Race Detection

cs378h

Pro Forma

- Questions?
- Administrivia:
  - Course/Instructor Survey : https://utdirect.utexas.edu/ctl/ecis/
  - Thoughts on exam
  - Thoughts on project presentation day
- Agenda
  - Linearizability clarification
  - Race Detection

- Acknowledgements:
  - https://ecksit.wordpress.com/2015/09/07/difference-between-sequential-consistency-serializability-and-linearizability/
  - https://www.cl.cam.ac.uk/teaching/1718/R204/slides-tharris-2-lock-free.pptx
  - http://concurrencyfreaks.blogspot.com/2013/05/lock-free-and-wait-free-definition-and.html
  - http://swtv.kaist.ac.kr/courses/cs492b-spring-16/lec6-data-race-bug.pptx
  - https://www.cs.cmu.edu/~clegoues/docs/static-analysis.pptx
  - http://www.cs.sfu.ca/~fedorova/Teaching/CMPT401/Summer2008/Lectures/Lecture8-GlobalClocks.pptx

Race Detection Faux Quiz

- Are linearizable objects composable? Why/why not? Is serializable code composable?
- What is a data race? What kinds of conditions make them difficult to detect automatically?
- What is a consistent cut in a distributed causality interaction graph?
- List some tradeoffs between static and dynamic race detection
- What are some pros and cons of happens-before analysis for race detection? Same for lockset analysis?
- Why might one use a vector clock instead of a logical clock?
- What are some advantages and disadvantages of combined lock-set and happens-before analysis?

Race Detection

Race Detection

Locks: a litany of problems

Race Detection

Locks: a litany of problems

- Deadlock

Race Detection

Locks: a litany of problems

- Deadlock
- Priority inversion

Race Detection

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys

Race Detection

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation

Race Detection

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance

Race Detection

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance

Race Detection

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance

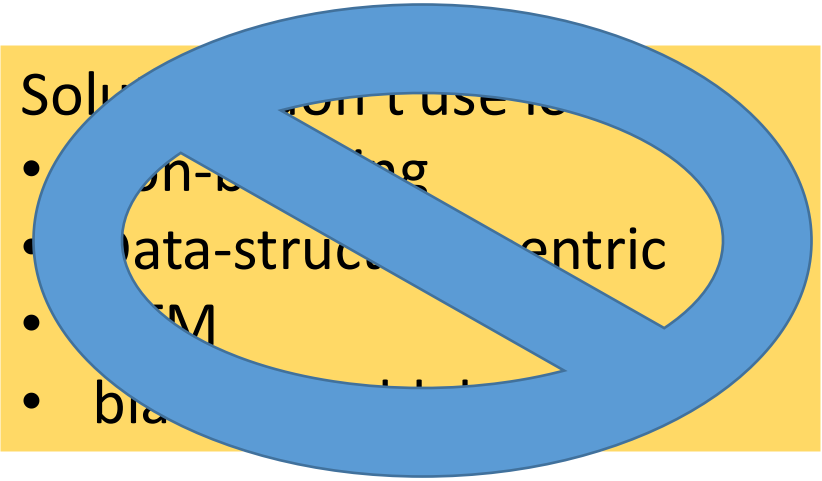
Solution: don't use locks

- non-blocking
- Data-structure-centric
- HTM
- blah, blah, blah..

Race Detection

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance



Race Detection

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance

Solu… on't use lo…
- …on-b… …ng
- …ata-struc… …entric
- …M
- bla…

Use locks!
- But automate bug-finding!

Races

```
Lock(lock);
Read-Write(X);              Read-Write(X);
Unlock(lock);
```

Races

```
Lock(lock);
Read-Write(X);          Read-Write(X);
Unlock(lock);
```

- Is there a race here?

Races

```
Lock(lock);
Read-Write(X);
Unlock(lock);
```

```

Read-Write(X);

```

- Is there a race here?
- What is a race?

Races

```
Lock(lock);
Read-Write(X);               Read-Write(X);
Unlock(lock);
```

- Is there a race here?
- What is a race?
- Informally: accesses with missing/incorrect synchronization

Races

```
Lock(lock);
Read-Write(X);             Read-Write(X);
Unlock(lock);
```

- Is there a race here?
- What is a race?
- Informally: accesses with missing/incorrect synchronization
- Formally:
  - >1 threads access same item
  - No intervening synchronization
  - At least one access is a write

Races

Thread 1	Thread 2
`Lock(lock);`	
`Read-Write(X);`	`Read-Write(X);`
`Unlock(lock);`	

- Is there a race here?
- What is a race?
- Informally: accesses with missing/incorrect sy
- Formally:
    - >1 threads access same item
    - No intervening synchronization
    - At least one access is a write

How to detect races:

```
forall(X) {
  if(not_synchronized(X))
    declare_race()
}
```

Races

```
read-write(X);
fork(thread-proc);
do_stuff();
do_more_stuff();
join(thread-proc);
read-Write(X);
```

```
thread-proc() {

  read-write(X);

}
```

Is there a race here?

How can a race detector tell?

Races

```
read-write(X);
fork(thread-proc);
do_stuff();
do_more_stuff();
join(thread-proc);
read-Write(X);
```

```
thread-proc() {

 read-write(X);

}
```

Is there a race here?

How can a race detector tell?

Races

```
read-write(X);
fork(thread-proc);
do_stuff();
do_more_stuff();
join(thread-proc);
read-Write(X);
```

```
thread-proc() {

 read-write(X);

}
```

Is there a race here?

How can a race detector tell?

Unsynchronized access can be

Races

```
read-write(X);
fork(thread-proc);
do_stuff();
do_more_stuff();
join(thread-proc);
read-Write(X);
```

```
thread-proc() {

  read-write(X);

}
```

Is there a race here?

How can a race detector tell?

Unsynchronized access can be

- Benign due to fork/join

Races

```
read-write(X);
fork(thread-proc);
do_stuff();
do_more_stuff();
join(thread-proc);
read-Write(X);
```

```
thread-proc() {

  read-write(X);

}
```

Is there a race here?
How can a race detector tell?

Unsynchronized access can be

- Benign due to fork/join
- Benign due to view serializability

Races

```
read-write(X);
fork(thread-proc);
do_stuff();
do_more_stuff();
join(thread-proc);
read-Write(X);
```

```
thread-proc() {

 read-write(X);

}
```

Is there a race here?
How can a race detector tell?

Unsynchronized access can be

- Benign due to fork/join
- Benign due to view serializability
- Benign due to application-level constraints

Races

```
read-write(X);
fork(thread-proc);
do_stuff();
do_more_stuff();
join(thread-proc);
read-Write(X);
```

```
thread-proc() {
  read-write(X);
}
```

Is there a race here?

How can a race detector tell?

Unsynchronized access can be

- Benign due to fork/join
- Benign due to view serializability
- Benign due to application-level constraints
- E.g. approximate stats counters

Detecting Races

- Static
  - Run a tool that analyses just code
  - Maybe code is annotated to help
  - Conservative: detect races that never occur
- Dynamic
  - Instrument code
  - Check synchronization invariants on accesses
  - More precise
  - Difficult to make fast
  - ***Lockset vs happens-before***

How to detect races:

```
forall(X) {
  if(not_synchronized(X))
    declare_race()
}
```

```
Lock(lock);
Read-Write(X);
Unlock(lock);
```

```

Read-Write(X);

```

Static Data Race Detection

- Type-based analysis
  - Language type system augmented
    - express common synchronization relationships”: correct typing→no data races
  - Difficult to do
  - Restricts the type of synchronization primitives
- Language features
  - e.g., use of monitors
  - Only works for static data – not dynamic data
- Model Checking
- Path analysis
  - Doesn’t scale well
  - Too many false positives

Static Data Race Detection

- Type-based analysis
  - Language type system augmented
    - express common synchronization relationships": correct typing→no data races
  - Difficult to do
  - Restricts the type of synchronization primitives
- Language features
  - e.g., use of monitors
  - Only works for static data – not dynamic data
- Model Checking
- Path analysis
  - Doesn't scale w
  - Too many false

```
Lock(lock);
Read-Write(X);              Read-Write(X);
Unlock(lock);
```

Static Data Race Detection

- Type-based analysis
  - Language type system augmented
    - express common synchronization relationships”: correct typing→no data races
  - Difficult to do
  - Restricts the type of synchronization primitives
- Language features
  - e.g., use of monitors
  - Only works for static data – not dynamic dat
- Model Checking
- Path analysis
  - Doesn’t scale w
  - Too many false

What if these *never* run concurrently? (False Positive)

```
Lock(lock);
Read-Write(X);              Read-Write(X);
Unlock(lock);
```

Lockset Algorithm

Lockset Algorithm

- Locking discipline
  - Every shared mutable variable is protected by some locks

Lockset Algorithm

- Locking discipline
  - Every shared mutable variable is protected by some locks
- Core idea

Lockset Algorithm

- Locking discipline
  - Every shared mutable variable is protected by some locks
- Core idea
  - Track locks held by thread t

Lockset Algorithm

- Locking discipline
  - Every shared mutable variable is protected by some locks
- Core idea
  - Track locks held by thread t
  - On access to var v, check if t holds the proper locks

Lockset Algorithm

- Locking discipline
  - Every shared mutable variable is protected by some locks
- Core idea
  - Track locks held by thread t
  - On access to var v, check if t holds the proper locks
  - Challenge: how to know what locks are required?

Lockset Algorithm

- Locking discipline
  - Every shared mutable variable is protected by some locks
- Core idea
  - Track locks held by thread t
  - On access to var v, check if t holds the proper locks
  - Challenge: how to know what locks are required?
- Infer protection relation
  - Infer which locks protect which variable from execution history.

Lockset Algorithm

- Locking discipline
  - Every shared mutable variable is protected by some locks
- Core idea
  - Track locks held by thread t
  - On access to var v, check if t holds the proper locks
  - Challenge: how to know what locks are required?
- Infer protection relation
  - Infer which locks protect which variable from execution history.
  - Assume every lock protects every variable

Lockset Algorithm

- Locking discipline
  - Every shared mutable variable is protected by some locks
- Core idea
  - Track locks held by thread t
  - On access to var v, check if t holds the proper locks
  - Challenge: how to know what locks are required?
- Infer protection relation
  - Infer which locks protect which variable from execution history.
  - Assume every lock protects every variable
  - On each access, use locks held by thread to narrow that assumption

Lockset Algorithm

- Locking discipline
  - Every shared mutable variable is protected by some locks
- Core idea
  - Track locks held by thread t

Let $locks\_held(t)$ be the set of locks held by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each access to v by thread t,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{\ \}$, then issue a warning.

Narrow down set of locks maybe protecting v

  - Assume every lock protects every variable
  - On each access, use locks held by thread to narrow that assumption

Lockset Algorithm Example

thread t

```
lock(lockA);
v++;
unlock(lockA);

lock(lockB);
v++;
unlock(lockB);
```

locks_held(t)	C(v)
{}	{lockA, lockB}

Lockset Algorithm Example

thread t

```
lock(lockA);
v++;
unlock(lockA);

lock(lockB);
v++;
unlock(lockB);
```

locks_held(t)	C(v)
{}	{lockA, lockB}

Lockset Algorithm Example

thread t	locks_held(t)	C(v)
	{}	{lockA, lockB}
`lock(lockA);`	{lockA}	
`v++;`		
`unlock(lockA);`		
`lock(lockB);`		
`v++;`		
`unlock(lockB);`		

Lockset Algorithm Example

thread t	locks_held(t)	C(v)
	{}	{lockA, lockB}
`lock(lockA);`	{lockA}	
→ `v++;`		{lockA} $C(v) \cap locks\_held(t)$
`unlock(lockA);`		
`lock(lockB);`		
`v++;`		
`unlock(lockB);`		

Lockset Algorithm Example

thread t	locks_held(t)	C(v)
	`{}`	`{lockA, lockB}`
`lock(lockA);`	`{lockA}`	
`v++;`		`{lockA}`
➡ `unlock(lockA);`	`{}`	
`lock(lockB);`		
`v++;`		
`unlock(lockB);`		

Lockset Algorithm Example

thread t	locks_held(t)	C(v)
	{}	{lockA, lockB}
`lock(lockA);`	{lockA}	
`v++;`		{lockA}
`unlock(lockA);`	{}	
`lock(lockB);`	{lockB}	
`v++;`		
`unlock(lockB);`		

Lockset Algorithm Example

thread t	locks_held(t)	C(v)
	`{}`	`{lockA, lockB}`
`lock(lockA);`	`{lockA}`	
`v++;`		`{lockA}`
`unlock(lockA);`	`{}`	
`lock(lockB);`	`{lockB}`	
`v++;`		`{}`
`unlock(lockB);`		

Lockset Algorithm Example

thread t	locks_held(t)	C(v)
	{}	{lockA, lockB}
`lock(lockA);`	{lockA}	
`v++;`		{lockA}
`unlock(lockA);`	{}	
`lock(lockB);`	{lockB}	
`v++;`		{} $C(v) \cap locks\_held(t)$
`unlock(lockB);`	{}	

Lockset Algorithm Example

thread t	locks_held(t)	C(v)
	{}	{lockA, lockB}
lock(lockA);	{lockA}	
v++;		{lockA}
unlock(lockA);	{}	
lock(lockB);	{lockB}	
v++;		{}
unlock(lockB);	{}	

ACK! race

Lockset Algorithm Example

thread t	locks_held(t)	C(v)
	{}	{lockA, lockB}
lock(lockA);	{lockA}	
v++;		{lockA}
unlock(lockA);	{}	
lock(lockB);	{lockB}	
v++;		{}
unlock(lockB);	{}	

ACK! race

Pretty clever! Why isn't this a complete solution?

Improving over lockset

thread A

```
read-write(X);
fork(thread-proc);
do_stuff();
do_more_stuff();
join(thread-proc);
read-Write(X);
```

thread B

```
thread-proc() {
 read-write(X);
}
```

Improving over lockset

thread A

```
read-write(X);
fork(thread-proc);
do_stuff();
do_more_stuff();
join(thread-proc);
read-Write(X);
```

thread B

```
thread-proc() {
 read-write(X);
}
```

Lockset detects a race

There is no race: why not?

Improving over lockset

thread A

```
1 read-write(X);
2 fork(thread-proc);
3 do_stuff();
4 do_more_stuff();
5 join(thread-proc);
6 read-Write(X);
```

thread B

```
1 thread-proc() {
2
3   read-write(X);
4
5 }
```

Lockset detects a race

There is no race: why not?

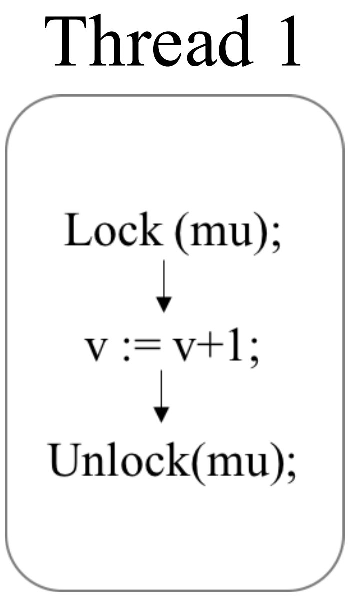
- A-1 happens before B-3
- B-3 happens before A-6
- Insight: races occur when “happens-before” cannot be known

Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing variables not ordered by “happens-before” is a race
- Captures locks and dynamism
- How to track “happens-before”?
  - Sync objects are ordering events
  - Generalizes to fork/join, etc

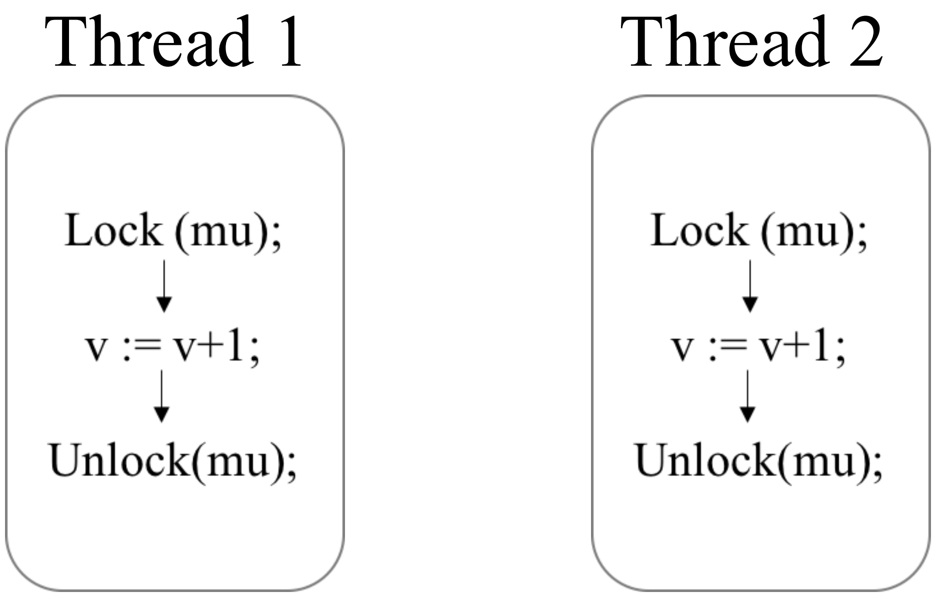
Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing variables not ordered by “happens-before” is a race
- Captures locks and dynamism
- How to track “happens-before”?
  - Sync objects are ordering events
  - Generalizes to fork/join, etc



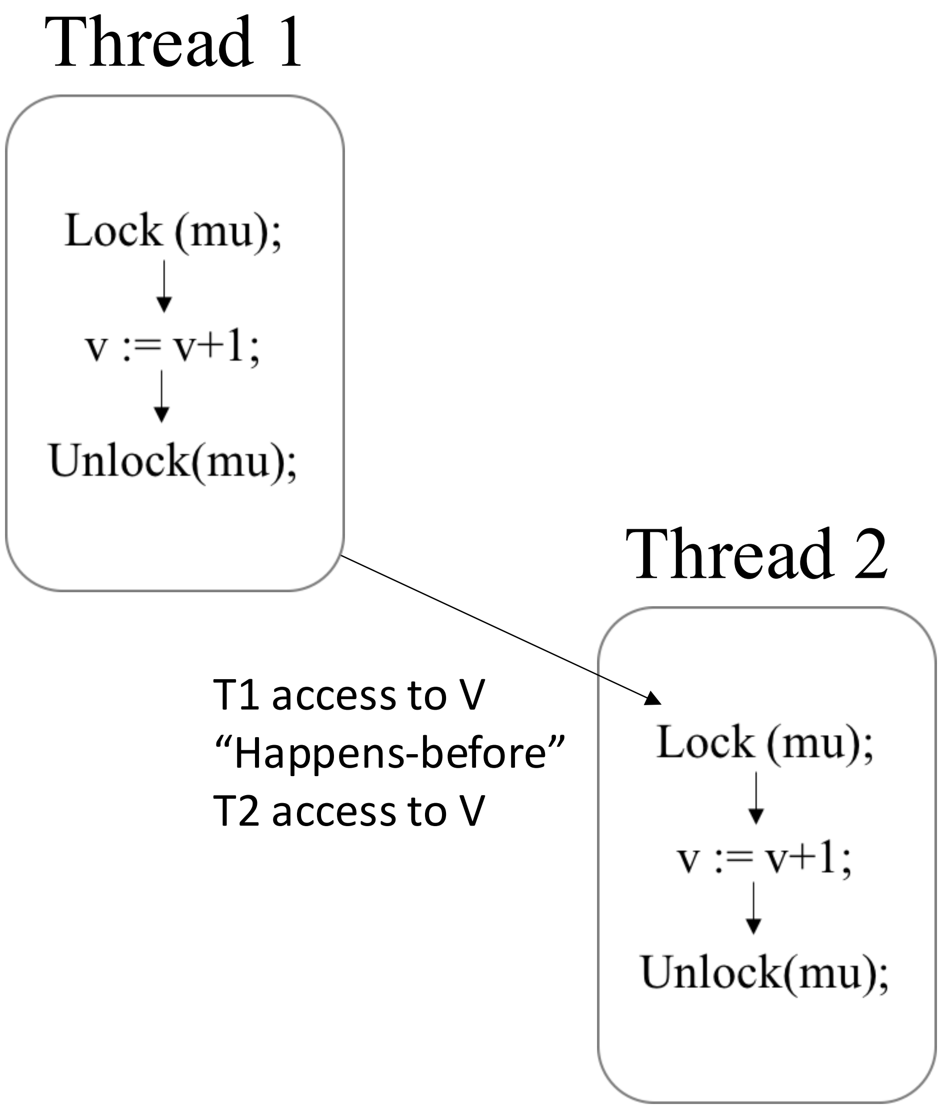
Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing variables not ordered by "happens-before" is a race
- Captures locks and dynamism
- How to track "happens-before"?
  - Sync objects are ordering events
  - Generalizes to fork/join, etc

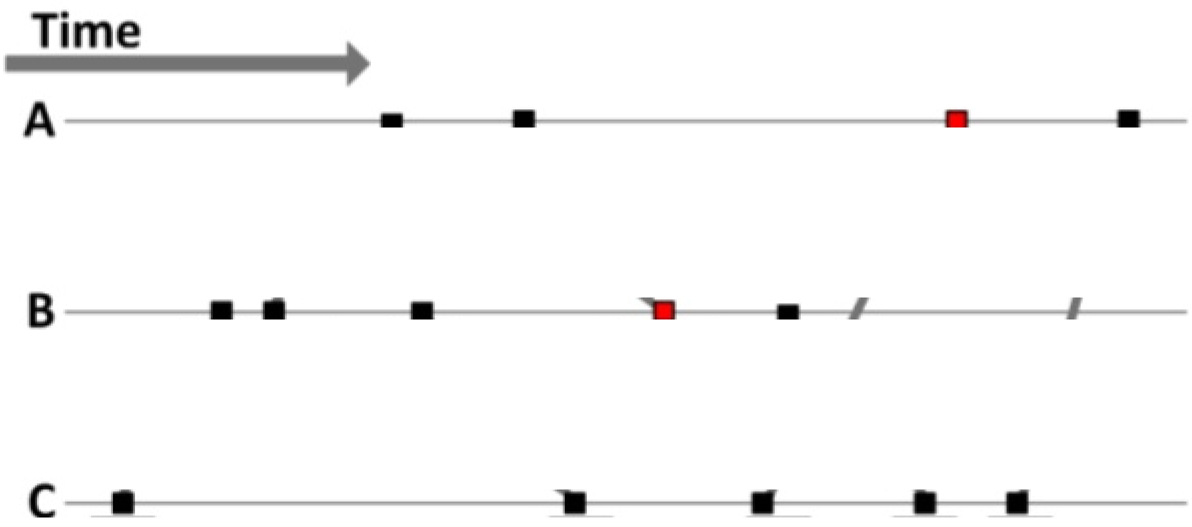


Happens-before

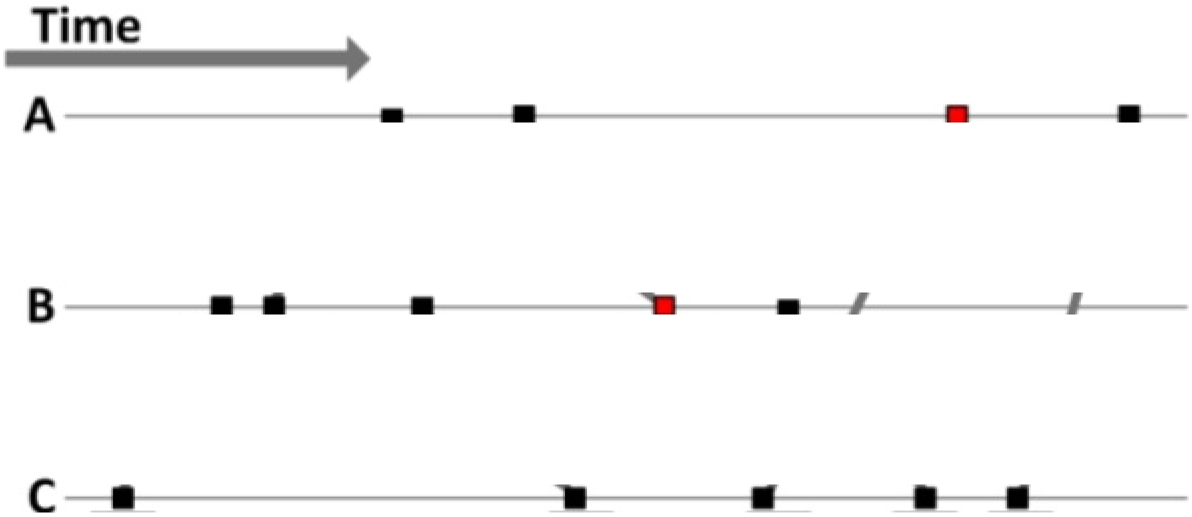
- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing variables not ordered by “happens-before” is a race
- Captures locks and dynamism
- How to track “happens-before”?
  - Sync objects are ordering events
  - Generalizes to fork/join, etc



Ordering and Causality

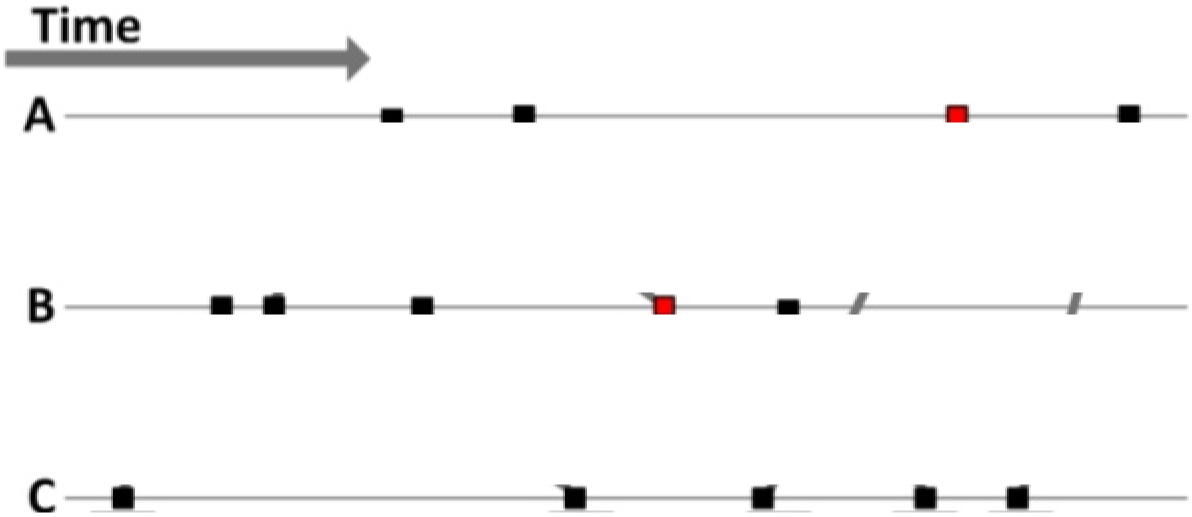


Ordering and Causality



A, B, C have local orders

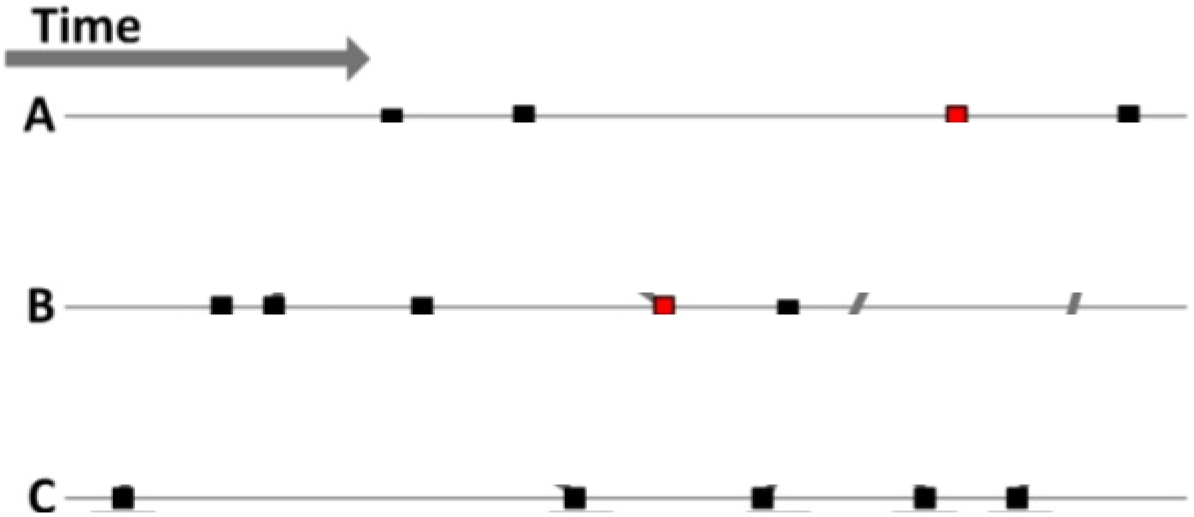
Ordering and Causality



A, B, C have local orders

- Want total order
  - But only for causality

Ordering and Causality

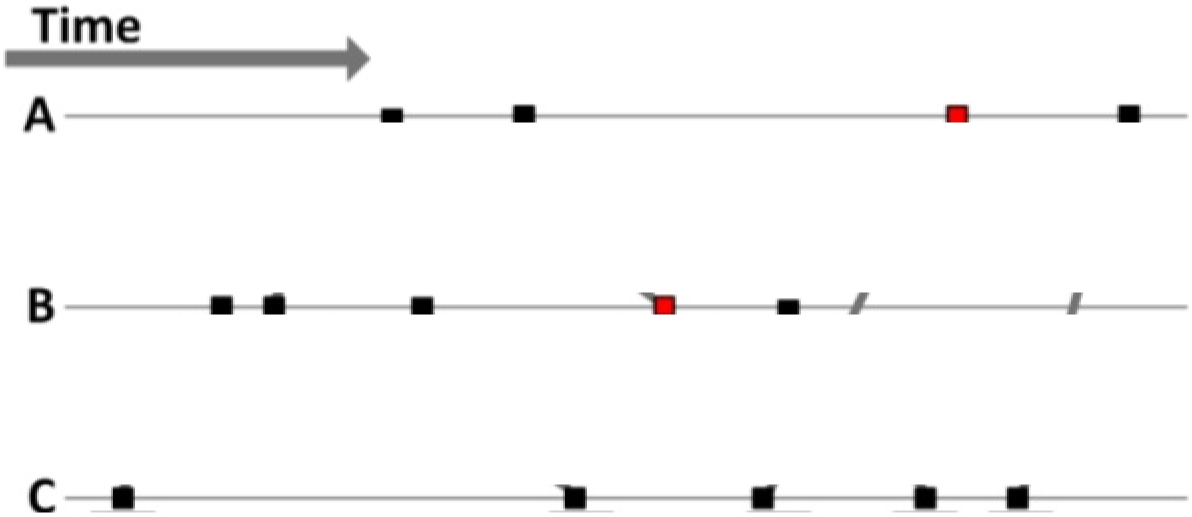


A, B, C have local orders

- Want total order
  - But only for causality

Different types of clocks

Ordering and Causality



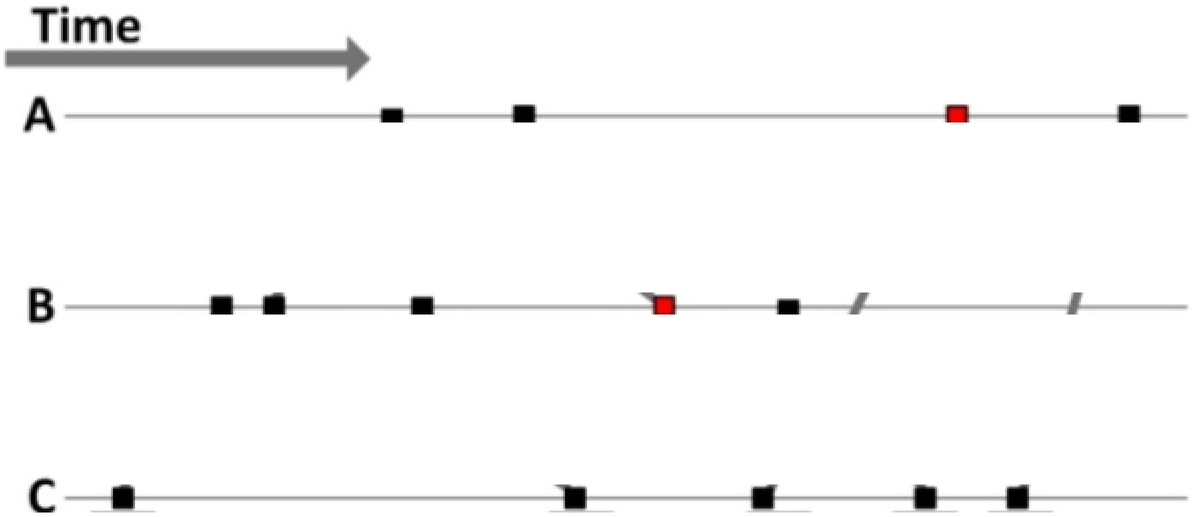
A, B, C have local orders

- Want total order
    - But only for causality

Different types of clocks

- Physical

Ordering and Causality



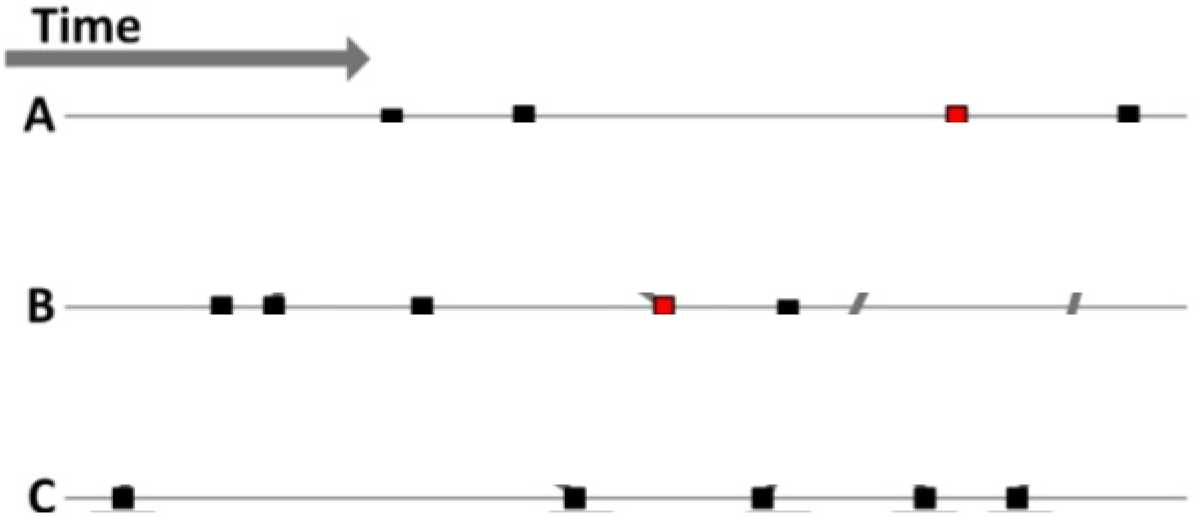
A, B, C have local orders

- Want total order
  - But only for causality

Different types of clocks

- Physical
- Logical
  - TS(A) later than others A knows about

Ordering and Causality



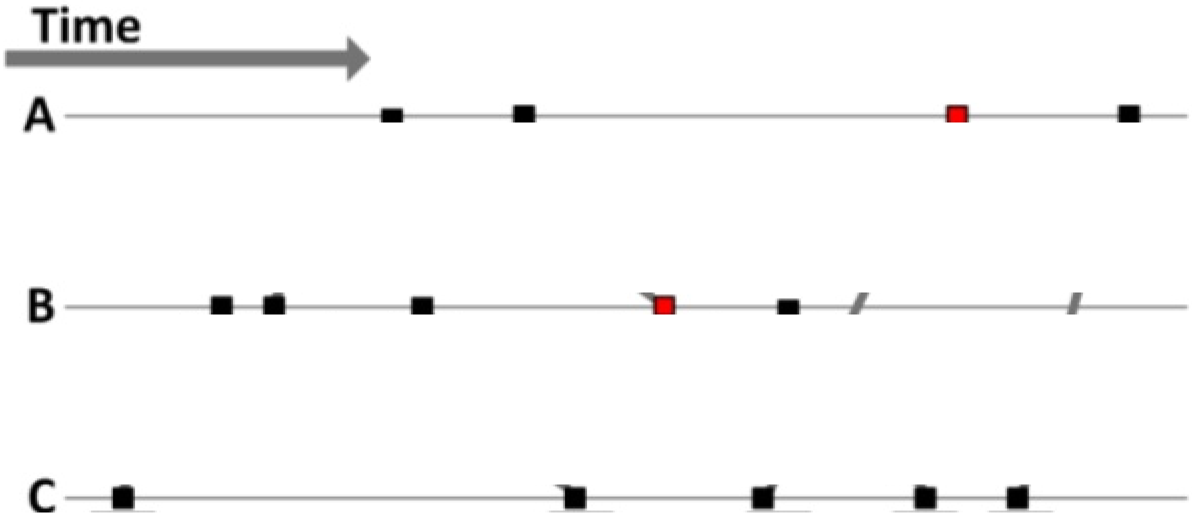
A, B, C have local orders

- Want total order
    - But only for causality

Different types of clocks

- Physical
- Logical
    - TS(A) later than others A knows about
- **Vector**
    - **TS(A): what A knows about other TS's**

