Race Detection

Chris Rossbach and Calvin Lin

cs380p

Outline

Agenda

- Race Detection

Acknowledgements:

- *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

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

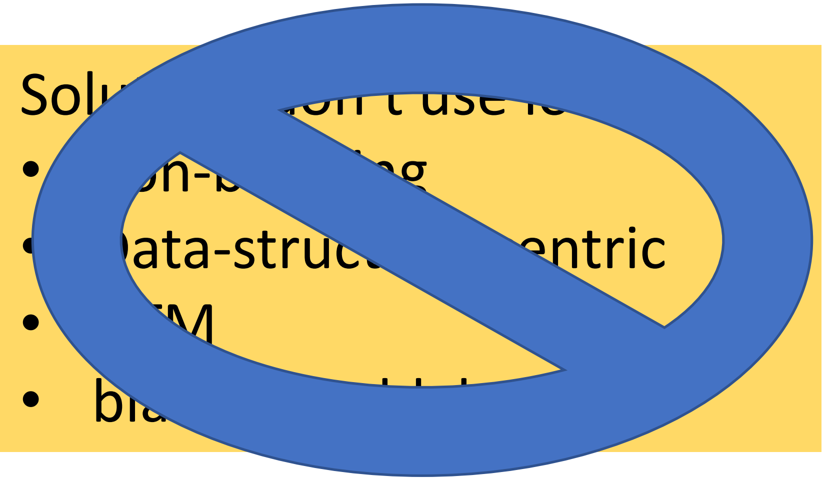
Solution: don’t use locks

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

Race Detection

Locks: a litany of problems

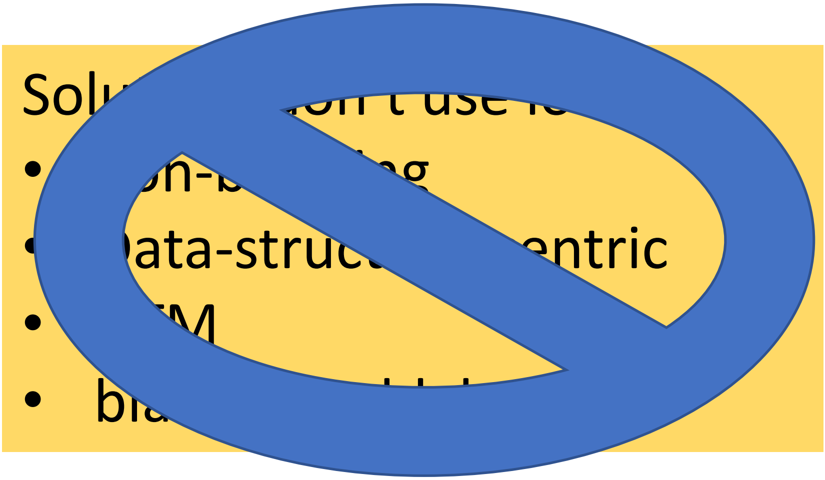
- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance



Race Detection

Locks: a litany of problems

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



Use locks!

- But automate bug-finding!

Races

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

```

Read-Write(X);

```

Races

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

- Is there a race here?

Races

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

- 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

```
1 Lock(lock);            1
2 Read-Write(X);         2 Read-Write(X);
3 Unlock(lock);          3
```

- Is there a race here?
- What is a race?
- Informally: accesses with r
- Formally:
  - >1 threads access same ite
  - No intervening synchroniza
  - At least one access is a writ

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);

}
```

Unsynchronized access can be

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);

}
```

Unsynchronized access can be

- Benign due to fork/join

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);

}
```

Unsynchronized access can be

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

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);

}
```

Unsynchronized access can be

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

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);

}
```

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

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

Detecting Races

How to detect races:

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

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

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

