Chapter 4
Causality, Logical Time, and Global States

Distributed Systems

SS 2015

Fabian Kuhn

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.
  - Might be the topic of a later chapter...

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: happen at 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'

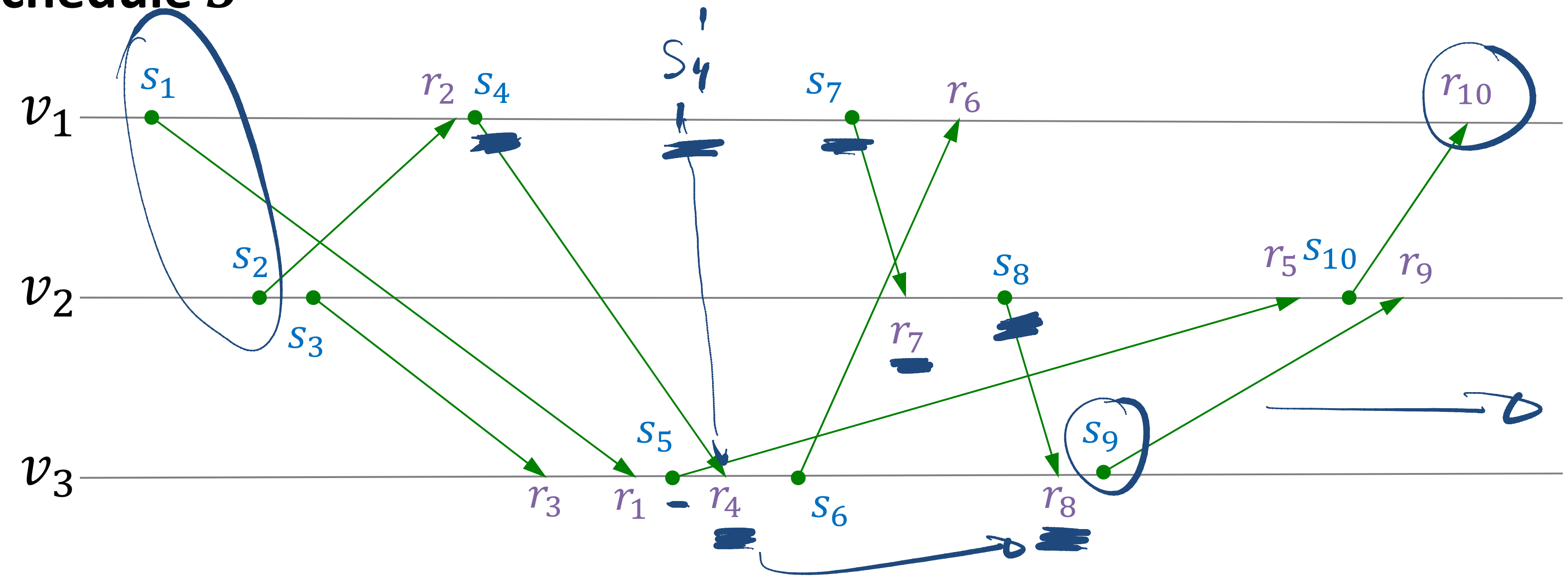
For a given schedule S:

