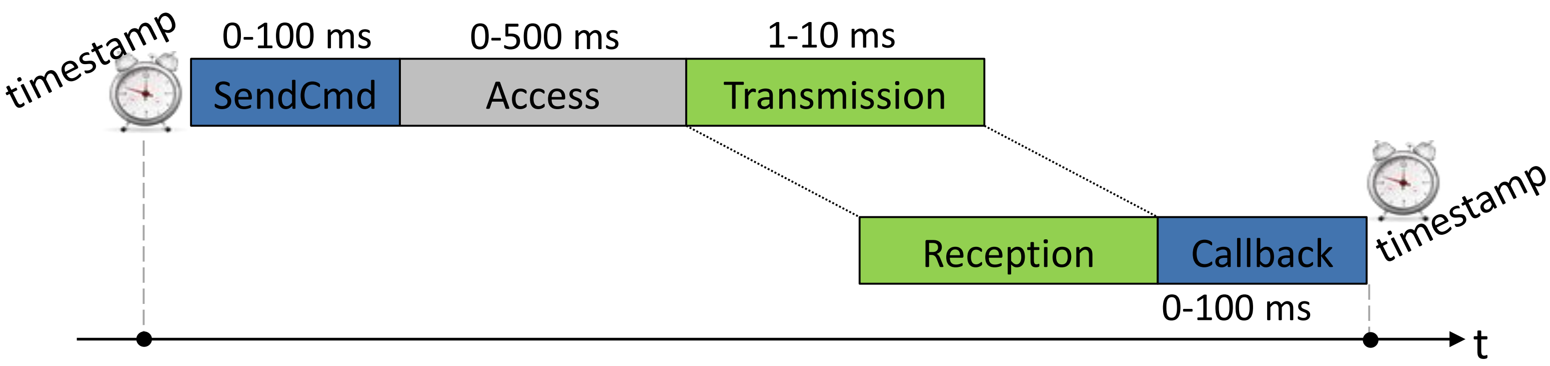
$$\delta = \frac{(t_4 - t_1) - (t_3 - t_2)}{2}$$

$$\Theta = \frac{(t_2 - (t_1 + \delta)) - (t_4 - (t_3 + \delta))}{2} = \frac{(t_2 - t_1) + (t_3 - t_4)}{2}$$

Messages Experience Jitter in the Delay

- Problem: Jitter in the message delay

  Various sources of errors (deterministic and non-deterministic)



- Solution: Timestamping packets at the MAC layer

  → Jitter in the message delay is reduced to a few clock ticks

Global vs. Local Time Synchronization

- Common time is essential for many applications:

Global

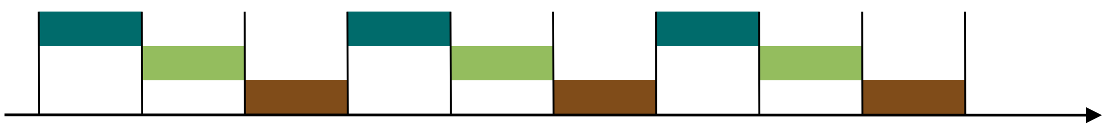
- Assigning a timestamp to a globally sensed event (e.g., earthquake)

Local

- Precise event localization (e.g., sensors networks, multiplayer games)

Local

- TDMA-based MAC layer in wireless networks



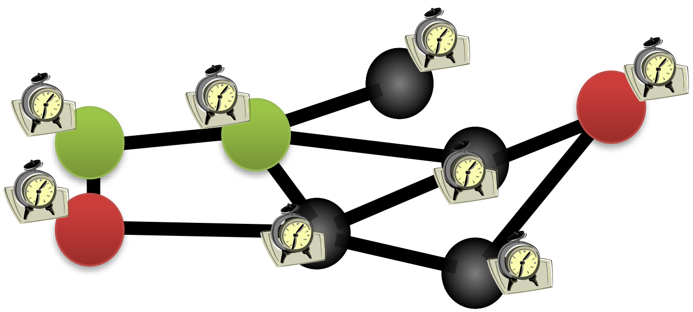
Local

- Coordination of wake-up and sleeping times (energy efficiency)

Theory of Clock Synchronization

- Given a communication network
  1. Each node equipped with hardware clock with drift
  2. Message delays with jitter

worst-case (but constant)



- Goal: Synchronize Clocks (“Logical Clocks”)
  - Both global and local synchronization!