```

Read-Write(X);

```

Static Data Race Detection

Static Data Race Detection

- Type-based analysis
  - Language type system augmented
    - express common synchronization relationships:
    - correct typing→no data races
  - Difficult to do *(although…cf. Rust)*
  - Often restricts the type of synchronization primitives

Static Data Race Detection

- Type-based analysis
  - Language type system augmented
    - express common synchronization relationships:
    - correct typing→no data races
  - Difficult to do *(although…cf. Rust)*
  - Often restricts the type of synchronization primitives
- Language features
  - e.g., use of monitors
  - Only works for static data – not dynamic data

Static Data Race Detection

- Type-based analysis
  - Language type system augmented
    - express common synchronization relationships:
    - correct typing→no data races
  - Difficult to do *(although…cf. Rust)*
  - Often restricts the type of synchronization primitives
- Language features
  - e.g., use of monitors
  - Only works for static data – not dynamic data
- Model Checking

Static Data Race Detection

- Type-based analysis
  - Language type system augmented
    - express common synchronization relationships:
    - correct typing→no data races
  - Difficult to do *(although…cf. Rust)*
  - Often 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 *(although…cf. Rust)*
  - Often 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

```
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 *(although…cf. Rust)*
  - Often 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

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

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

```

Read-Write(X);

```

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
  - 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

Narrow down set of locks maybe protecting v

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.

Lockset Algorithm Example

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

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

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

Lockset Algorithm Example

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

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

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

Lockset Algorithm Example

thread t

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

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

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

Lockset Algorithm Example

thread t

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

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

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

$C(v) \cap locks\_held(t)$

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++;		{} ACK! race
unlock(lockB);	{}	

Lockset Algorithm Example

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

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 when *“happens-before”* cannot be known

Happens-before

Happens-before

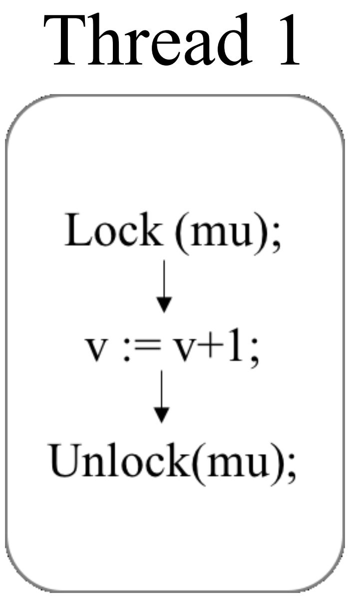
- *Happens-before* relation
  - Within single thread
  - Between threads

Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing vars not ordered by *happens-before* → race

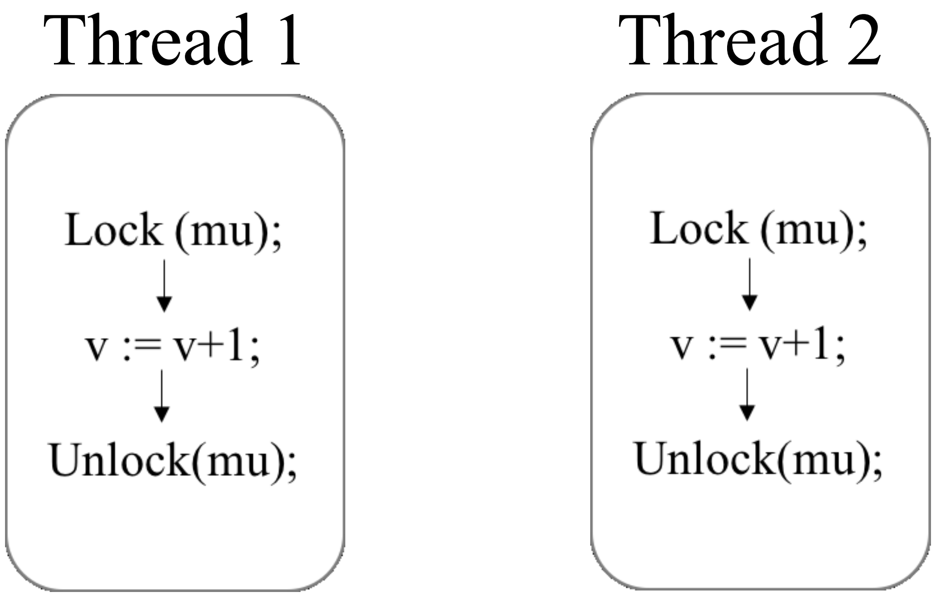
Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing vars not ordered by *happens-before* → race