Ordering and Causality



A, B, C have local orders

- Want total order
  - But only for causality

Different types of clocks

- Physical
- Logical
  - TS(A) later than others A knows about
- **Vector**
  - **TS(A): what A knows about other TS's**
- Matrix
  - TS(A) is N^2 showing pairwise knowledge

A Naïve Approach

- Each system records each event it performed and its timestamp
- Suppose events in the this system happened in this real order:

A Naïve Approach

- Each system records each event it performed and its timestamp
- Suppose events in the this system happened in this real order:
  - **Time Tc0:** System C sent data to System B (before C stopped responding)

A Naïve Approach

- Each system records each event it performed and its timestamp
- Suppose events in the this system happened in this real order:
  - **Time Tc0:** System C sent data to System B (before C stopped responding)
  - **Time Ta0:** System A asked for work from System B

A Naïve Approach

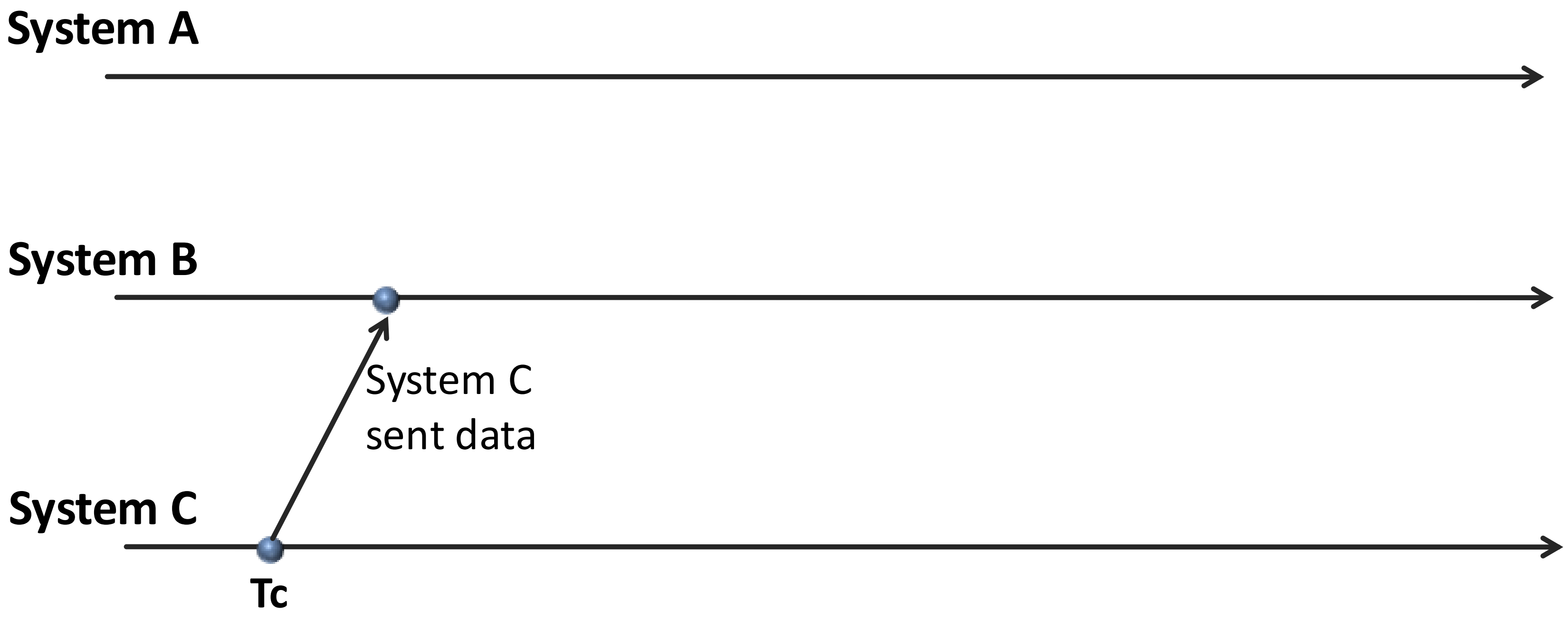
- Each system records each event it performed and its timestamp
- Suppose events in the this system happened in this real order:
  - **Time Tc0:** System C sent data to System B (before C stopped responding)
  - **Time Ta0:** System A asked for work from System B
  - **Time Tb0:** System B asked for data from System C

A Naïve Approach (cont)

- Ideally, we will construct real order of events from local timestamps and detect this dependency chain:

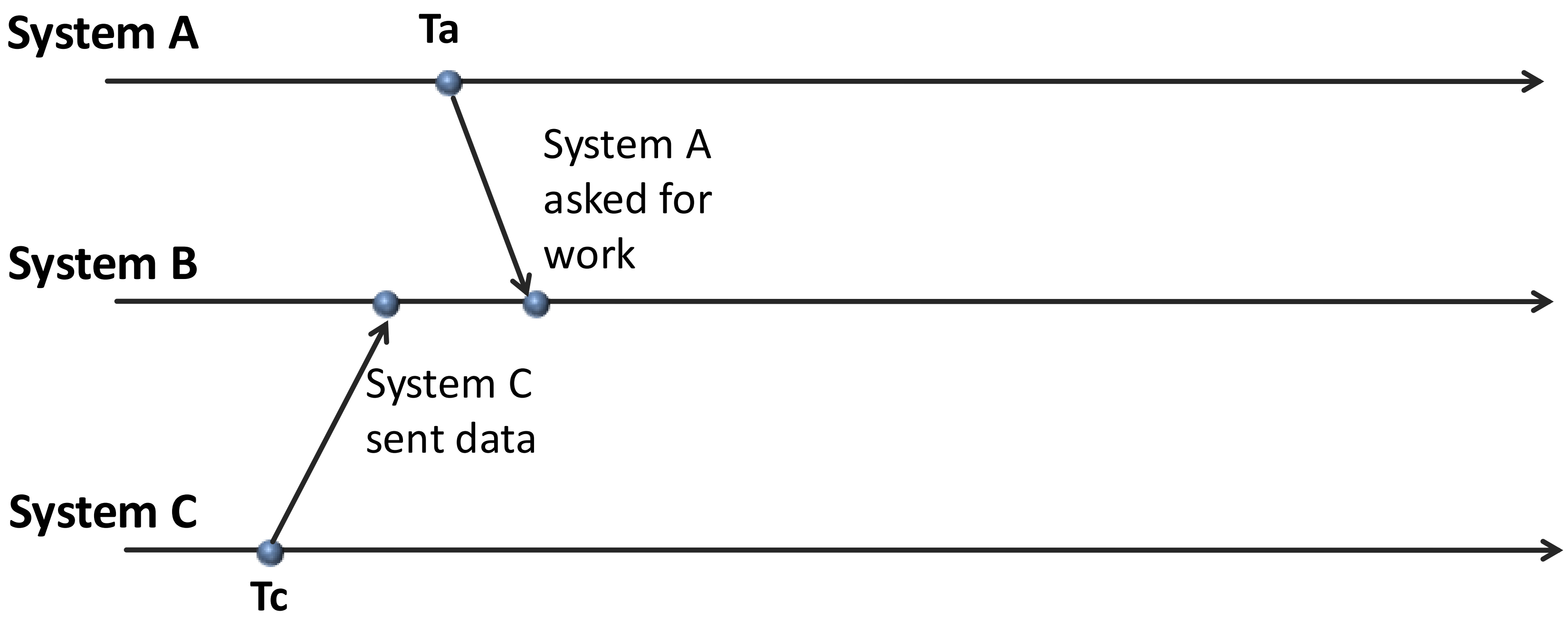
A Naïve Approach (cont)

- Ideally, we will construct real order of events from local timestamps and detect this dependency chain:



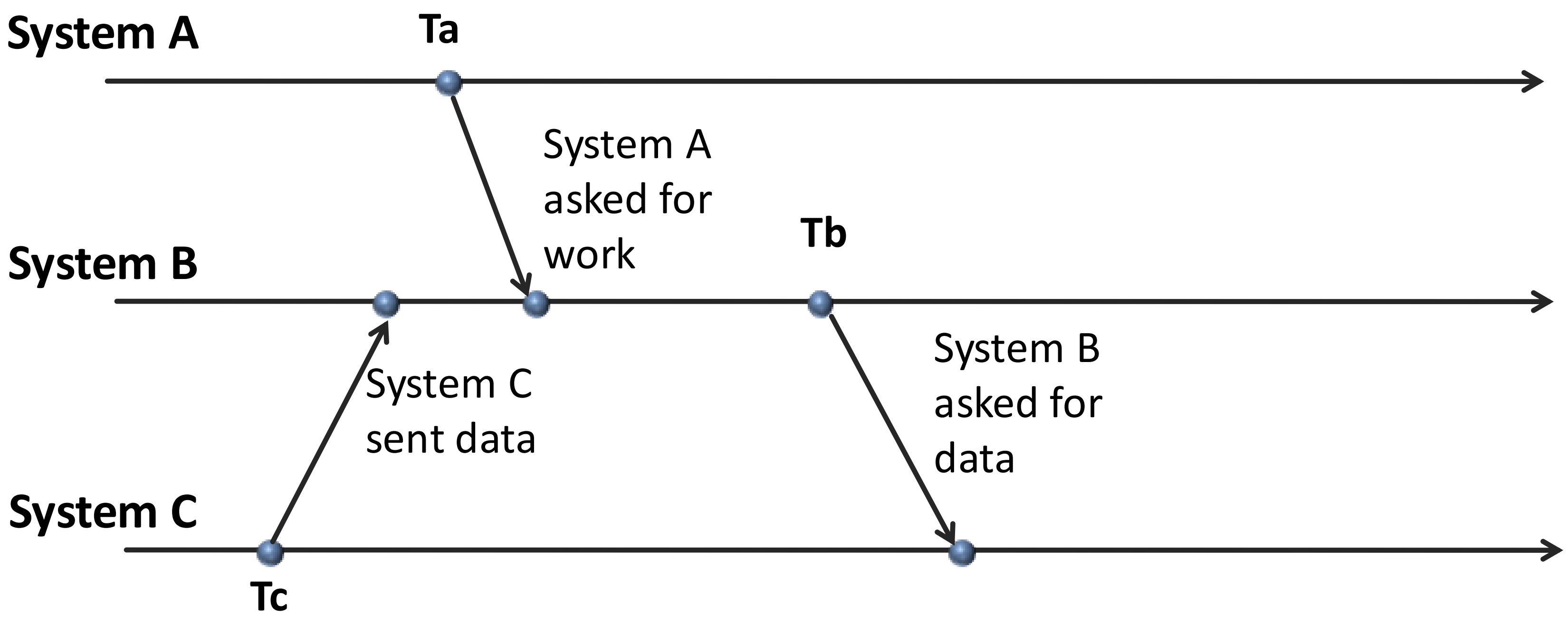
A Naïve Approach (cont)

- Ideally, we will construct real order of events from local timestamps and detect this dependency chain:



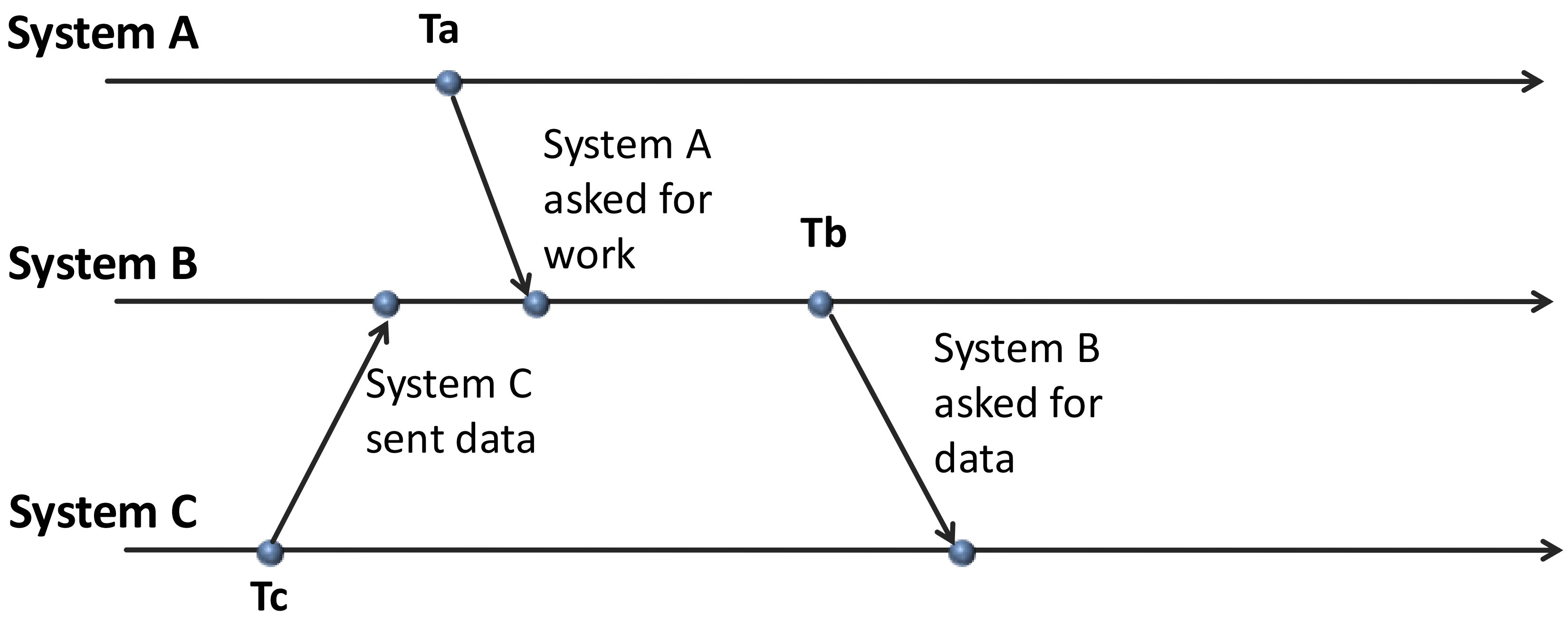
A Naïve Approach (cont)

- Ideally, we will construct real order of events from local timestamps and detect this dependency chain:



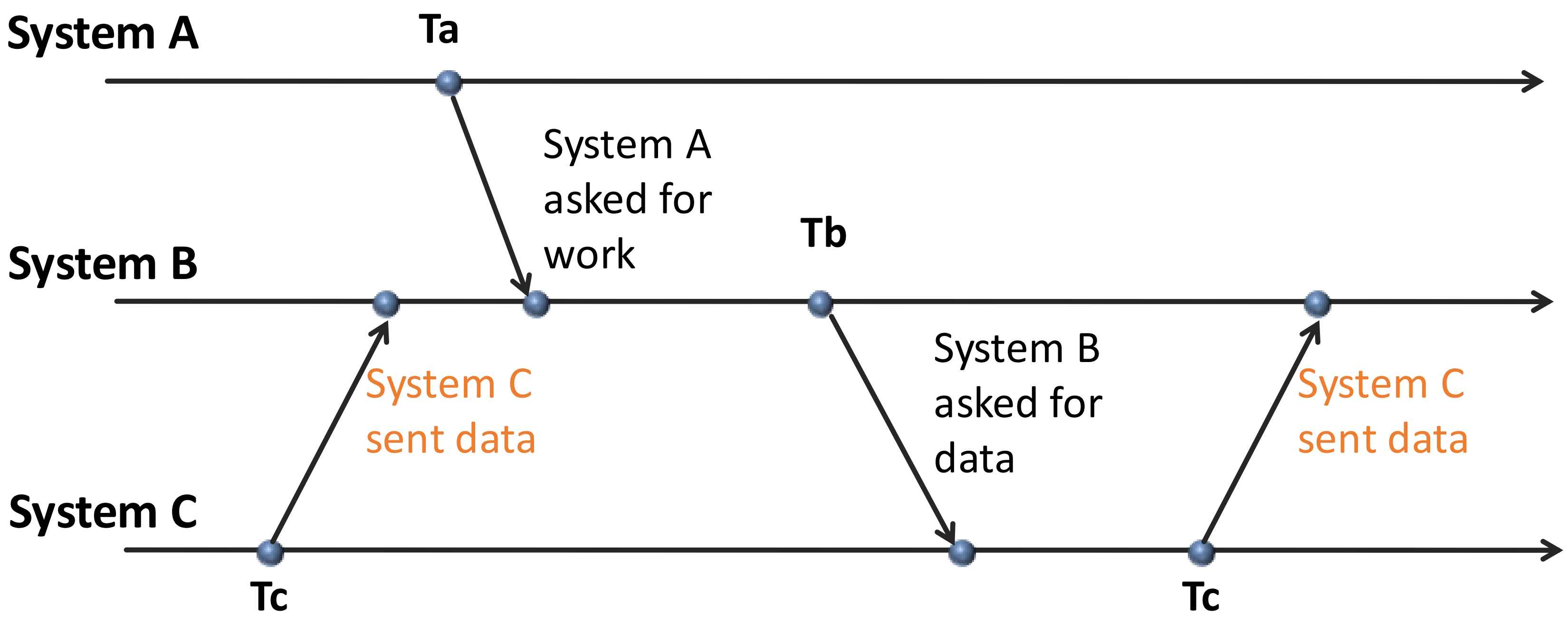
A Naïve Approach (cont)

- But in reality, we do not know if Tc occurred **before** Ta and Tb, because in an asynchronous distributed system **clocks are not synchronized**!



A Naïve Approach (cont)

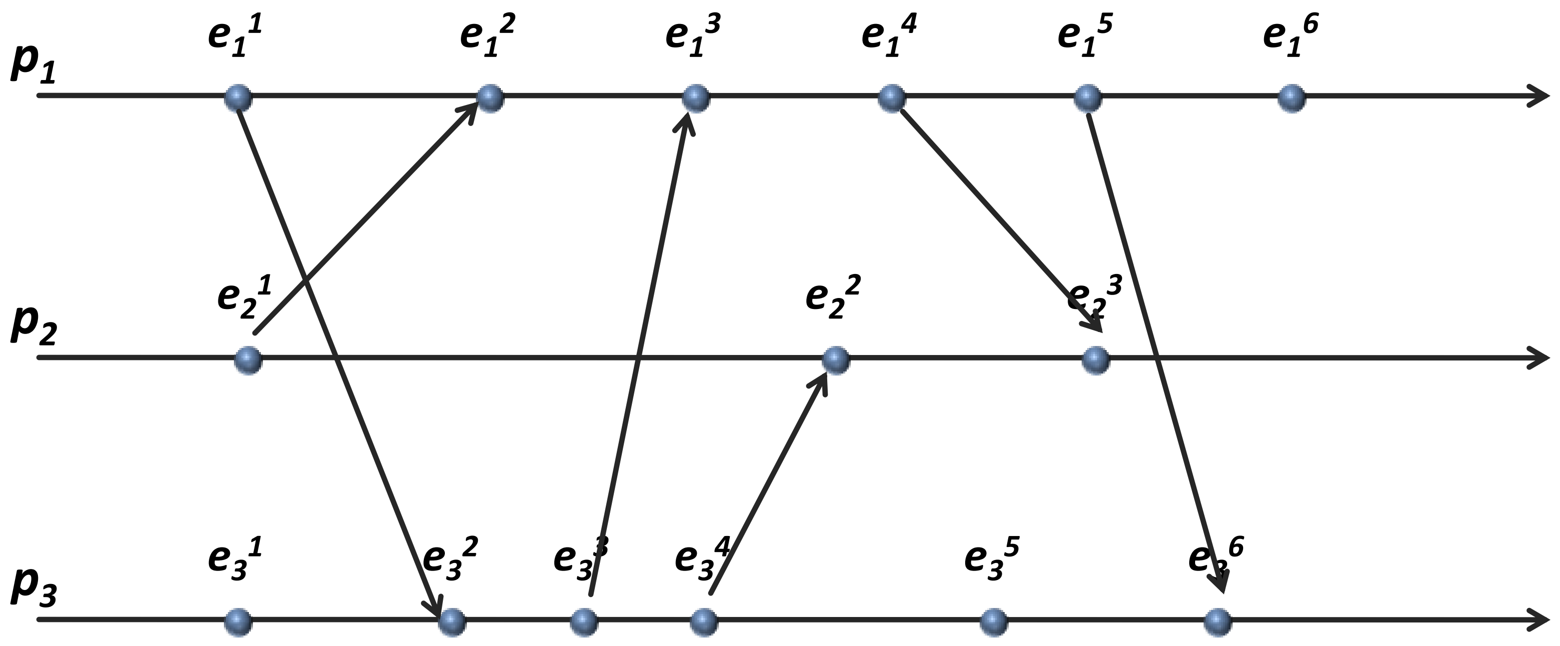
- But in reality, we do not know if Tc occurred **before** Ta and Tb, because in an asynchronous distributed system **clocks are not synchronized**!



Rules for Ordering of Events

- local events precede one another → precede one another globally:
  - If $e_i^k, e_i^m \in h_i$ and $k < m$, then $e_i^k \rightarrow e_i^m$
- Sending a message always precedes receipt of that message:
  - If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$
- Event ordering is transitive:
  - If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Space-time Diagram for Distributed Computation



local events precede one another → precede one another globally:

If $e_i^k, e_i^m \in h_i$ and $k < m$, then $e_i^k \rightarrow e_i^m$

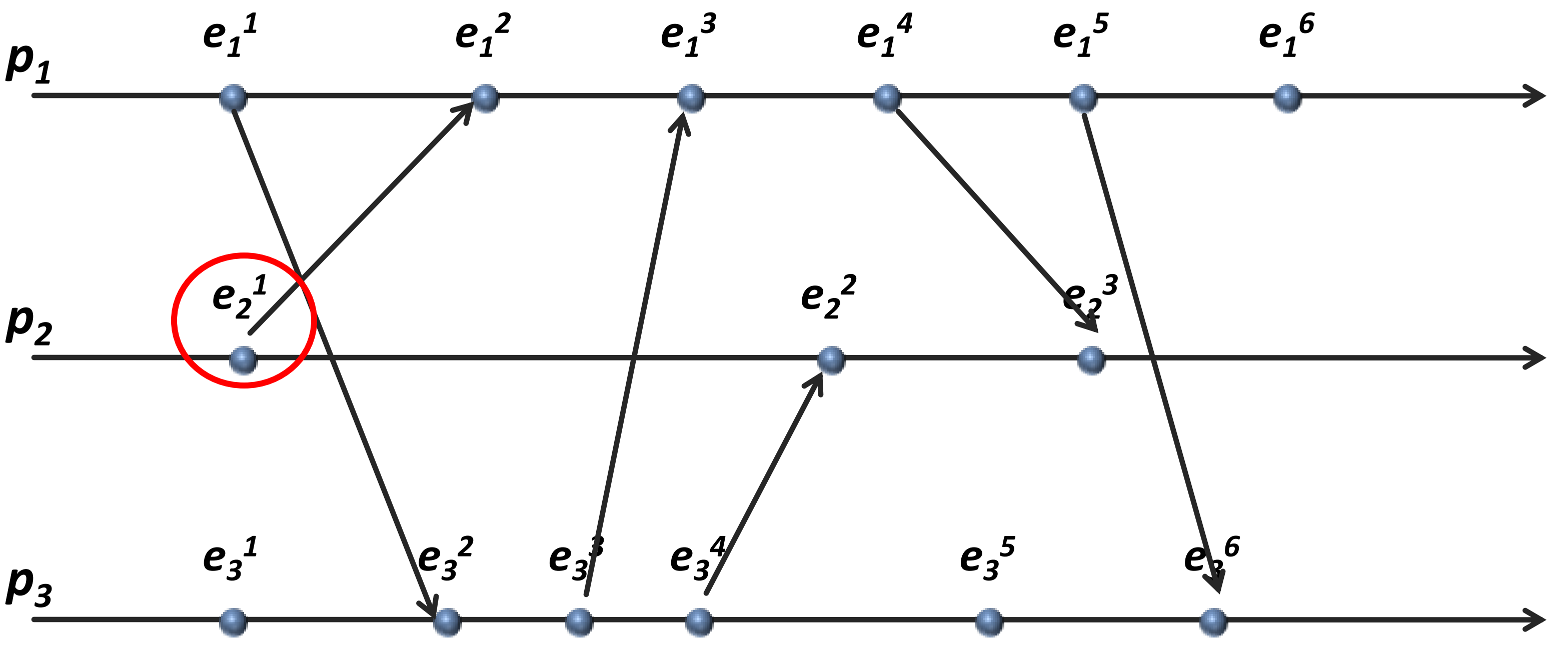
Sending a message always precedes receipt of that message:

If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$

Event ordering is associative:

If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Space-time Diagram for Distributed Computation



local events precede one another → precede one another globally:

If $e_i^k, e_i^m \in h_i$ *and* $k < m$, *then* $e_i^k \rightarrow e_i^m$

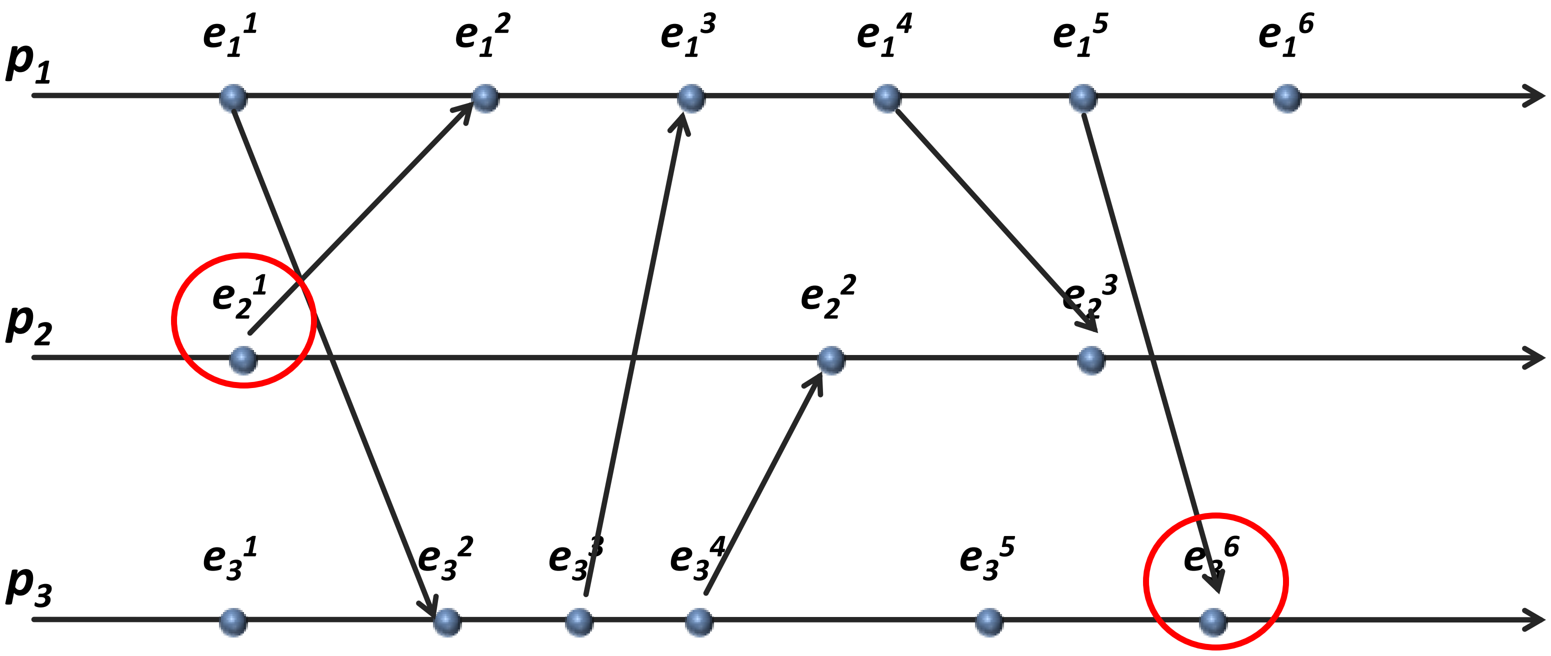
Sending a message always precedes receipt of that message:

If $e_i = send(m)$ *and* $e_j = receive(m)$, *then* $e_i \rightarrow e_j$

Event ordering is associative:

If $e \rightarrow e'$ *and* $e' \rightarrow e''$, *then* $e \rightarrow e''$

Space-time Diagram for Distributed Computation



local events precede one another → precede one another globally:

If $e_i^k, e_i^m \in h_i$ and $k < m$, then $e_i^k \rightarrow e_i^m$

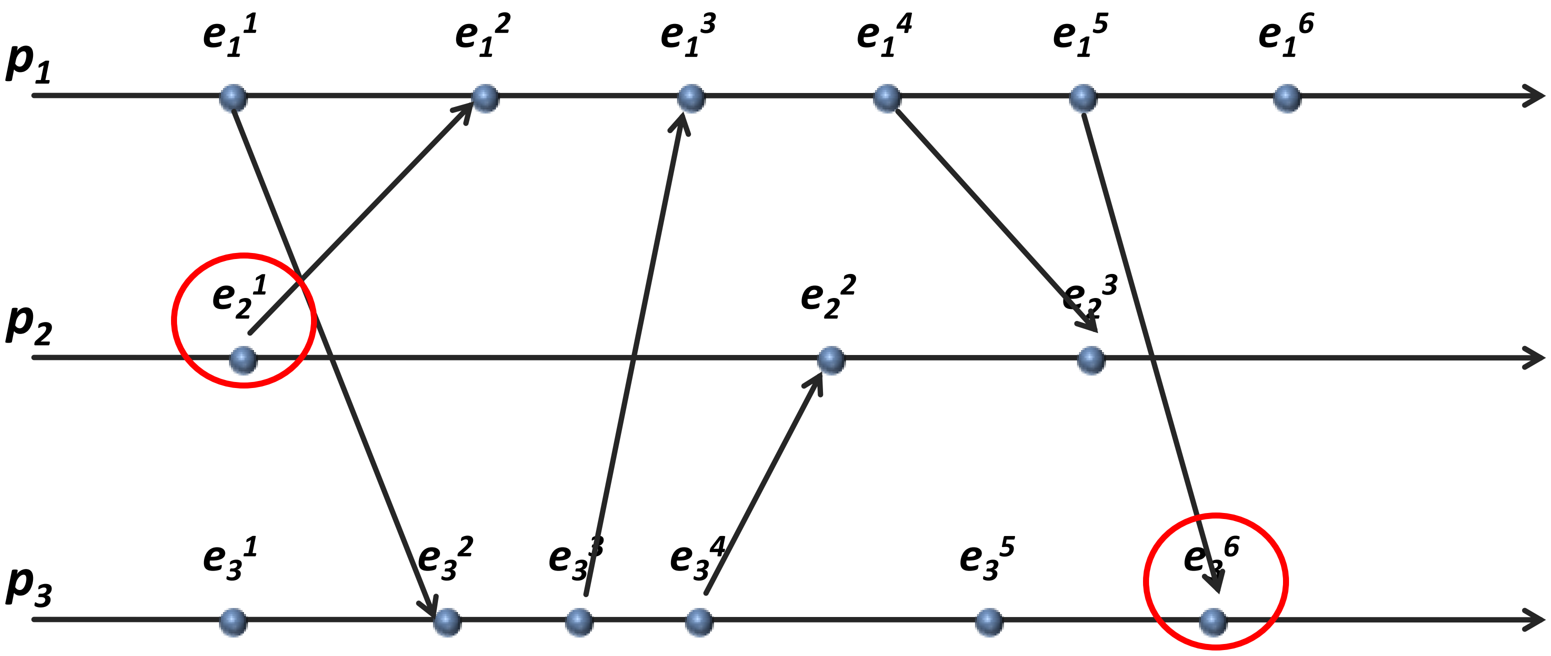
Sending a message always precedes receipt of that message:

If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$

Event ordering is associative:

If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Space-time Diagram for Distributed Computation



e_2^1 e_3^6

local events precede one another → precede one another globally:

If $e_i^k, e_i^m \in h_i$ and $k < m$, then $e_i^k \rightarrow e_i^m$

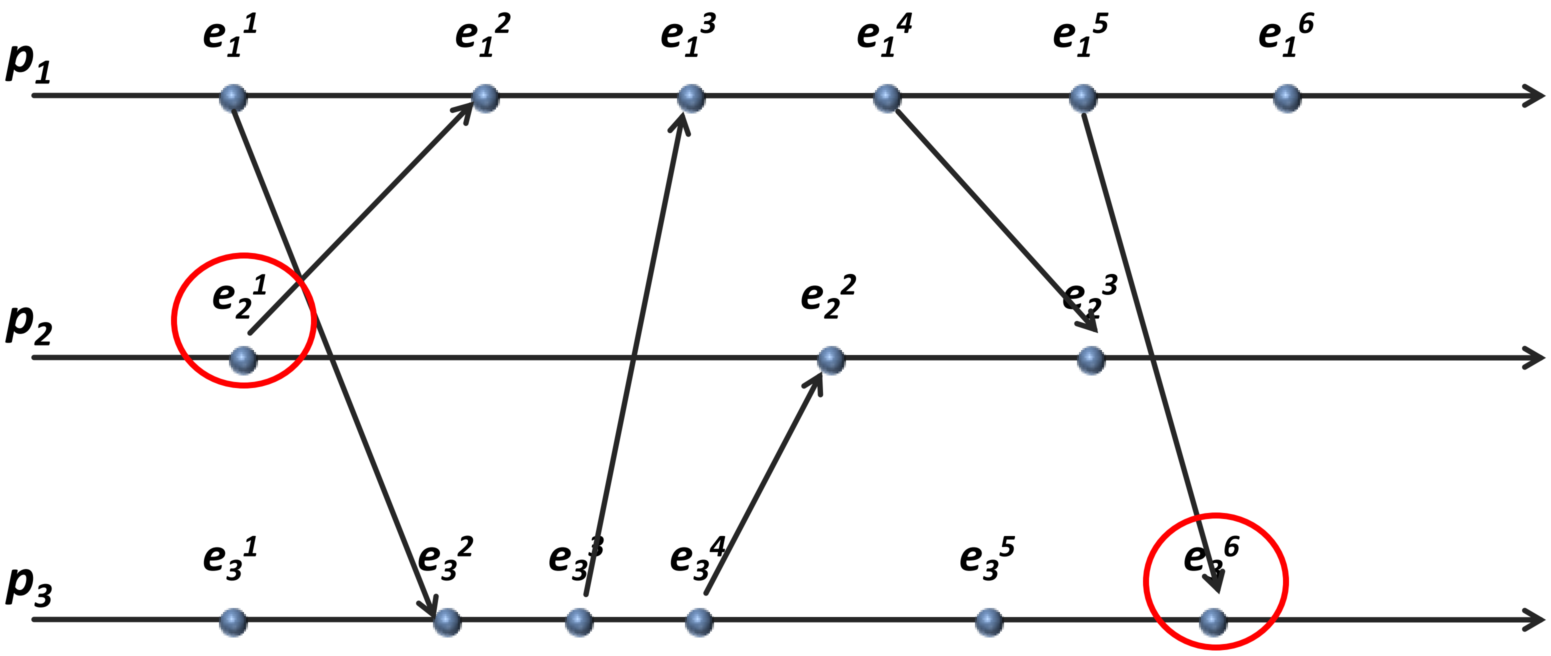
Sending a message always precedes receipt of that message:

