# Chapter 4
# Causality, Time, and Global States

## Distributed Systems

## SS 2019

no exercise tutorial on Wed, May 22

**Fabian Kuhn**

# Time in Distributed Systems

**Goal:** Establish a notion of time in (partially) asynchronous systems

**Physical time:**

- Establish an approximation of real time in a network
- Synchronize local clocks in a network
- 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.)

**Logical time:**

- Determine an order on the events in a distributed system
- Establish a global view on the system

# Logical Clocks

**Goal:** Assign a timestamp to all events in an asynchronous message-passing system

- Allows to give the nodes some notion of time
  - which can be used by algorithms
- **Logical clock values:** numerical values that increase over time and which are consistent with the observable behavior of the system
- The objective here is **not** to do **clock synchronization**:

  **Clock Synchronization:** compute logical clocks at all nodes which simulate real time and which are tightly synchronized.
  - We will briefly talk about clock synchronization later...

# 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 $v$:

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

## 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 $e'$

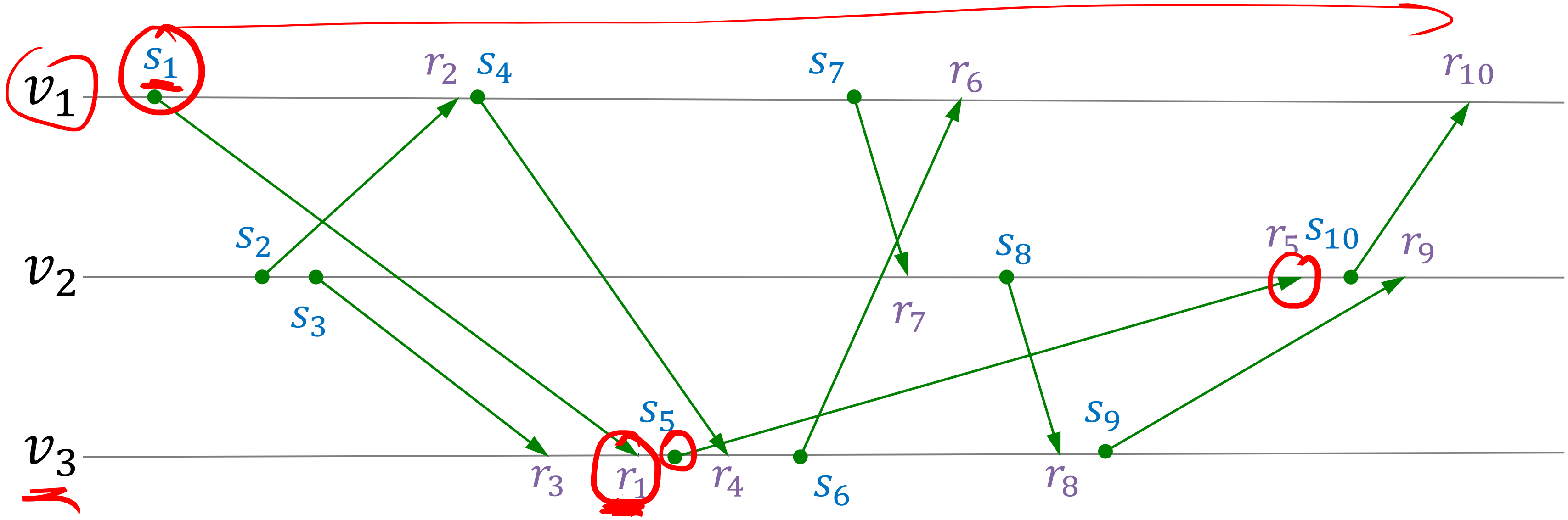
**For a given schedule $\boldsymbol{S}$:**

- 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 $\boldsymbol{e}$ provably occurs before $\boldsymbol{e'}$ if and only if $\boldsymbol{e}$ appears before $\boldsymbol{e'}$ in all causal shuffles of $\boldsymbol{S}$**