Time Must Behave!

- Time (logical clocks) should
  not be allowed to stand still or jump

- Let’s be more careful (and ambitious):
- Logical clocks should always move forward
  - Sometimes faster, sometimes slower is OK.
  - But there should be a minimum and a maximum speed.
  - As close to correct time as possible!

Formal Model

- Hardware clock $H_v(t) = \int_0^t h_v(\tau)d\tau$ with clock rate $h_v(t) \in [1-\rho, 1+\rho]$

Clock drift ρ is typically small, e.g., $\rho \approx 10^{-4}$ for a cheap quartz oscillator

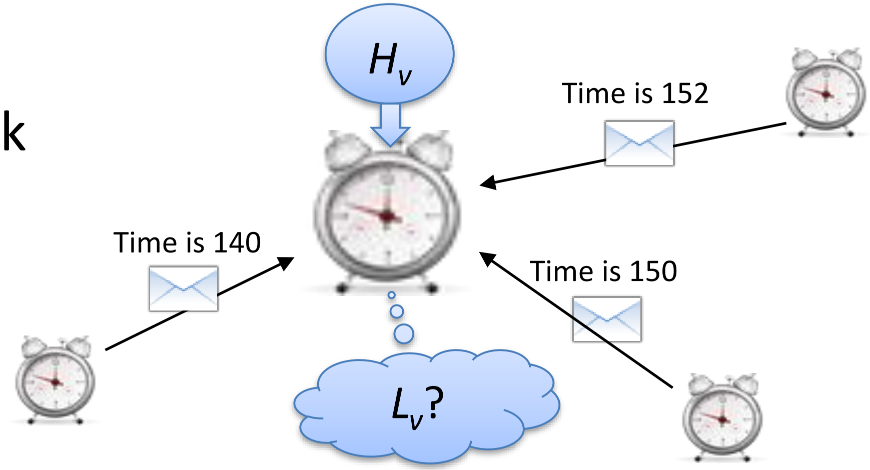
- Logical clock $L_v(t)$ which increases at rate at least $1-\rho$ and at most β

Logical clocks should run at least as fast as hardware clocks

- Message delays $\in [0,1]$

Neglect fixed part of delay, normalize jitter to 1

- Goal: a distributed synchronization algorithm to update the logical clock according to hardware clock and messages from neighbors



Global and Local Clock Skew

Clock Skew of a Clock Synchronization Algorithm

- Maximum possible difference between two clock values during an execution.

Global Skew

- Maximum possible clock skew between any two nodes in network

Local Skew

- Maximum possible clock skew between two neighbors

- Global and local skew are both important
- We will focus on global skew here
  - Because it is much easier to handle…

Synchronization Algorithm $\mathcal{A}^{\max}$

Task: How to update logical clocks based on msg. from neighbors

Idea: Minimize skew to the fastest neighbor

Algorithm $\mathcal{A}^{\max}$

- Set logical clock to the maximum clock value received from any neighbor (if larger than local logical clock value)
- If recv. value $>$ previously forwarded value, forward immediately
- at least forward local logical clock value once every T time steps
  - send out local logical clock value if hardware clock proceeds by $1-\rho$ since the last time the clock value was sent

Remark: Algorithm allows $\beta = \infty$
(clock values can jump to larger values)

Synchronization Algorithm $\mathcal{A}^{\max}$

Theorem: Alg. $\mathcal{A}^{\max}$ guarantees a global clock skew of at most

$$(1 + \rho) \cdot D + 2\rho \cdot T.$$

(global clock skew = max. diff. between two clock values, D: diameter)