- 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

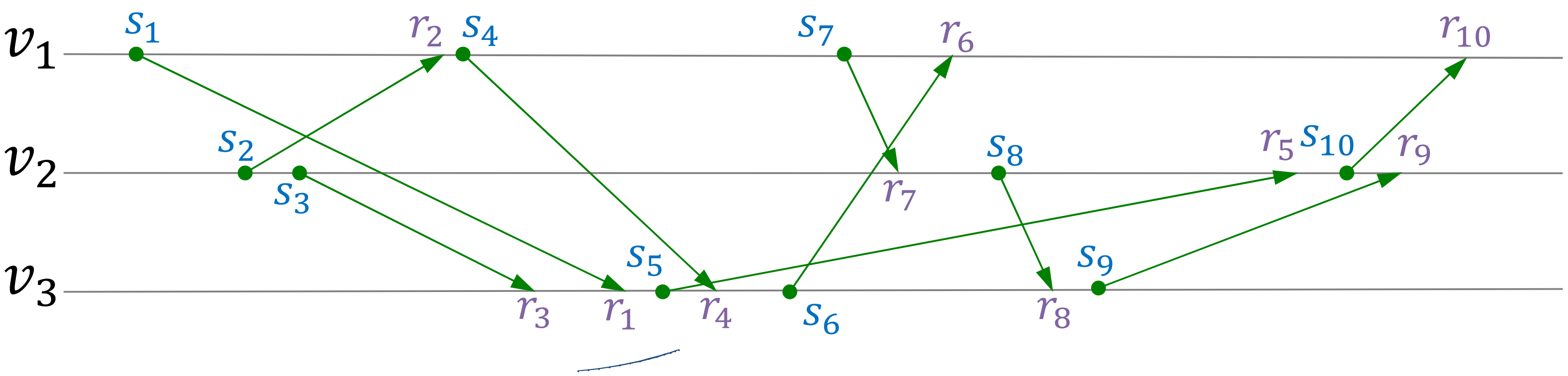
Causal Shuffles / Causal Order Example

Schedule S

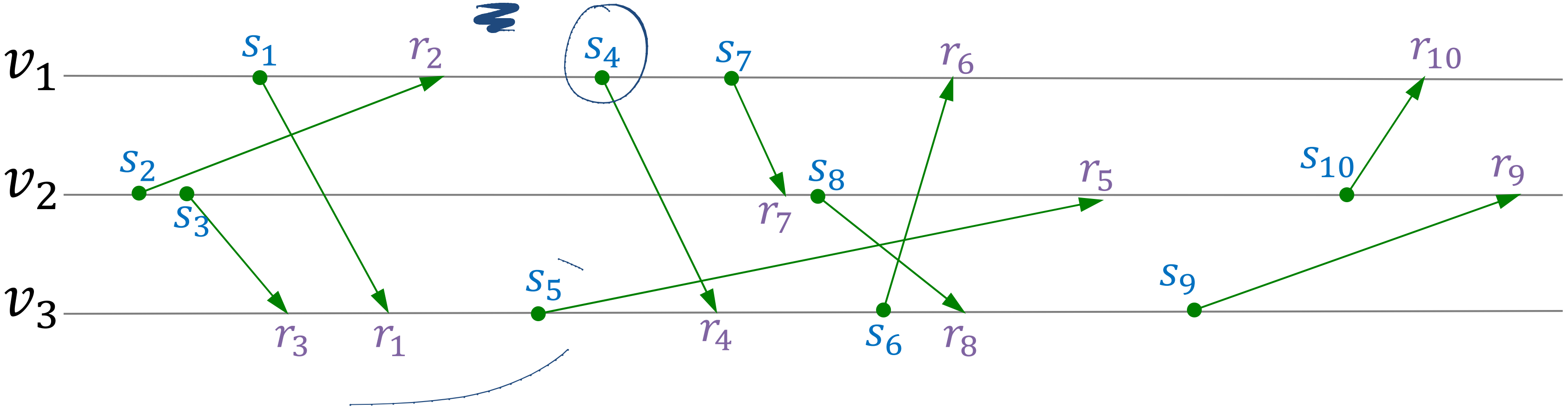


Causal Shuffles / Causal Order Example

Schedule S



Some Causal Shuffle S'



Lamport's Happens-Before Relation

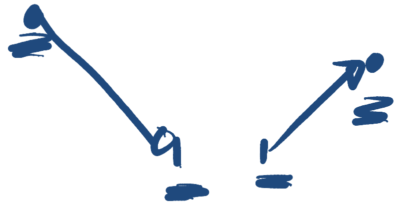
Assumption: message passing system, only send and receive events

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

in schedule S

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'

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.

$e \Rightarrow_S e'$

2. All pairs (e, e') where e is a send event and e' the receive event for the same message.

$e \Rightarrow_S e'$

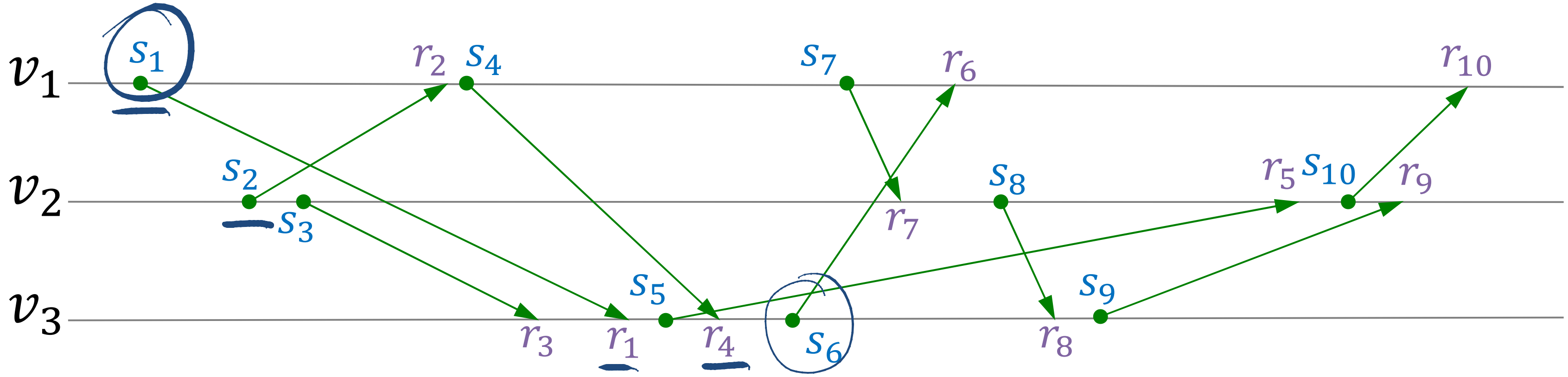
3. All pairs (e, e') where there is a third event e'' such that