# Causal Shuffles / Causal Order Example

**Schedule $S$**



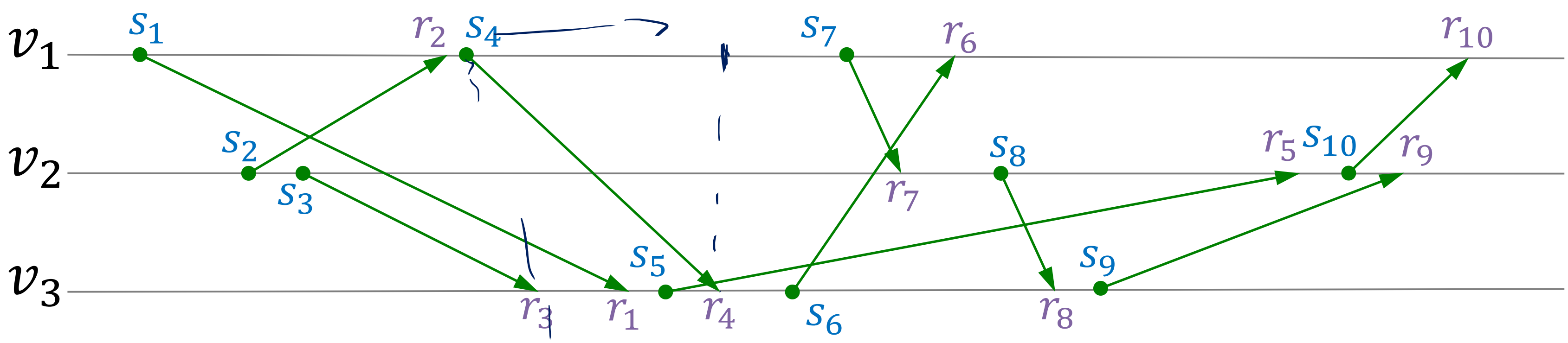

schedule $s_1, s_2, s_3, r_2, s_4, r_3, r_1, s_5$

$S|v_1 = s_1, r_2, s_4, s_7, r_6, r_{10}$

# Causal Shuffles / Causal Order Example

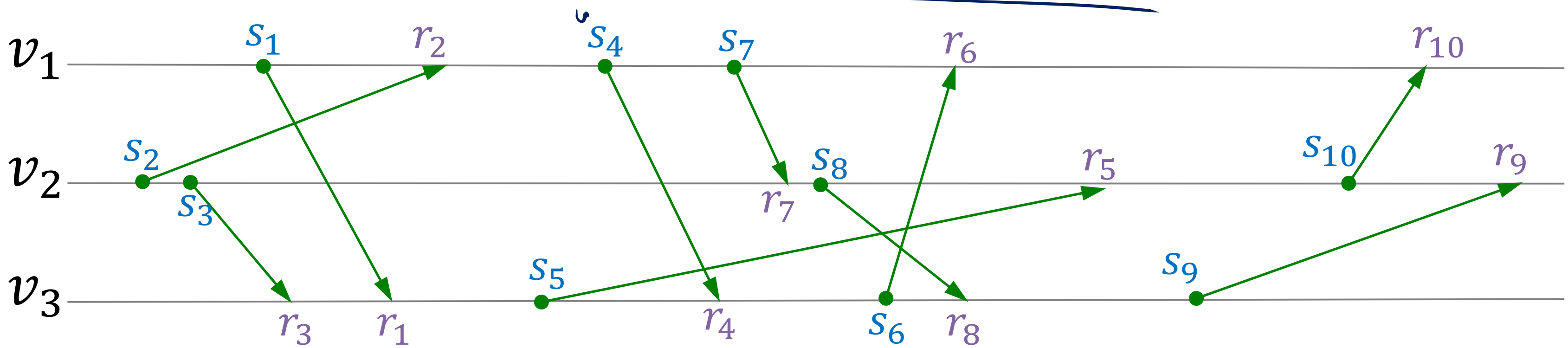
**Schedule $S$**

$s_1, s_2, s_3, r_2, s_4, s_3$




**Some Causal Shuffle $S'$**

$s_2, s_3, s_1, r_3, r_1, r_2, s_5$ ...



# Lamport's Happens-Before Relation

**Assumption:** message passing system, only send and receive events

**Consider two events $e$ and $e'$ occurring at nodes $u$ and $u'$**

– send event occurs at sending node, recv. event at receiving node

– let's define $t$ and $t'$ be the (real) times when $e$ and $e'$ occur

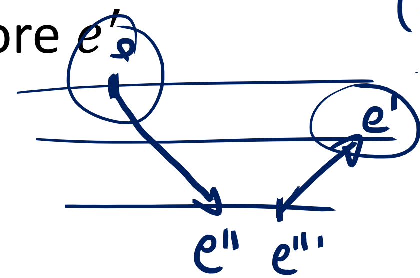
**We know that $e$ provably occurs before $e'$ if**

1. The events occur at the same node and $e$ occurs before $e'$

2. Event $e$ is a send event, $e'$ the recv. event of the same message

3. There is an event $e''$ for which we know that provably, $e$ occurs before $e''$ and $e''$ occurs before $e'$ (transitivity)

$e \to e'' \to e'$



# 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

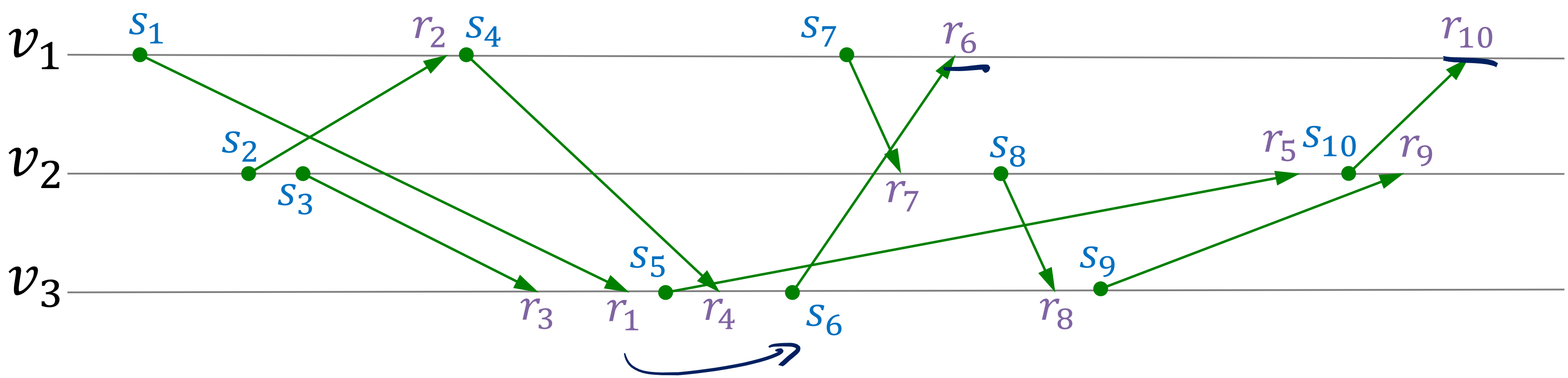
1. All pairs $(e, e')$ where $e$ precedes $e'$ in $S$ and $e$ and $e'$ are events of the same node/process.
2. All pairs $(e, e')$ where $e$ is a send event and $e'$ the receive event for the same message.
3. All pairs $(e, e')$ where there is a third event $e''$ such that

$$e \Rightarrow_S e'' \quad \land \quad e'' \Rightarrow_S e'$$

   – Hence, we take the **transitive closure** of the relation defined by 1. and 2.