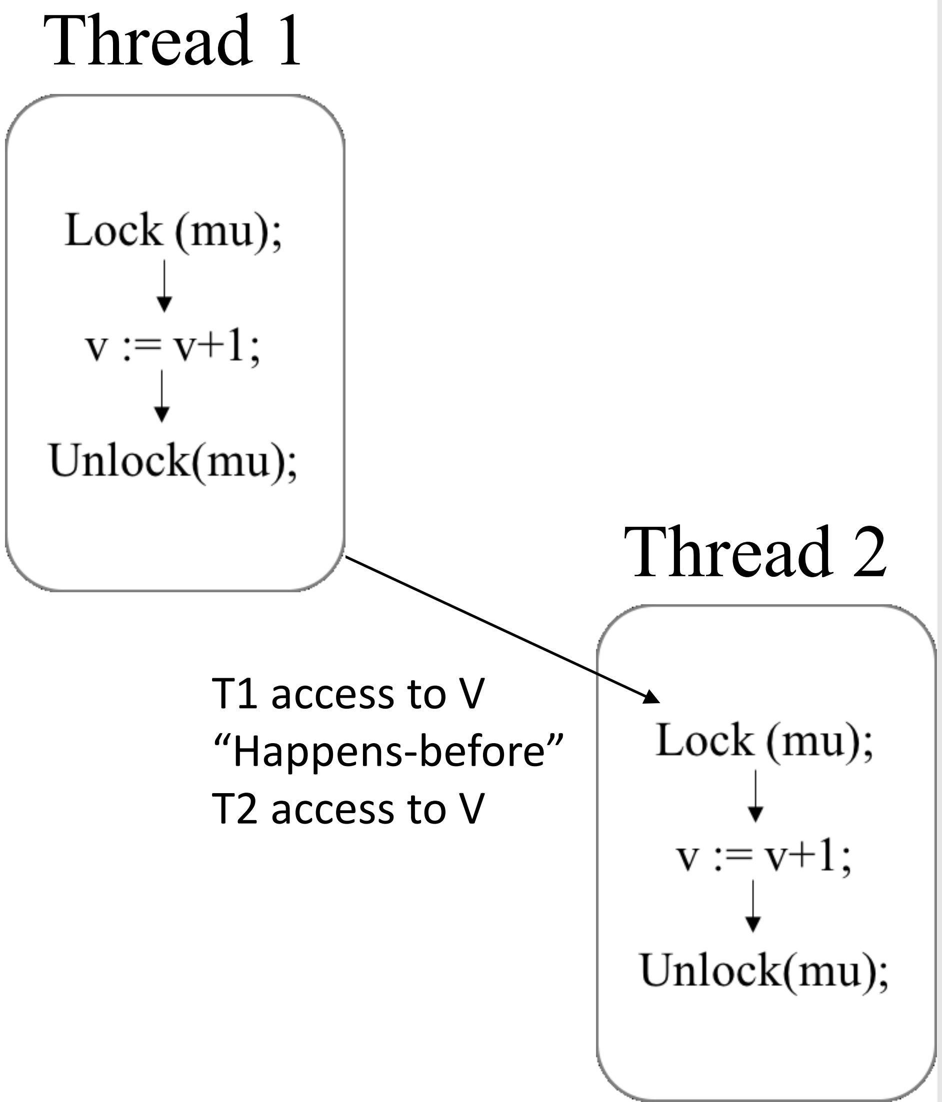
Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing vars not ordered by *happens-before* → race



