Chapter 4
Causality, Logical Time, and Global States

Distributed Systems

SS 2015

Fabian Kuhn

Causal Shuffles

Causal Shuffles

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

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

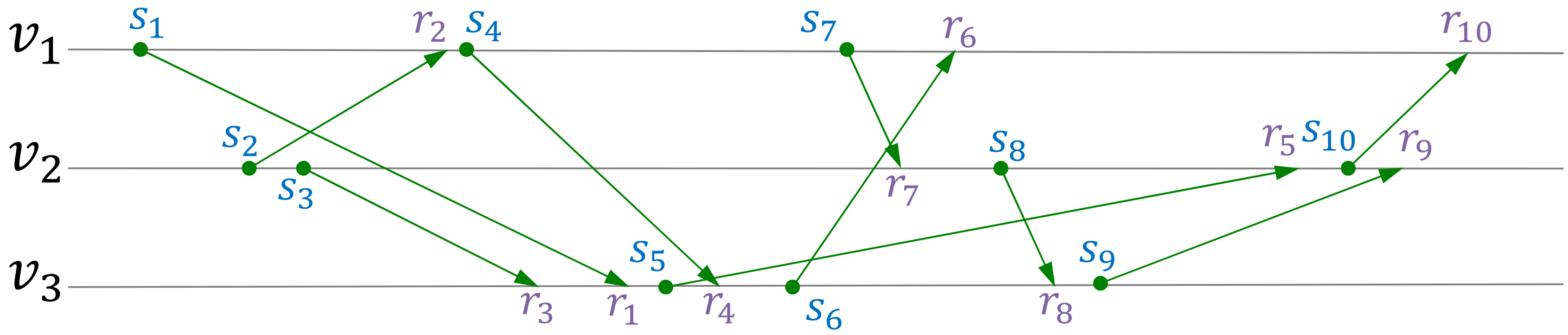
For a given schedule $\boldsymbol{S}$:

- 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

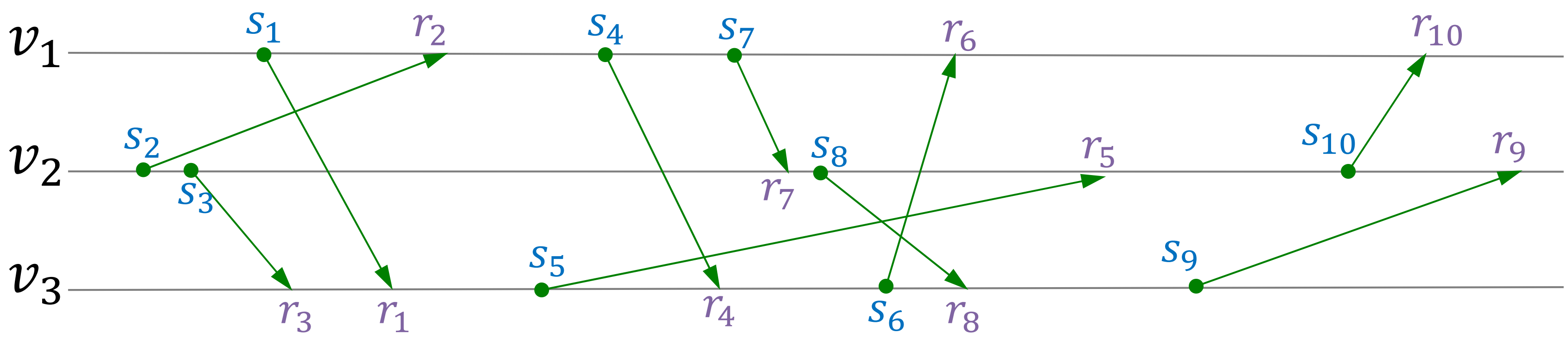
Event $\boldsymbol{e}$ provably occurs before $\boldsymbol{e'}$ if and only if $\boldsymbol{e}$ appears before $\boldsymbol{e'}$ in all causal shuffles of $\boldsymbol{S}$

Causal Shuffles / Causal Order Example

Schedule S



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

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

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

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.

- 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

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 $\Rightarrow_S$

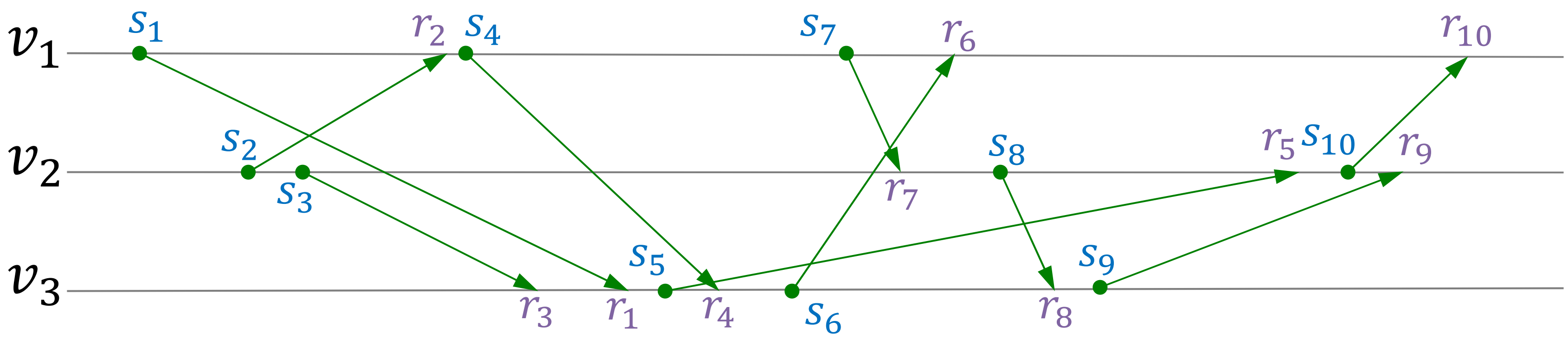
Global States

- 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 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 $S|v$.