# Happens-Before Relation: Example

**Schedule $S$**




$\forall i: \; s_i \Rightarrow_S r_i$

$s_1 \Rightarrow_S r_1, \; r_1 \Rightarrow_S s_6, \; s_6 \Rightarrow_S r_6, \; r_6 \Rightarrow r_{10}$

$\Downarrow$

$s_1 \Rightarrow_S r_{10}$

# Happens-Before and Causal Shuffles

**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 $S$.

$\neg(e \Rightarrow_S e') \wedge \neg(e' \Rightarrow_S e)$

**Some remarks before proving the theorem…**

- Shows that the happens-before relation is exactly capturing what we need about the causality between events
  - It captures exactly what is observable about the order of events

- To prove the theorem, we show that
  1. a) $\longrightarrow$ b)
  2. b) $\longrightarrow$ a)

Remark:

$\Rightarrow_S$ is a partial order

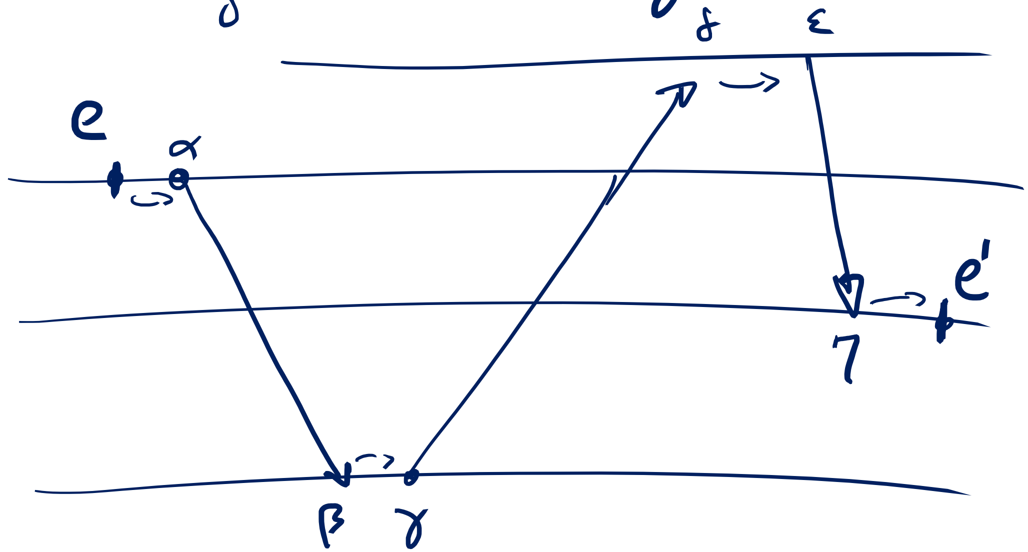
# Happens-Before and Causal Shuffles

**If $e \Rightarrow_S e'$, then $e$ precedes $e'$ in all causal shuffles $S'$ of $S$.**

- 1) $e, e'$ occur at the same node ✓
- 2) $e, e'$ belong to the same msg. ($e$: send, $e'$: receive) ✓

3)

S:




$e \Rightarrow_S e'$

take shortest chain $\underbrace{e \Rightarrow_S e_1 \Rightarrow_S e_2}\ \underbrace{\dots \Rightarrow_S e_k} \Rightarrow_S e'$

$\underbrace{e \Rightarrow_S e''} \Rightarrow_S e'$

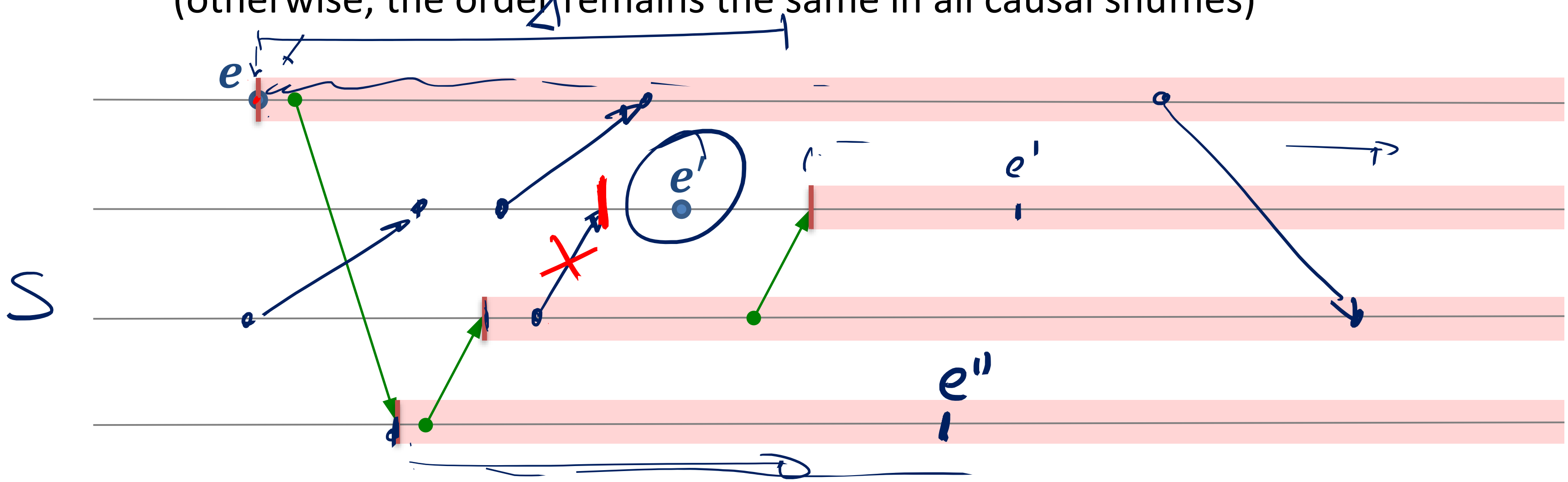
# Happens-Before and Causal Shuffles

**If $e$ precedes $e'$ in all causal shuffles $S'$ of $S$, then $e \Rightarrow_S e'$.**

$A \rightarrow B \quad \neg B \rightarrow \neg A$

**Proof:**

- Show: $e \not\Rightarrow_S e'$, there is a shuffle $S'$ such that $e'$ precedes $e$ in $S$
- W.l.o.g., assume that $e$ precedes $e'$ in $S$
  - Consequently, $e$ and $e'$ happen at different nodes (otherwise, the order remains the same in all causal shuffles)