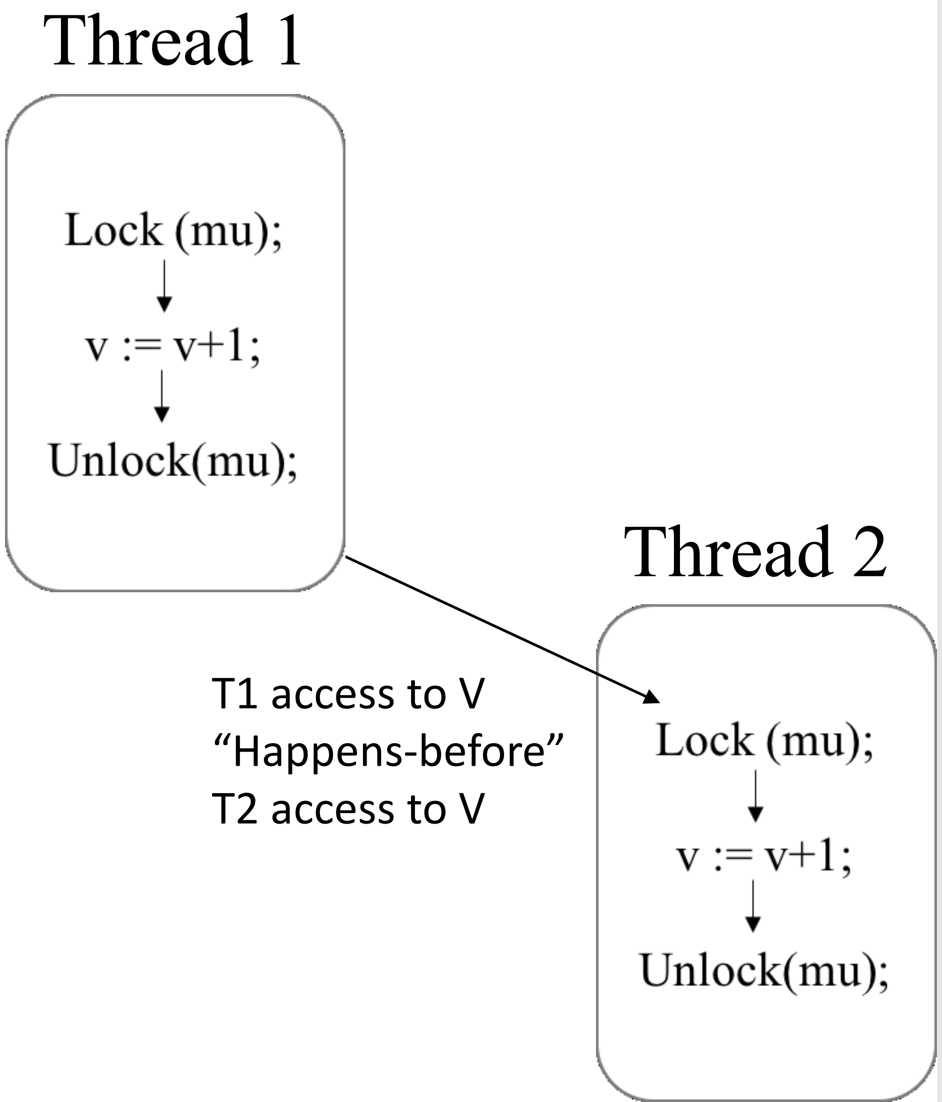
Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing vars not ordered by *happens-before* → race