$$e \Rightarrow_S e'' \quad \wedge \quad e'' \Rightarrow_S e'$$

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

Happens-Before Relation: Example

Schedule S



$s_1 \Rightarrow_S s_7$ $\quad$ $s_1 \Rightarrow_S r_2$

$\forall i: s_i \Rightarrow_S r_i$ $\quad$ $s_1 \Rightarrow_S s_6$

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. ← e provably occurs before 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) ⟶ b)
  2. b) ⟶ a)

Remark:
$\Rightarrow_S$ is a partial order

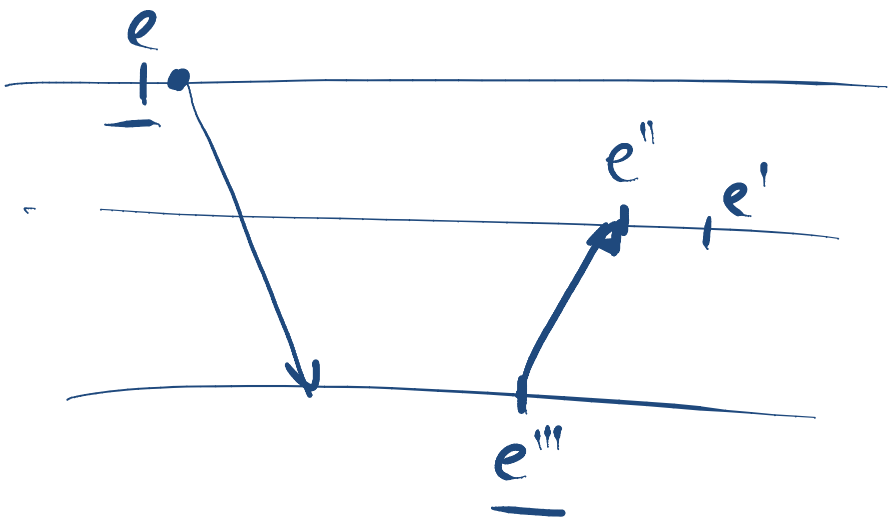
Happens-Before and Causal Shuffles

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': recv.

3)