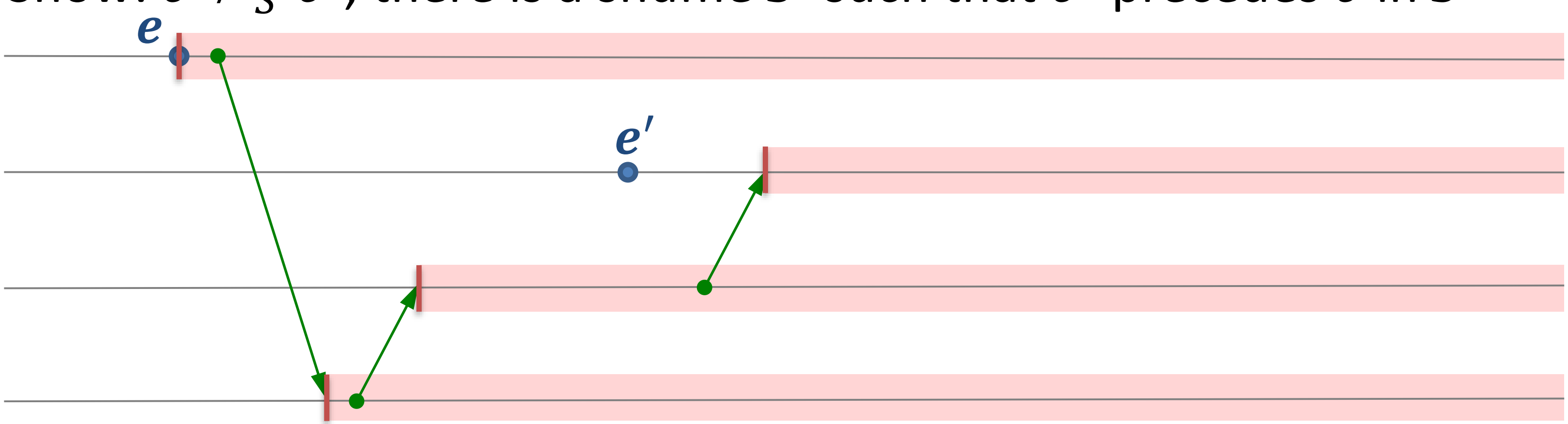

- **Events in red part can be shifted by fixed amount Δ**

# Happens-Before and Causal Shuffles

**If $e$ precedes $e'$ in all causal shuffles $S'$ of $S$, then $e \Rightarrow_S e'$.**

**Proof:**

- Show: $e \nRightarrow_S e'$, there is a shuffle $S'$ such that $e'$ precedes $e$ in $S$




- **Events in red part can be shifted by fixed amount $\Delta$**
  - Consider some message $M$ with send/receive events $s_M, r_M$
  - If $s_M$ and $r_M$ or only $r_M$ are shifted, message delay gets larger → OK
  - It is not possible to only shift $s_M$
  - Choose $\Delta$ large enough to move $e$ past $e'$

# 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 $\Rightarrow_S$

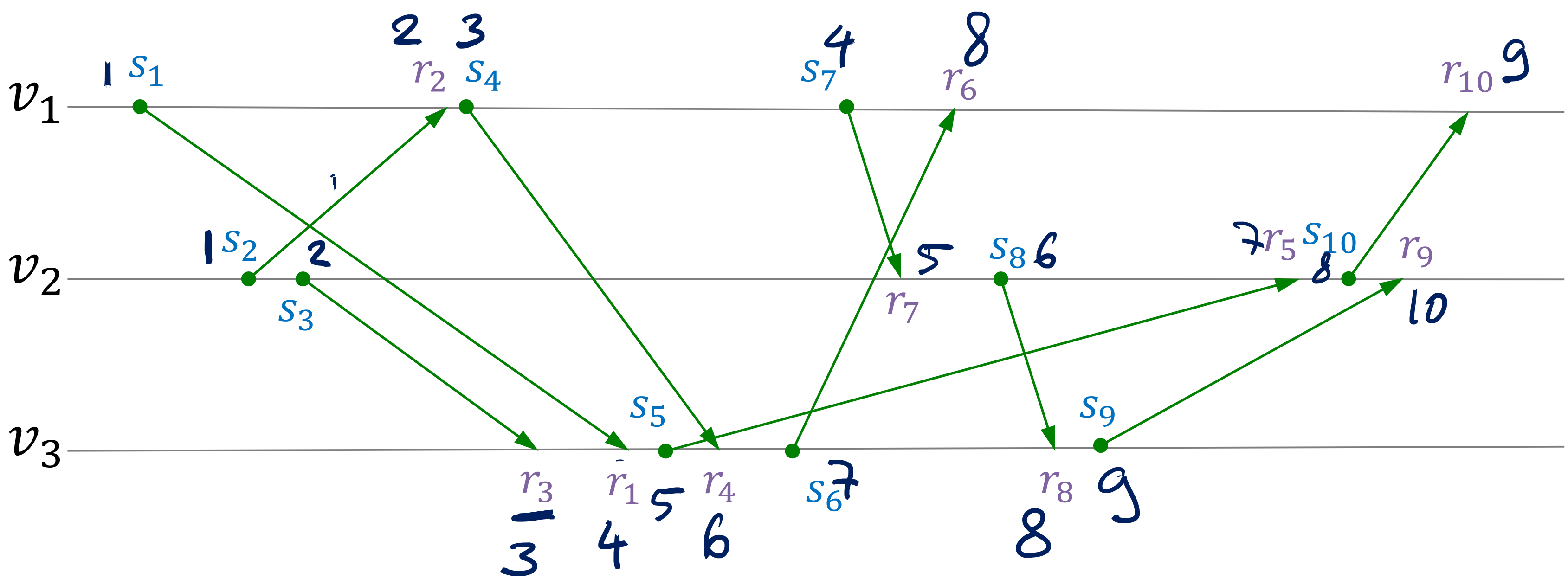
# Lamport Clocks

**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
$$c_u := c_u + 1;\ \tau(e) := c_u$$
- 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
$$c_u := \max\{c_u, \tau(s)\} + 1;\ \ \tau(r) := c_u$$