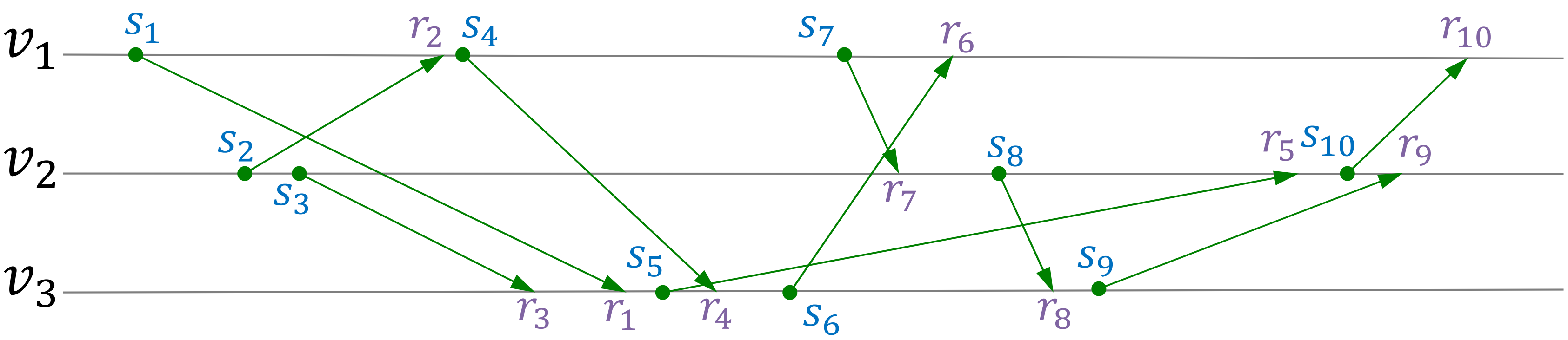


Consistent Cut

Consistent Cut

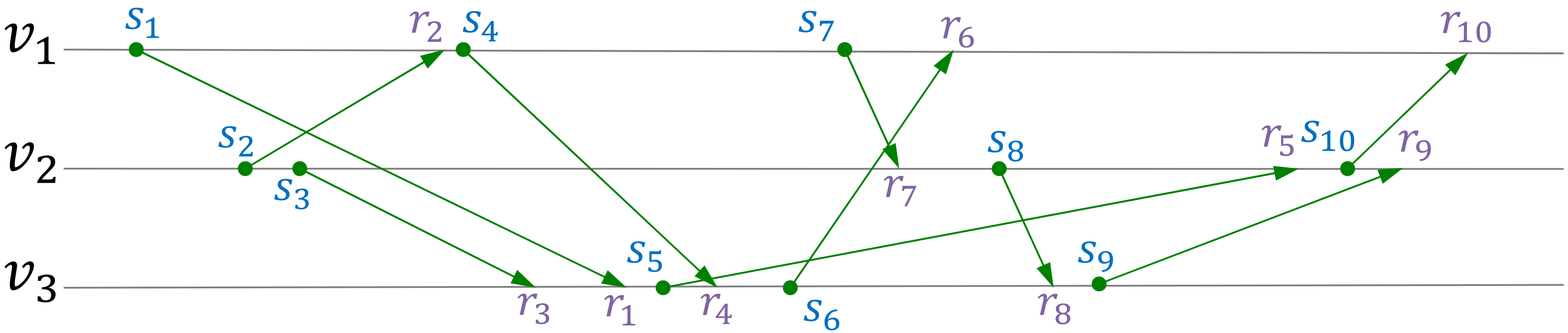
Given a schedule S, a consistent cut C is cut such that for all events $e \in C$ and all events f in S, it holds that

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

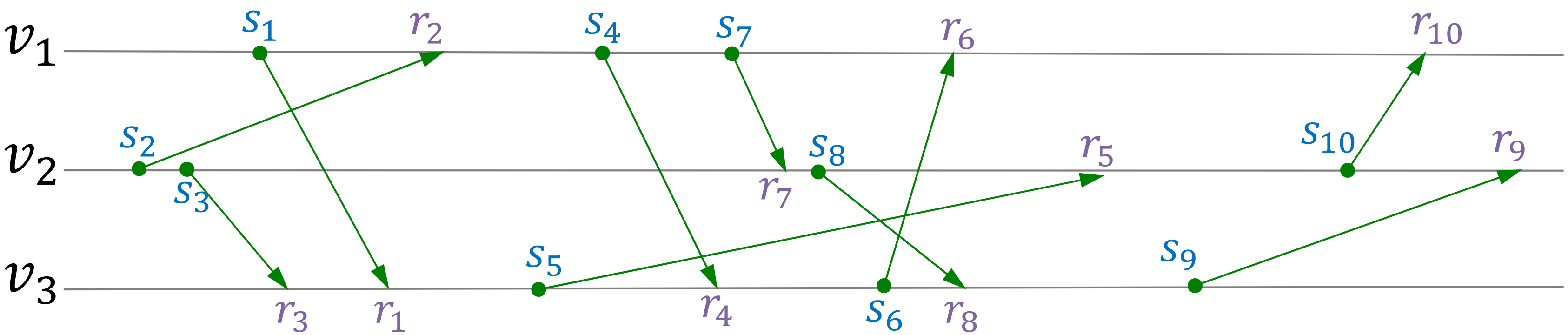


Consistent Cut

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

- τ large: not clear how long it takes until snapshot is computed
- τ 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