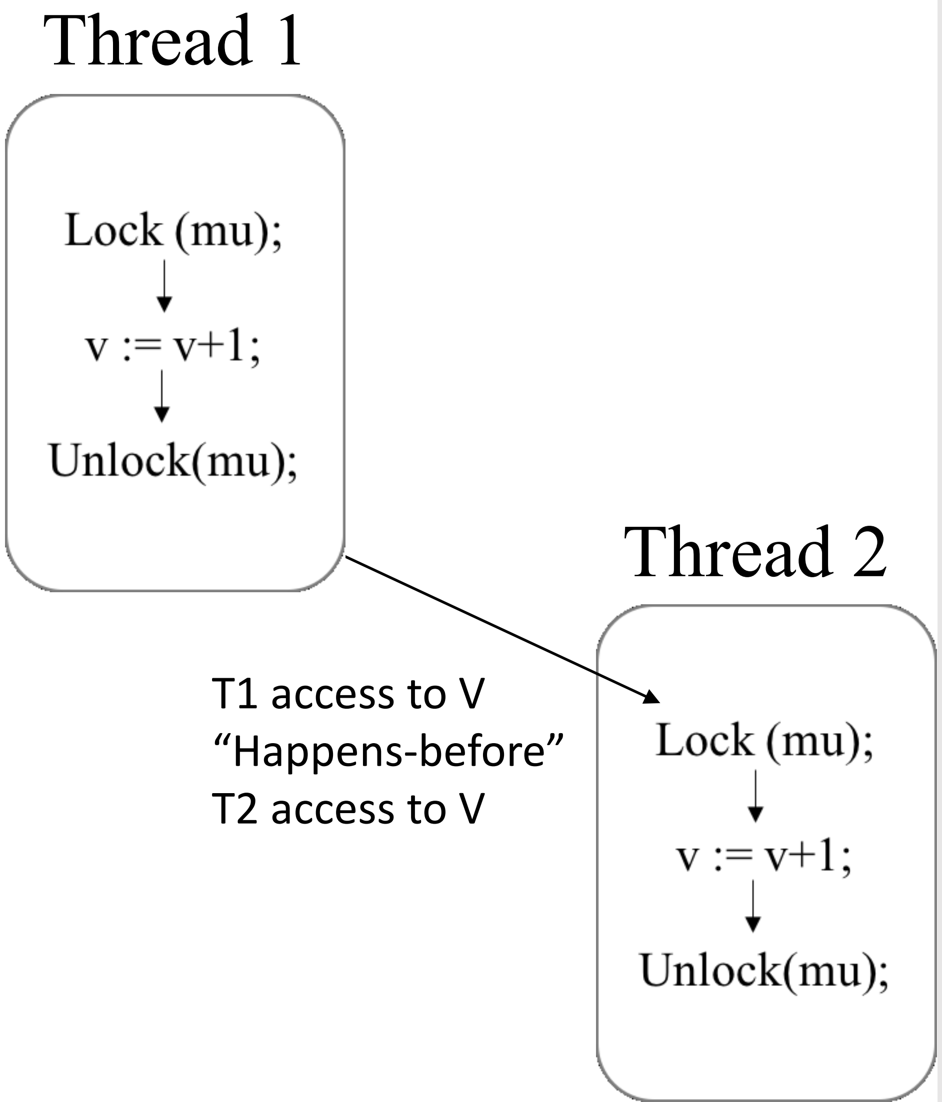
Happens-before

- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing vars not ordered by *happens-before* → race
- Captures locks + dynamism

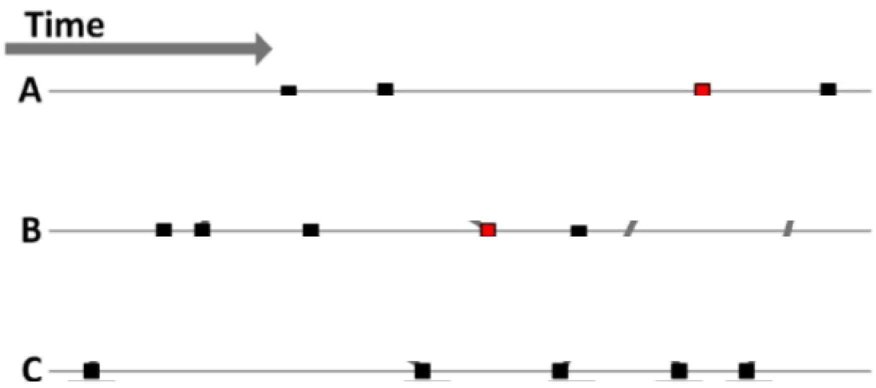


Happens-before

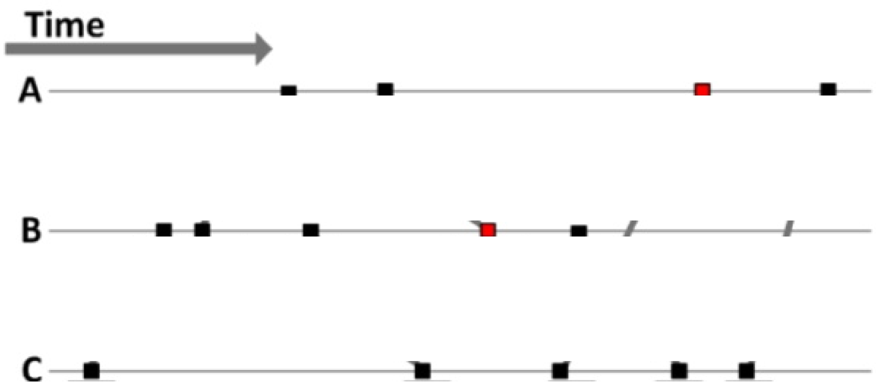
- *Happens-before* relation
  - Within single thread
  - Between threads
- Accessing vars not ordered by *happens-before* → race
- Captures locks + dynamism
- How to track *happens-before?*
  - Sync objects → ordering
  - fork/join/etc → ordering
  - *But how to order events across different threads/CPUs?*