# Lamport Clocks: Example

**Schedule $S$**



# Neiger-Toueg-Welch Clocks

**Discussion Lamport Clocks:**

- Advantage: no changes in the behavior of the underlying protocol
- Disadvantage: clocks might make huge jumps (when recv. a msg.)

**Idea by Neiger, Toueg, and Welch:**

- Assume nodes have some approximate knowledge of real time
  - e.g., by using a clock synchronization algorithm
- Nodes increase their clock value periodically
- Combine with Lamport clock ideas to ensure safety
- When receiving a message with a time stamp which is larger than the current local clock value, wait with processing the message.

# Fidge-Mattern Vector Clocks

- Lamport clocks give a superset of the happens-before relation
- Can we compute logical clocks to get $\Rightarrow_S$ exactly?

**Vector Clocks:**

- Each node $u$ maintains an vector $\mathrm{VC}(u)$ *of* clock values
  - one entry $\mathrm{VC}_v(u)$ for each node $v \in V$
- In the vector $\mathrm{VC}(e)$ assigned (by $u$) to some event $e$ happening at node $u$, the component $x_v$ corresponding to $v \in V$ refers to the

  **number of events at node $v$, $u$ knows about when $e$ occurs**

# Vector Clocks Algorithm

- All Nodes $u$ keep a vector $\mathrm{VC}(u)$ with an entry for all nodes in $V$
  - all components are initialized to $0$
  - component corresponding to node $v$: $\mathrm{VC}_v(u)$
- For any non-receive event $e$ at node $u$, node $u$ computes

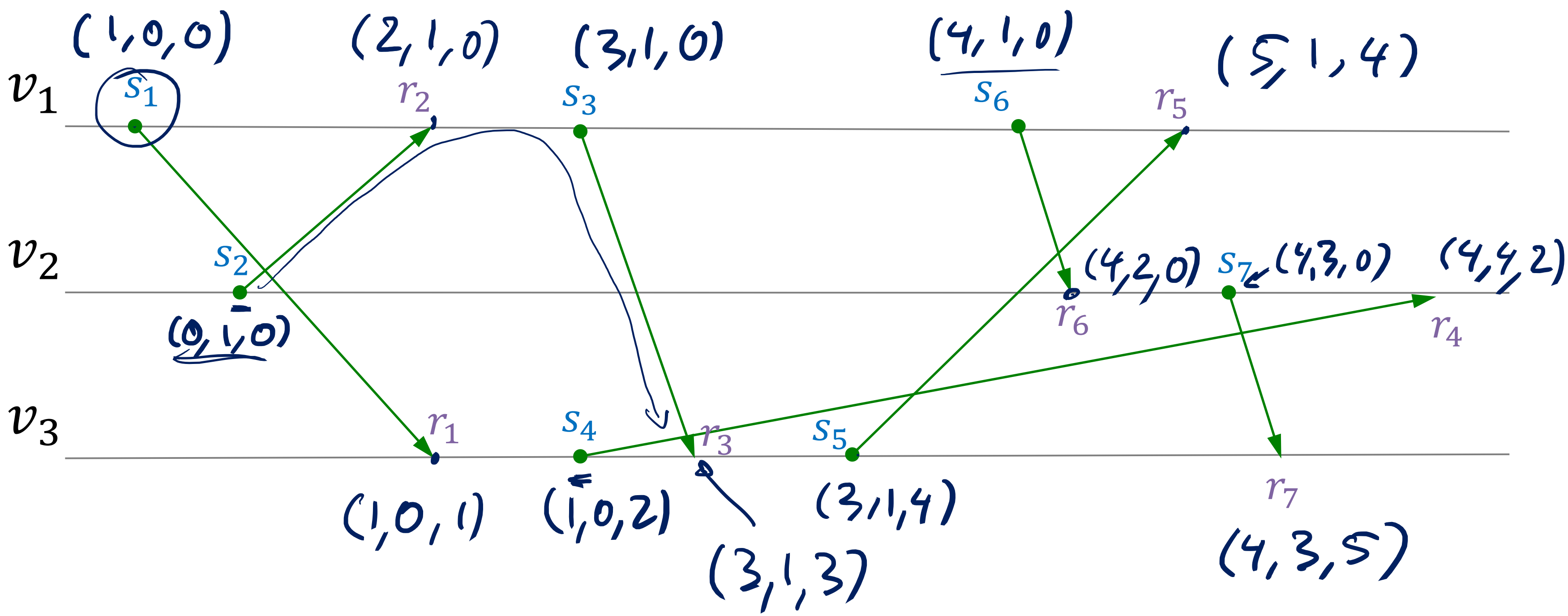
$$\mathrm{VC}_u(u) \coloneqq \mathrm{VC}_u(u) + 1; \quad \mathrm{VC}(e) \coloneqq \mathrm{VC}(u)$$

- For any send event $s$ at node $u$, node $u$ attaches the value of $\mathrm{VC}(s)$ to the message
- For any receive event $r$ at node $u$ (with corresponding send event $s$), node $u$ computes

$$\begin{aligned}
&\forall v \neq u: \mathrm{VC}_v(u) \coloneqq \max\{\mathrm{VC}_v(s), \mathrm{VC}_v(u)\}; \\
&\mathrm{VC}_u(u) \coloneqq \mathrm{VC}_u(u) + 1; \\
&\mathrm{VC}(r) \coloneqq \mathrm{VC}(u)
\end{aligned}$$