If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$

Event ordering is associative:

If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Space-time Diagram for Distributed Computation



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

If $e_i^k, e_i^m \in h_i$ and $k < m$, then $e_i^k \rightarrow e_i^m$

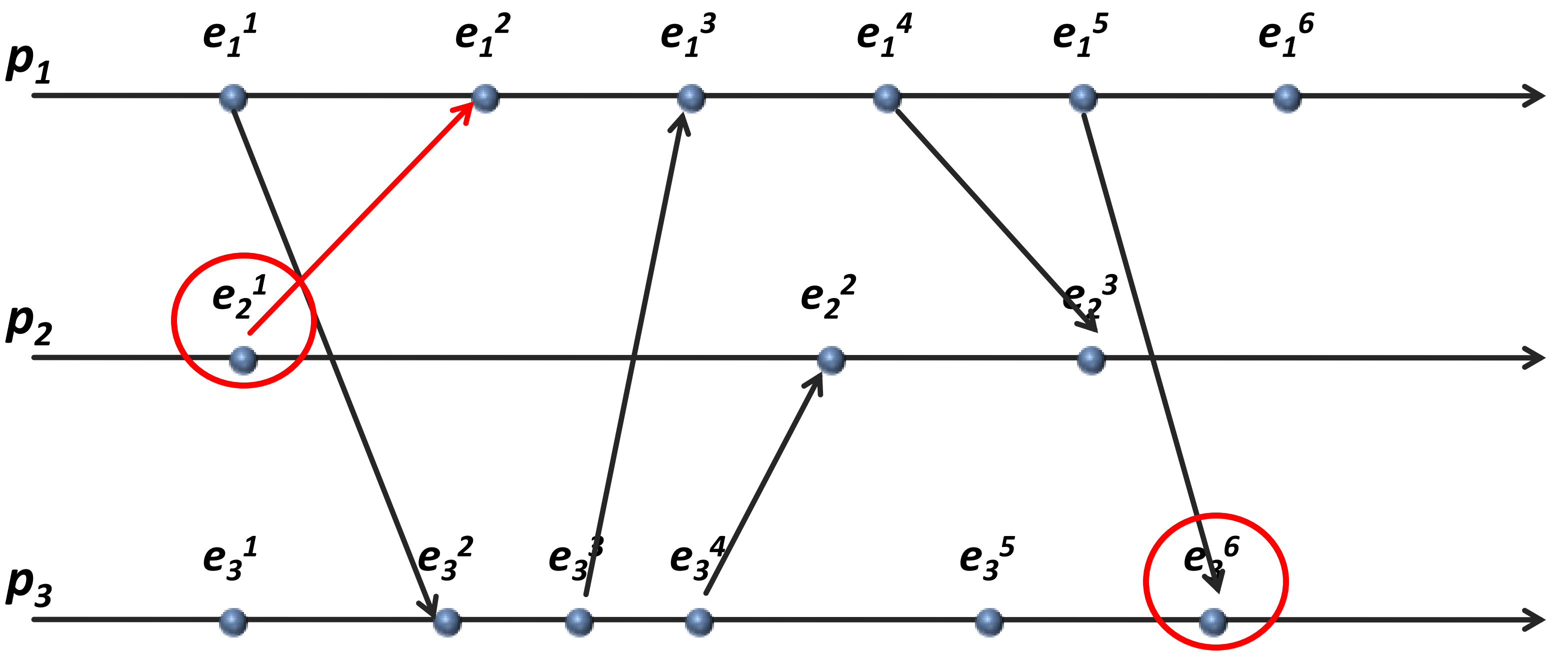
Sending a message always precedes receipt of that message:

If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$

Event ordering is associative:

If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Space-time Diagram for Distributed Computation



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

If $e_i^k, e_i^m \in h_i$ and $k < m$, then $e_i^k \rightarrow e_i^m$

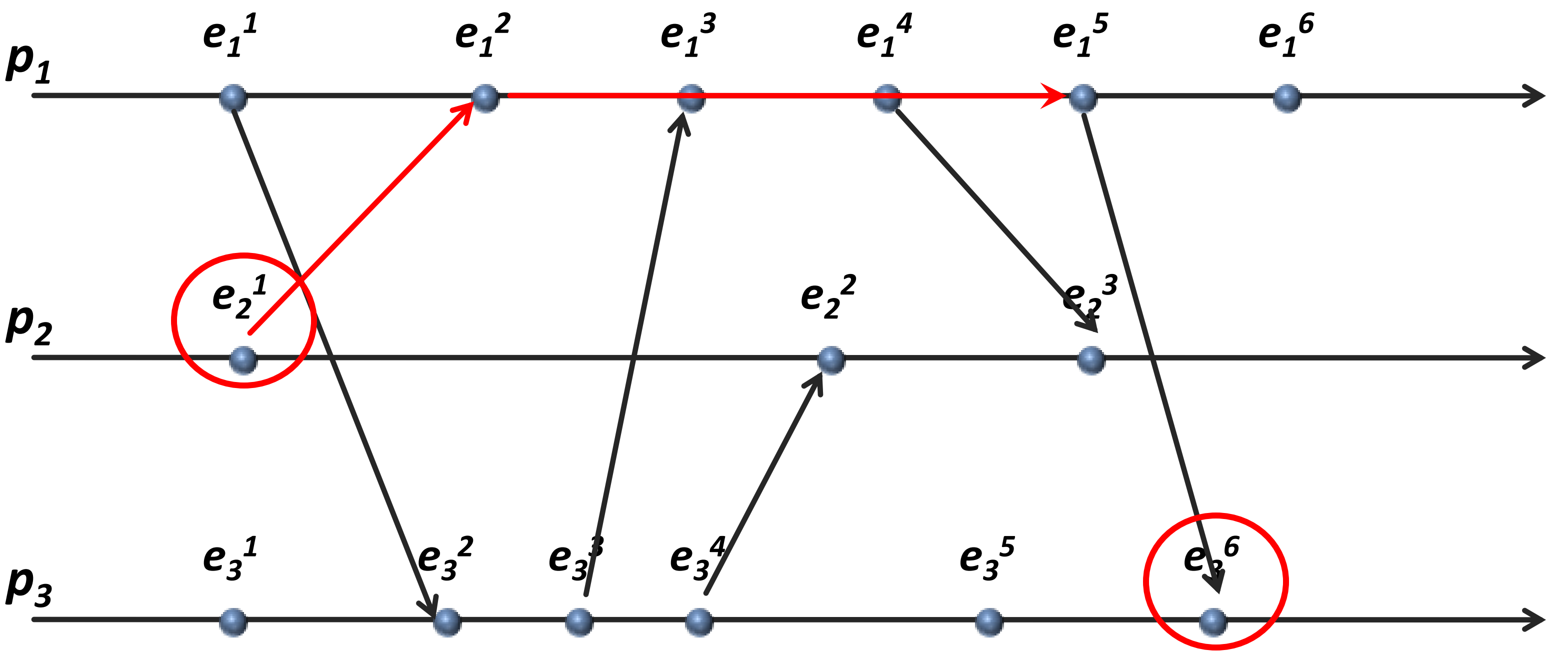
Sending a message always precedes receipt of that message:

If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$

Event ordering is associative:

If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Space-time Diagram for Distributed Computation



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

If $e_i^k, e_i^m \in h_i$ and $k < m$, then $e_i^k \rightarrow e_i^m$

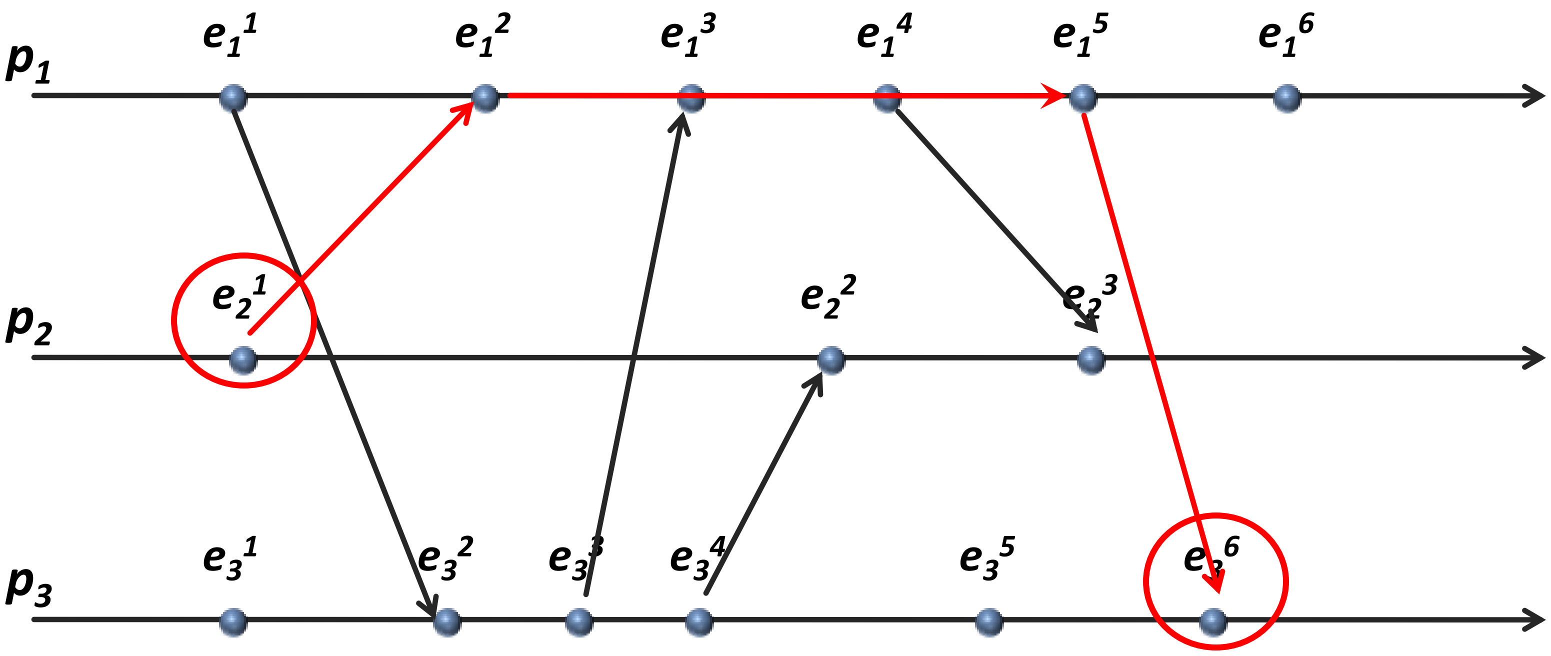
Sending a message always precedes receipt of that message:

If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$

Event ordering is associative:

If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Space-time Diagram for Distributed Computation



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

If $e_i^k, e_i^m \in h_i$ and $k < m$, then $e_i^k \rightarrow e_i^m$

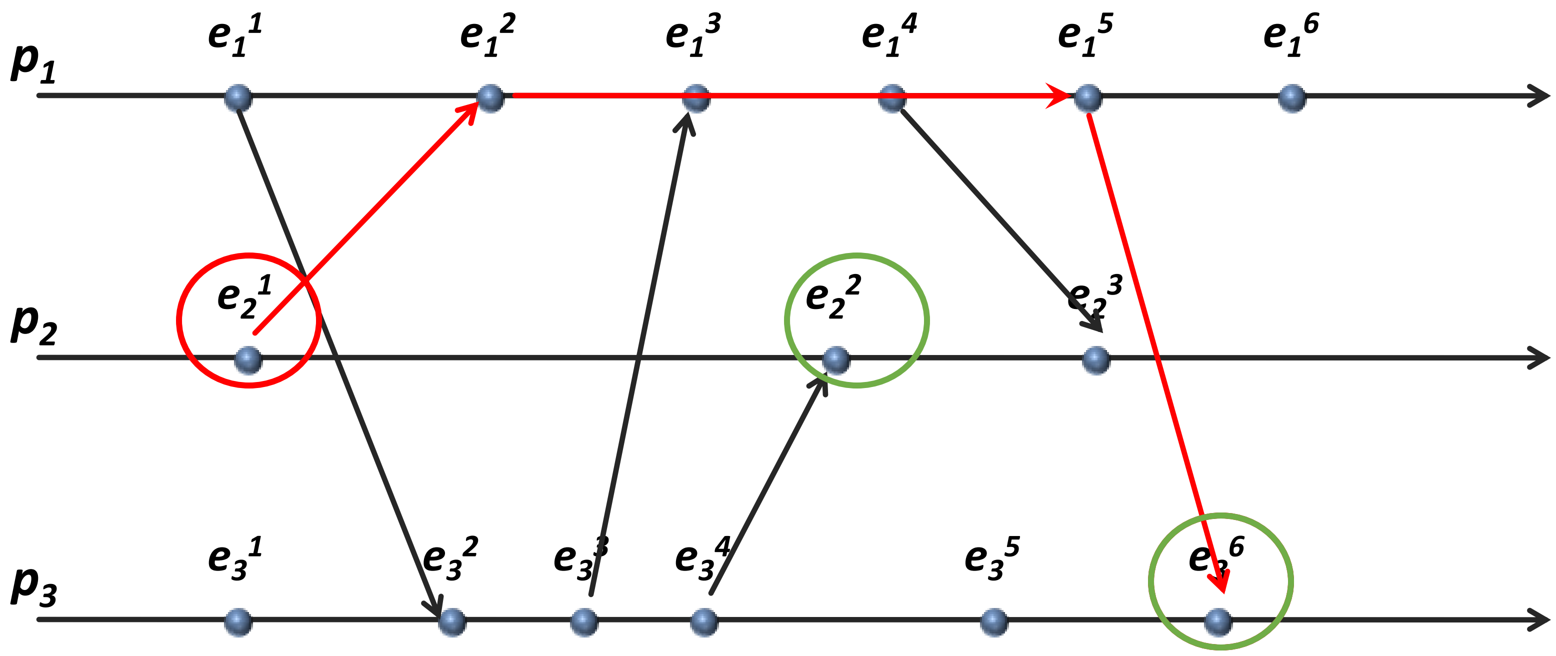
Sending a message always precedes receipt of that message:

If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$

Event ordering is associative:

If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Space-time Diagram for Distributed Computation



$e_2^1 \rightarrow e_3^6$

e_2^2 e_3^6

local events precede one another → precede one another globally:

If $e_i^k, e_i^m \in h_i$ *and* $k < m$, *then* $e_i^k \rightarrow e_i^m$

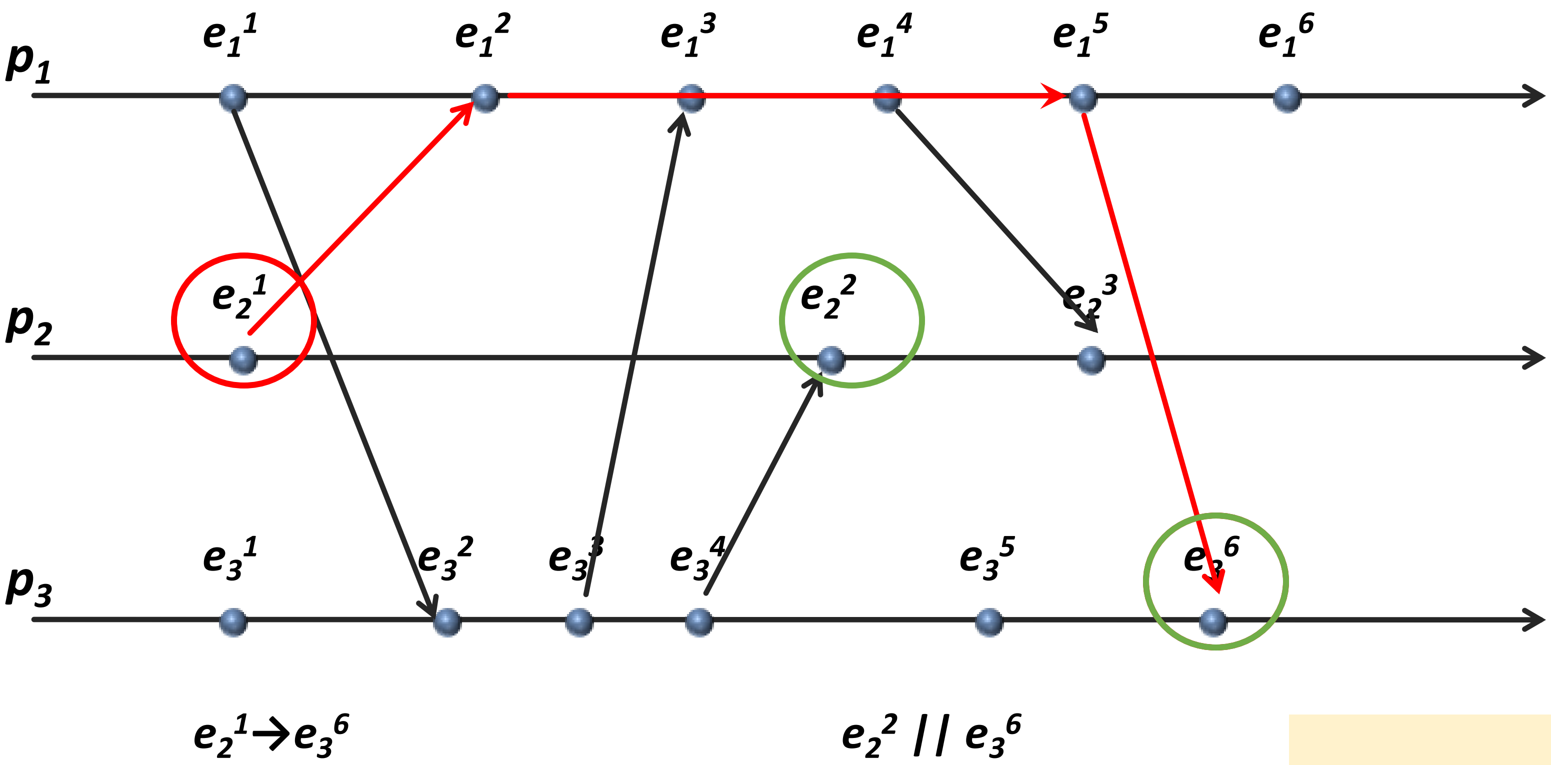
Sending a message always precedes receipt of that message:

If $e_i = send(m)$ *and* $e_j = receive(m)$, *then* $e_i \rightarrow e_j$

Event ordering is associative:

If $e \rightarrow e'$ *and* $e' \rightarrow e''$, *then* $e \rightarrow e''$

Space-time Diagram for Distributed Computation



local events precede one another → precede one another globally:

If $e_i^k, e_i^m \in h_i$ and $k < m$, then $e_i^k \rightarrow e_i^m$

Sending a message always precedes receipt of that message:

If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$

Event ordering is associative:

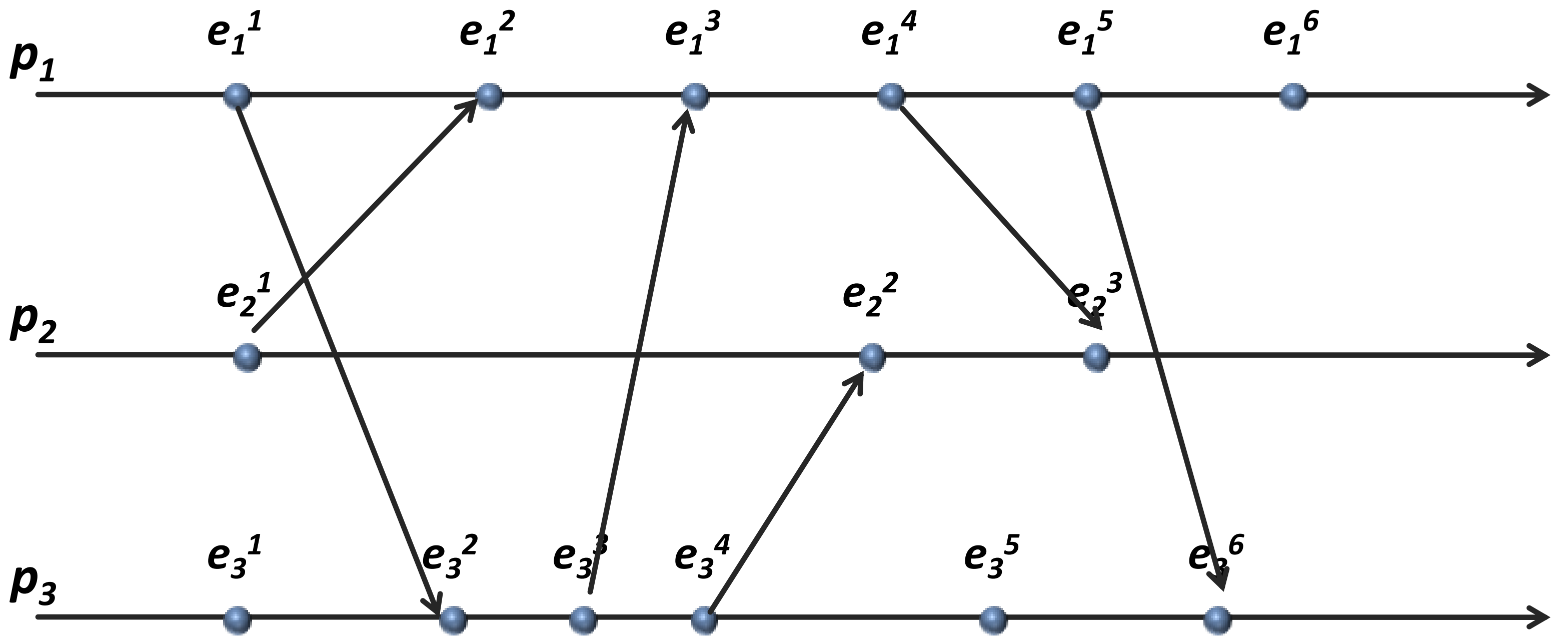
If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Cuts of a Distributed Computation

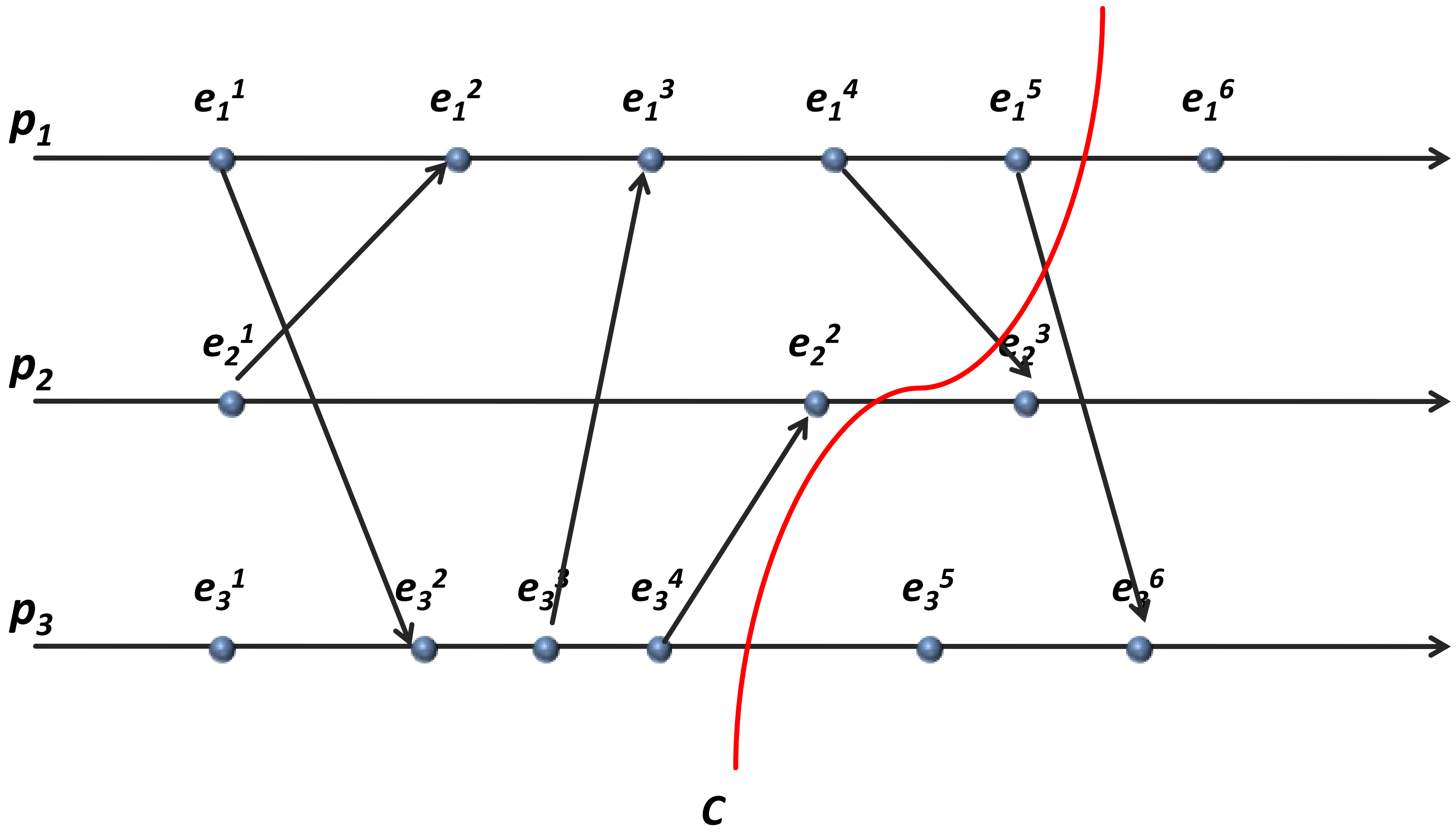
- Suppose there is an ***external monitor*** process
- External monitor constructs a global state:
  - Asks processes to send it local history
- Global state constructed from these local histories is:

a ***cut of a distributed computation***

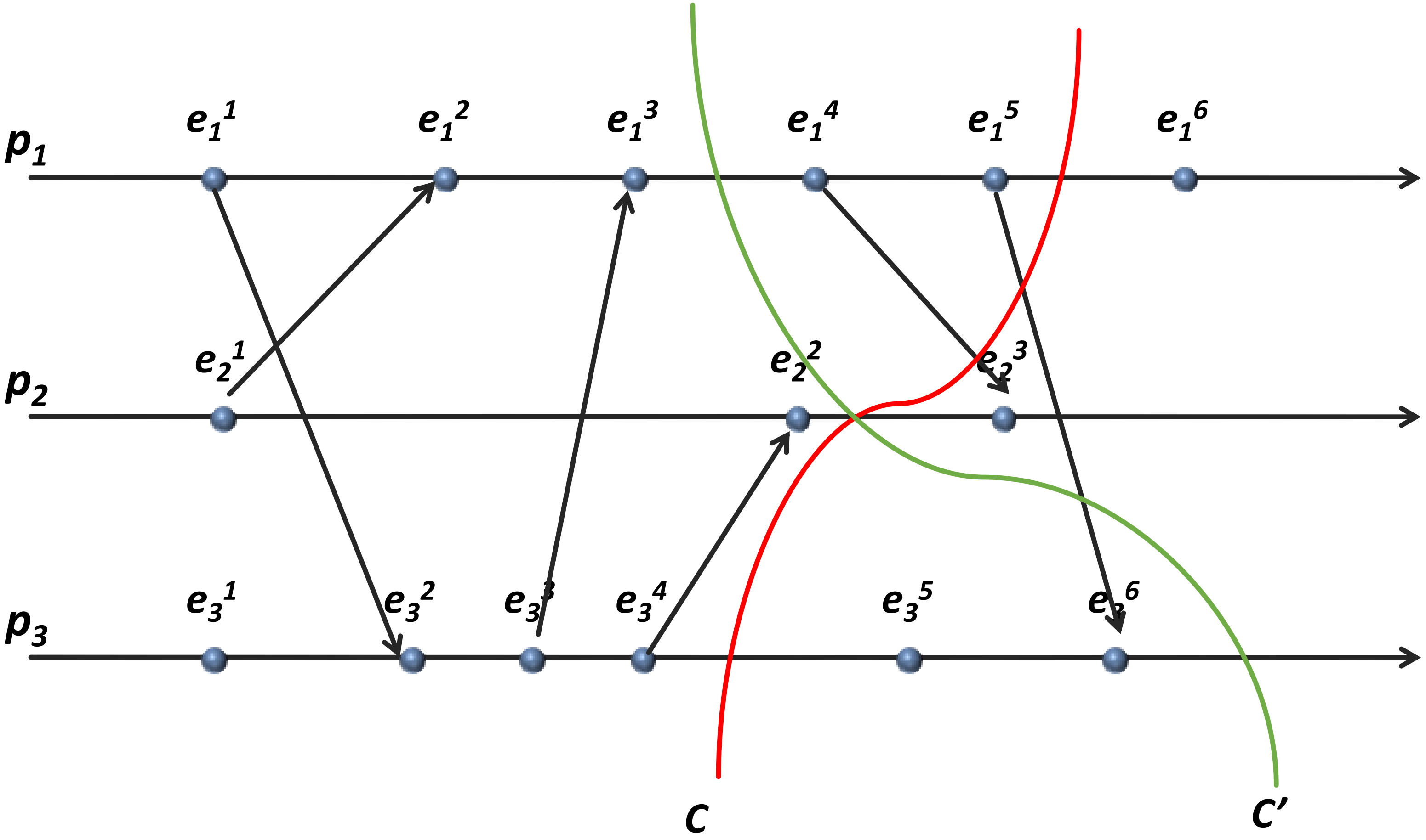
Example Cuts



Example Cuts



Example Cuts

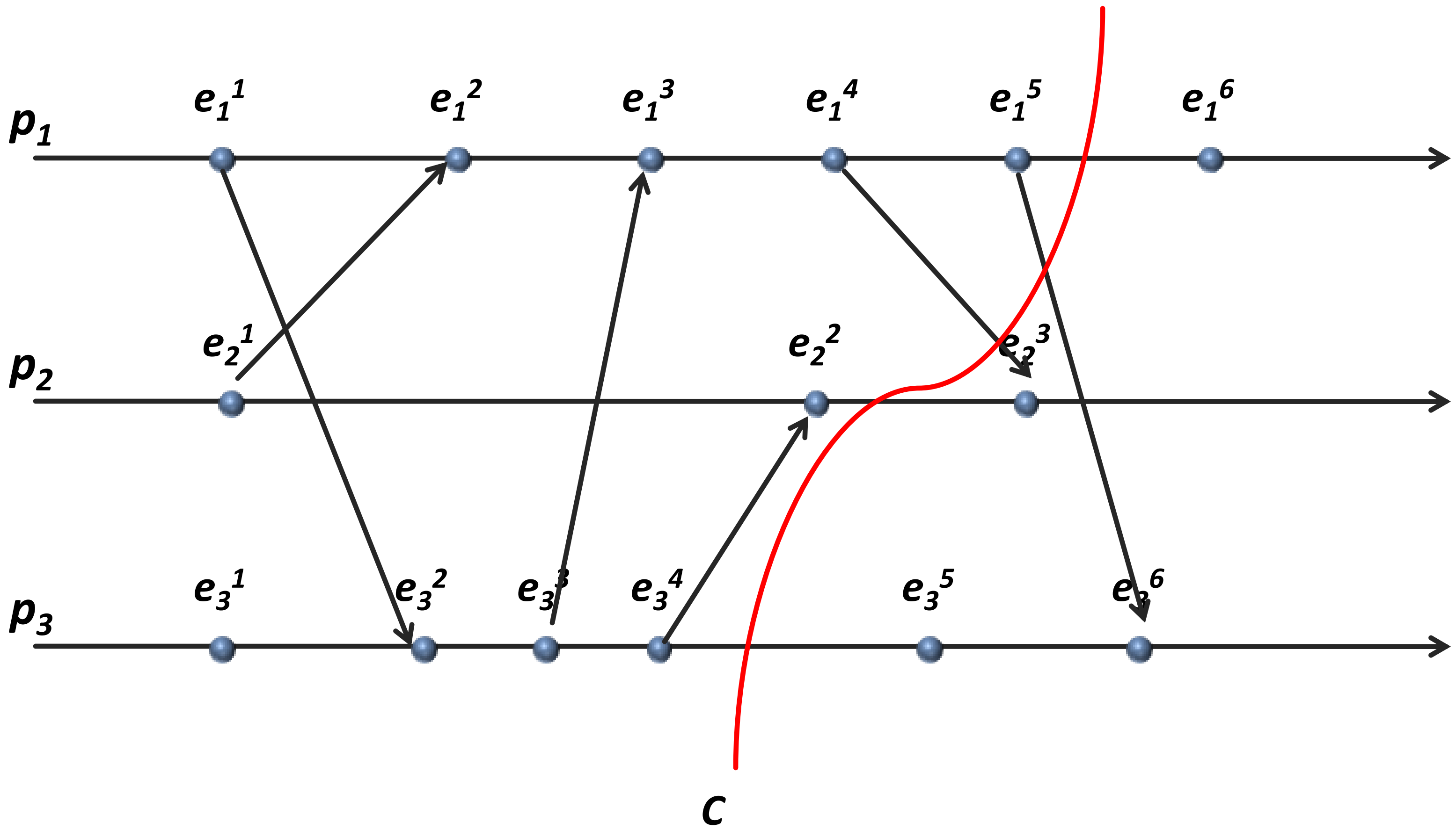


Consistent vs. Inconsistent Cuts

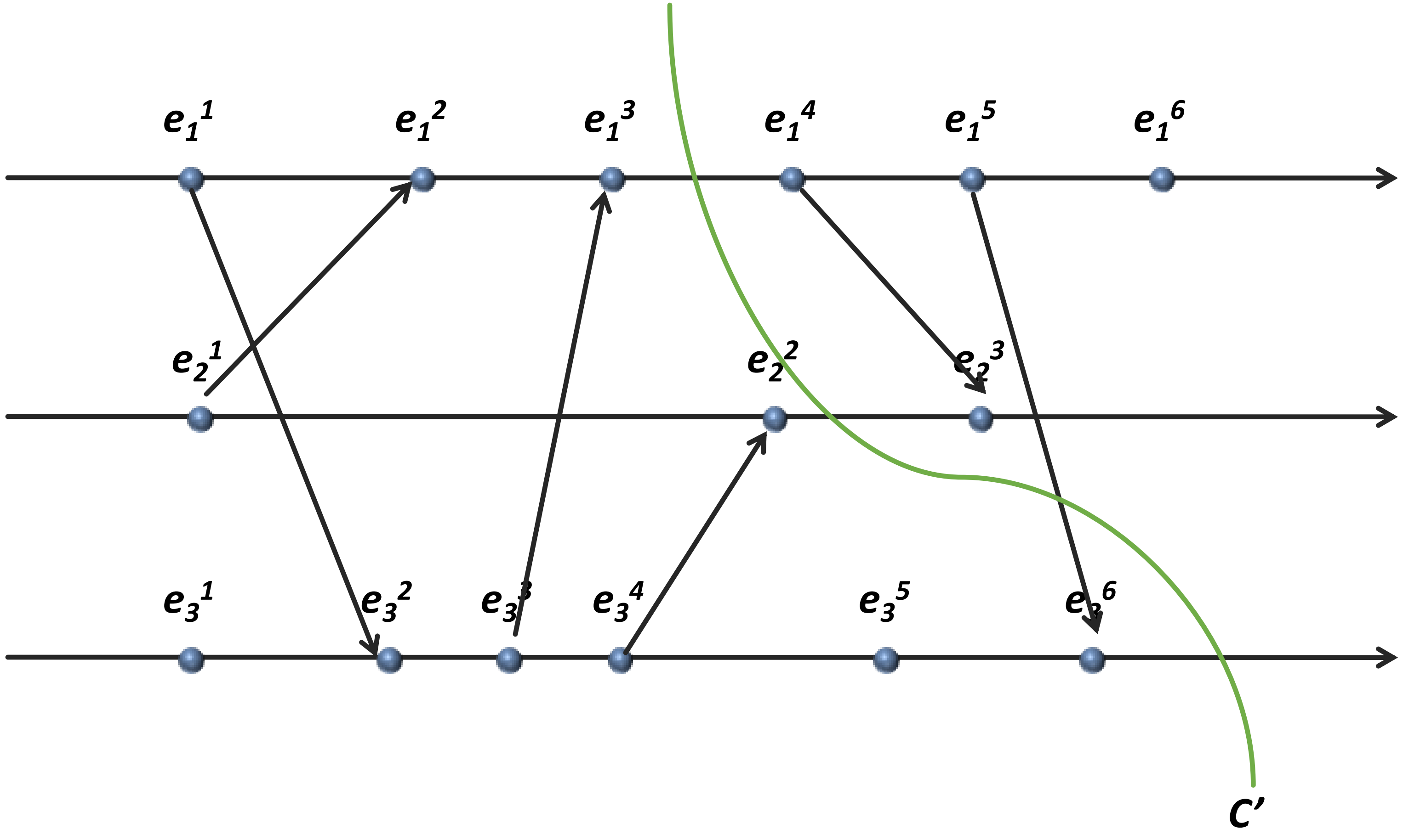
- A cut is consistent if
  - for any event *e* included in the cut
  - any event *e'* that causally precedes *e* is also included in that cut
- For cut *C:*