Ordering and Causality

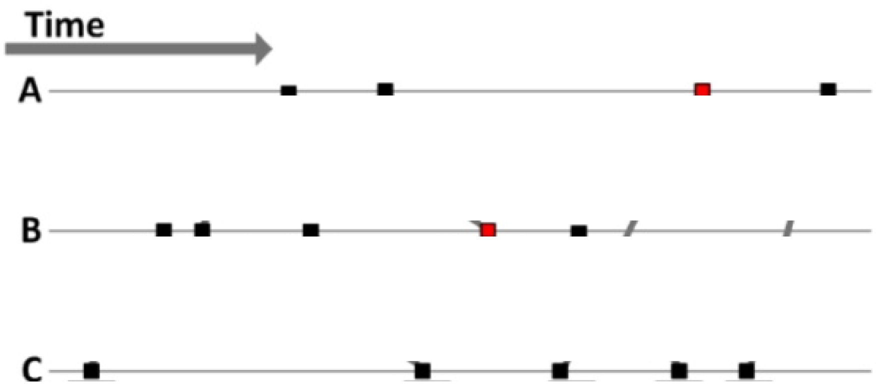


Ordering and Causality



A, B, C have local orders

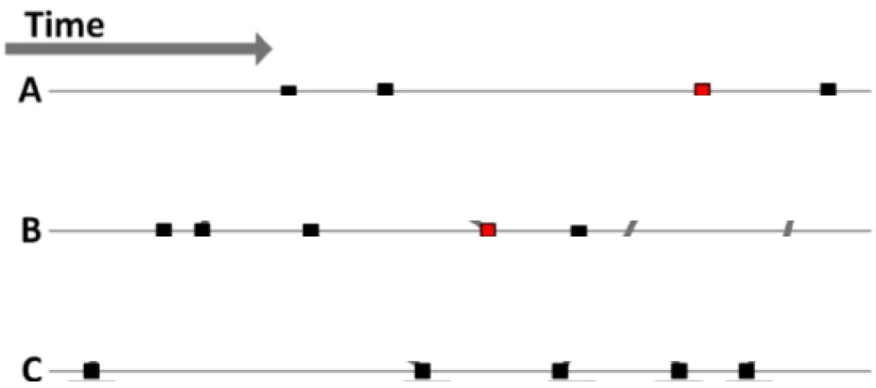
Ordering and Causality



A, B, C have local orders

- Want total order
  - *(Need happens-before)*
  - But only for causality

Ordering and Causality

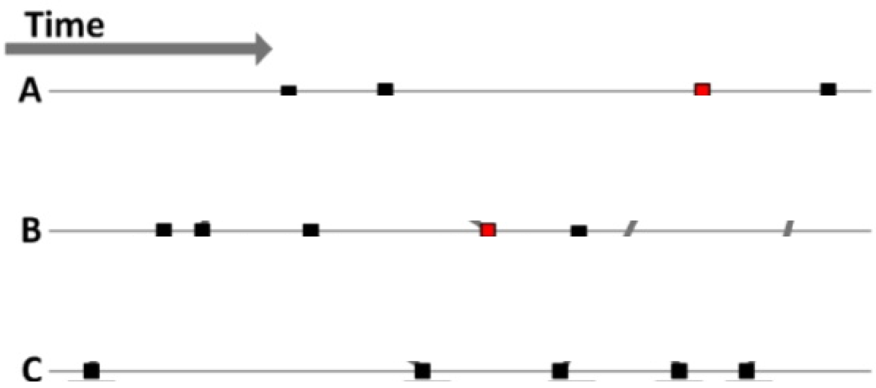


A, B, C have local orders

- Want total order
  - *(Need happens-before)*
  - But only for causality

Different types of clocks

Ordering and Causality



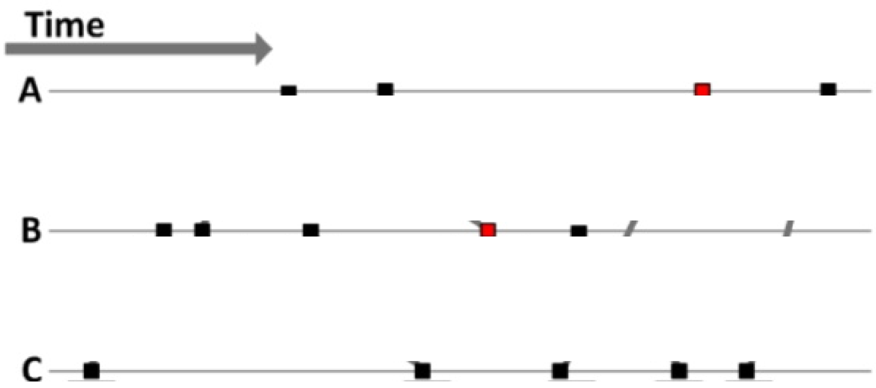
A, B, C have local orders

- Want total order
  - *(Need happens-before)*