# Vector Clocks Example

**Schedule $S$**



# Vector Clocks and Happens-Before

$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} < \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$

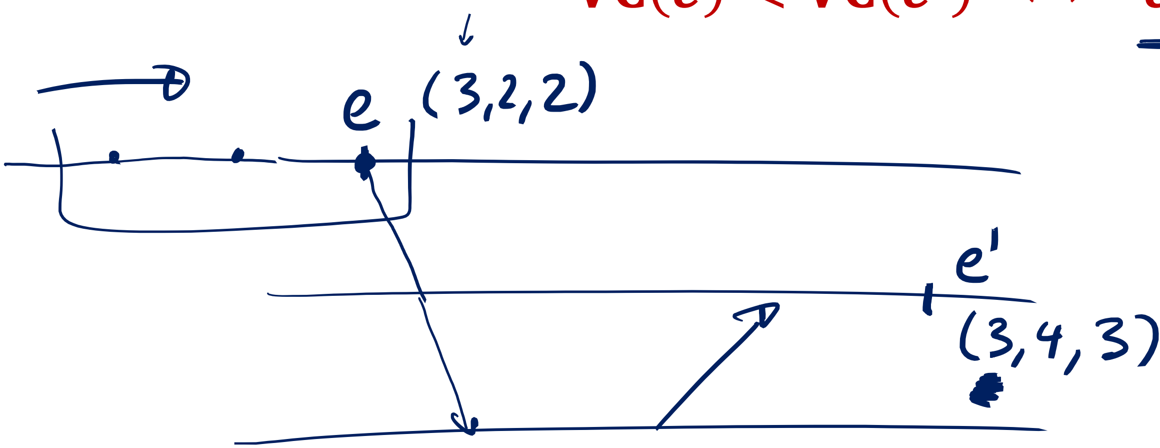
$x_1 \leq y_1 \land x_2 \leq y_2 \land x_3 \leq y_3$

**Definition:** $\mathbf{VC}(e) < \mathbf{VC}(e') :=$

$$\left(\forall v \in V: \mathbf{VC}_v(e) \leq \mathbf{VC}_v(e')\right) \land \left(\mathbf{VC}(e) \neq \mathbf{VC}(e')\right)$$

**Theorem:** Given a schedule $S$, for any two events $e$ and $e'$,

$$\mathbf{VC}(e) < \mathbf{VC}(e') \quad \longleftrightarrow \quad e \Rightarrow_s e'$$



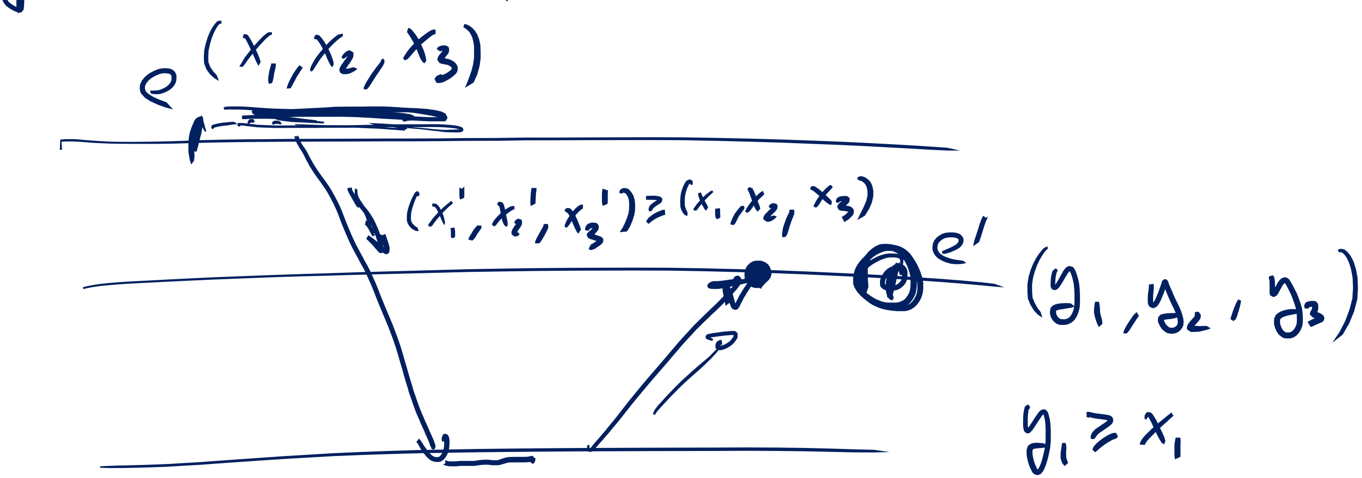
# Vector Clocks and Happens-Before

**Definition:** $\mathbf{VC}(e) < \mathbf{VC}(e') :=$

$$\big(\forall v \in V: \mathbf{VC}_v(e) \leq \mathbf{VC}_v(e')\big) \land \big(\mathbf{VC}(e) \neq \mathbf{VC}(e')\big)$$

**Theorem:** Given a schedule $S$, for any two events $e$ and $e'$,

$$\mathbf{VC}(e) < \mathbf{VC}(e') \quad \longleftrightarrow \quad e \Rightarrow_S e'$$




$y_1 \geq x_1$

$y_2 \geq x_2 \quad \rightarrow \quad \underline{y_2} > x_2$

$y_3 \geq x_3$

# Logical Clocks vs. Synchronizers

**Synchronizer:**

- Algorithm that generates clock pulses that allow to run an synchronous algorithm in an asynchronous network
  - We will discuss synchronizers later