  $(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$

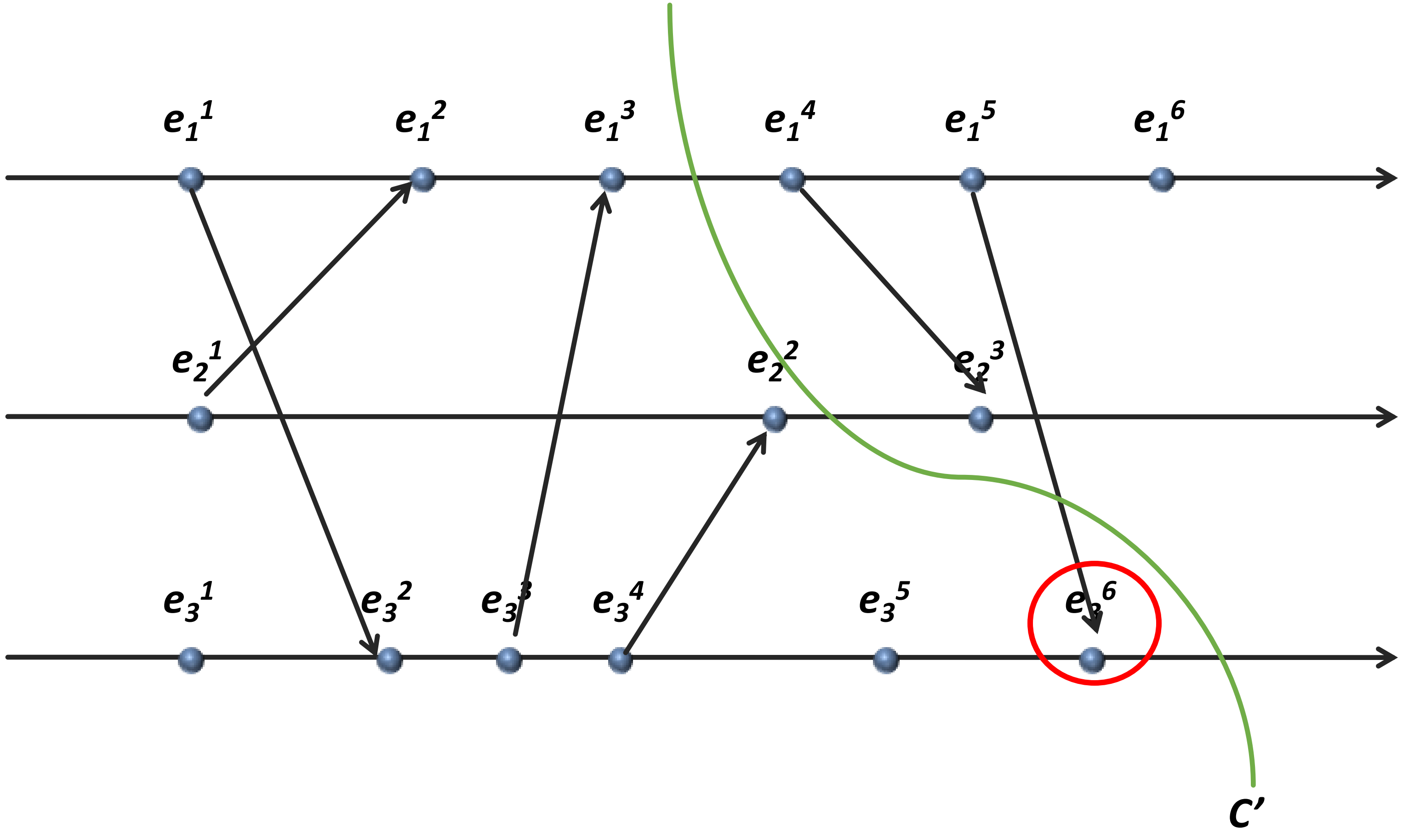
Are These Cuts Consistent?



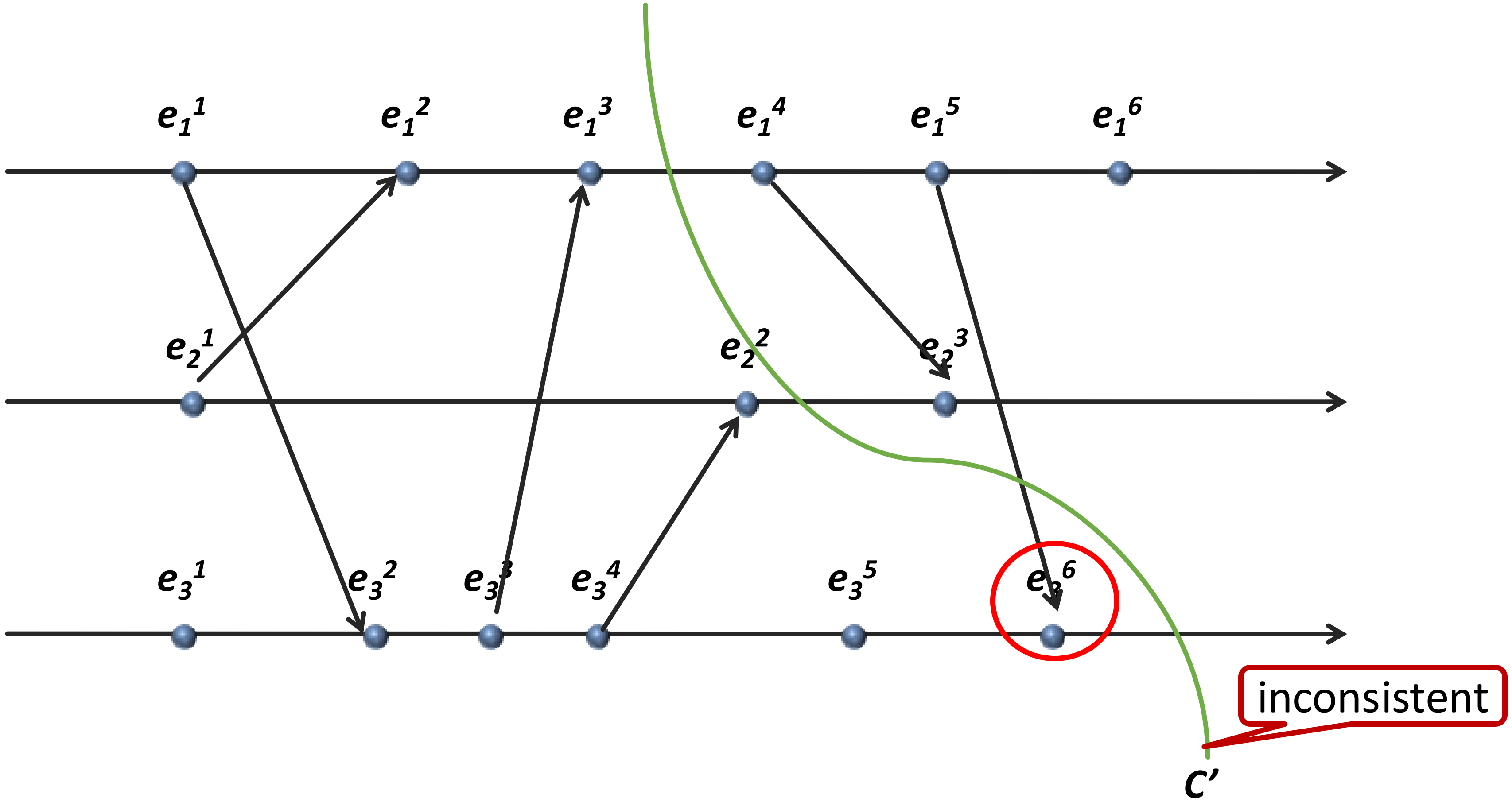
Are These Cuts Consistent?



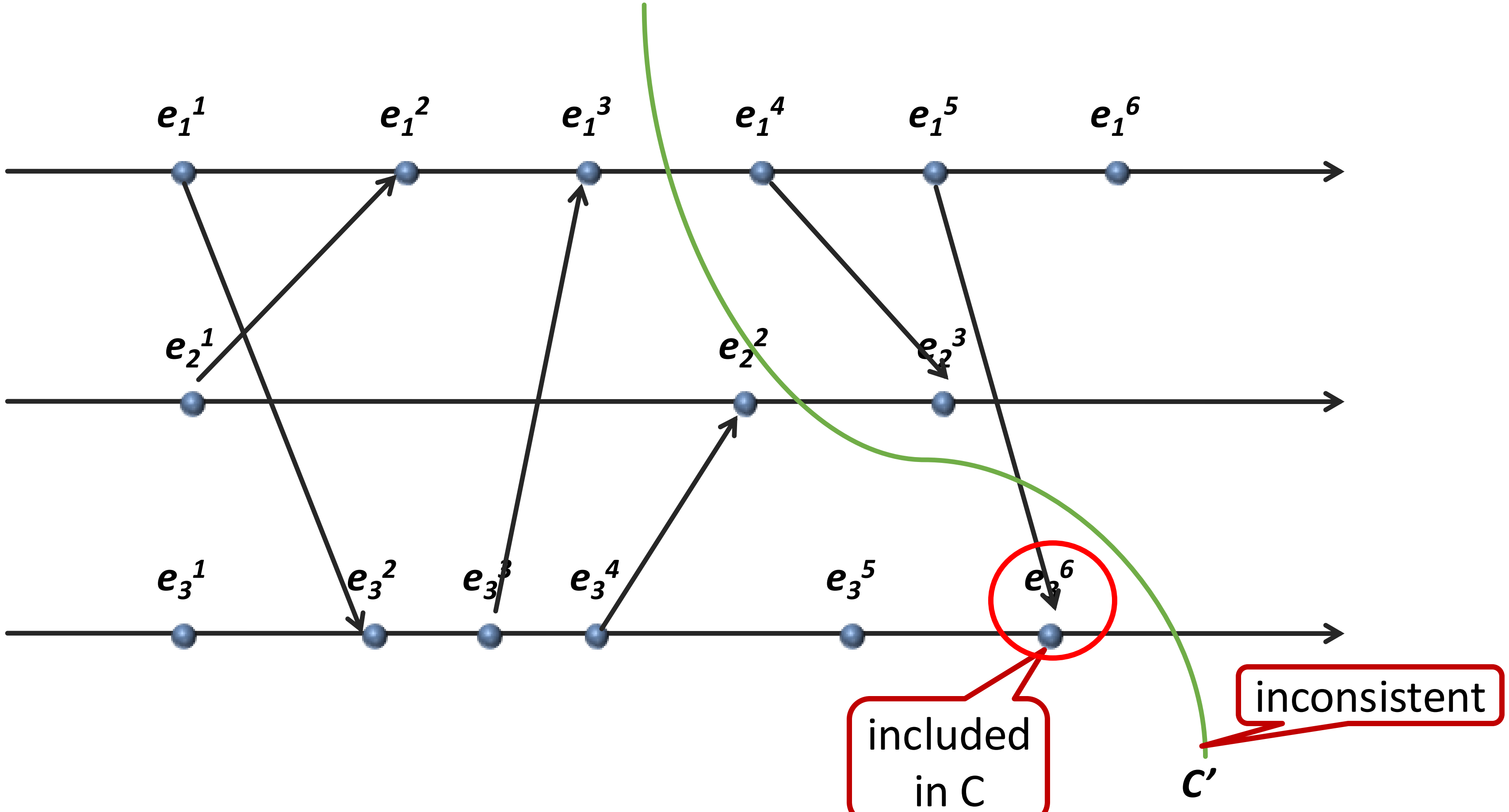
Are These Cuts Consistent?



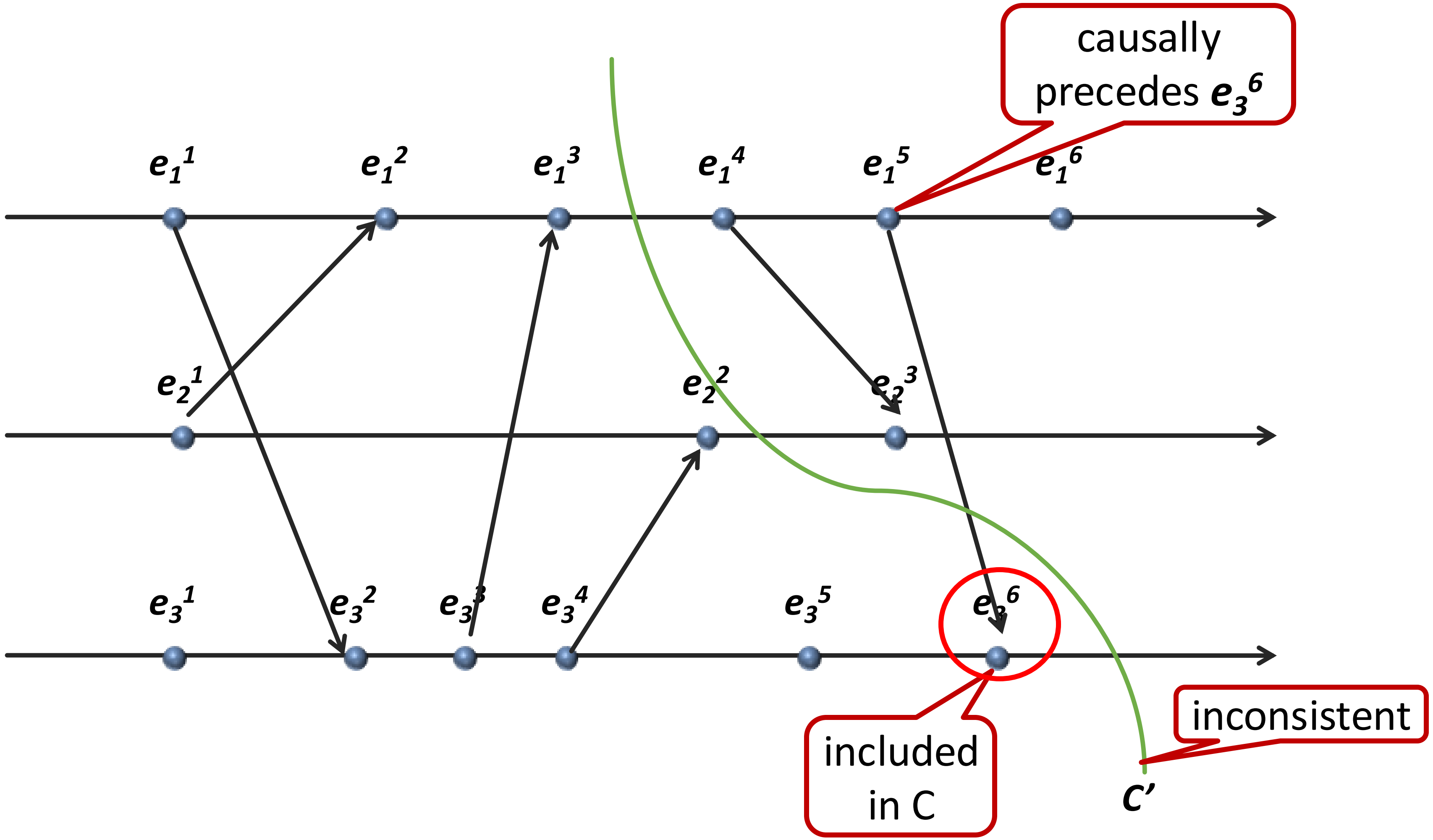
Are These Cuts Consistent?



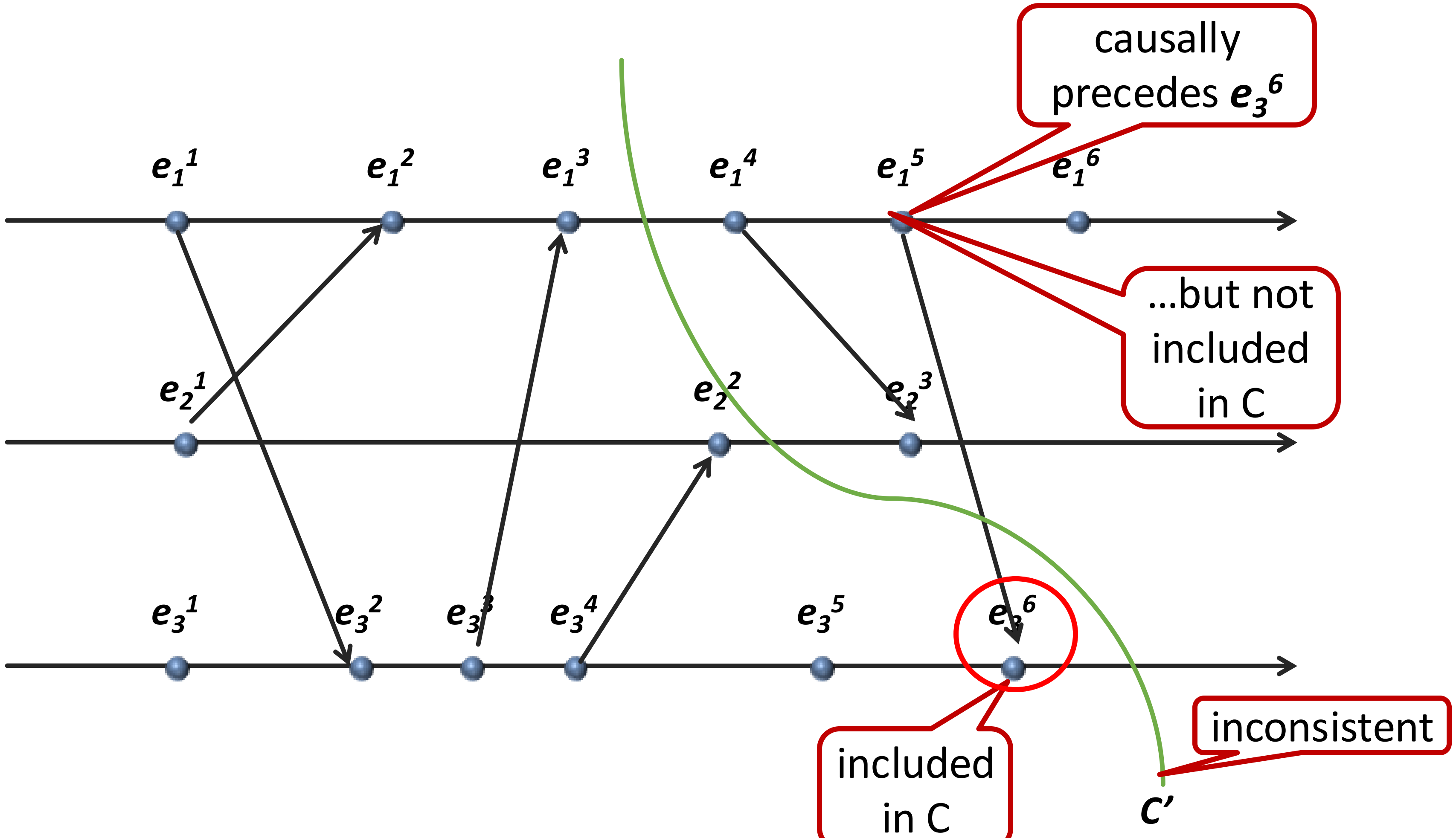
Are These Cuts Consistent?



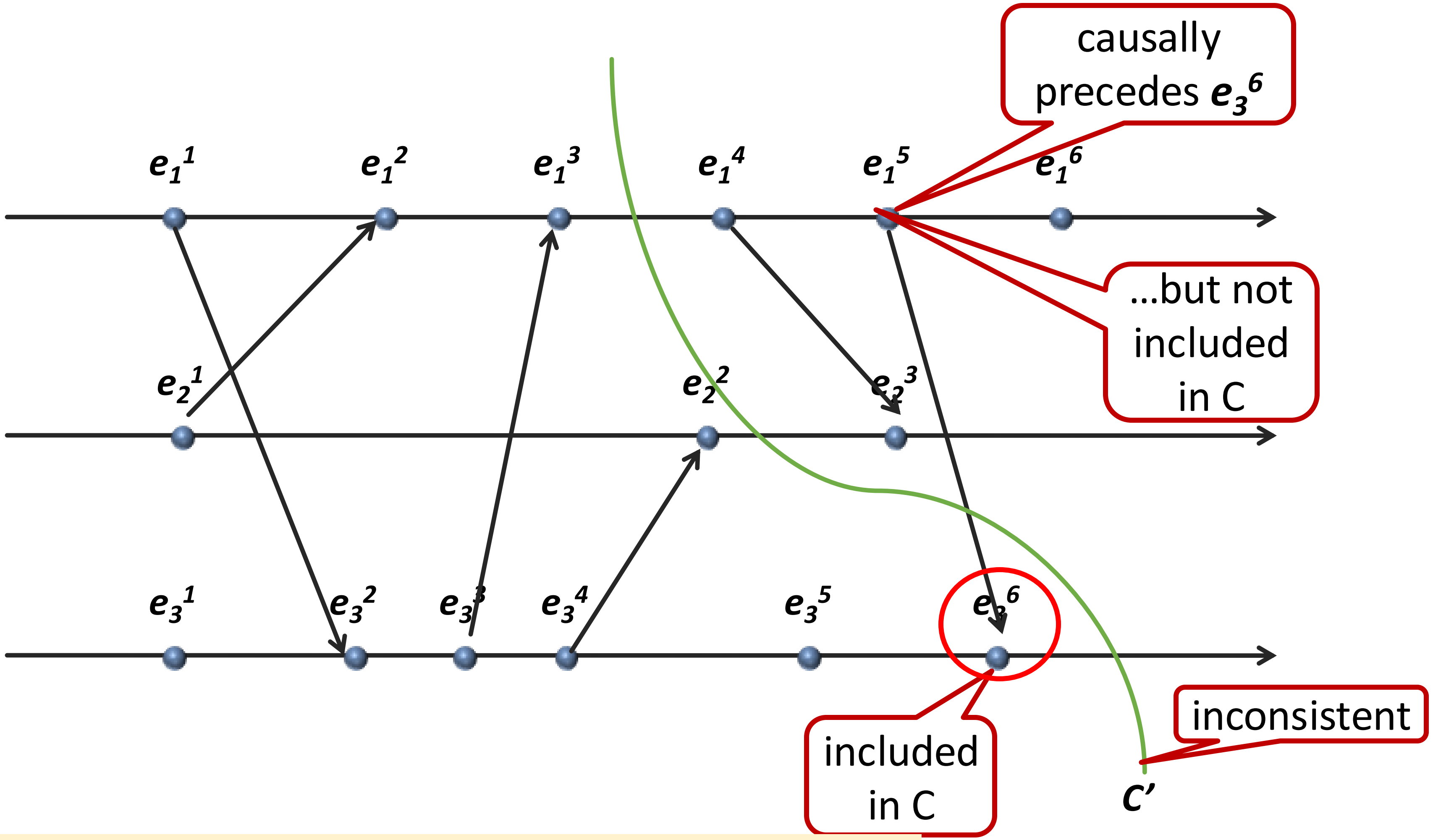
Are These Cuts Consistent?



Are These Cuts Consistent?

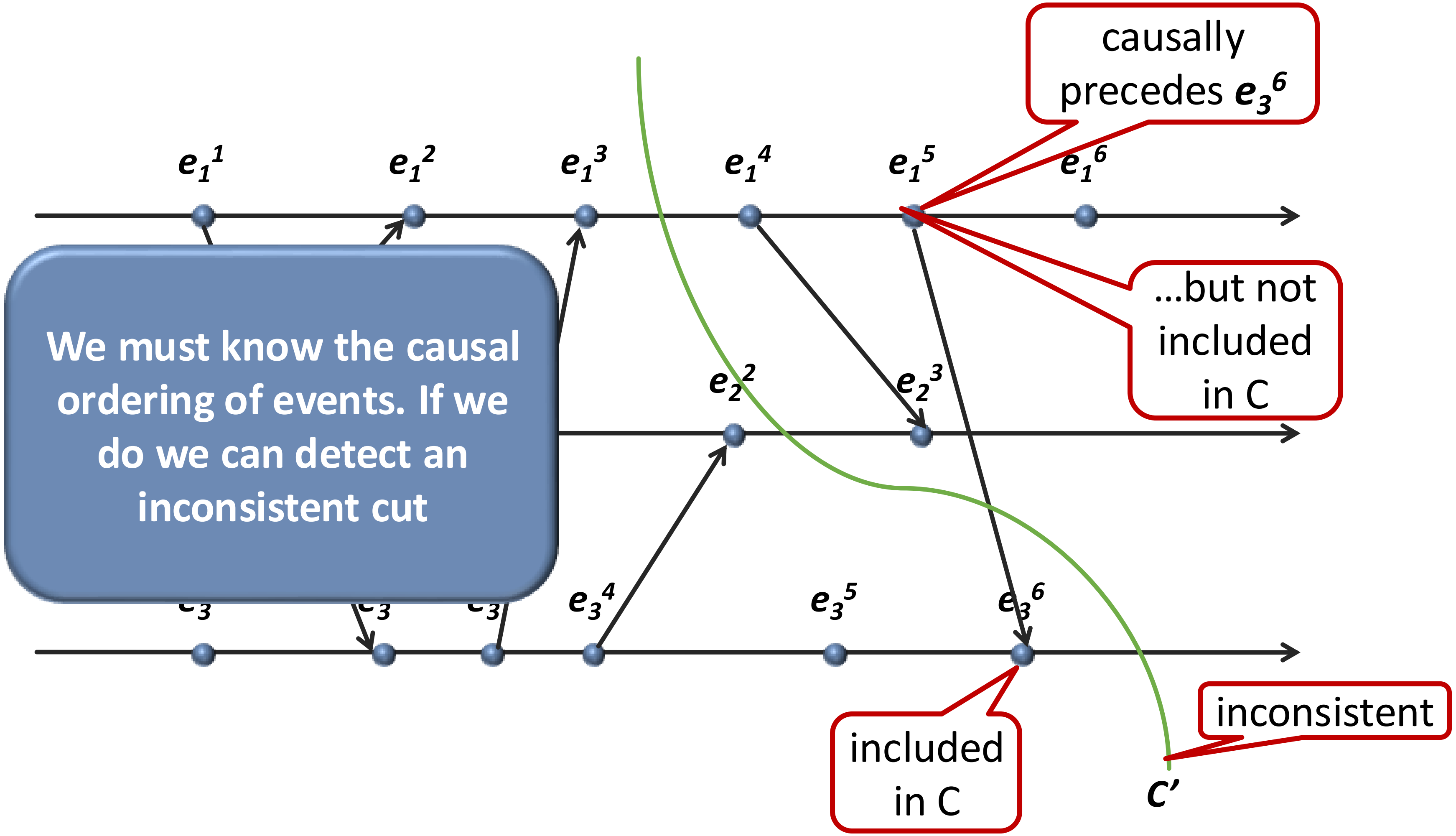


Are These Cuts Consistent?



A consistent cut corresponds to a consistent global state

What Do We Need to Know to Construct a Consistent Cut?



Logical Clocks

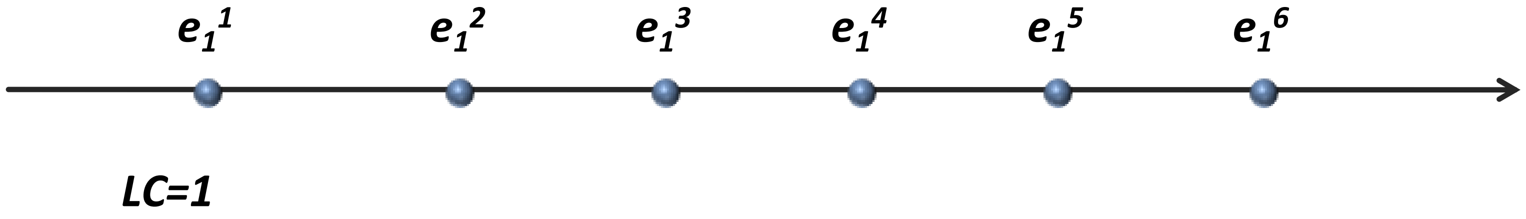
- Each process maintains a local value of a logical clock ***LC***
- Logical clock of process *p* counts **how many events in a distributed computation causally preceded the current event at *p*** (including the current event).
- $LC(e_i)$ – the logical clock value at process p_i at event e_i
- Suppose we had a distributed system with only a single process

Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of process *p* counts **how many events in a distributed computation causally preceded the current event at *p*** (including the current event).
- $LC(e_i)$ – the logical clock value at process p_i at event e_i
- Suppose we had a distributed system with only a single process

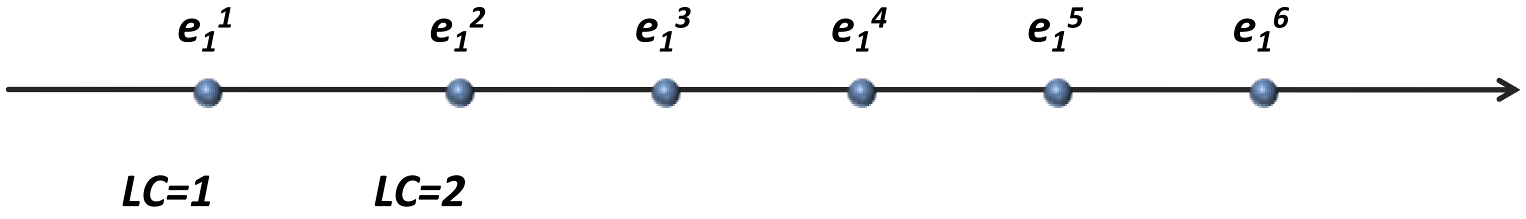
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of process *p* counts **how many events in a distributed computation causally preceded the current event at *p*** (including the current event).
- $LC(e_i)$ – the logical clock value at process p_i at event e_i
- Suppose we had a distributed system with only a single process



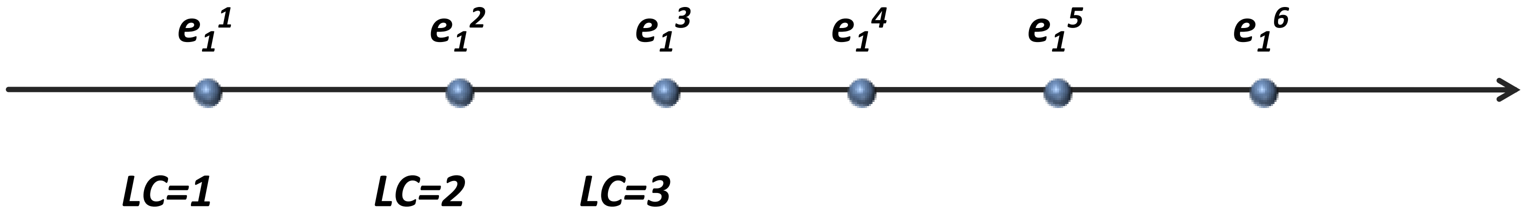
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of process *p* counts **how many events in a distributed computation causally preceded the current event at *p*** (including the current event).
- $LC(e_i)$ – the logical clock value at process p_i at event e_i
- Suppose we had a distributed system with only a single process



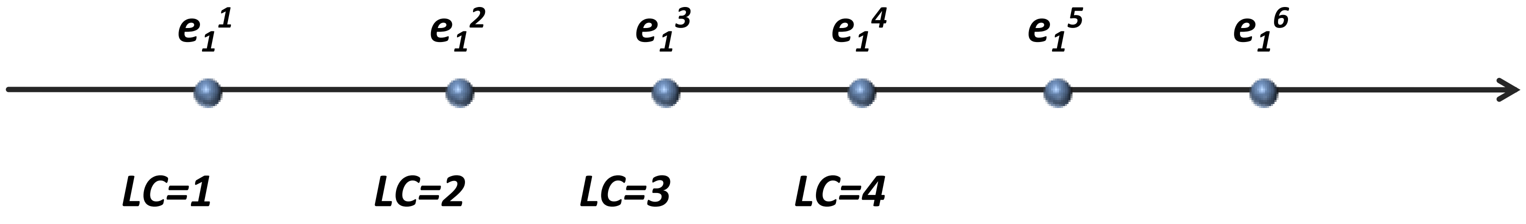
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of process *p* counts **how many events in a distributed computation causally preceded the current event at *p*** (including the current event).
- $LC(e_i)$ – the logical clock value at process p_i at event e_i
- Suppose we had a distributed system with only a single process



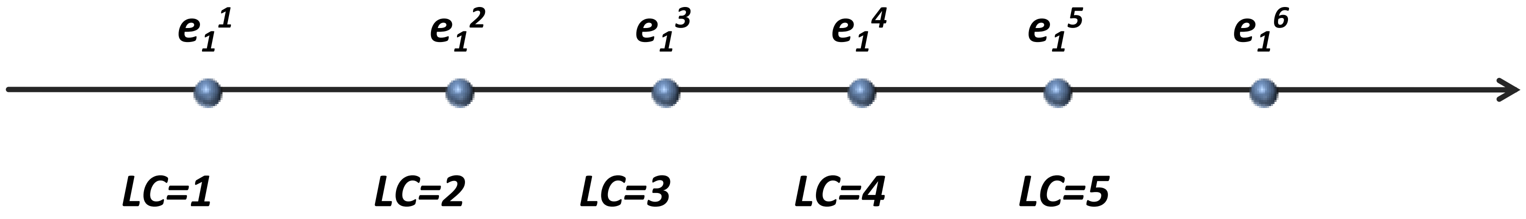
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of process *p* counts **how many events in a distributed computation causally preceded the current event at *p*** (including the current event).
- $LC(e_i)$ – the logical clock value at process p_i at event e_i
- Suppose we had a distributed system with only a single process



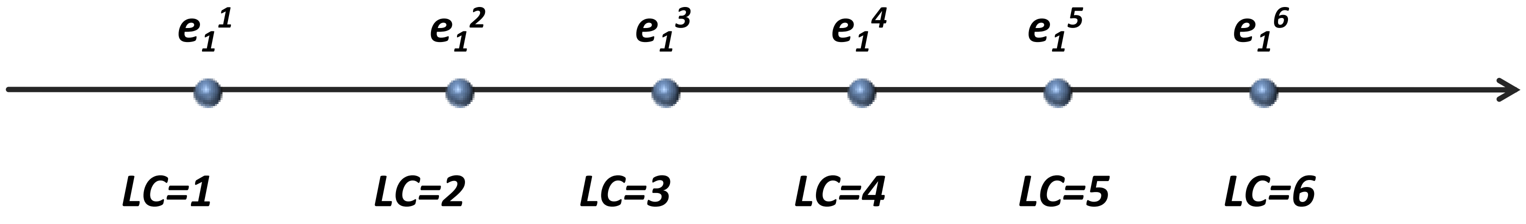
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of process *p* counts **how many events in a distributed computation causally preceded the current event at *p*** (including the current event).
- $LC(e_i)$ – the logical clock value at process p_i at event e_i
- Suppose we had a distributed system with only a single process



Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of process *p* counts **how many events in a distributed computation causally preceded the current event at *p*** (including the current event).
- $LC(e_i)$ – the logical clock value at process p_i at event e_i
- Suppose we had a distributed system with only a single process

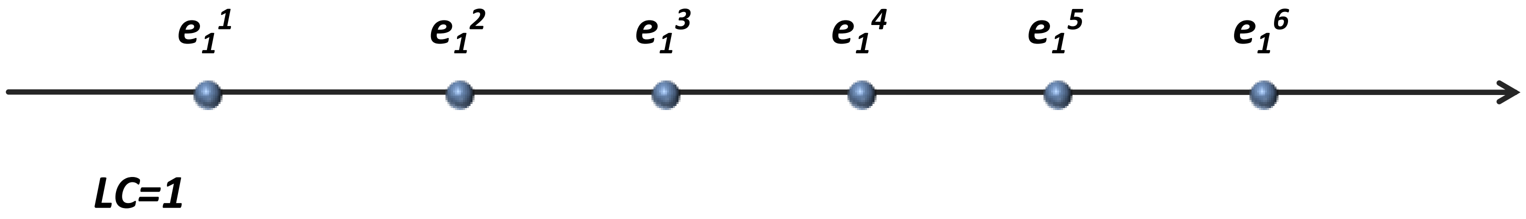


Logical Clocks (cont.)