  - But only for causality

Different types of clocks

- Physical

Ordering and Causality



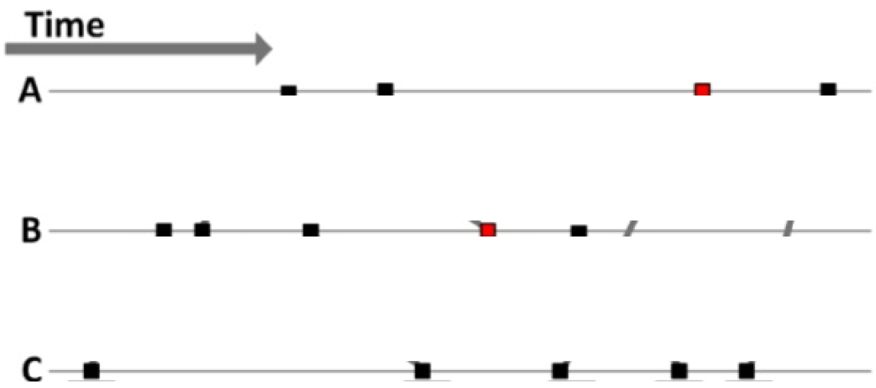
A, B, C have local orders

- Want total order
  - *(Need happens-before)*
  - But only for causality

Different types of clocks

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

Ordering and Causality



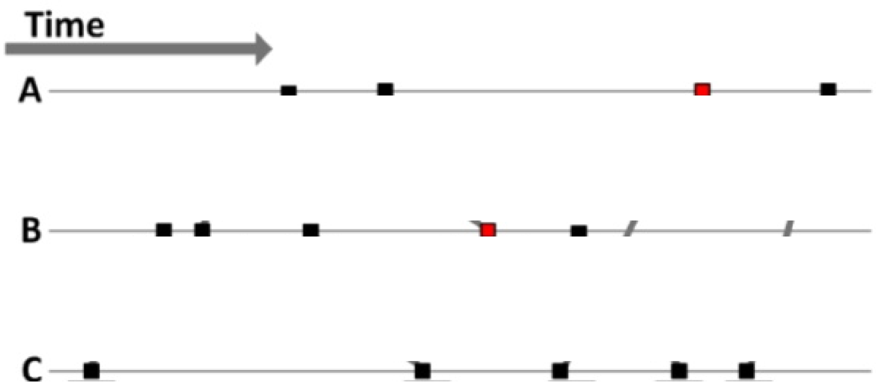
A, B, C have local orders

- Want total order
  - *(Need happens-before)*
  - 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
  - *(Need happens-before)*
  - 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: pairwise knowledge