**The clock pulses (local round numbers) generated by a synchronizer can also used as logical clocks**

- Send events of round $r$ get clock value $2r - 1$
- Receive events of round $r$ get clock value $2r$

- superset of the happens-before relation
- requires to drastically change the protocol and its behavior
  - synchronizer determines when messages can be sent
- a very heavy-weight method to get logical clock values
  - requires a lot of messages

# Application of Logical Times

**Replicated State Machine**

- main application suggested by Lamport in his original paper
- a shared state machine where every node can issue operations
- state machine is simulated by several replicas

**Solution:**

- add current clock value (and issuer node ID) to every operation
- operations have to be carried out in order of clock values / IDs
- ***Safety:***
  - all replicas use same order of operations
  - order of operations is a possible actual order (consistent with local views)
- ***Liveness:***
  - progress is guaranteed if nodes regularly send messages to each other

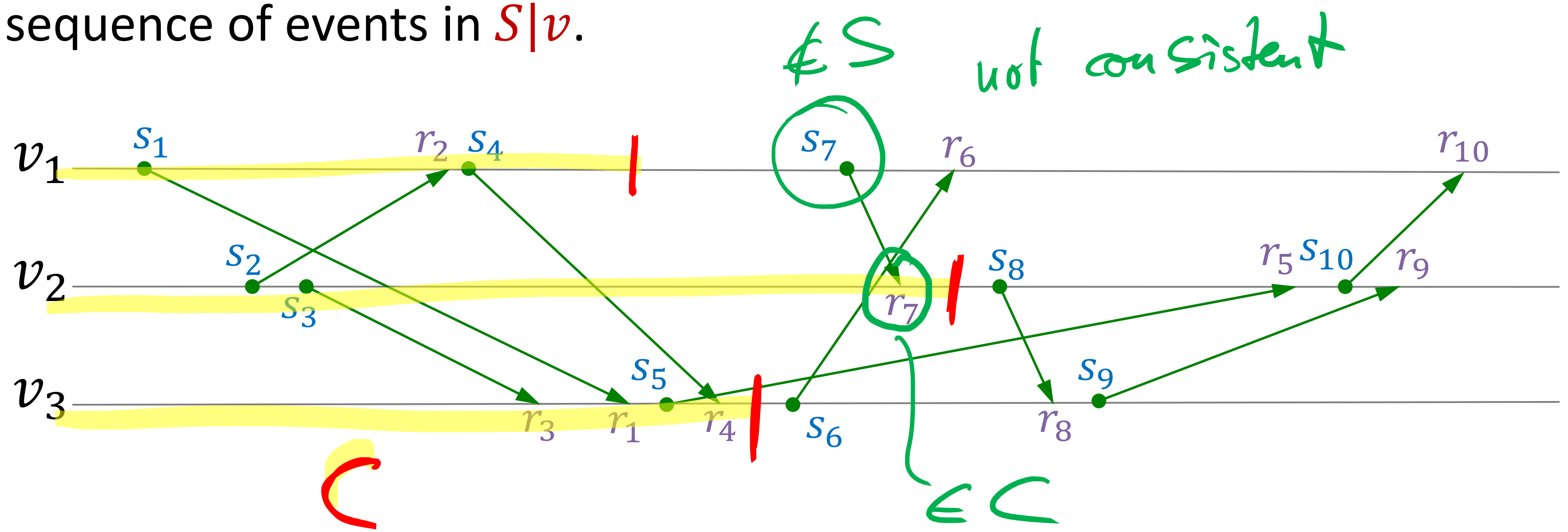
# Global States

- Sometimes the nodes of a distributed system need to figure out the global state of the system
  – e.g., to find out if some property about the system state is true
- Executions/schedules which lead to the same happens-before relation (i.e., causal shifts) cannot be distinguished by the system.
- Generally not possible to record the global state at any given time of the execution
- Best solution: A global state which is consistent with all local views
  – i.e., a state which could have been true at some time
- Called a **consistent** or **global snapshot** of the system and based on **consistent cuts** of the schedule