- In a system with more than one process logical clocks are updated as follows:
- Each message m that is sent contains a timestamp TS(m)
- TS(m) is the logical clock value associated with sending event at the sending process

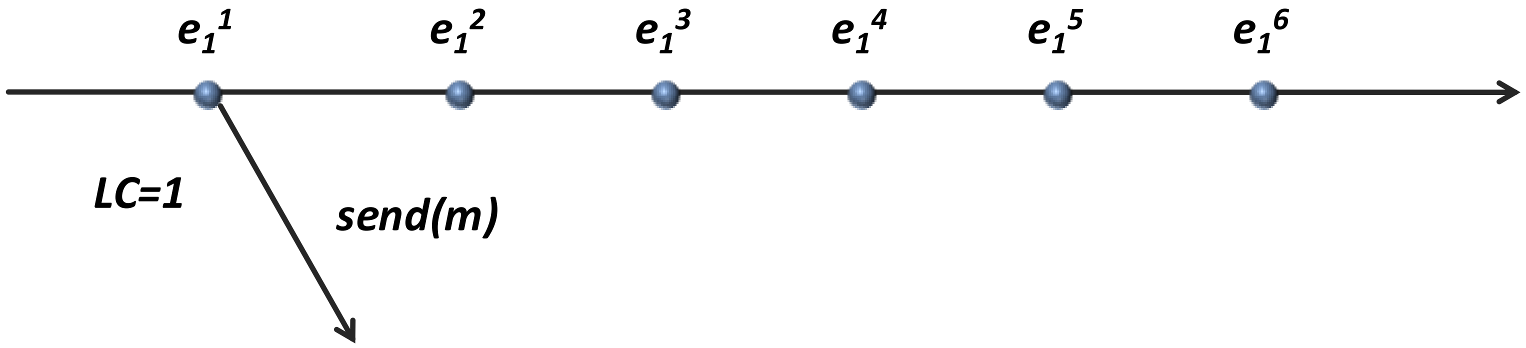
Logical Clocks (cont.)

- In a system with more than one process logical clocks are updated as follows:
- Each message m that is sent contains a timestamp TS(m)
- TS(m) is the logical clock value associated with sending event at the sending process



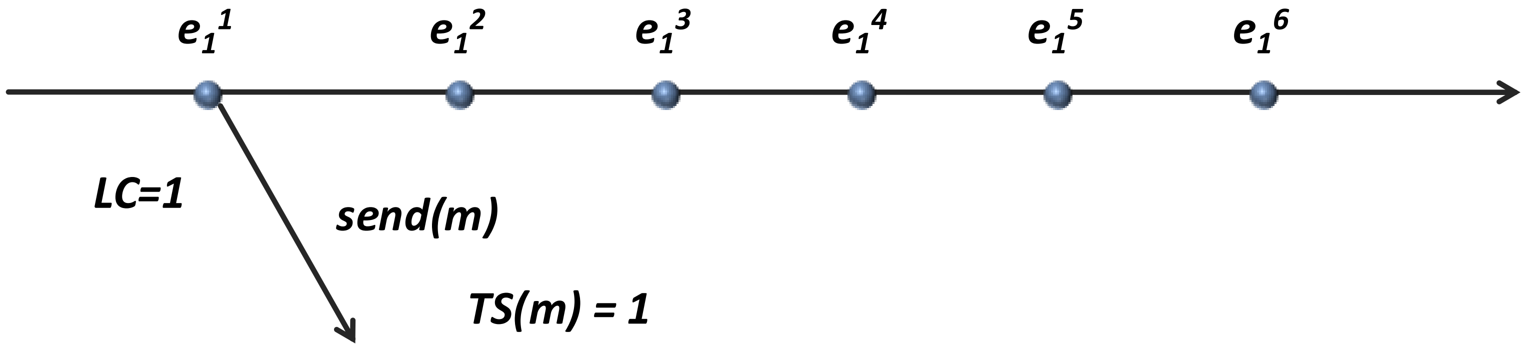
Logical Clocks (cont.)

- In a system with more than one process logical clocks are updated as follows:
- Each message m that is sent contains a timestamp TS(m)
- TS(m) is the logical clock value associated with sending event at the sending process



Logical Clocks (cont.)

- In a system with more than one process logical clocks are updated as follows:
- Each message m that is sent contains a timestamp TS(m)
- TS(m) is the logical clock value associated with sending event at the sending process



Logical Clocks (cont)

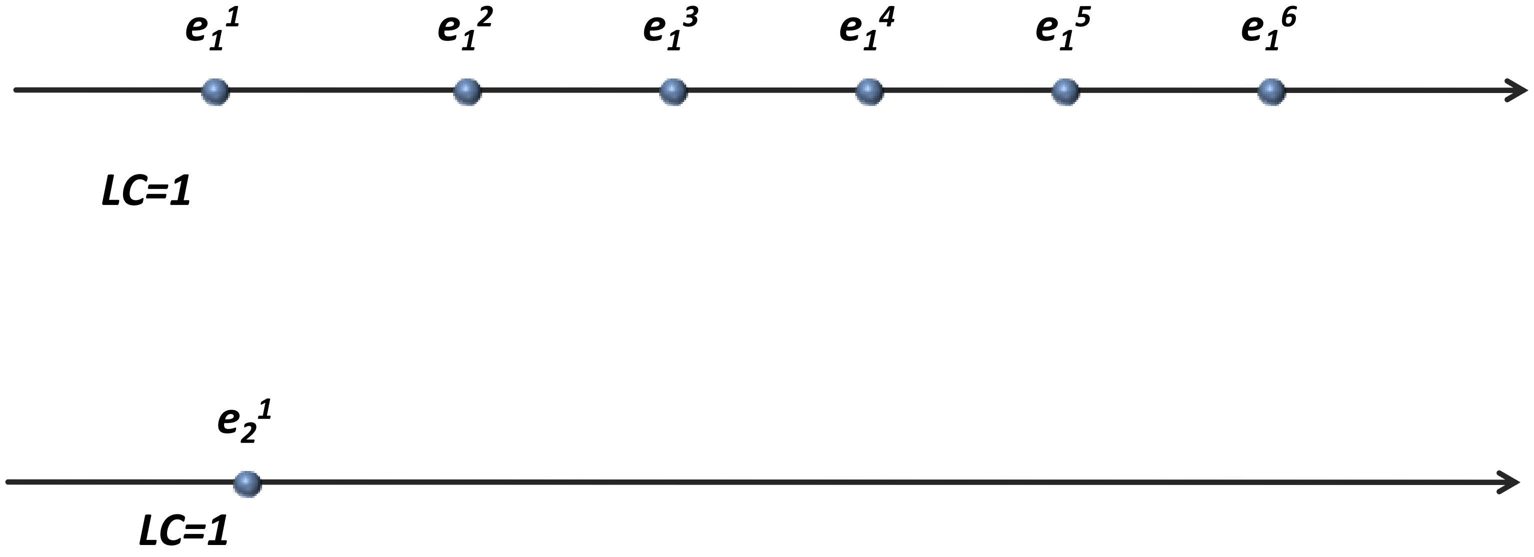
- When the receiving process receives message m, it updates its logical clock to:

$$max\{LC, TS(m)\} + 1$$

Logical Clocks (cont)

- When the receiving process receives message m, it updates its logical clock to:

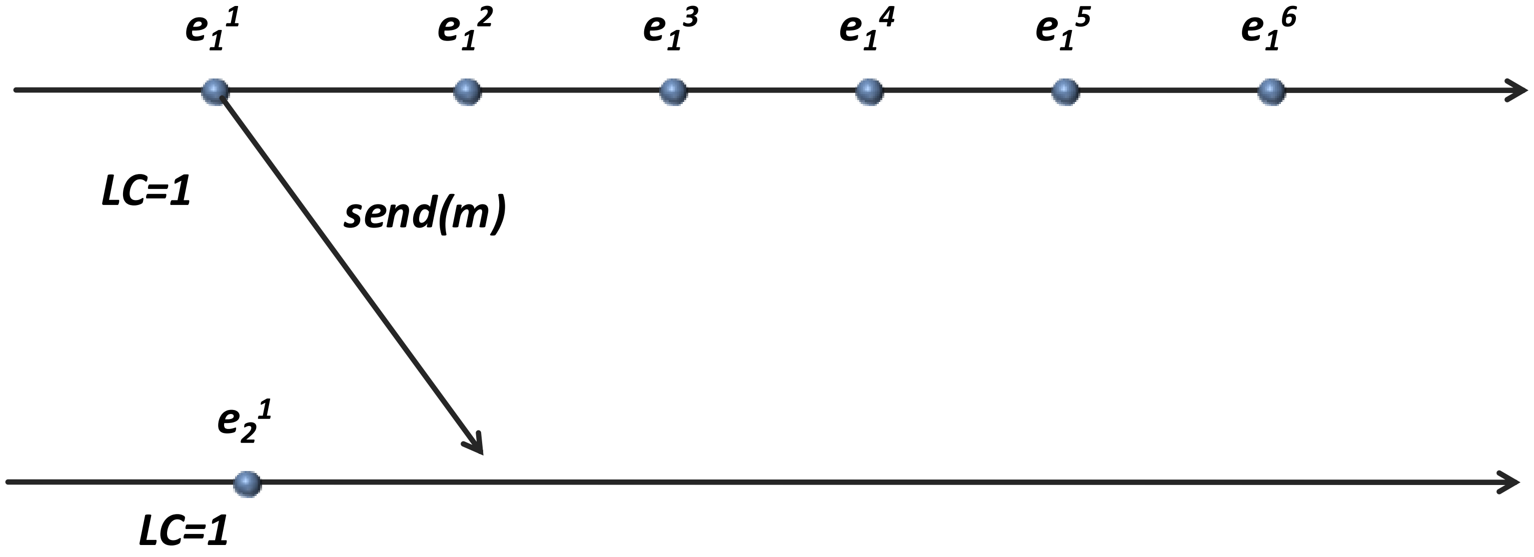
$$max\{LC, TS(m)\} + 1$$



Logical Clocks (cont)

- When the receiving process receives message m, it updates its logical clock to:

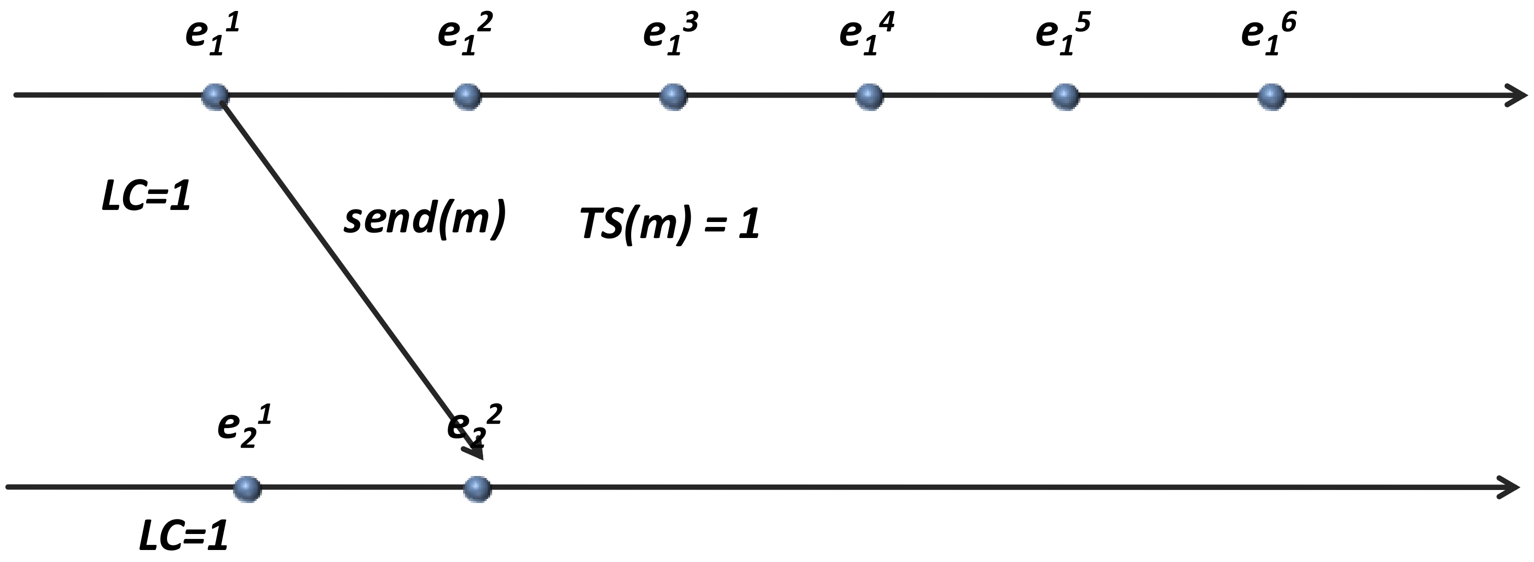
$$max\{LC, TS(m)\} + 1$$



Logical Clocks (cont)

- When the receiving process receives message m, it updates its logical clock to:

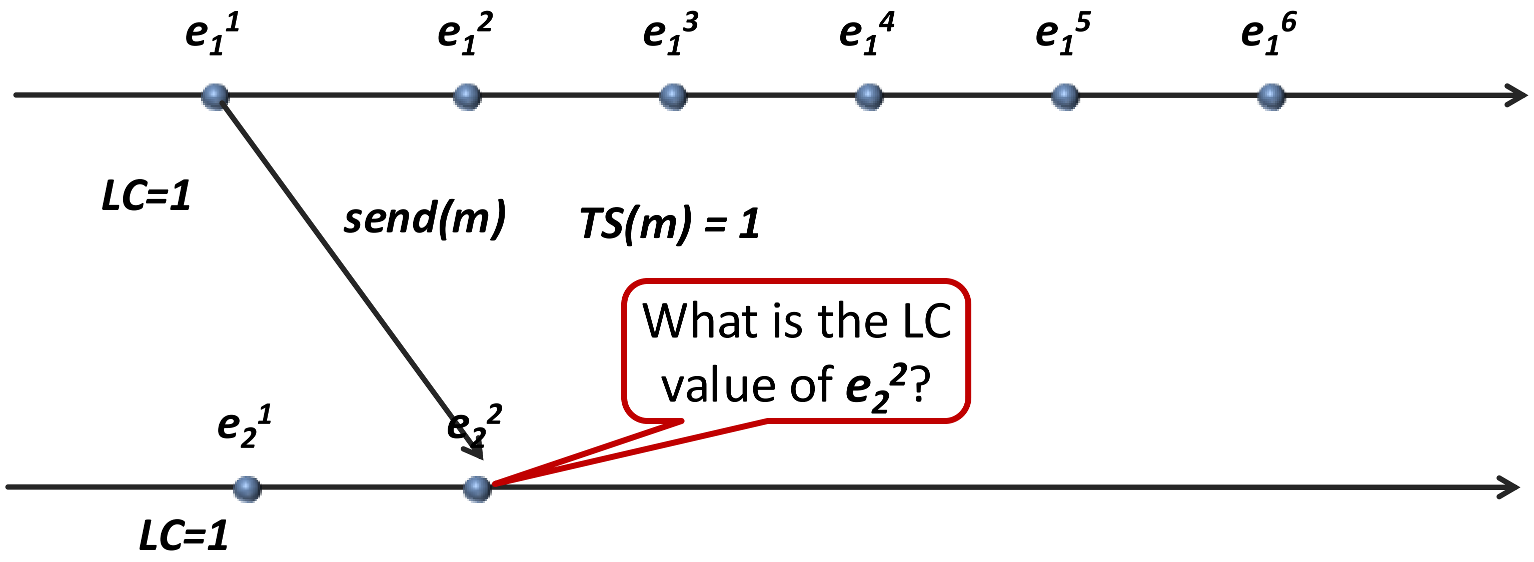
$$max\{LC, TS(m)\} + 1$$



Logical Clocks (cont)

- When the receiving process receives message m, it updates its logical clock to:

$$max\{LC, TS(m)\} + 1$$



Logical Clocks (cont)

- When the receiving process receives message m, it updates its logical clock to:

$$max\{LC, TS(m)\} + 1$$

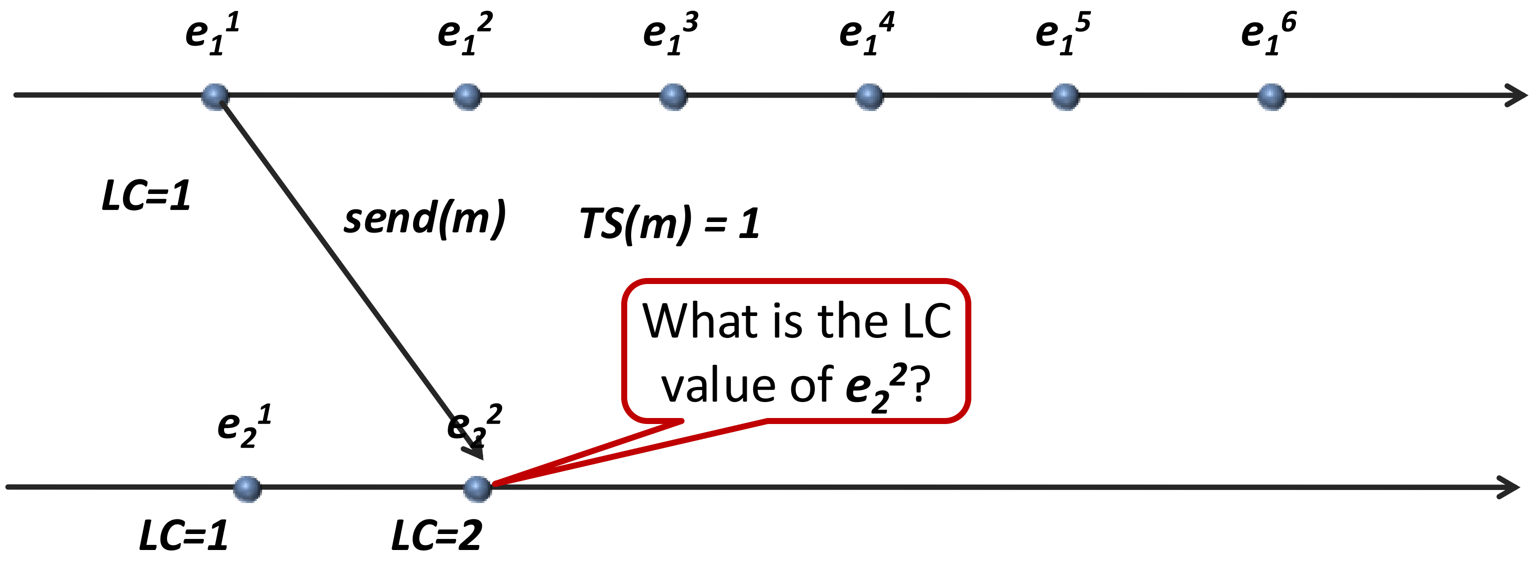


Illustration of a Logical Clock

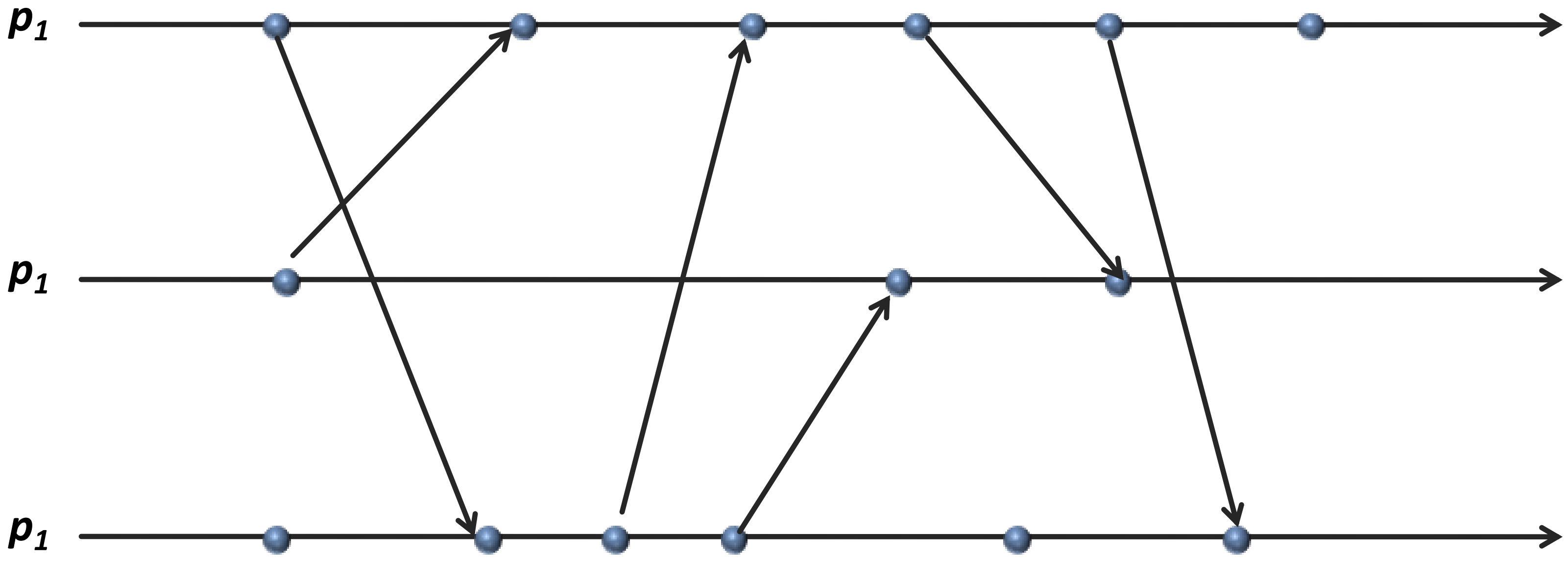


Illustration of a Logical Clock

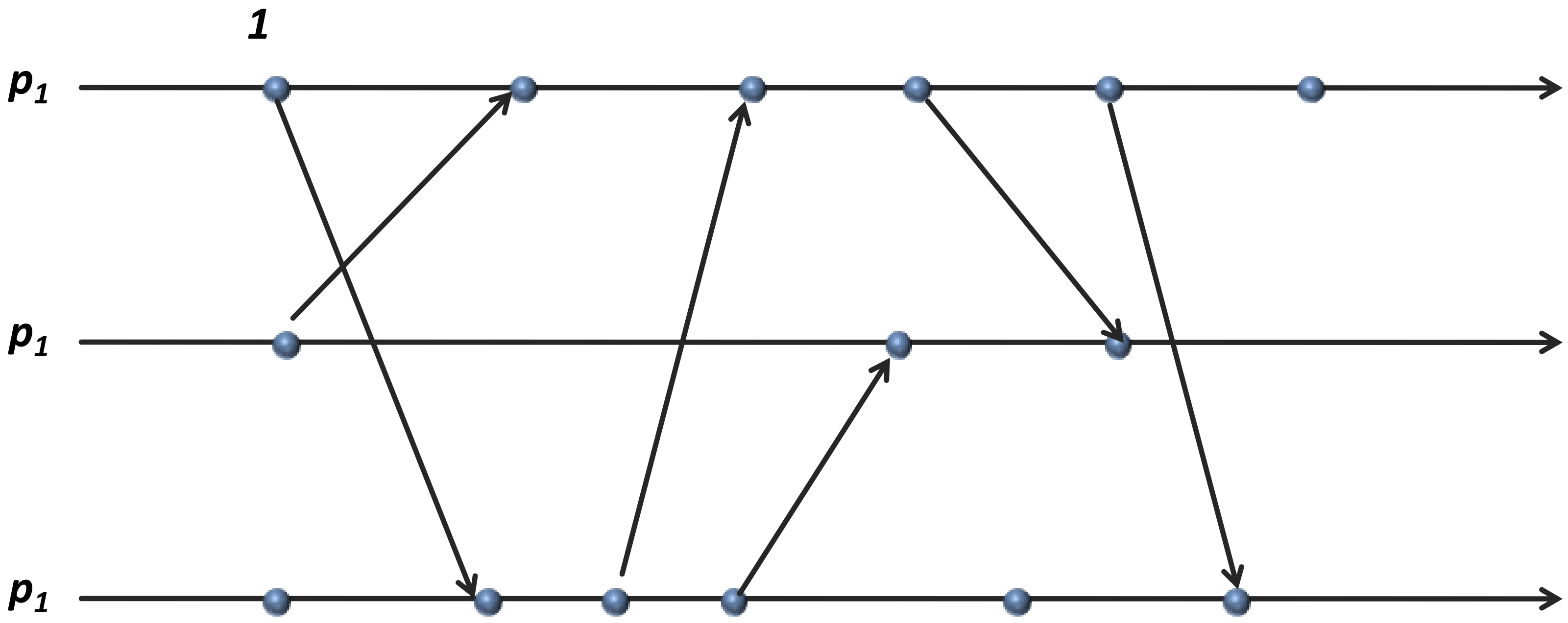


Illustration of a Logical Clock

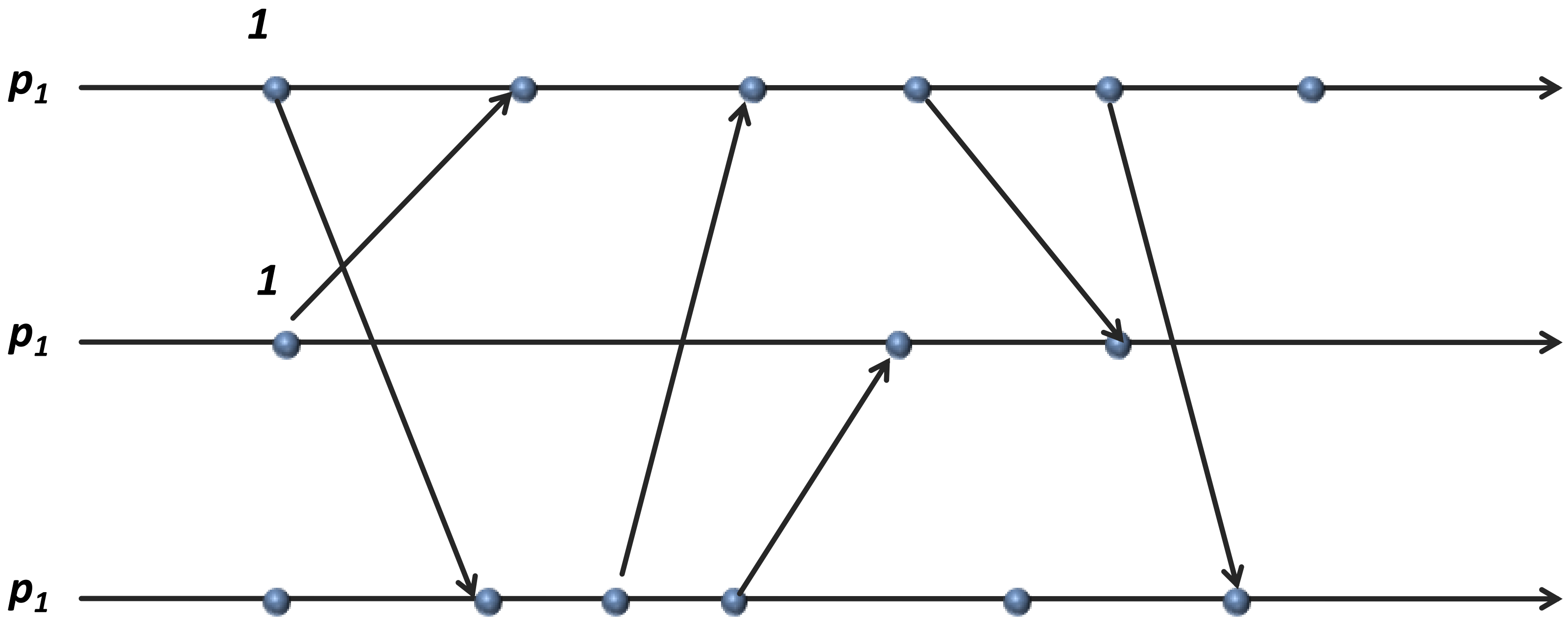


Illustration of a Logical Clock

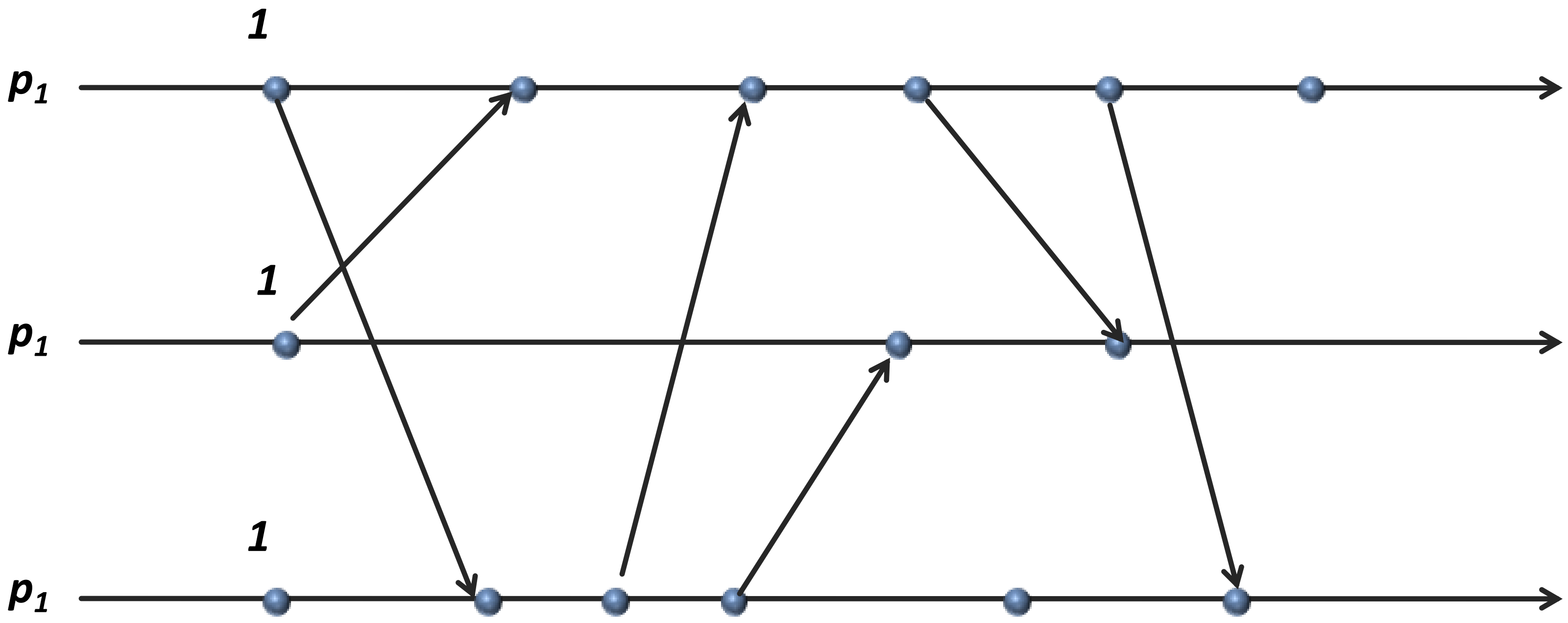


Illustration of a Logical Clock

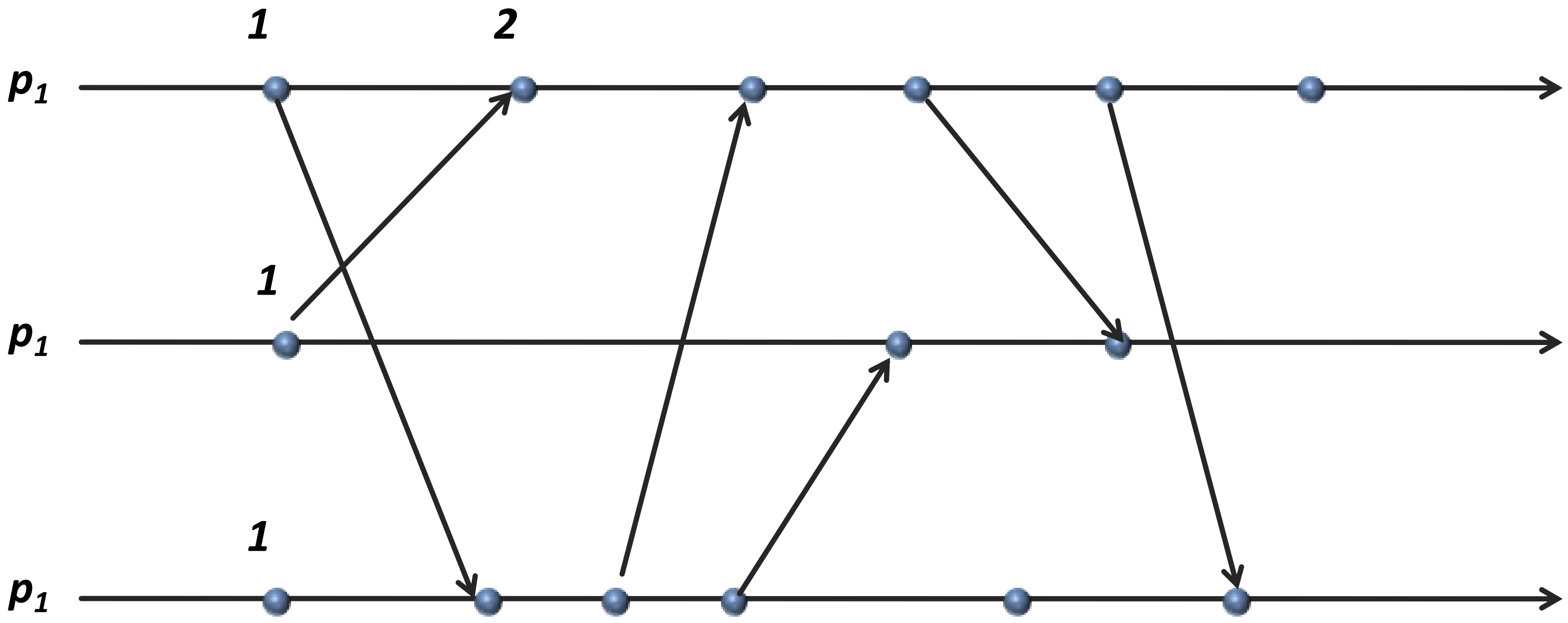


Illustration of a Logical Clock

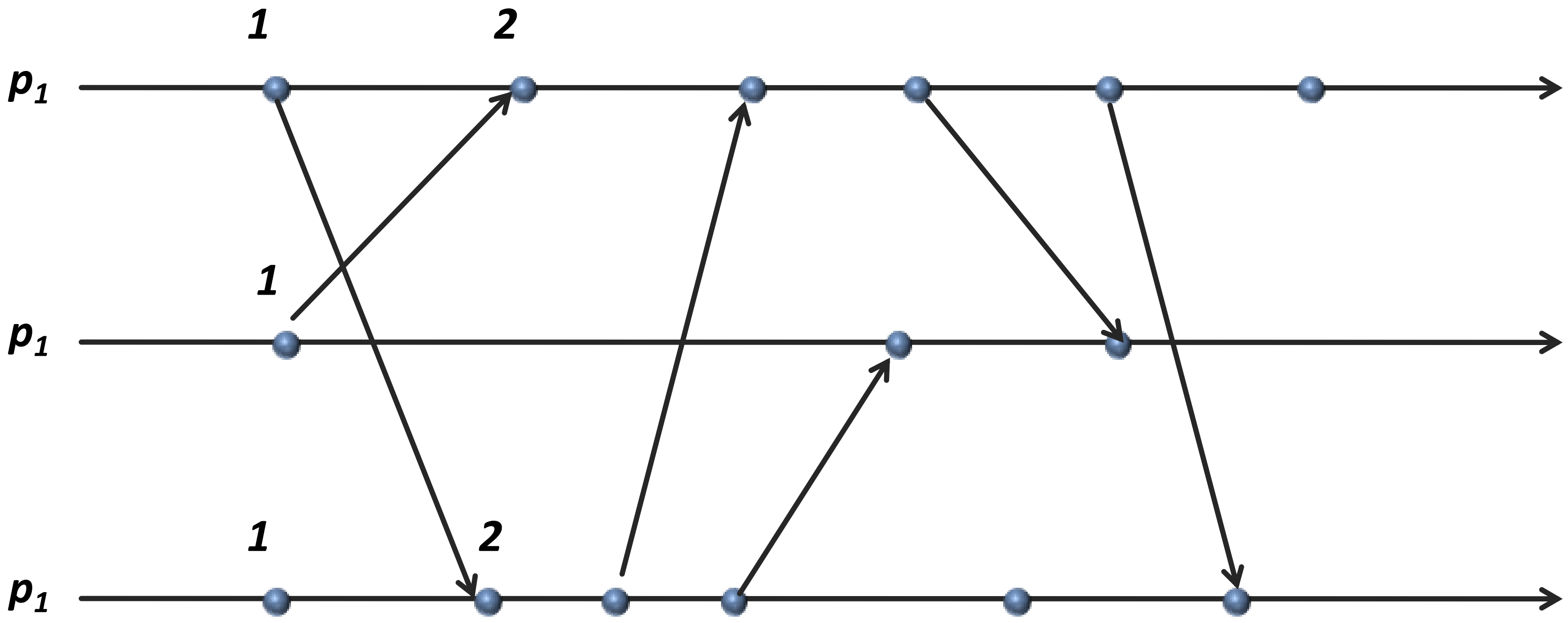


Illustration of a Logical Clock

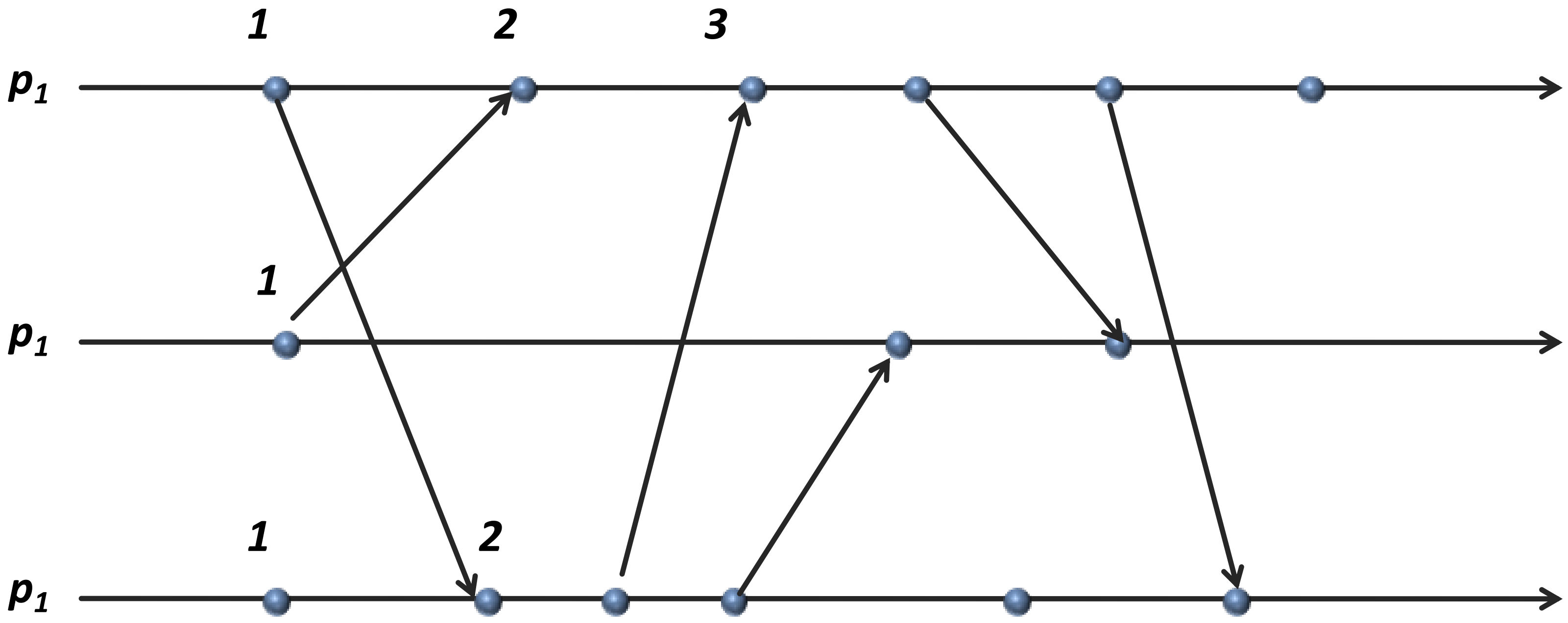


Illustration of a Logical Clock

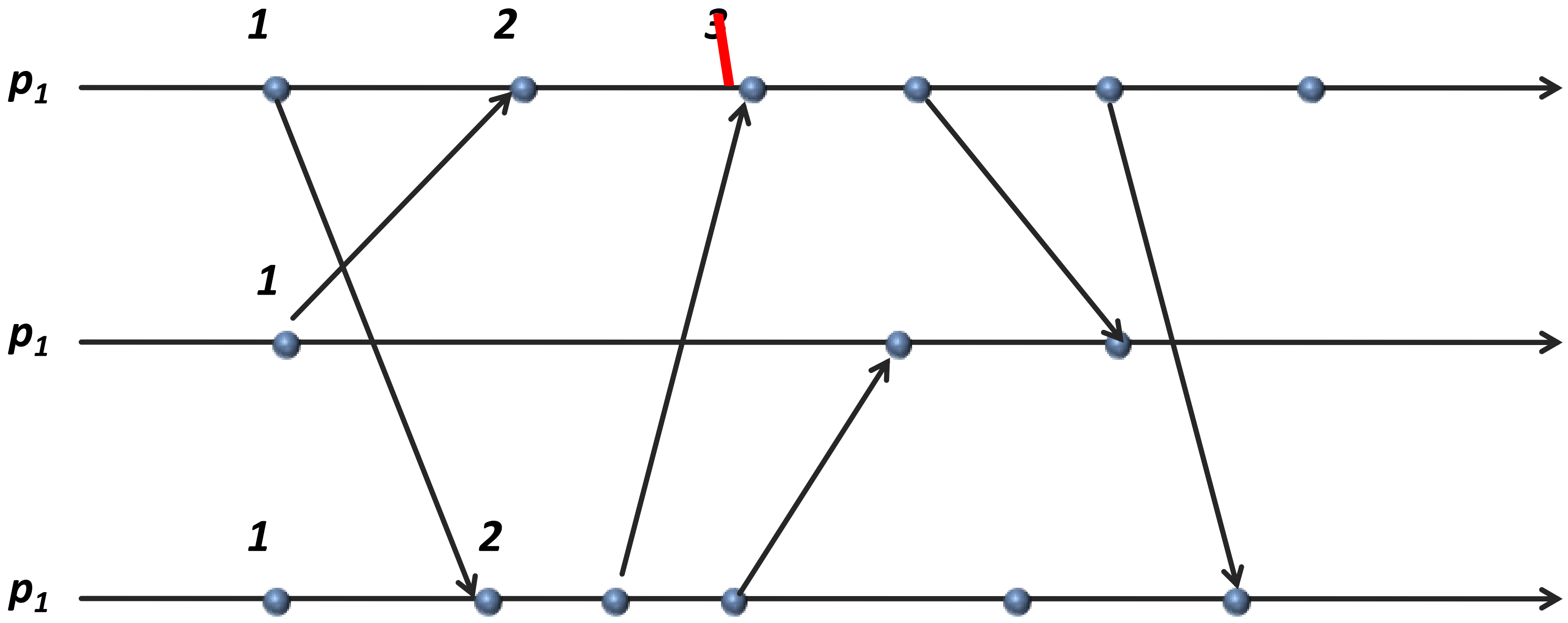


Illustration of a Logical Clock

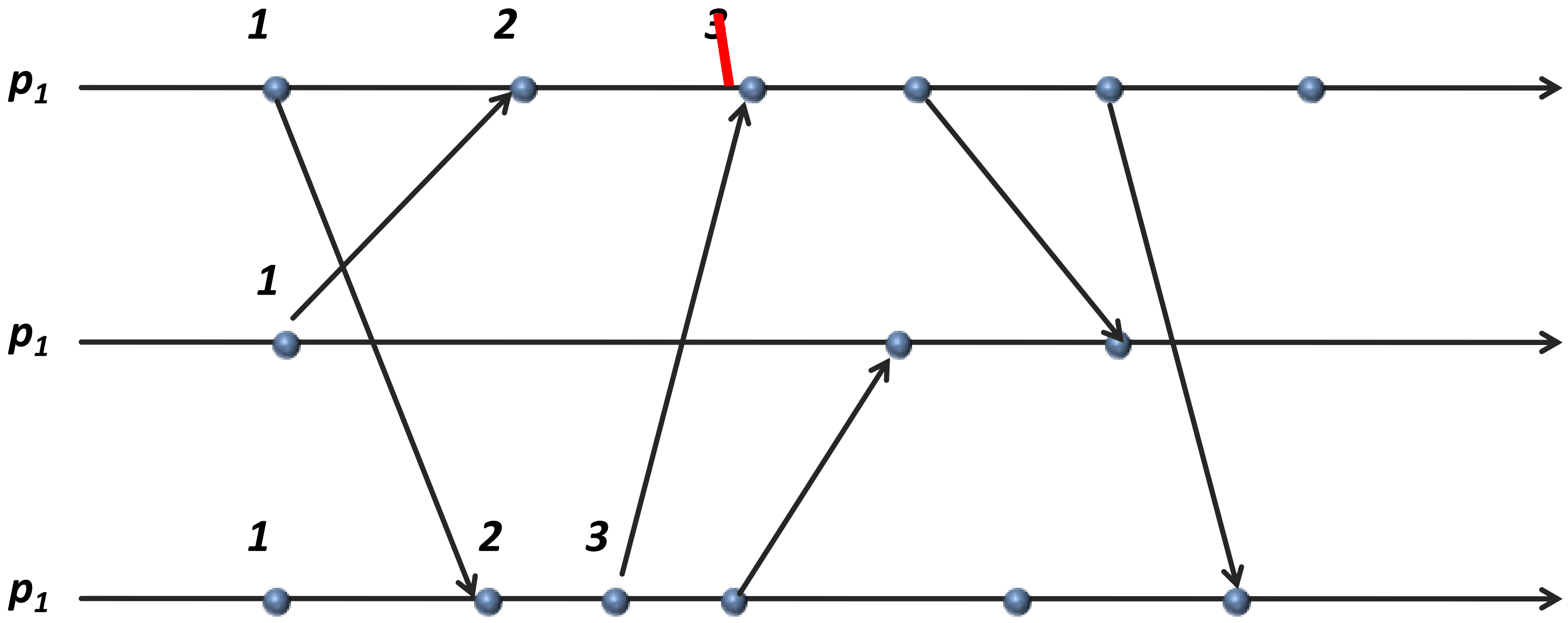


Illustration of a Logical Clock

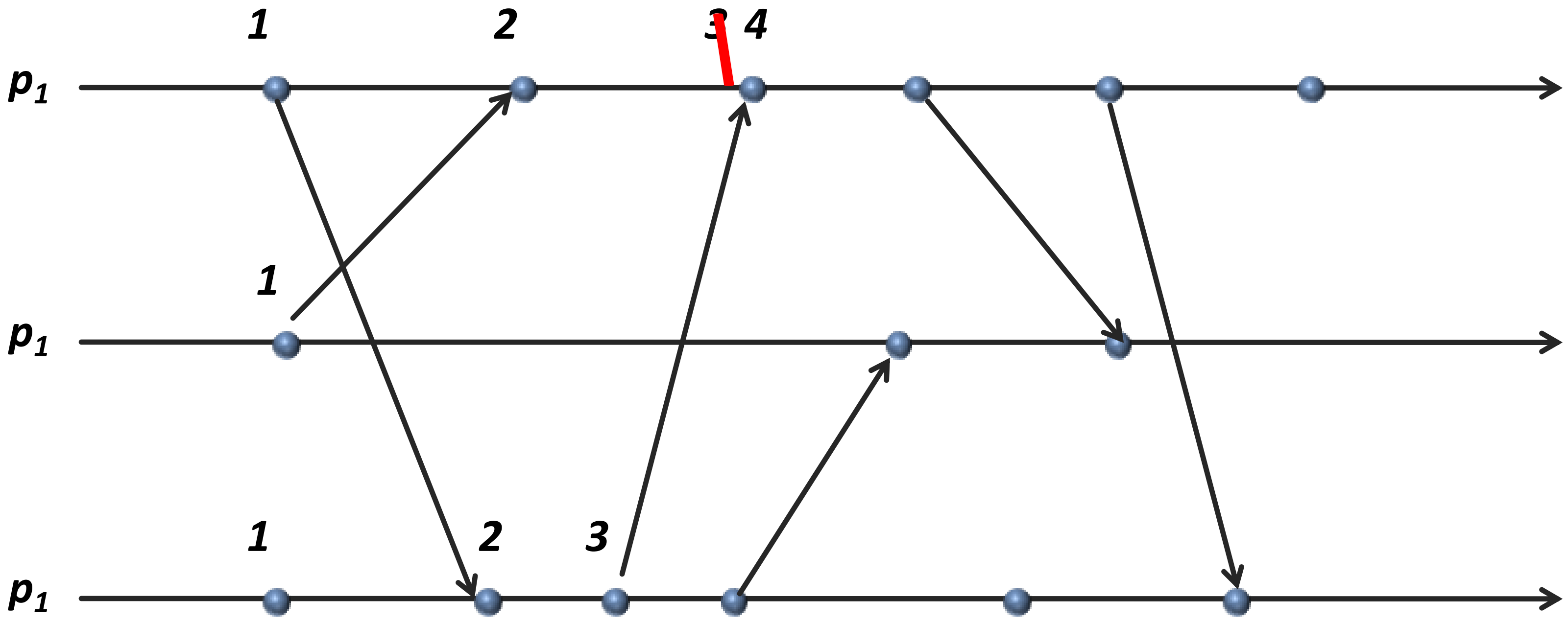


Illustration of a Logical Clock

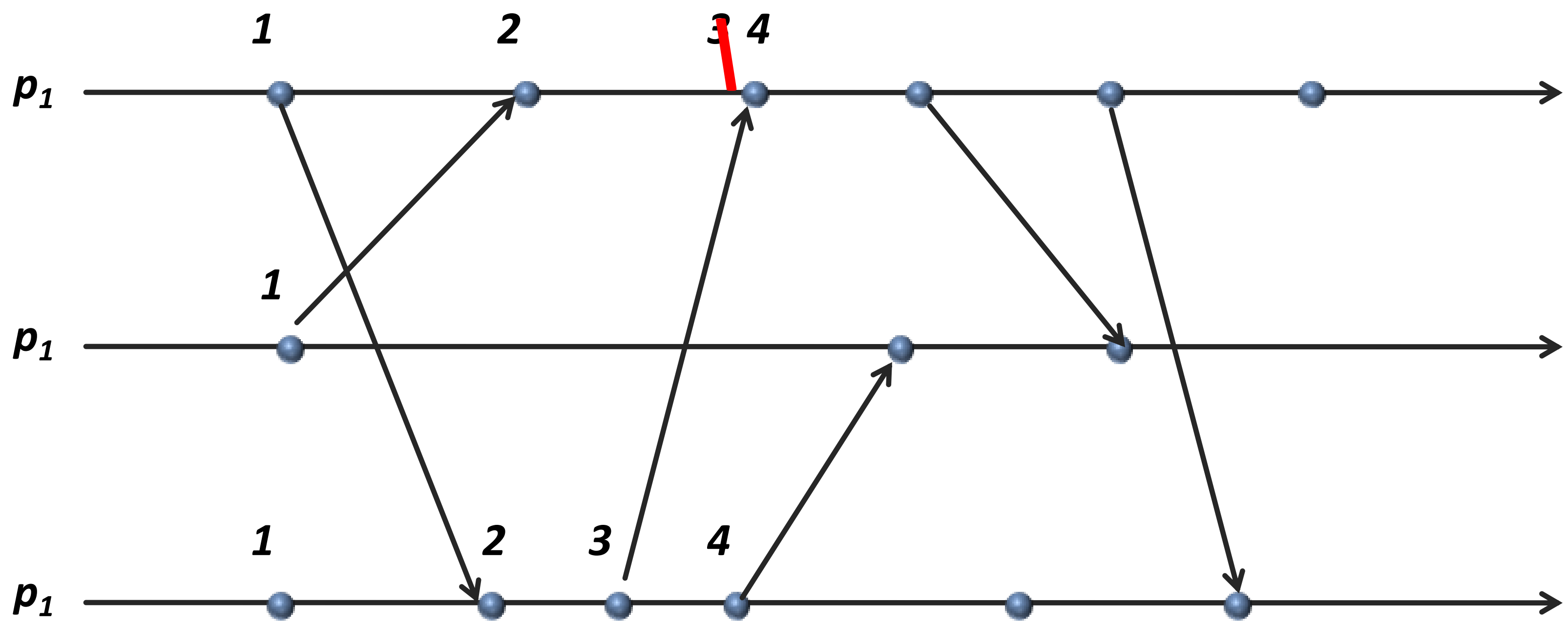


Illustration of a Logical Clock

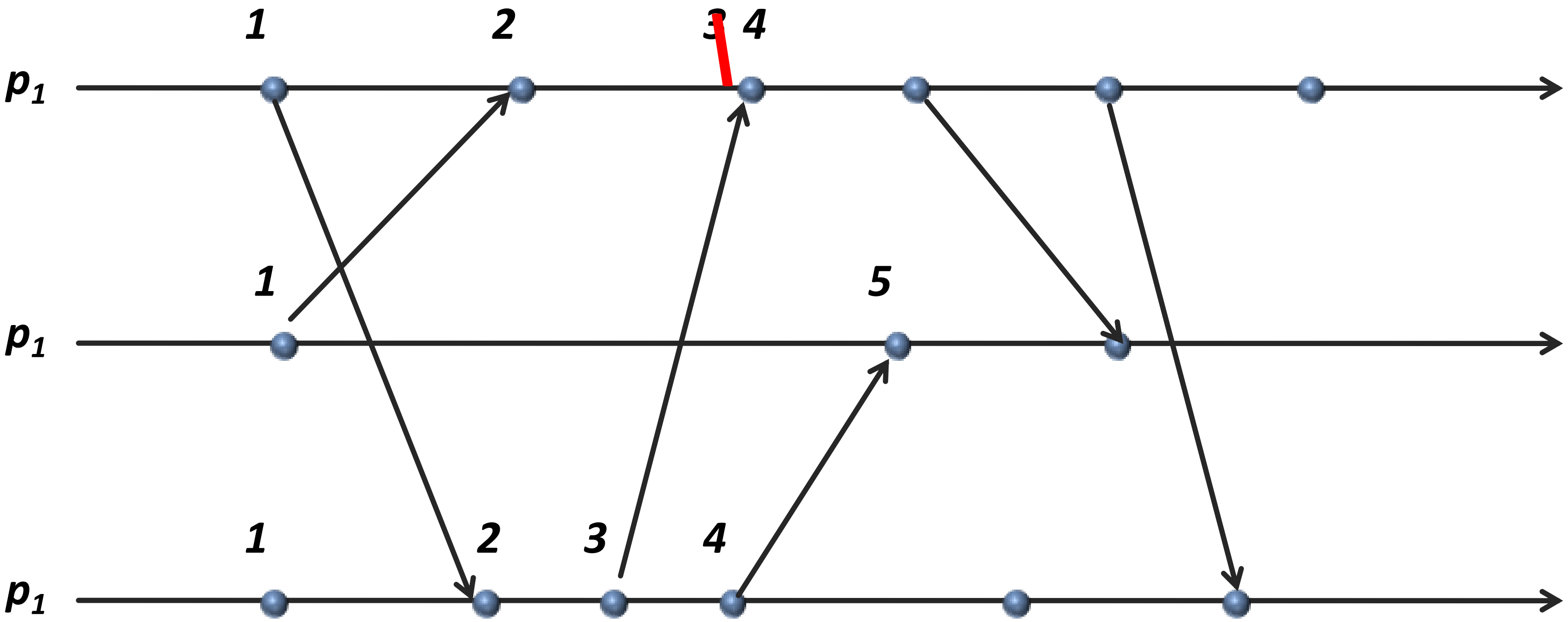


Illustration of a Logical Clock

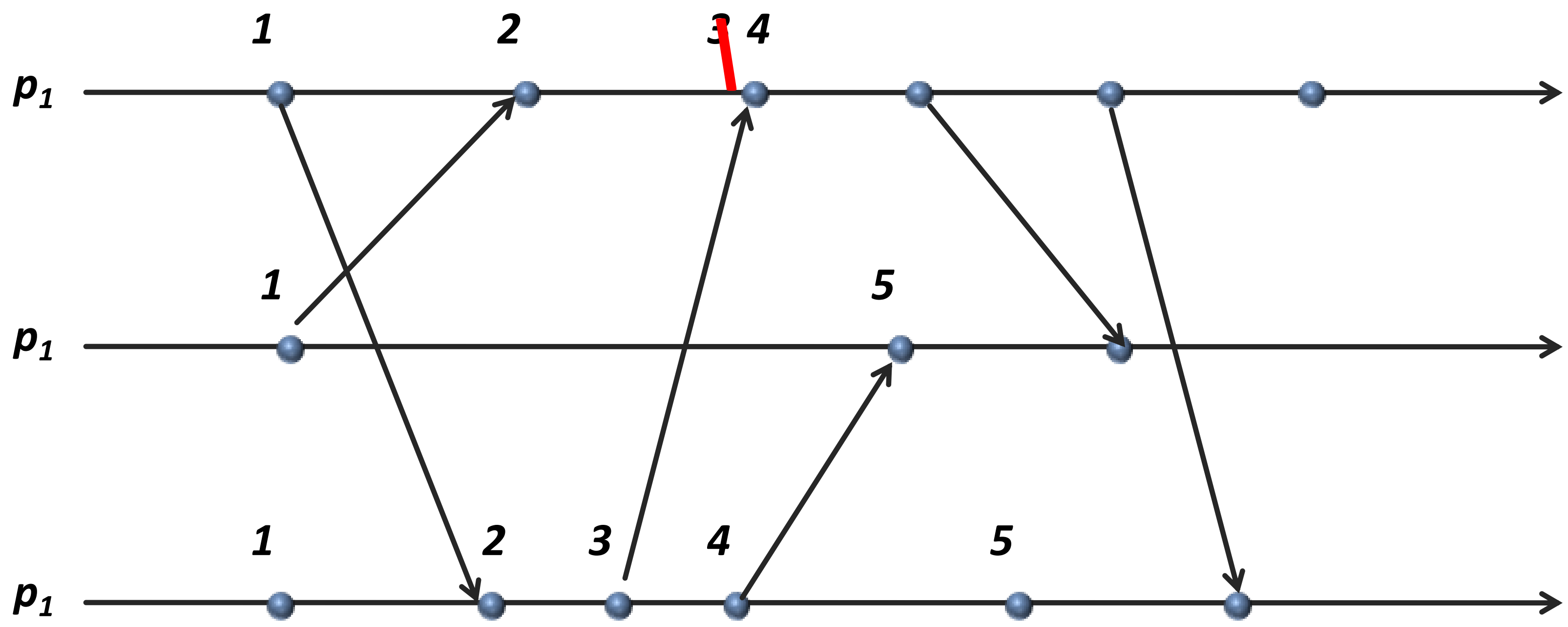


Illustration of a Logical Clock

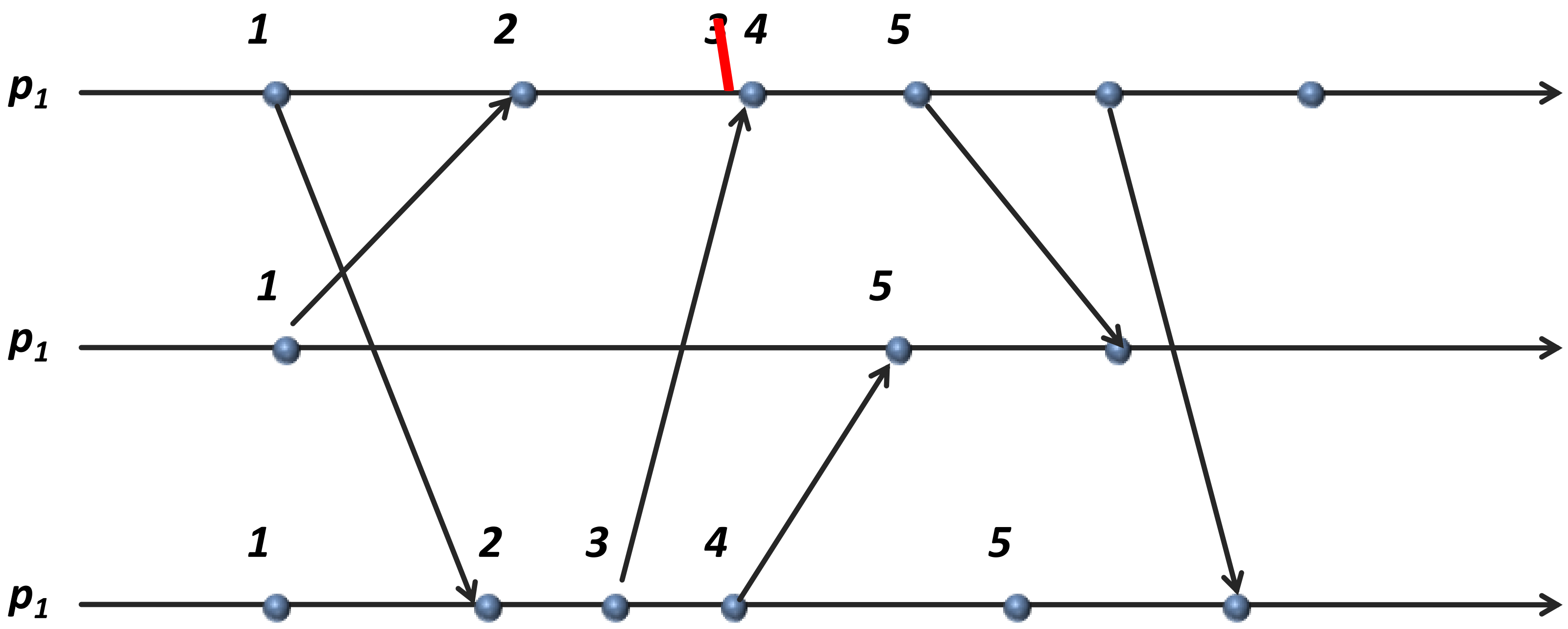


Illustration of a Logical Clock

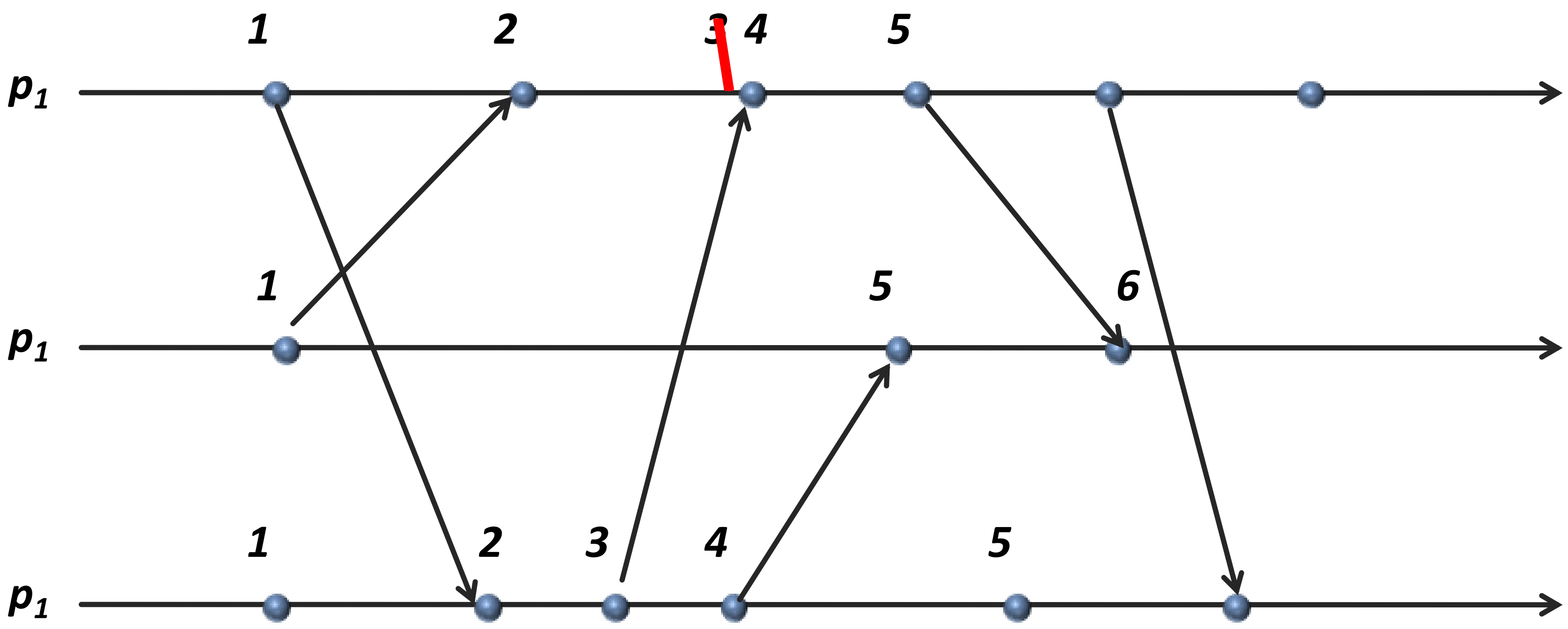


Illustration of a Logical Clock

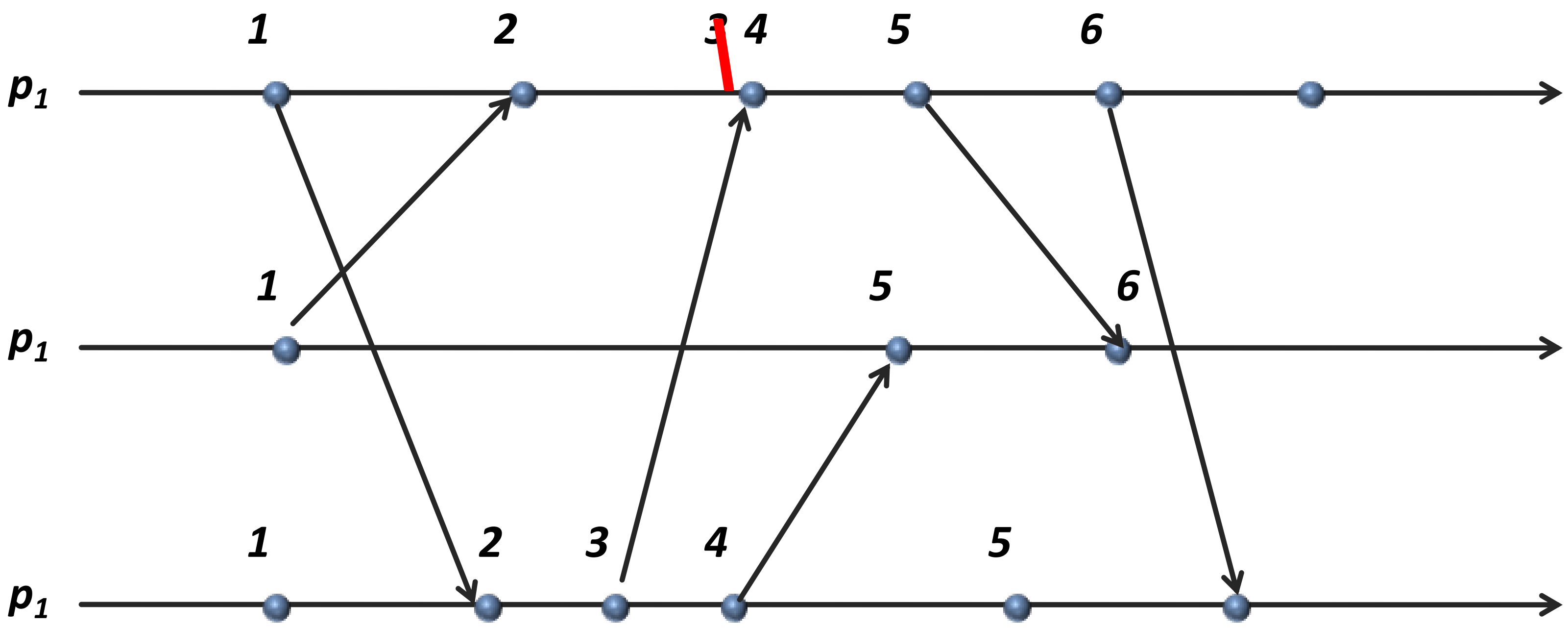


Illustration of a Logical Clock

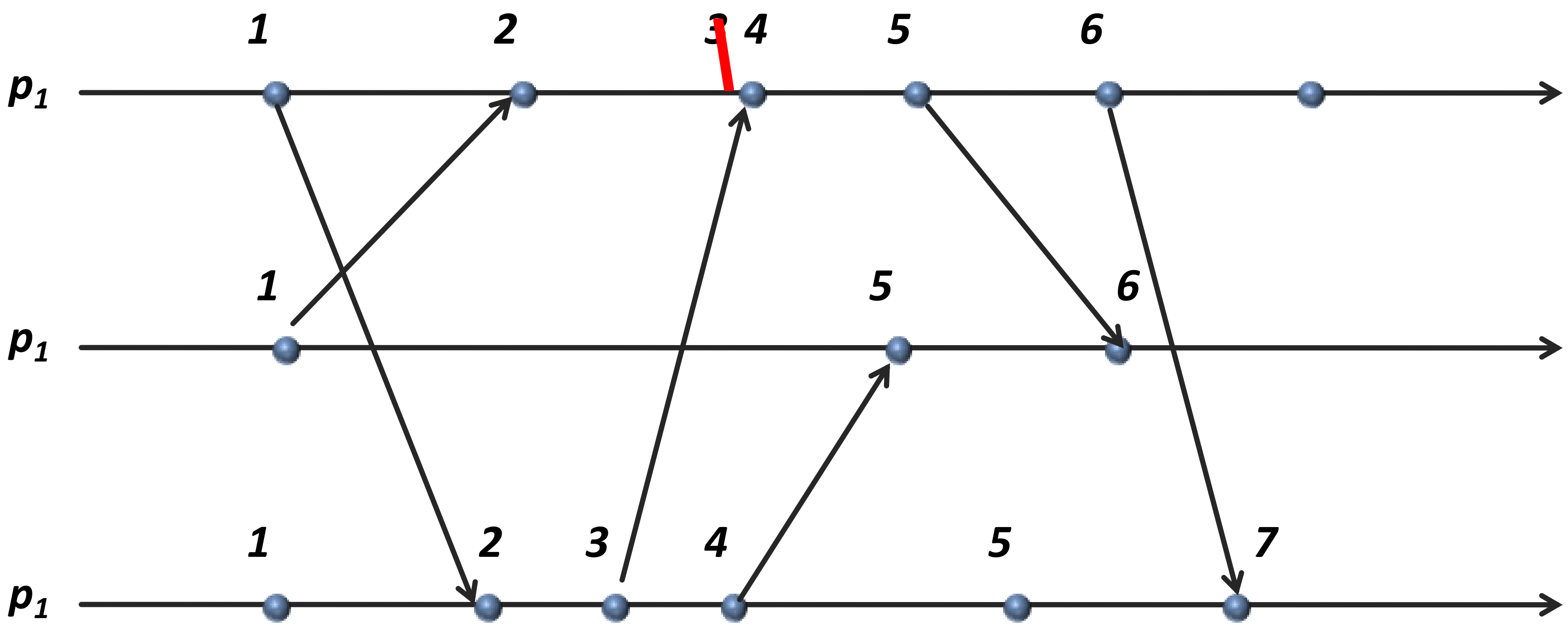


Illustration of a Logical Clock

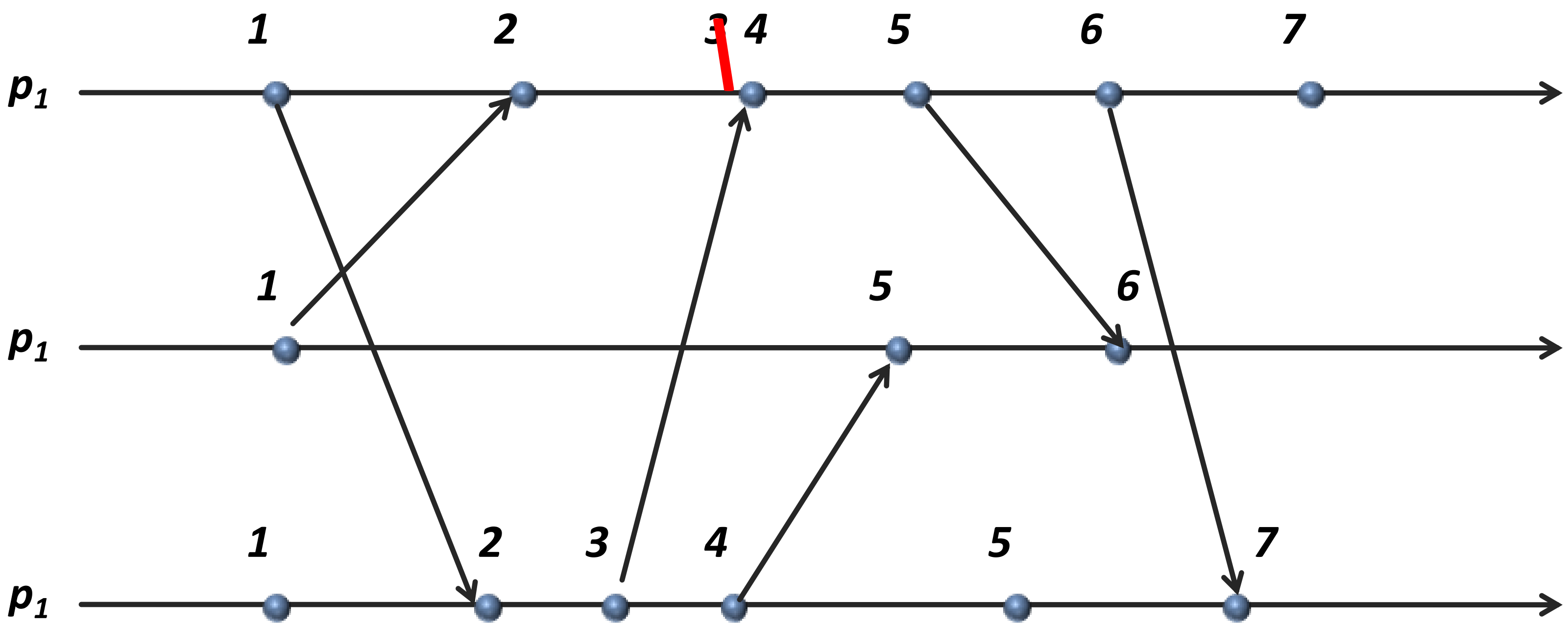
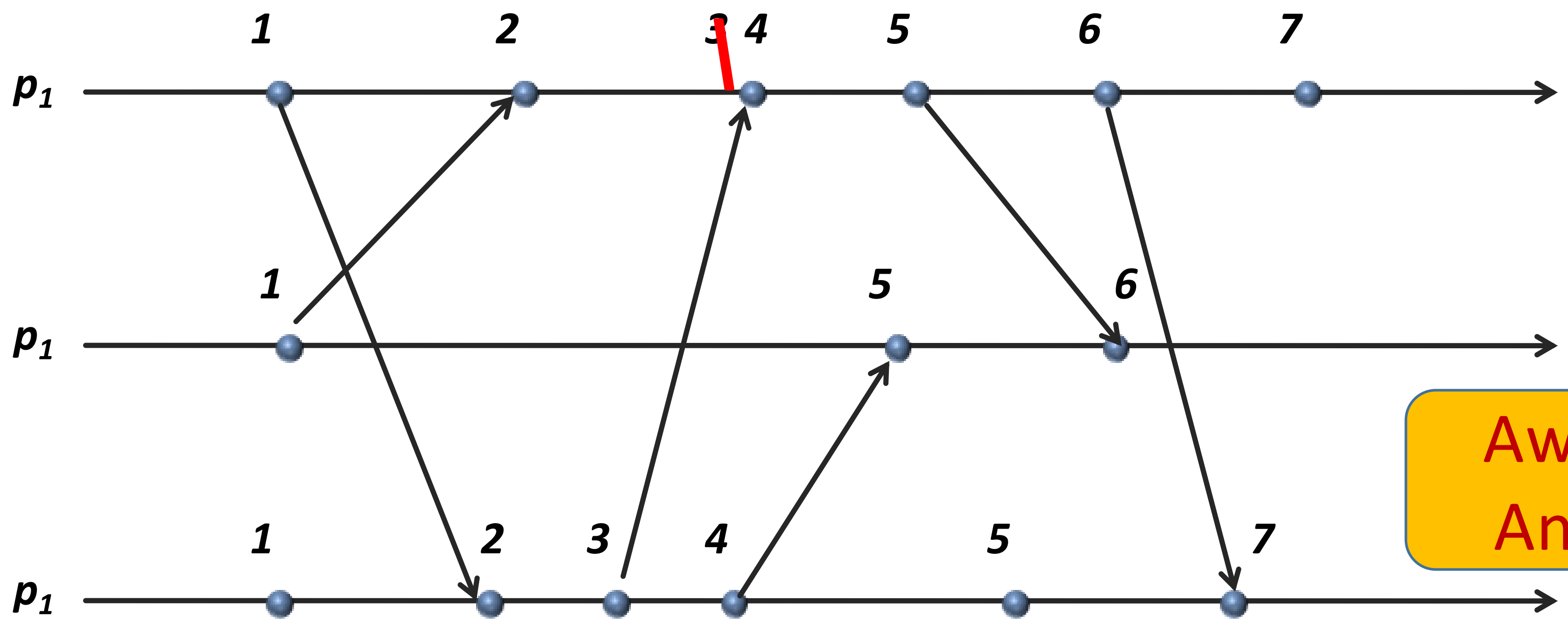
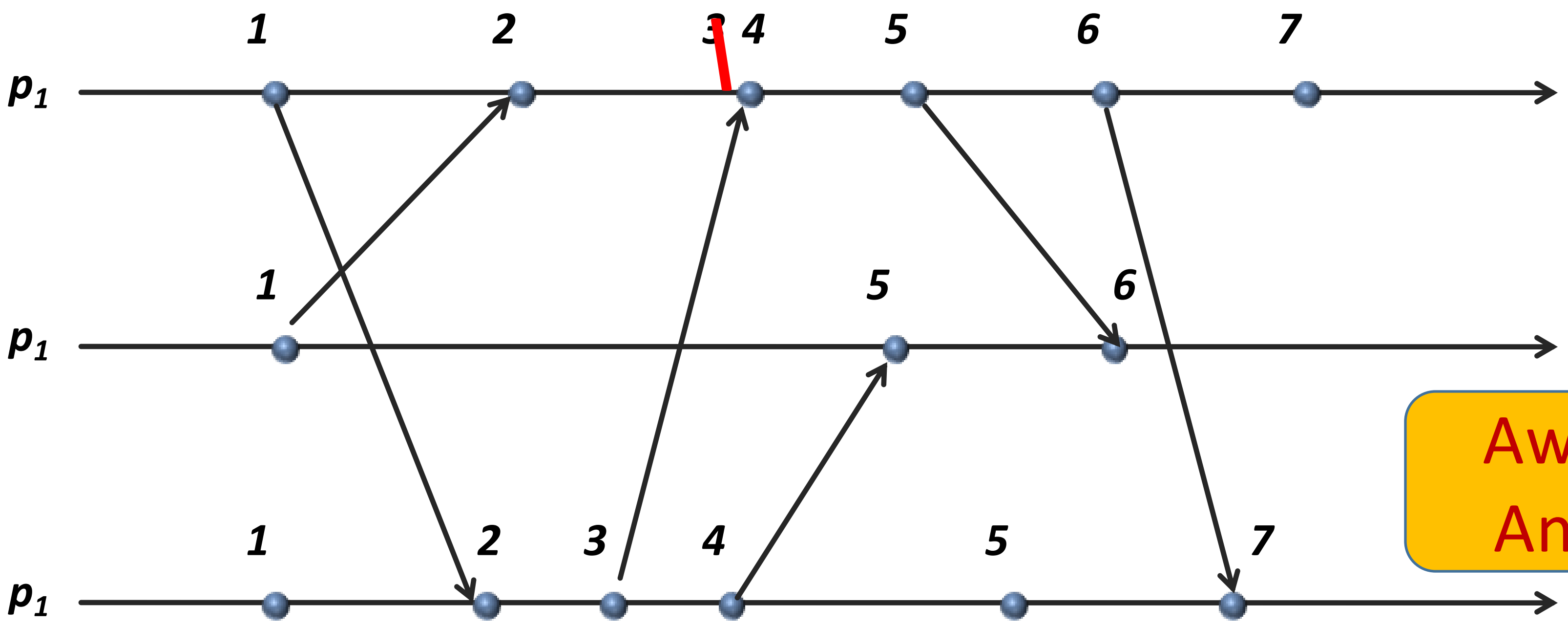


Illustration of a Logical Clock



Awesome, right?
Any drawbacks?

Illustration of a Logical Clock



Awesome, right?
Any drawbacks?

e_x < e_y → TS(e_x) < TS(e_y), but
TS(e_x) < TS(e_y) doesn't guarantee e_x < e_y

Vector Clock

Vector Clock

Replace Single Logical value with Vector!

Vector Clock

Replace Single Logical value with Vector!

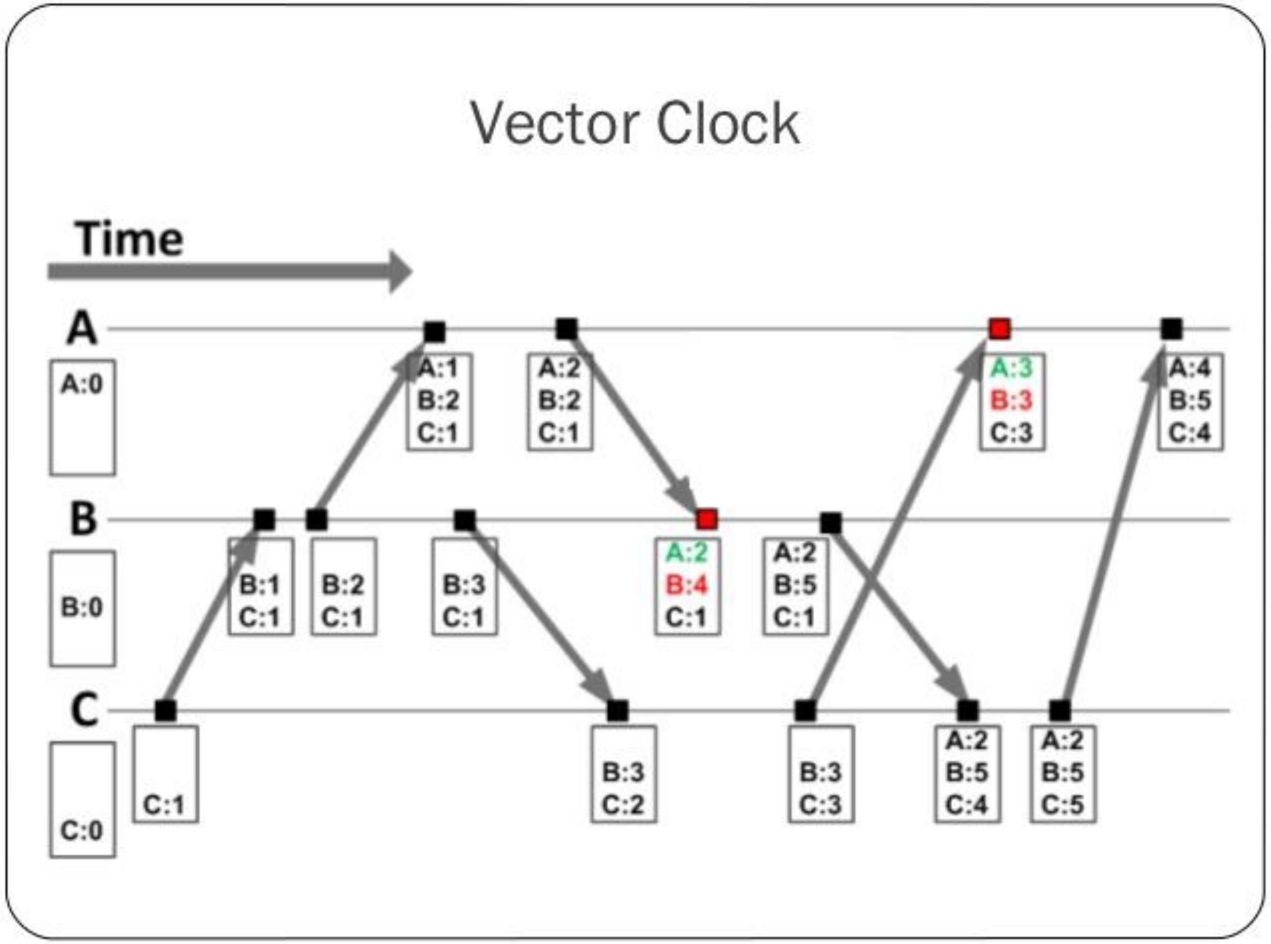
$V_i[i]$: #events occurred at i

$V_i[j]$: #events i knows occurred at j

Update

- On local-event: increment $V_i[I]$
- On send-message: increment, piggyback entire local vector V
- On recv-message: $V_j[k]$ = max($V_j[k], V_i[k]$)
  - $V_j[i] = V_j[i]+1$ *(increment local clock)*
  - Receiver learns about number of events sender knows occurred elsewhere

Vector Clock



Replace Single Logical value with Vector!

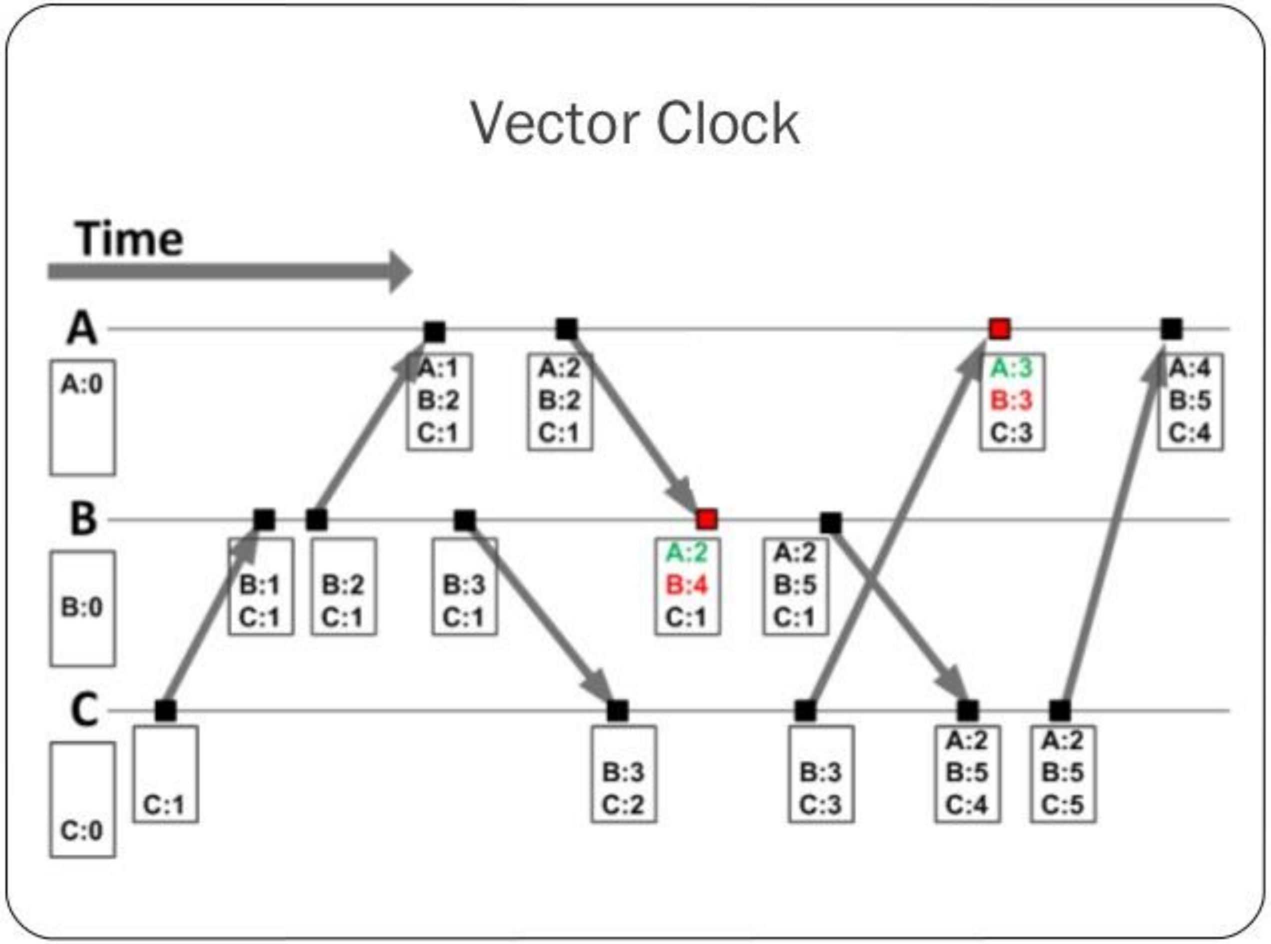
$V_i[i]$: #events occurred at i

$V_i[j]$: #events i knows occurred at j

Update

- On local-event: increment $V_i[I]$
- On send-message: increment, piggyback entire local vector V