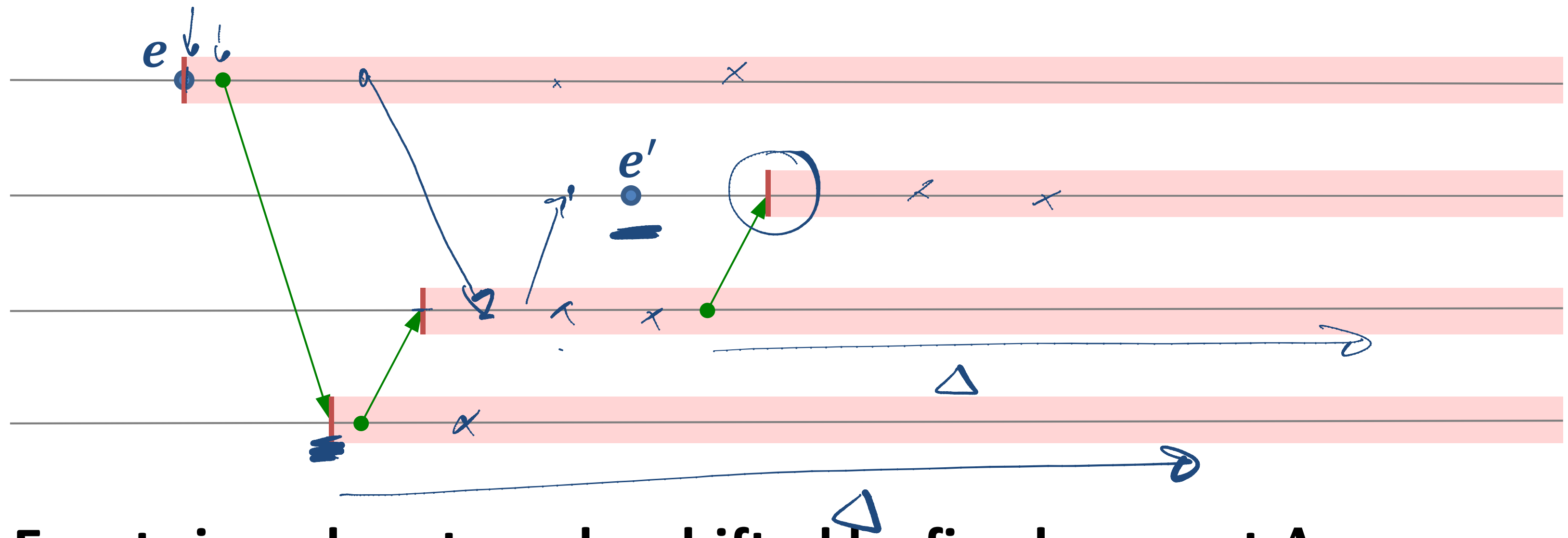
proof by induction on # msg. in chain

Happens-Before and Causal Shuffles

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

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 (else: $S' = S$)
  - Consequently, e and e' happen at different nodes (otherwise, the order remains the same in all causal shuffles)



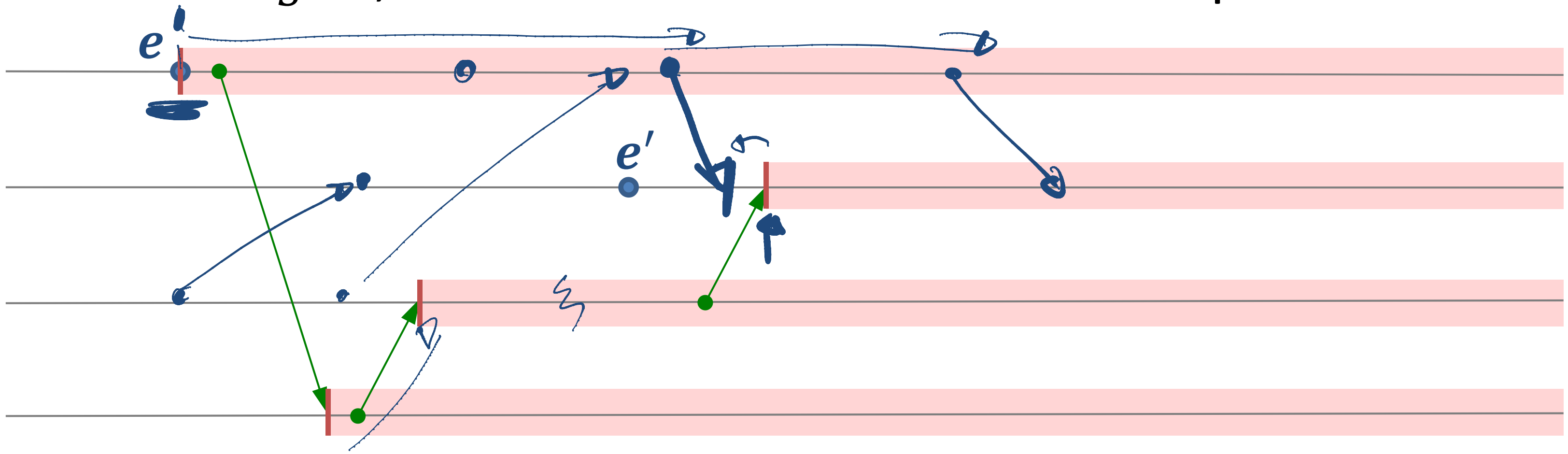
- **Events in red part can be shifted by fixed amount Δ**

Happens-Before and Causal Shuffles

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

Proof:

- Show: $e \not\Rightarrow_S e'$, there is a shuffle S' such that e' precedes e in S