Synchronization Algorithm $\mathcal{A}^{\max}$

Theorem: Alg. $\mathcal{A}^{\max}$ guarantees a global clock skew of at most

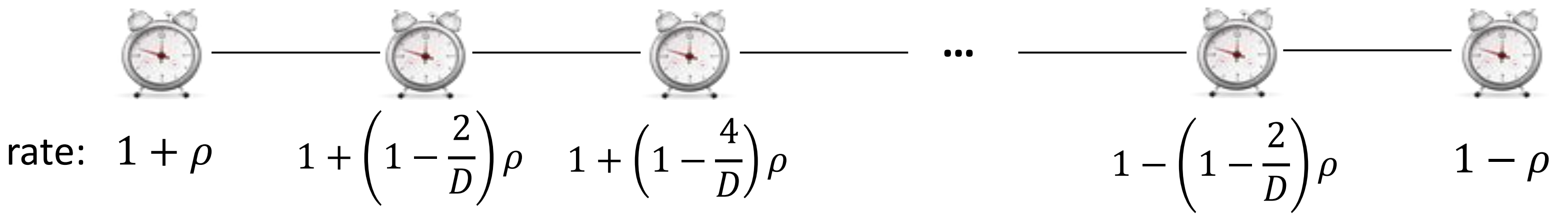
$$(1+\rho)\cdot D + 2\rho \cdot T.$$

(global clock skew = max. diff. between two clock values, D: diameter)

Synchronization Algorithm $\mathcal{A}^{\max}$

Global Skew can be D

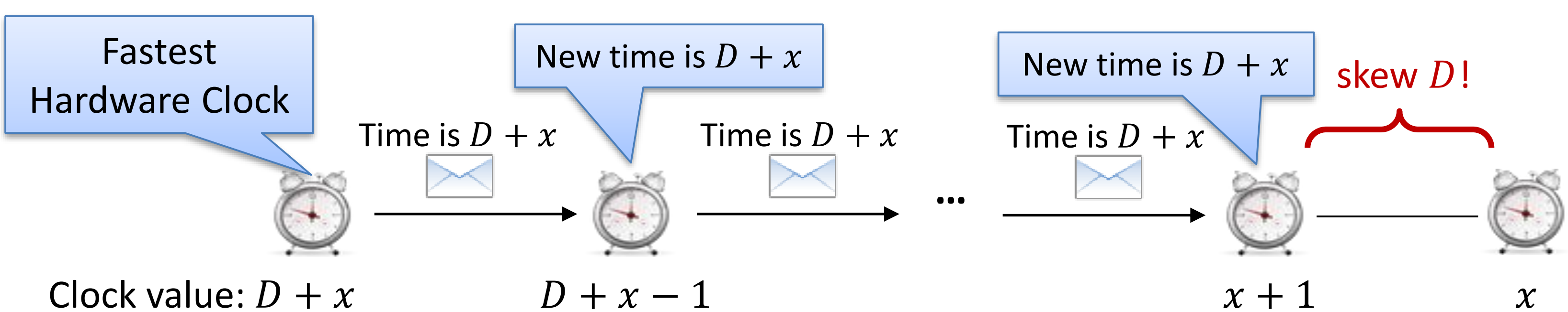
- path of length D, all message delays are 1



- skew between any 2 neighbors grows to 1 before detecting any skew

Local Skew can also be D...

- first all messages have delay 1 $\Longrightarrow$ skew D between ends of path
- then, messages become very fast (delay ≈ 0)



Synchronization Algorithm $\mathcal{A}^{\max}$

Problems

- Global and local skew can both be $\Theta(D)$
- Clock values can jump (i.e., $\beta = \infty$)

Can we do better?