# 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 $S|v$.

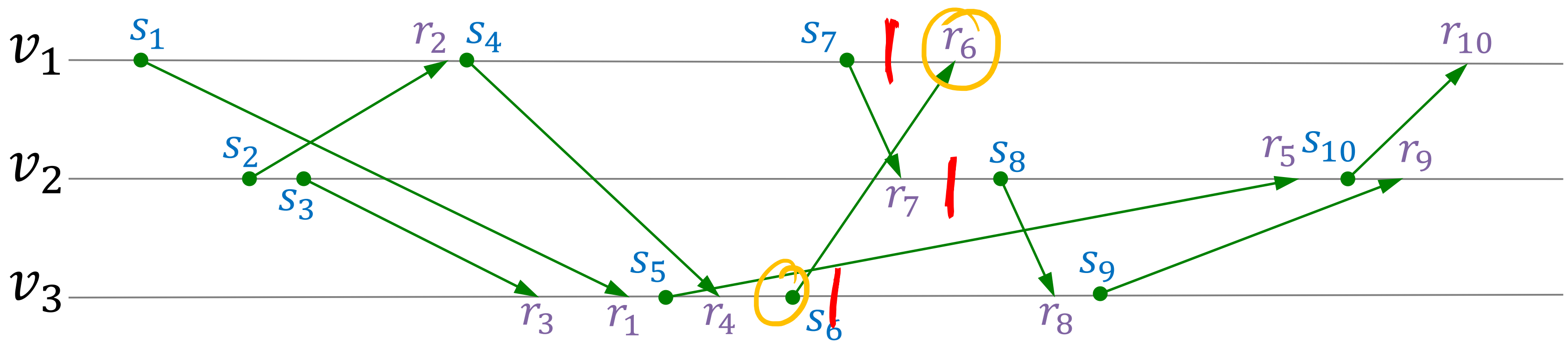


# Consistent Cut

## Consistent Cut

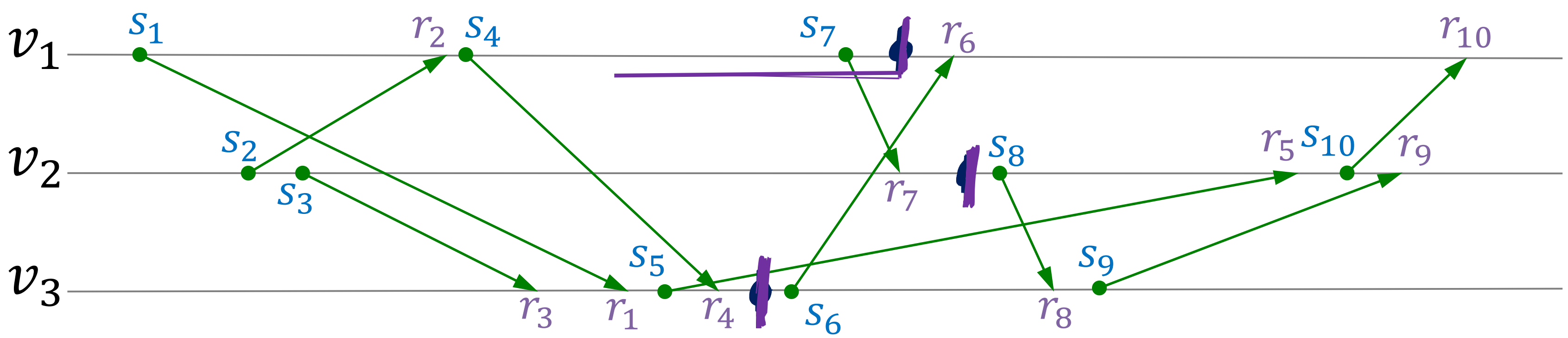
Given a schedule $S$, a consistent cut $C$ is cut such that for all events $e \in C$ and all events $f$ in $S$, it holds that

$$f \Rightarrow_S e \;\longrightarrow\; f \in C$$

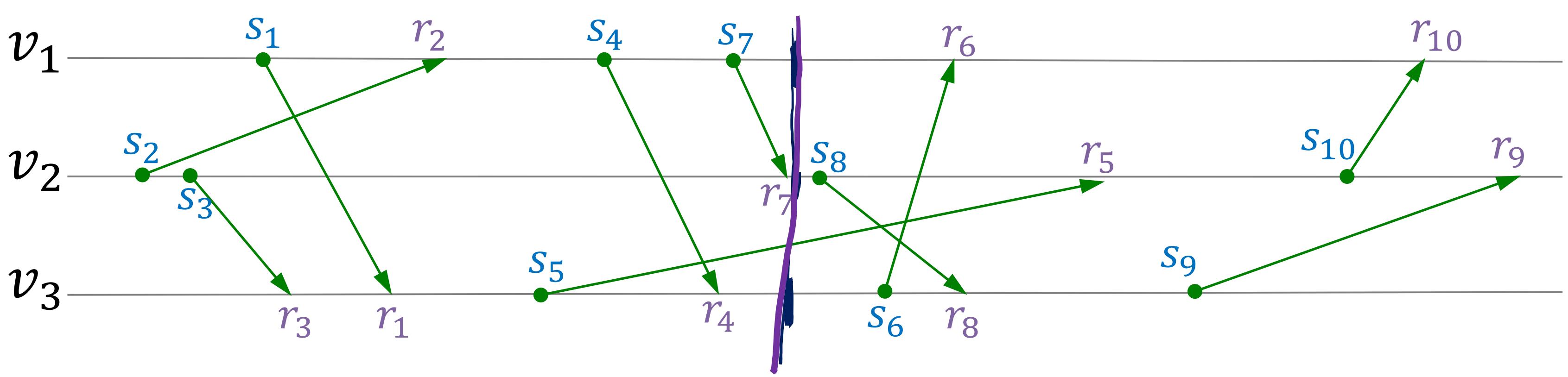


# Consistent Cut

### 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 $s_M \in C$.

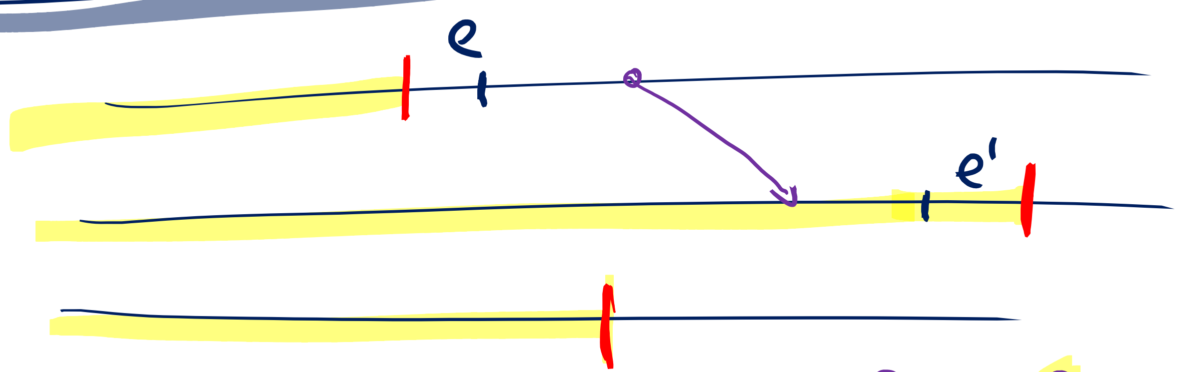
$\forall M: \; r_M \in C \rightarrow s_M \in C \iff C$ consistent

✓

def. of consistent cuts

other dir. $(\longrightarrow)$

assume $C$ not consistent:




$e \Rightarrow_s e'$

$e \Rightarrow_s e_1 \Rightarrow_s e_2 \Rightarrow_s e_3 \cdots \Rightarrow_s e'$

# 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)