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

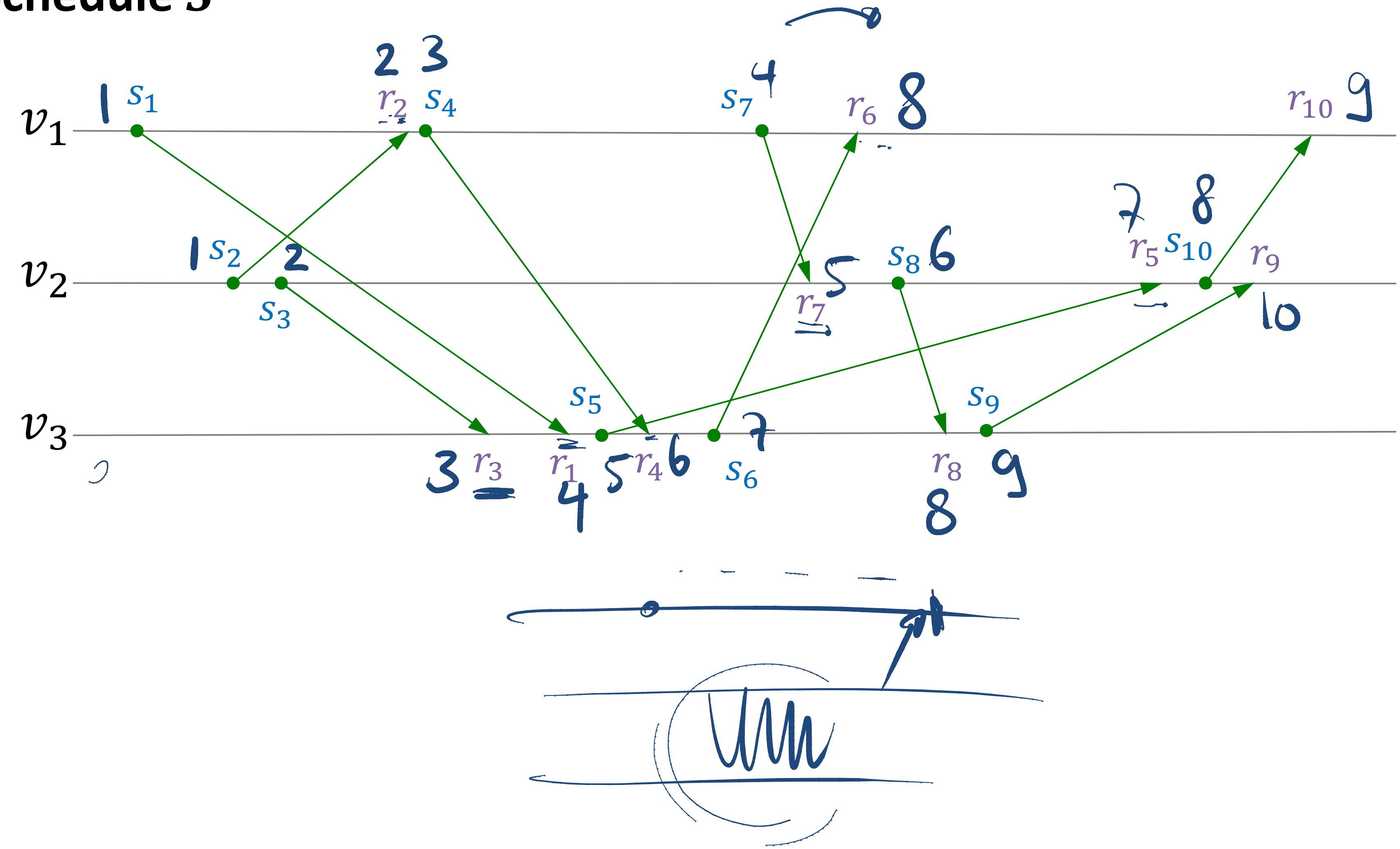
$$c_u := c_u + 1;\ \tau(e) := c_u$$

- 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?

$VC(u)$

$$\begin{array}{c} v_1 \\ v_2 \\ \vdots \\ v_n \end{array} \begin{pmatrix} \cdot \\ \cdot \\ \vdots \\ \cdot \end{pmatrix}$$

$VC(e)$

$VC_v(u)$

Vector Clocks:

- Each node u maintains an vector of clock values
  - one value for each node $v \in V$
- In the vector of node u assigned to some event e happening at node u, the component x_v corresponding to $v \in V$ refers to the

$VC_v(u)$ $VC_v(e)$

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

$$VC_v(e) = |\{\text{events } e' \text{ at } v : e' \Rightarrow_S e\}|$$

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: $\mathrm{VC}_v(u)$

(send)

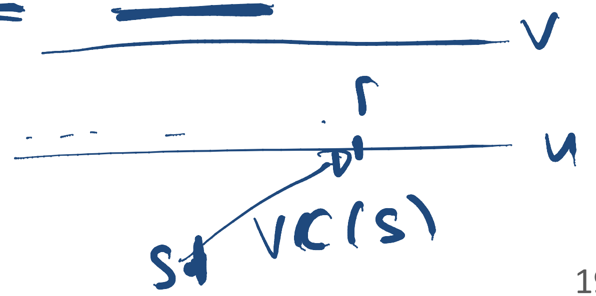
- For any non-receive event e at node u, node u computes