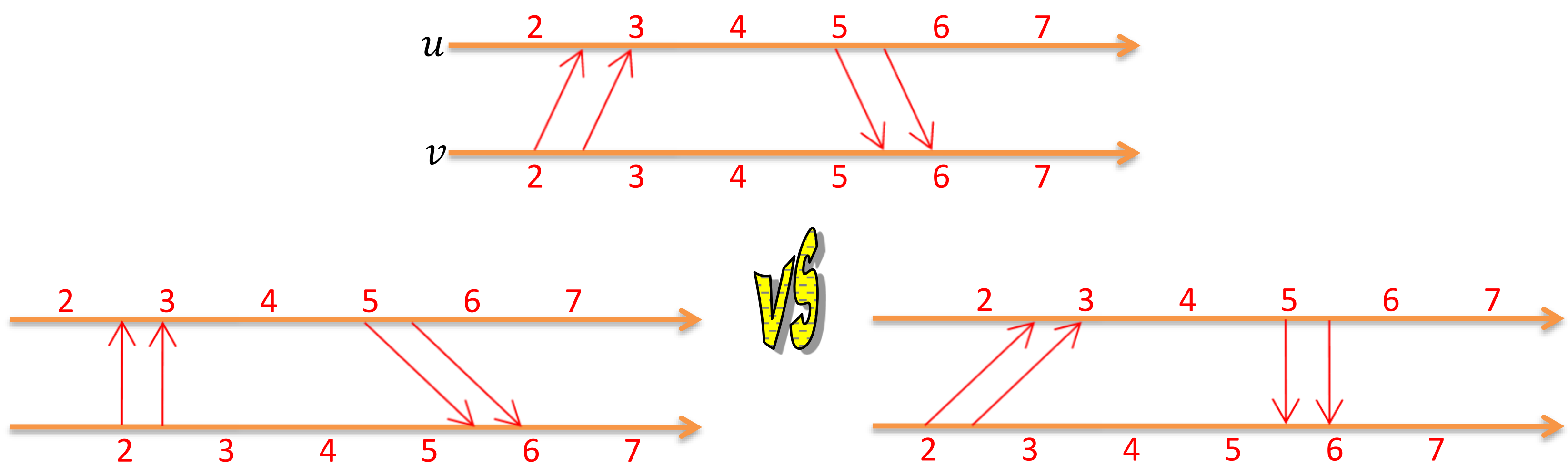
- We can make clocks continuous, any $\beta > 2\rho \cdot \frac{1+\rho}{1-\rho}$ works
  - Intuition: If a node u knows of a larger clock value, it sets its logical clock rate to $\frac{\beta}{1+\rho} \cdot h_u(t)$ to catch up $\Longrightarrow$ see exercises!
- Global skew cannot be improved $\Longrightarrow$ see next slides!
- Local skew can be improved, however
  - straightforward, simple ideas don't work [Locher et al., 2006]
  - somewhat surprisingly, $O(1)$ local skew is not possible [Fan et al., 2004]

Global Skew Lower Bound

Theorem: The global skew guarantee of any clock synchronization algorithm is at least $D/2$ (where D is the diameter of the network).

How to Enforce Clock Skew?

- Make messages fast in one direction and slow in the other dir.
- This allows to “hide” a constant amount of skew per edge

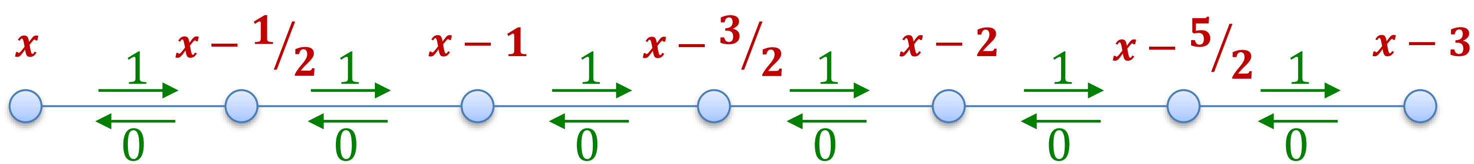


Global Skew Lower Bound

Theorem: The global skew guarantee of any clock synchronization algorithm is at least $D/2$ (where D is the diameter of the network).