- On recv-message: $V_j[k] = \max(V_j[k], V_i[k])$
  - $V_j[i] = V_j[i]+1$ *(increment local clock)*
  - Receiver learns about number of events sender knows occurred elsewhere

Vector Clock



Replace Single Logical value with Vector!

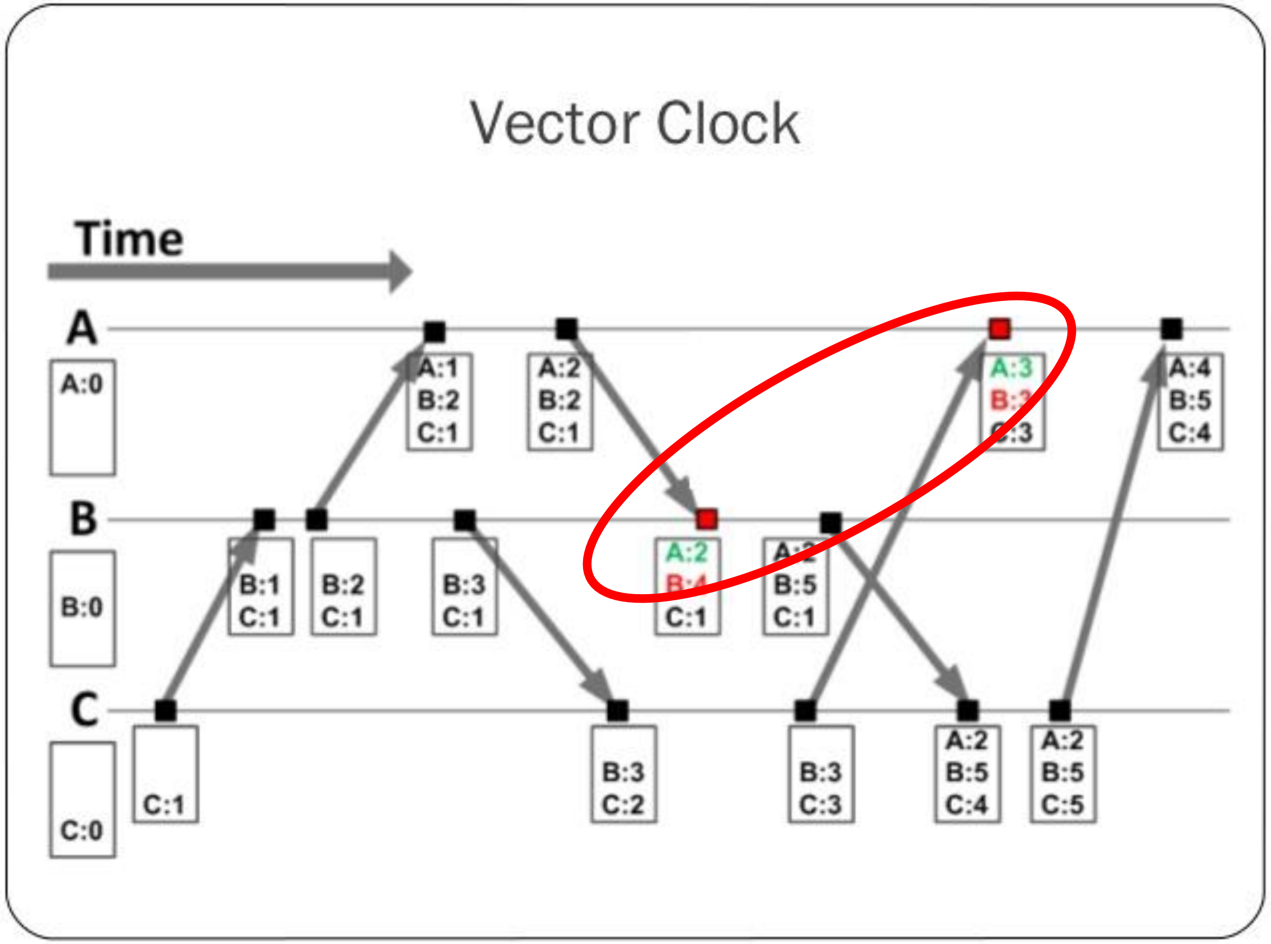
$V_i[i]$: #events occurred at i

$V_i[j]$: #events i knows occurred at j

Update

- On local-event: increment $V_i[I]$
- On send-message: increment, piggyback entire local vector V
- On recv-message: $V_j[k]$ = max($V_j[k],V_i[k]$)
  - $V_j[i] = V_j[i]+1$ *(increment local clock)*
  - Receiver learns about number of events sender knows occurred elsewhere

Vector Clock



Replace Single Logical value with Vector!

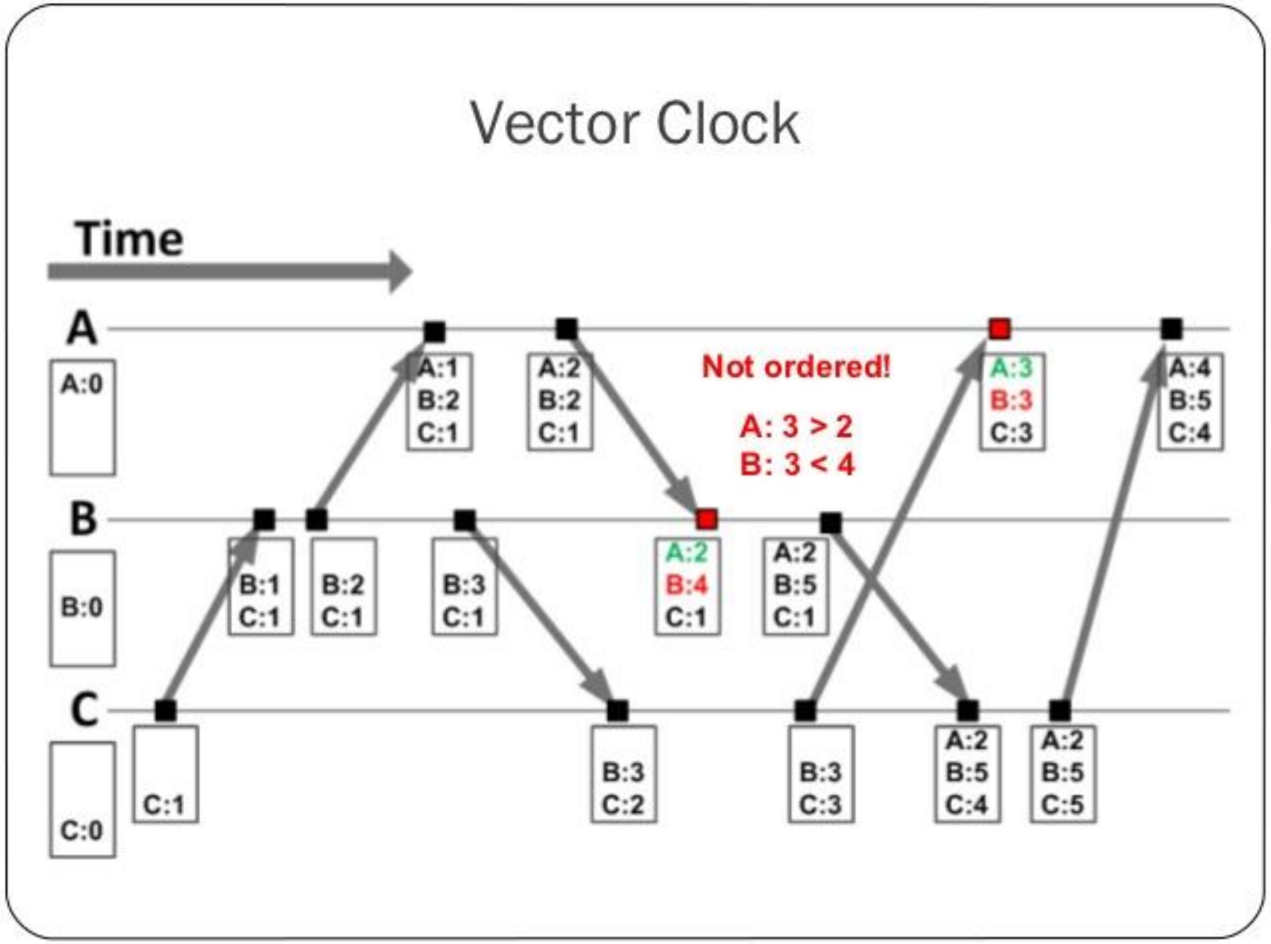
$V_i[i]$: #events occurred at i

$V_i[j]$: #events i knows occurred at j

Update

- On local-event: increment $V_i[I]$
- On send-message: increment, piggyback entire local vector V
- On recv-message: $V_j[k]$ = max($V_j[k]$,$V_i[k]$)
  - *$V_j[i] = V_j[i]+1$ (increment local clock)*
  - Receiver learns about number of events sender knows occurred elsewhere

Vector Clock



Replace Single Logical value with Vector!

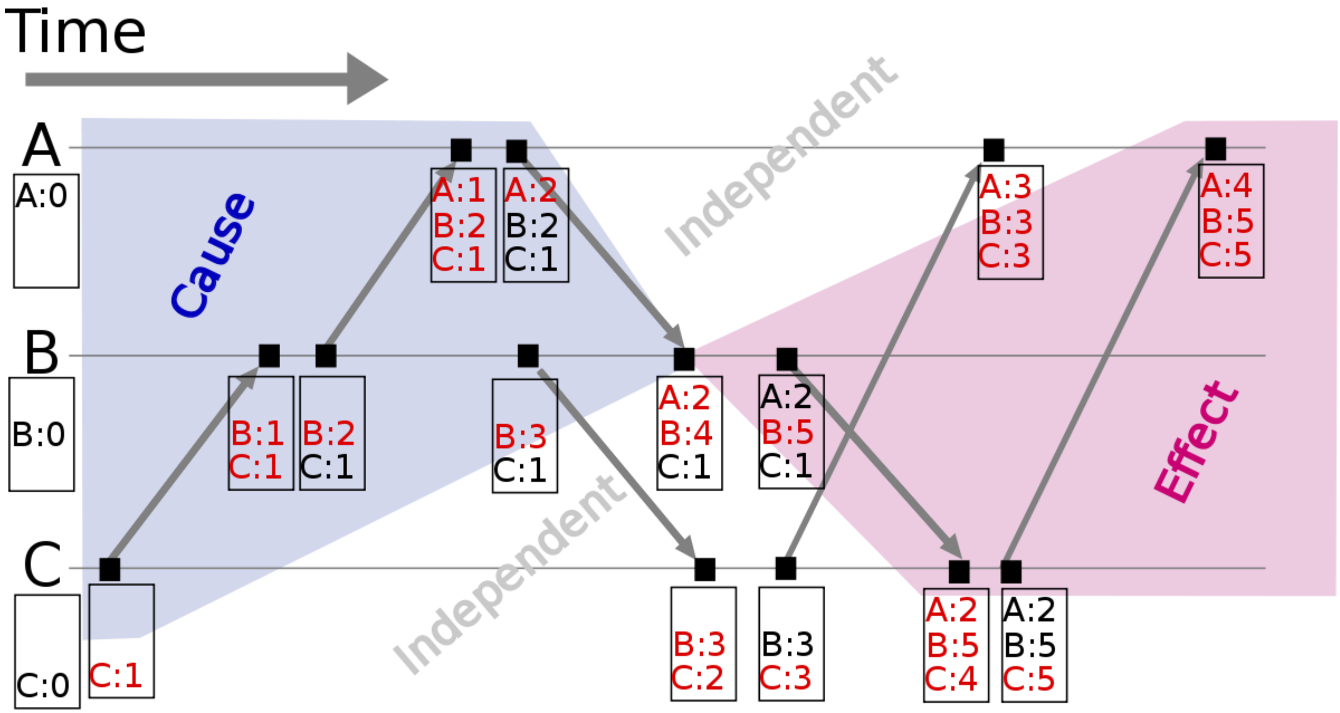
$V_i[i]$: #events occurred at i

$V_i[j]$: #events i knows occurred at j

Update

- On local-event: increment $V_i[I]$
- On send-message: increment, piggyback entire local vector V
- On recv-message: $V_j[k]$ = max($V_j[k], V_i[k]$)
  - *$V_j[i] = V_j[i]+1$ (increment local clock)*
  - Receiver learns about number of events sender knows occurred elsewhere

Vector Clock Example



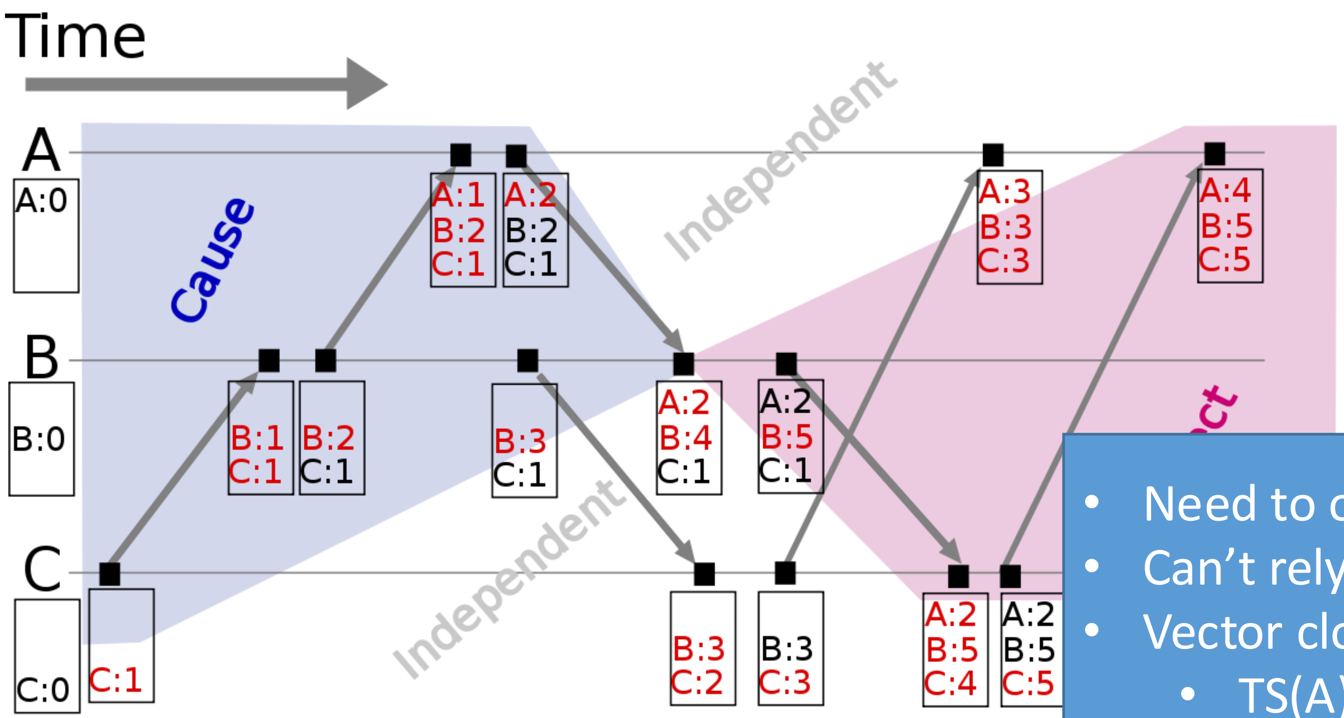
Each process *i* maintains a vector V_i

- $V_i[i]$: number of events that have occurred at i
- $V_i[j]$: number of events I knows have occurred at process j

Update

- Local event: increment $V_i[I]$
- Send a message :piggyback entire vector V
- Receipt of a message: $V_j[k]$ = max($V_j[k], V_i[k]$)
  - Receiver is told about how many events the sender knows occurred at another process *k*
  - Also $V_j[i] = V_j[i]+1$

Vector Clock Example



Each process *i* maintains a vector V_i

- $V_i[i]$: number of events that have occurred at i
- $V_i[j]$: number of events I knows have occurred at process j

Update

- Local event: increment $V_i[I]$

…or V

…$V_i[k]$)

…s the

…cess *k*

- Need to order operations
- Can't rely on real-time
- Vector clock: timestamping algorithm s.t.
  - TS(A) < TS(B) → A happens before B
  - Independent ops remain unordered

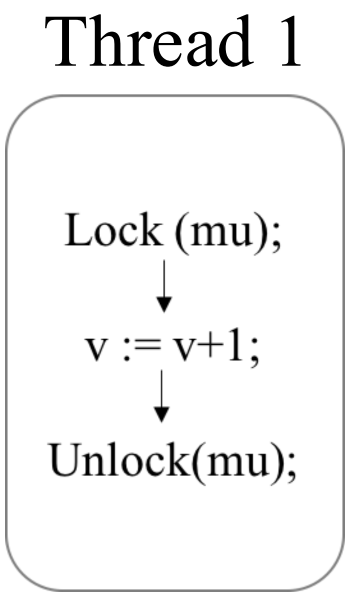
See any drawbacks?

Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing variables not ordered by "happens-before" is a race
- Captures locks and dynamism
- How to track "happens-before"?
  - Sync objects are ordering events
  - Generalizes to fork/join, etc

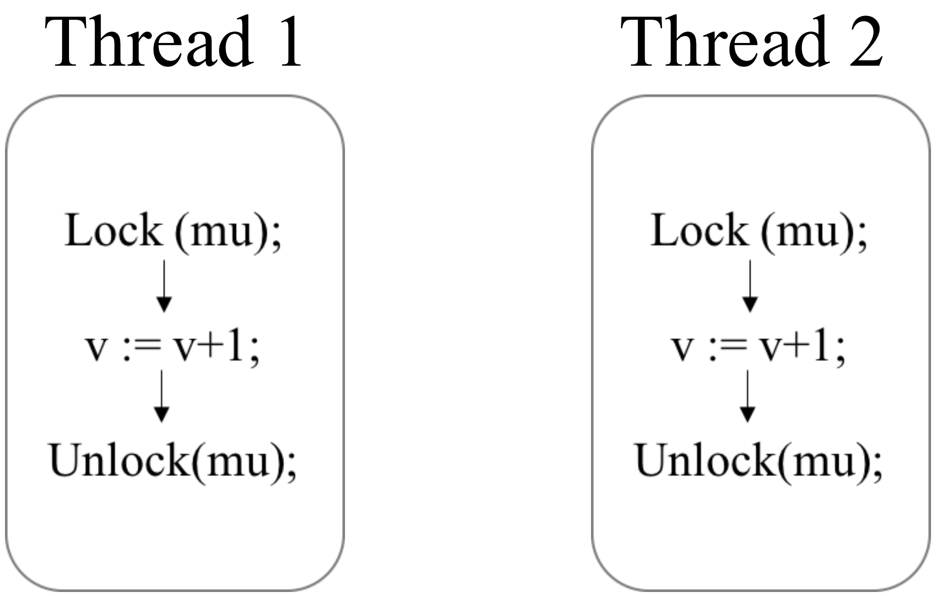
Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing variables not ordered by “happens-before” is a race
- Captures locks and dynamism
- How to track “happens-before”?
  - Sync objects are ordering events
  - Generalizes to fork/join, etc



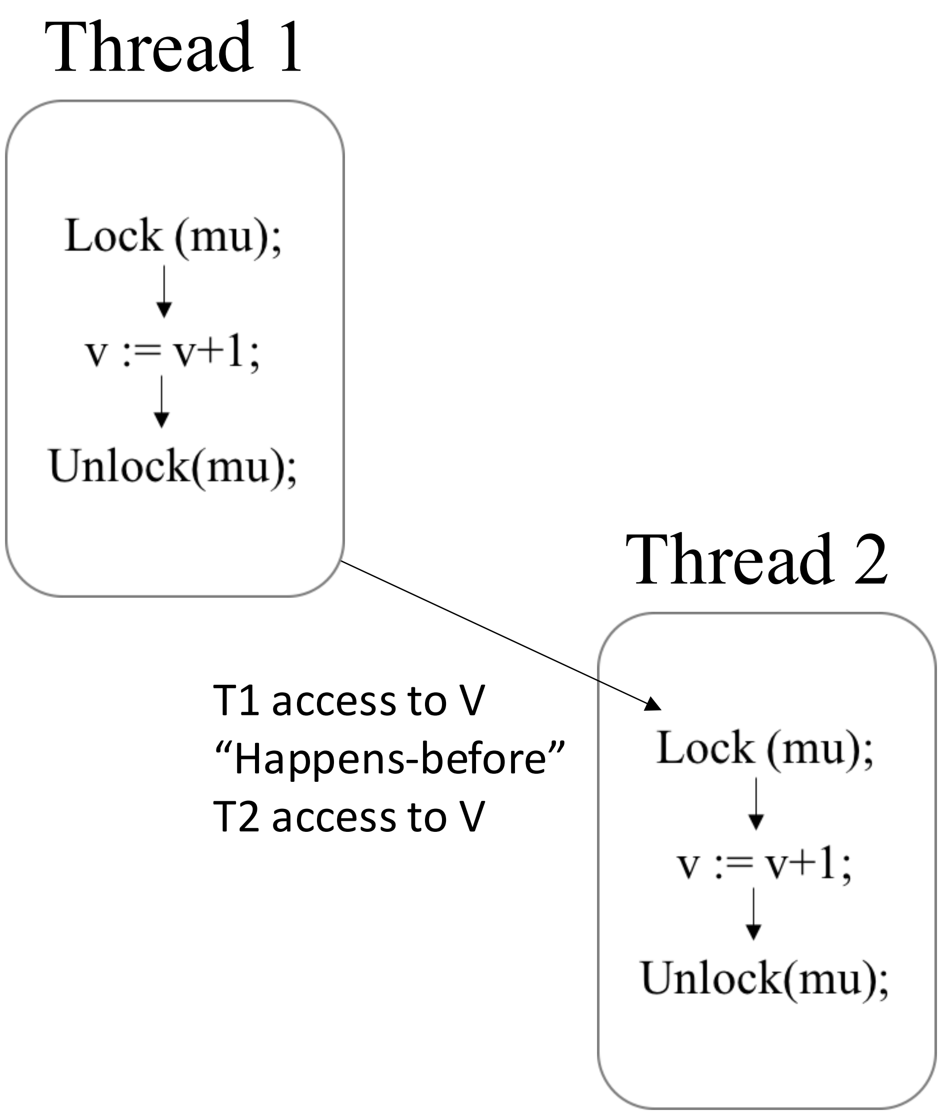
Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing variables not ordered by “happens-before” is a race
- Captures locks and dynamism
- How to track “happens-before”?
  - Sync objects are ordering events
  - Generalizes to fork/join, etc



Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing variables not ordered by “happens-before” is a race
- Captures locks and dynamism
- How to track “happens-before”?
  - Sync objects are ordering events
  - Generalizes to fork/join, etc

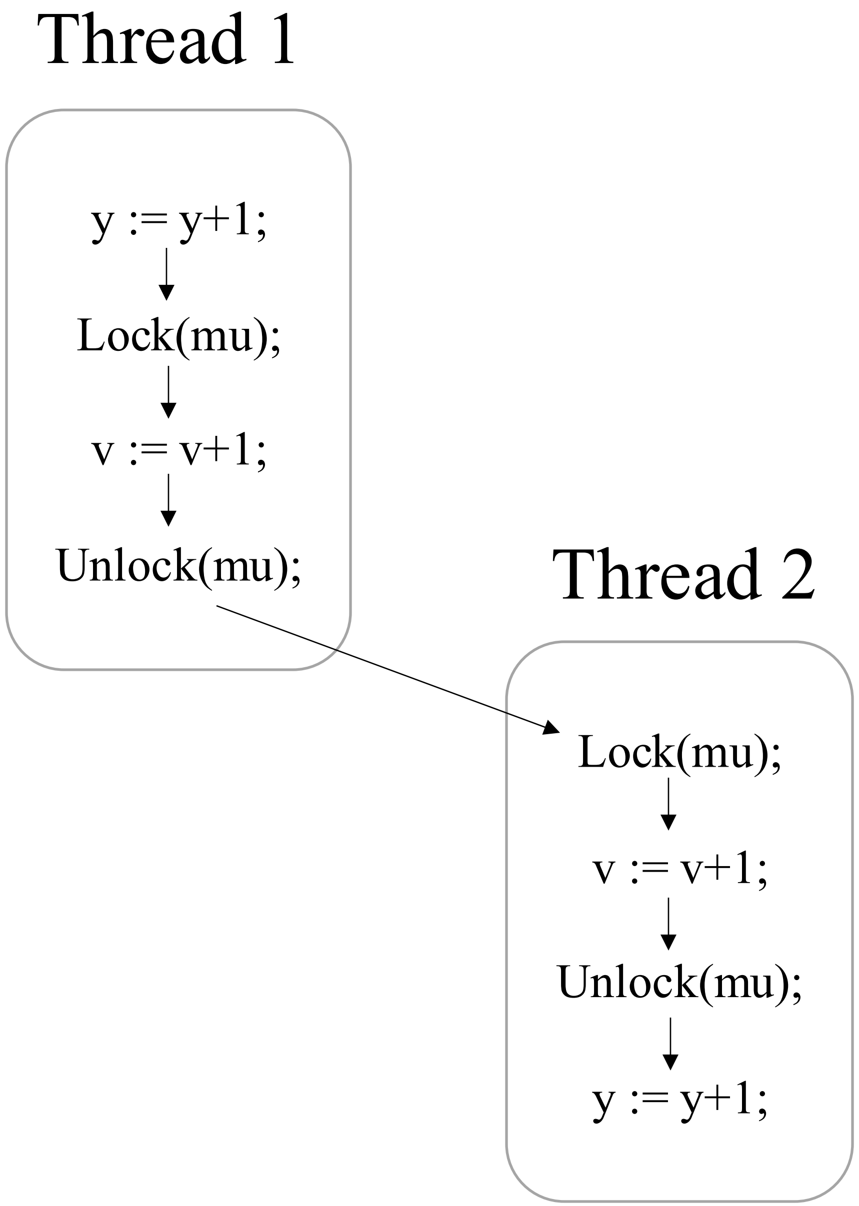


Flaws of *Happens-before*

- Difficult to implement
  - Requires per-thread information
- Dependent on the interleaving produced by the scheduler
- Example

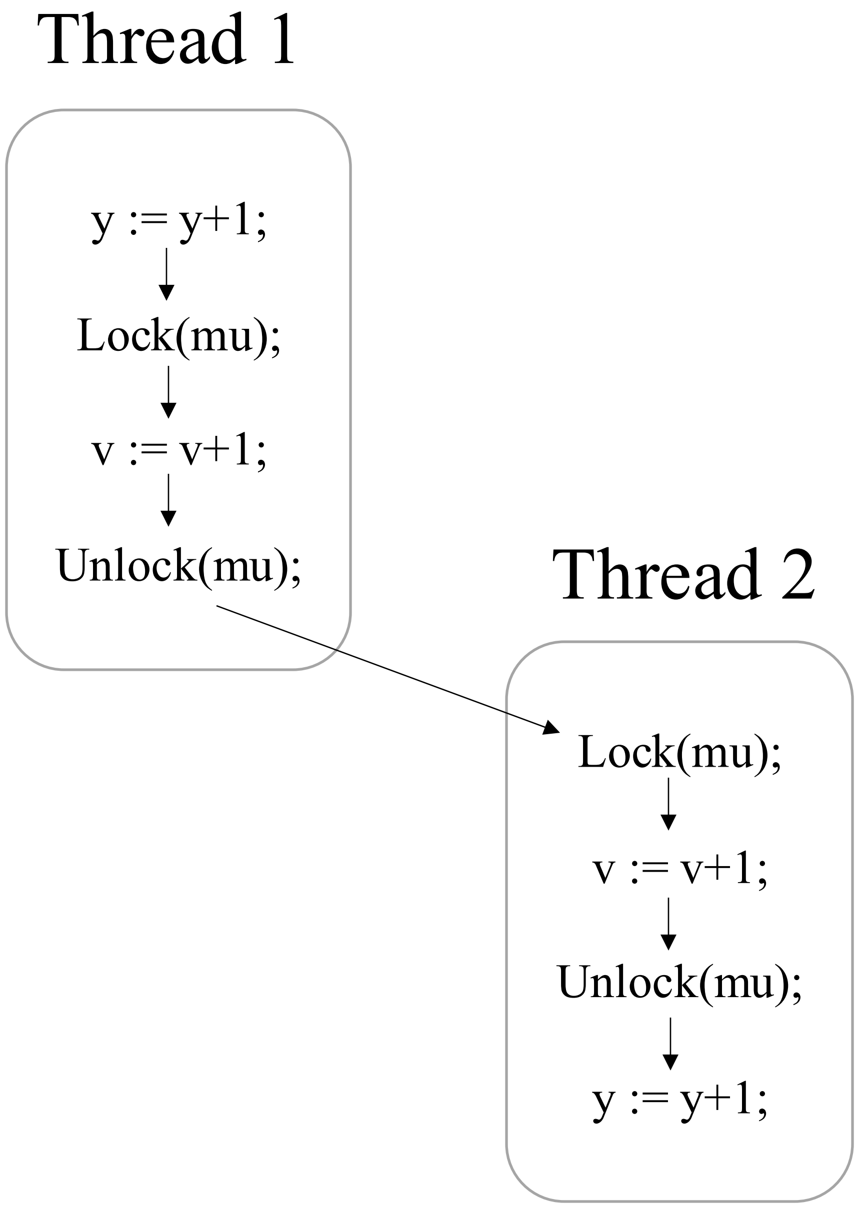
Flaws of *Happens-before*

- Difficult to implement
  - Requires per-thread information
- Dependent on the interleaving produced by the scheduler
- Example



Flaws of *Happens-before*

- Difficult to implement
  - Requires per-thread information
- Dependent on the interleaving produced by the scheduler
- Example
  - T1-acc(v) happens before T2-acc(v)
  - T1-acc(y) happens before T1-acc(v)
  - T2-acc(v) happens before T2-acc(y)
  - Conclusion: no race on Y!
  - Finding doesn't generalize



Flaws of *Happens-before*

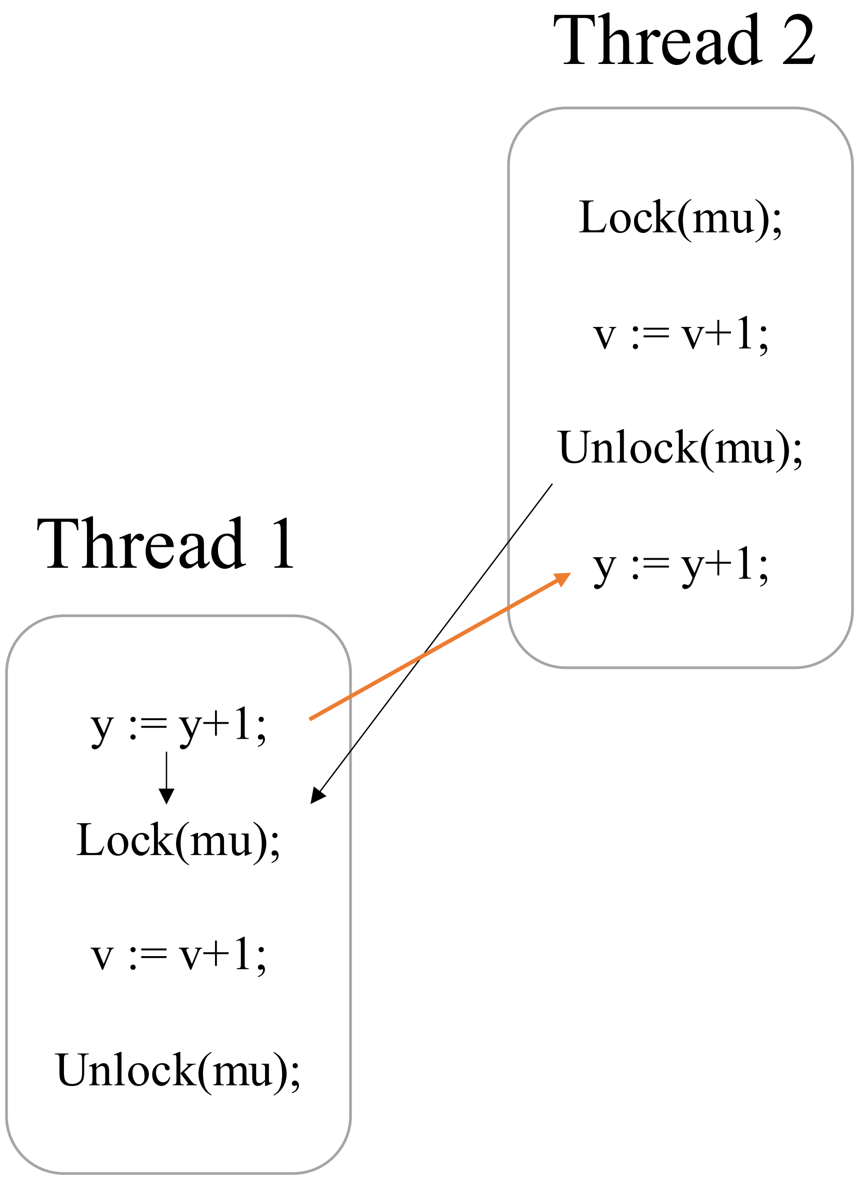
- Difficult to implement
  - Requires per-thread information

- Dependent on the interleaving produced by the scheduler

- Example
  - T1-acc(v) happens before T2-acc(v)
  - T1-acc(y) happens before T1-acc(v)
  - T2-acc(v) happens before T2-acc(y)
  - Conclusion: no race on Y!
  - Finding doesn’t generalize

Flaws of *Happens-before*

- Difficult to implement
  - Requires per-thread information
- Dependent on the interleaving produced by the scheduler
- Example
  - T1-acc(v) happens before T2-acc(v)
  - T1-acc(y) happens before T1-acc(v)
  - T2-acc(v) happens before T2-acc(y)
  - Conclusion: no race on Y!
  - Finding doesn’t generalize

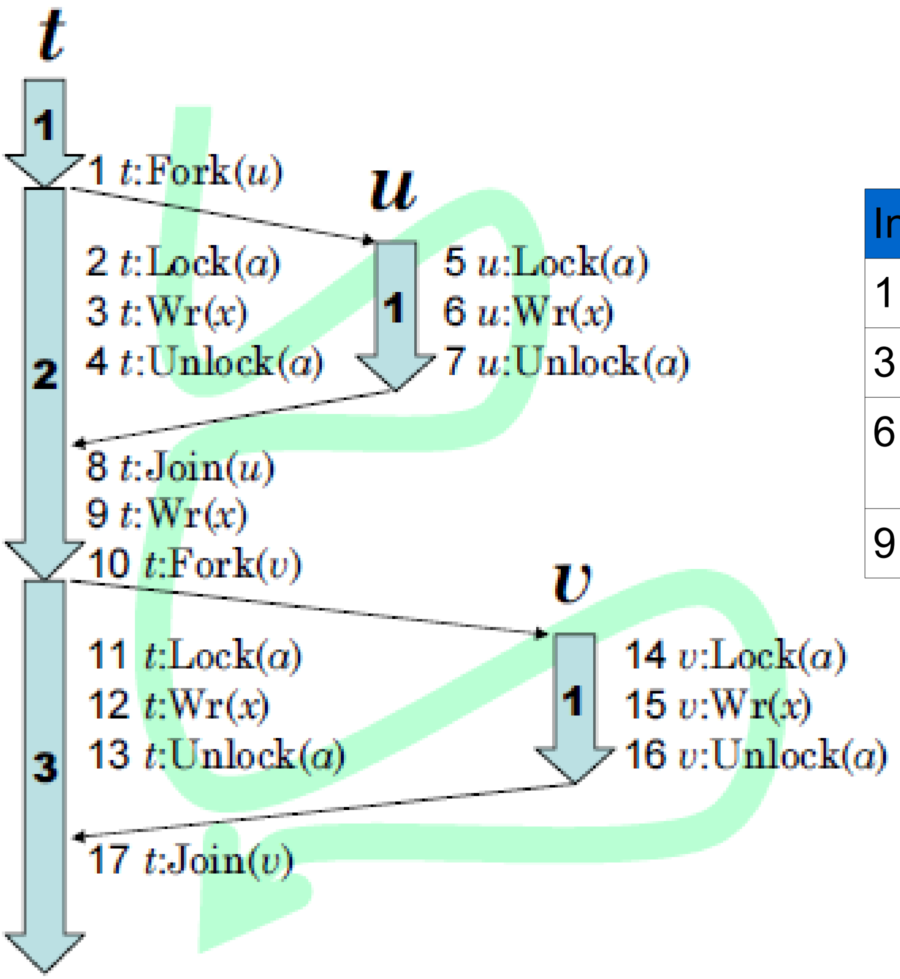


Dynamic Race Detection Summary

- Lockset: verify locking discipline for shared memory
  - ✓ Detect race regardless of thread scheduling
  - ✗ False positives because other synchronization primitives (fork/join, signal/wait) not supported
- Happens-before: track partial order of program events
  - ✓ Supports general synchronization primitives
  - ✗ Higher overhead compared to lockset
  - ✗ False negatives due to sensitivity to thread scheduling

RaceTrack = Lockset + Happens-before

False positive using Lockset



Tracking accesses to X

Inst	State	Lockset
1	Virgin	{ }
3	Exclusive:**t**	{ }
6	Shared Modified	{**a**}
9	Report race	{ }

RaceTrack Notations

Notation	Meaning
L_t	Lockset of thread **t**
C_x	Lockset of memory **x**
B_u	Vector clock of thread **u**
S_x	Threadset of memory **x**
$\mathbf{t_i}$	Thread **t** at clock time **i**

$$
\begin{aligned}
|V| &\triangleq |\{t \in T : V(t) > 0\}| \\
Inc(V,t) &\triangleq u \mapsto \text{if } u = t \text{ then } V(u) + 1 \text{ else } V(u) \\
Merge(V,W) &\triangleq u \mapsto max(V(u), W(u)) \\
Remove(V,W) &\triangleq u \mapsto \text{if } V(u) \leq W(u) \text{ then } 0 \text{ else } V(u)
\end{aligned}
$$

RaceTrack Algorithm

Notation	Meaning
L_t	Lockset of thread **t**
C_x	Lockset of memory **x**
B_t	Vector clock of thread **t**
S_x	Threadset of memory **x**
$\mathbf{t_1}$	Thread **t** at clock time 1

$$\begin{aligned}
|V| &\triangleq |\{t \in T : V(t) > 0\}| \\
Inc(V,t) &\triangleq u \mapsto \text{if } u = t \text{ then } V(u)+1 \text{ else } V(u) \\
Merge(V,W) &\triangleq u \mapsto max(V(u), W(u)) \\
Remove(V,W) &\triangleq u \mapsto \text{if } V(u) \leq W(u) \text{ then } 0 \text{ else } V(u)
\end{aligned}$$

At t:Lock(l):
$\quad L_t \leftarrow L_t \cup \{l\}$

At t:Unlock(l):
$\quad L_t \leftarrow L_t - \{l\}$

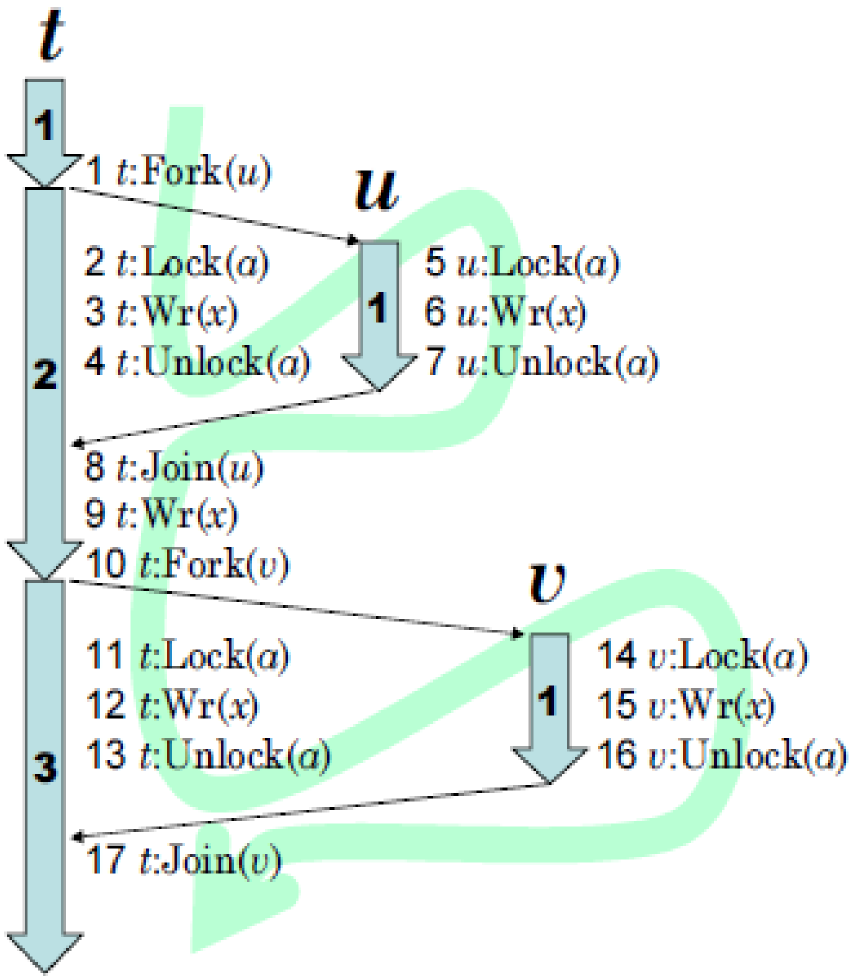
At t:Fork(u):
$\quad L_u \leftarrow \{\}$
$\quad B_u \leftarrow Merge(\{\langle u, 1\rangle\}, B_t)$
$\quad B_t \leftarrow Inc(B_t, t)$

At t:Join(u):
$\quad B_t \leftarrow Merge(B_t, B_u)$

At t:Rd(x) or t:Wr(x):
$\quad S_x \leftarrow Merge(Remove(S_x, B_t), \{\langle t, B_t(t)\rangle\})$
$\quad$ if $|S_x| > 1$
$\quad\quad$ then $C_x \leftarrow C_x \cap L_t$
$\quad\quad$ else $C_x \leftarrow L_t$
$\quad$ if $|S_x| > 1 \wedge C_x = \{\}$ then report race

Avoiding Lockset's false positive (1)

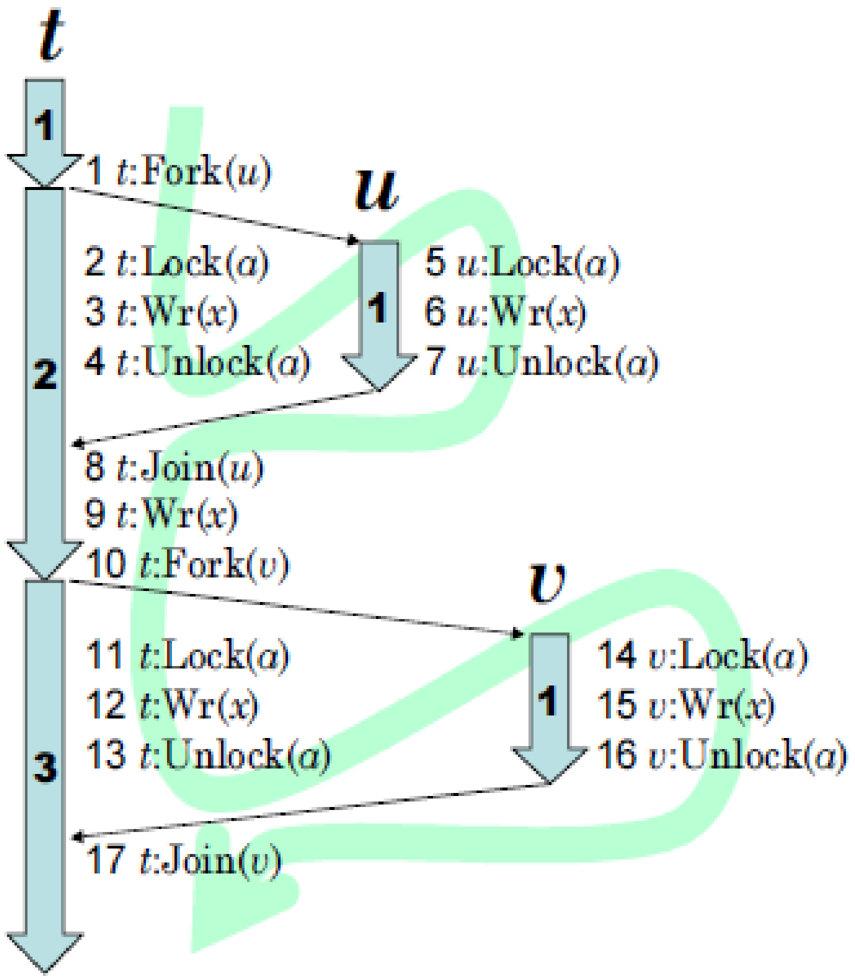
Notation	Meaning
L_t	Lockset of thread **t**
C_x	Lockset of memory **x**
B_t	Vector clock of thread **t**
S_x	Threadset of memory **x**
t_1	Thread **t** at clock time 1



Inst	C_x	S_x	L_t	B_t	L_u	B_u
0	All	{ }	{ }	$\{t_1\}$	-	-
1				$\{t_2\}$	{ }	$\{t_1, u_1\}$
2			{a}			
3	{a}	$\{t_2\}$				
4			{ }			
5					{a}	
6		$\{t_2, u_1\}$				
7					{ }	
8				$\{t_2, u_1\}$	-	-

Avoiding Lockset's false positive (2)

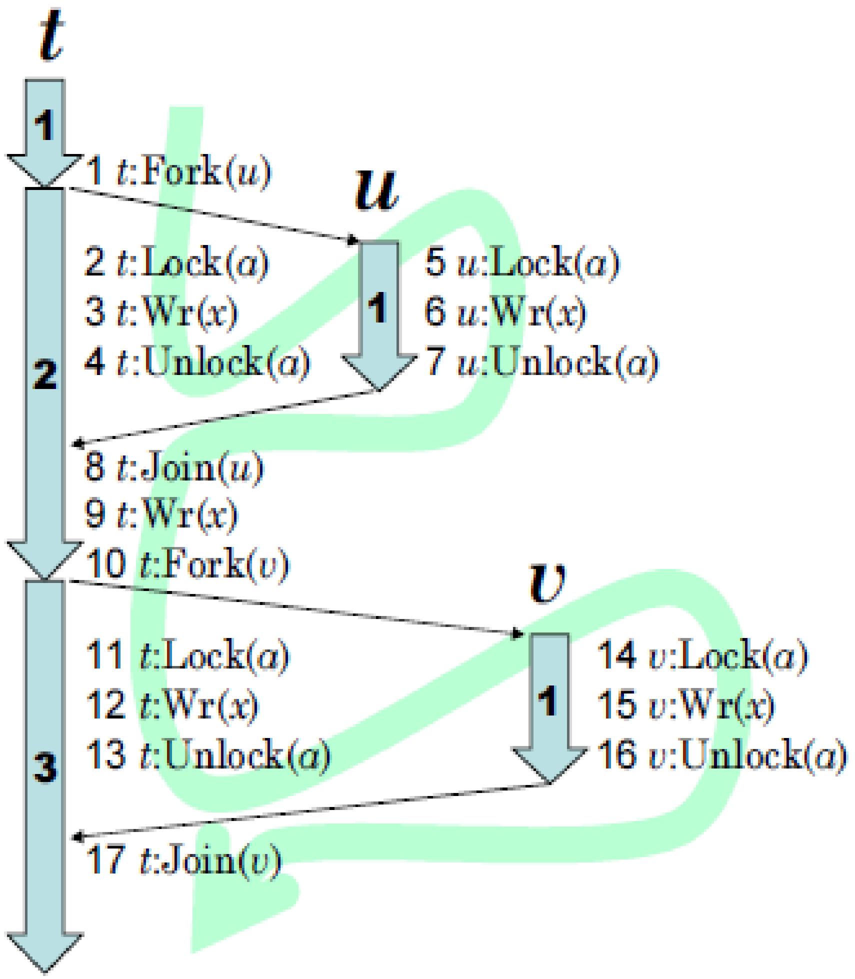
Notation	Meaning
L_t	Lockset of thread **t**
C_x	Lockset of memory **x**
B_t	Vector clock of thread **t**
S_x	Threadset of memory **x**
$\mathbf{t_1}$	Thread **t** at clock time 1



Inst	C_x	S_x	L_t	B_t	L_v	B_v
8	{a}	$\{t_2,u_1\}$	{ }	$\{t_2,u_1\}$	-	-
9	{ }	$\{t_2\}$				
10				$\{t_3,u_1\}$	{ }	$\{t_2,v_1\}$
11			{a}			
12	{a}	$\{t_3\}$				
13			{ }			
14					{a}	
15		$\{t_3,v_1\}$				
16					{ }	

Notation	Meaning
L_t	Lockset of thread **t**
C_x	Lockset of memory **x**
B_t	Vector clock of thread **t**
S_x	Threadset of memory **x**
t_1	Thread **t** at clock time 1

Avoiding Lockset's false positive (2)



Inst	C_x	S_x	L_t	B_t	L_v	B_v
8	{a}	$\{t_2,u_1\}$	{ }	$\{t_2,u_1\}$	-	-
9	{ }	$\{t_2\}$				
10				$\{t_3,u_1\}$	{ }	$\{t_2,v_1\}$
11			{a}			
12	{a}	$\{t_3\}$				
13			{ }			
14					{a}	
15		$\{t_3,v_1\}$				
16					{ }	

Only one thread!

Are we done?