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

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

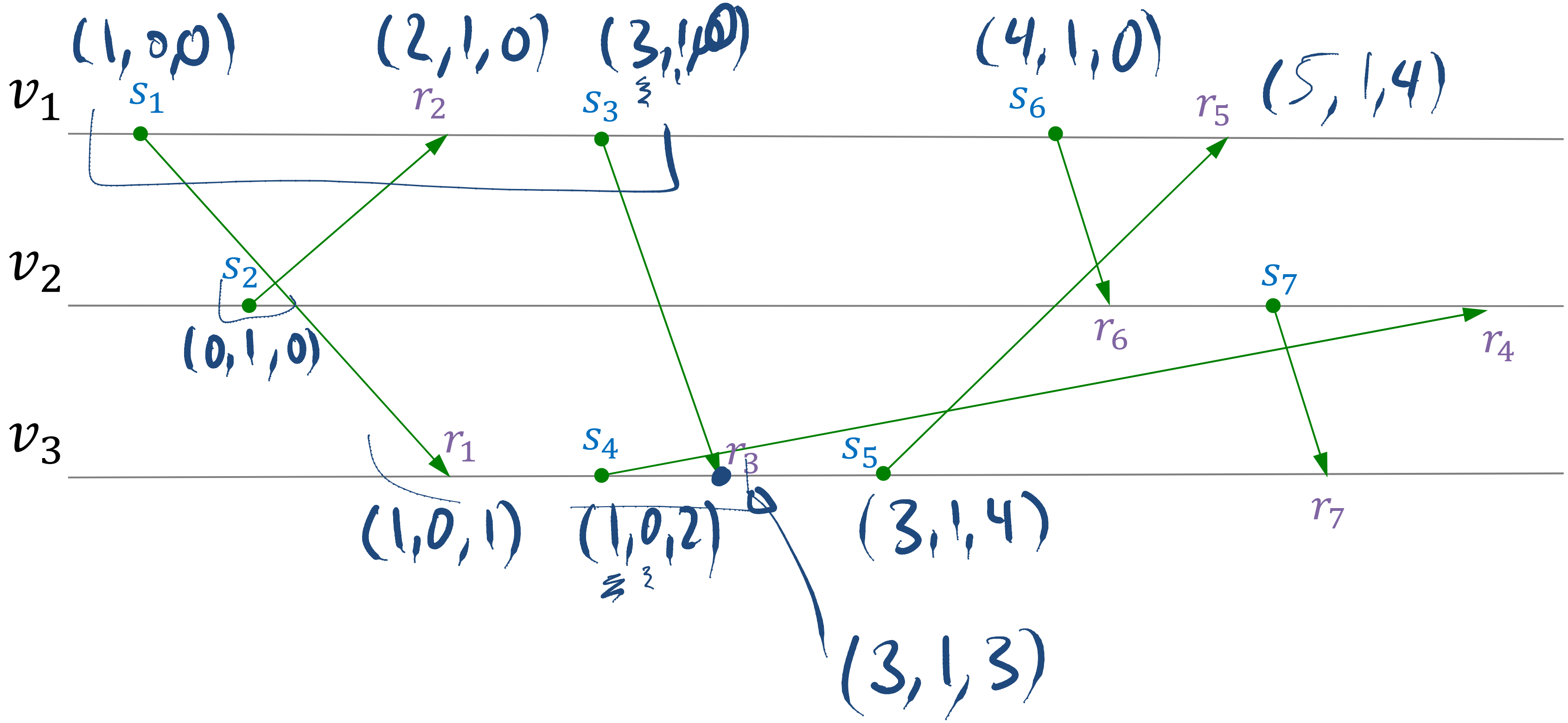
- For any receive event r at node u (with corresponding send event s), node u computes

$$\forall v \neq u\colon \mathrm{VC}_v(u) \coloneqq \max\{\mathrm{VC}_v(s), \mathrm{VC}_v(u)\};$$
$$\mathrm{VC}_u(u) \coloneqq \mathrm{VC}_u(u) + 1;$$
$$\mathrm{VC}(e) \coloneqq \mathrm{VC}(u)$$



Vector Clocks Example

Schedule S



Vector Clocks and Happens-Before

$$\begin{pmatrix}1\\2\\2\end{pmatrix} < \begin{pmatrix}2\\2\\3\end{pmatrix}$$

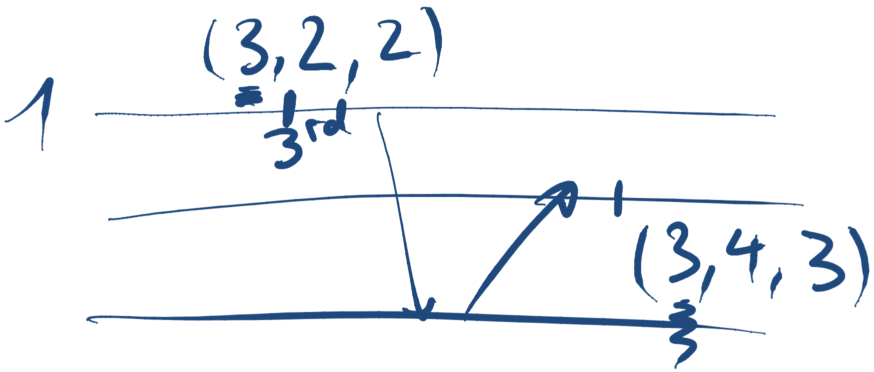
Definition: $\mathbf{VC}(\boldsymbol{e}) < \mathbf{VC}(\boldsymbol{e}') :=$