Proof Idea:

- Assume that all hardware clocks run at rate 1 (no drift)
- Create two indistinguishable executions (causal shuffles):
  1. Initially: going from left two right, clock skew $-{}^{1}/_{2}$ between neighbors
     Message delays: left to right: 1, right to left: 0

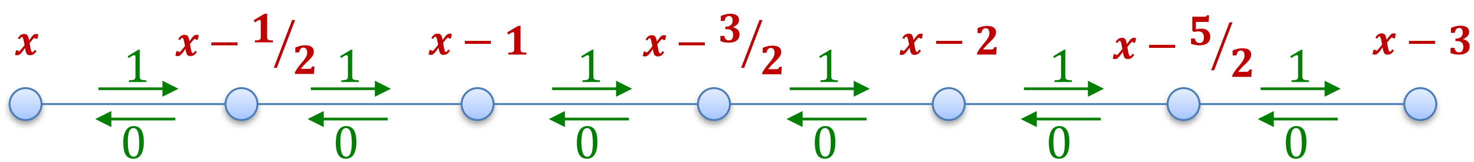


Global Skew Lower Bound

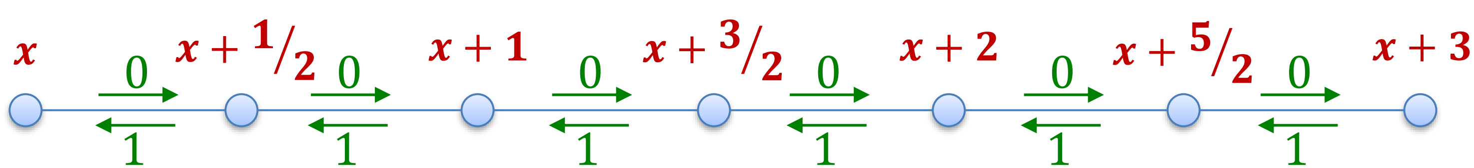
Theorem: The global skew guarantee of any clock synchronization algorithm is at least $D/2$ (where D is the diameter of the network).

Proof Idea:

- Create two indistinguishable executions (causal shuffles):
  1. Initially: going from left two right, clock skew $-{}^{1}/_{2}$ between neighbors
     Message delays: left to right: 1, right to left: 0



  2. Initially: going from left two right, clock skew $+{}^{1}/_{2}$ between neighbors
     Message delays: left to right: 0, right to left: 1



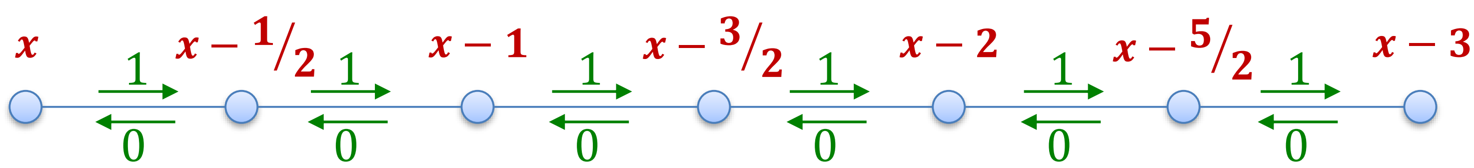
Global Skew Lower Bound

Theorem: The global skew guarantee of any clock synchronization algorithm is at least $D/2$ (where D is the diameter of the network).

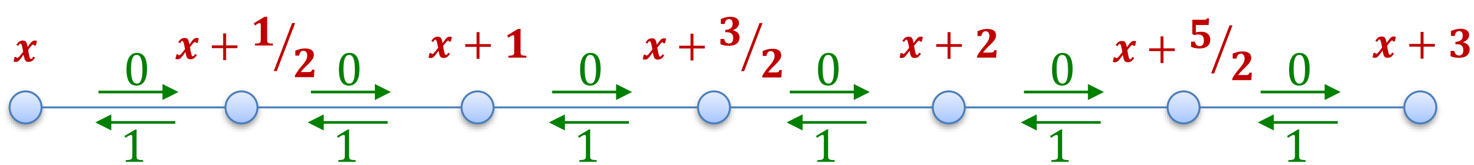
Proof Idea:

- Create two indistinguishable executions (causal shuffles):

1.



2.



- If in execution 1, $L_{v_R}(t) - L_{v_L}(t) = S$,
  in execution 2, we have $L_{v_R}(t) - L_{v_L}(t) = S + D$.