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

$\exists v \in V: VC_v(e) < VC_v(e')$

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

$$\mathbf{VC}(\boldsymbol{e}) < \mathbf{VC}(\boldsymbol{e}') \quad \longleftrightarrow \quad \boldsymbol{e} \Rightarrow_{\boldsymbol{s}} \boldsymbol{e}'$$



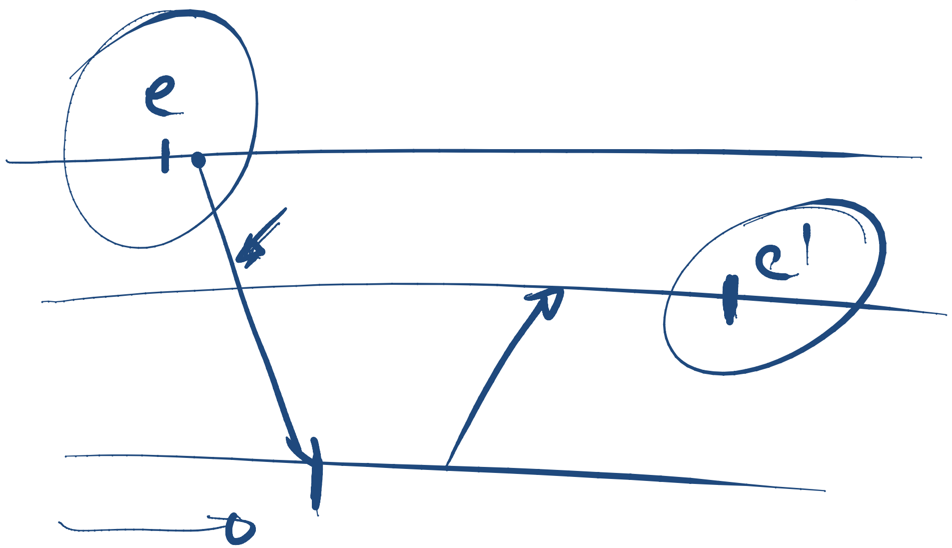
Vector Clocks and Happens-Before

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

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

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

$$\mathbf{VC}(\boldsymbol{e}) < \mathbf{VC}(\boldsymbol{e'}) \quad \leftrightarrow \quad \boldsymbol{e} \Rightarrow_{\boldsymbol{s}} \boldsymbol{e'}$$



e, e' at same node

$\forall v: VC_v(e) \leq VC_v(e')$

$VC_u(e) < VC_u(e')$

Logical Clocks vs. Synchronizers

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

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

Properties:

- 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

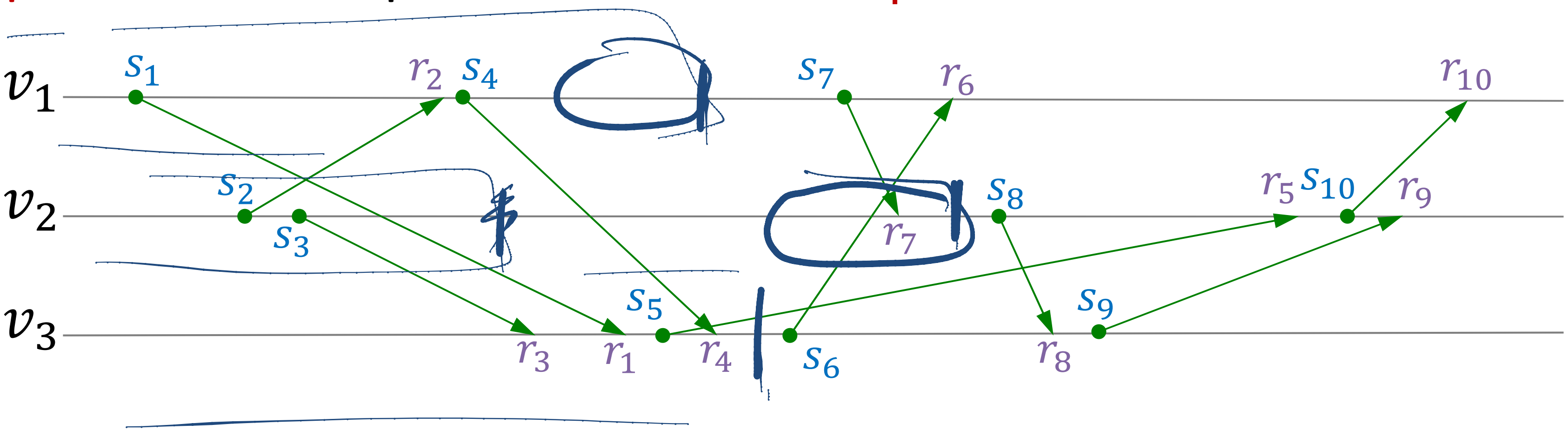
Global States

- 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: Record a global state which is consistent with all local views
  - i.e., a state which could have been tr~~e~~ue 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 $S|v$.

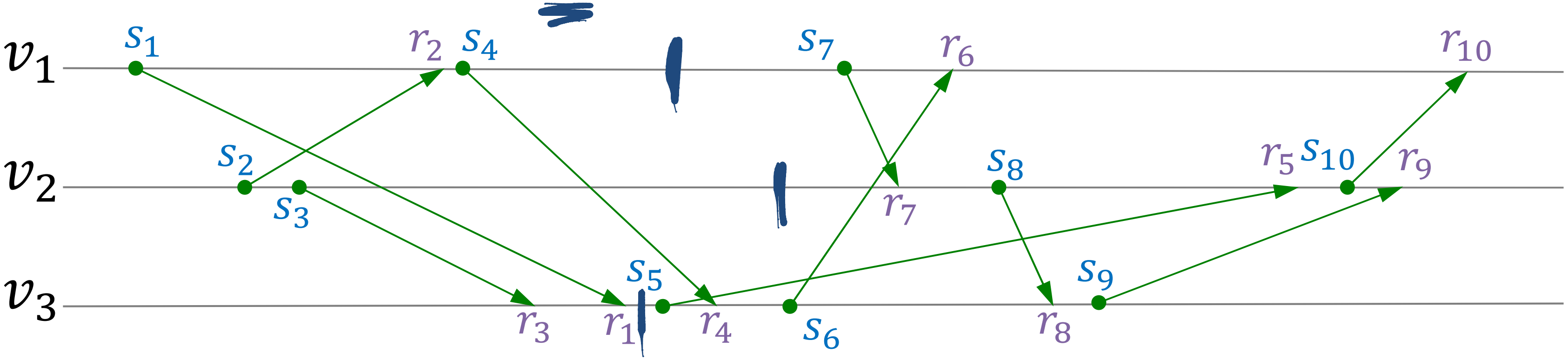


Consistent Cut

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

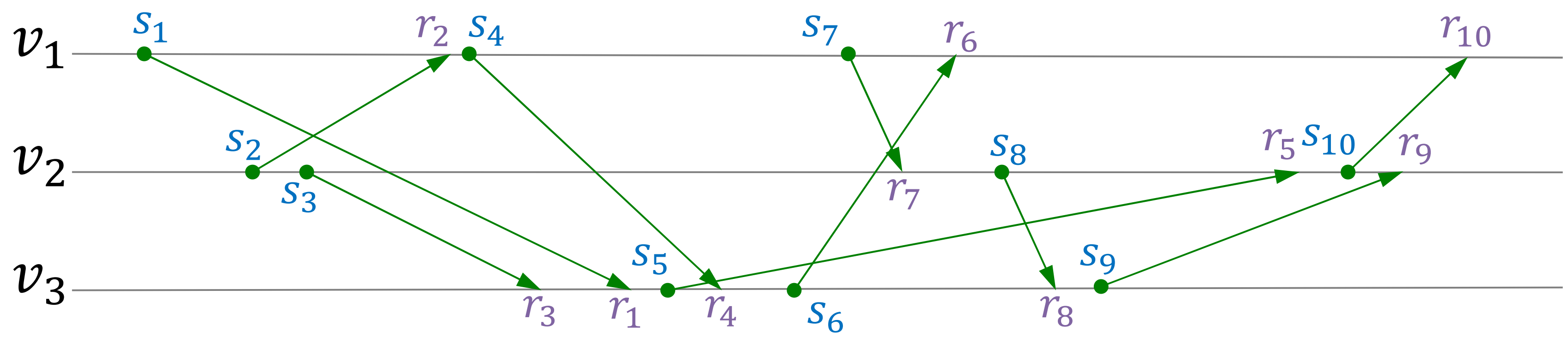
$$f \Rightarrow_S e \longrightarrow f \in C$$



$C = \{ s_1, r_2, s_4, s_2, s_3, r_1, r_3 \}$

Consistent Cut

Schedule S



Some Causal Shuffle S'