Strawperson Approach

- Each system records each event, timestamp
- Suppose events occur in *this* real order:

Strawperson Approach

- Each system records each event, timestamp
- Suppose events occur in *this* real order:
  - **Time Tc0:** C sends data to B (before C stops responding)

Strawperson Approach

- Each system records each event, timestamp
- Suppose events occur in *this* real order:
  - **Time Tc0:** C sends data to B (before C stops responding)
  - **Time Ta0:** A asks for work from B

Strawperson Approach

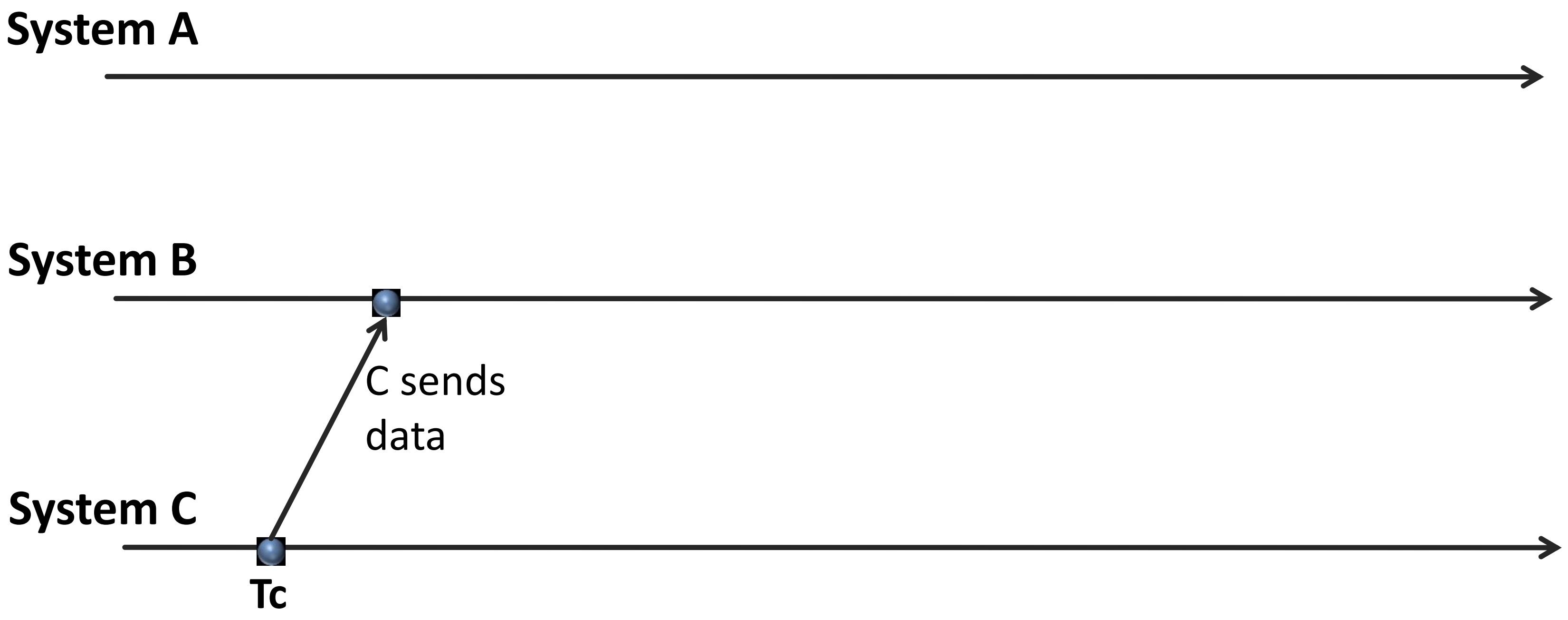
- Each system records each event, timestamp
- Suppose events occur in *this* real order:
  - **Time Tc0:** C sends data to B (before C stops responding)
  - **Time Ta0:** A asks for work from B
  - **Time Tb0:** B asks for data from C

Strawperson Approach

- *Ideally*, construct real order from local timestamps
- Thus, detect *actual* dependency chain Tc→Ta→Tb:

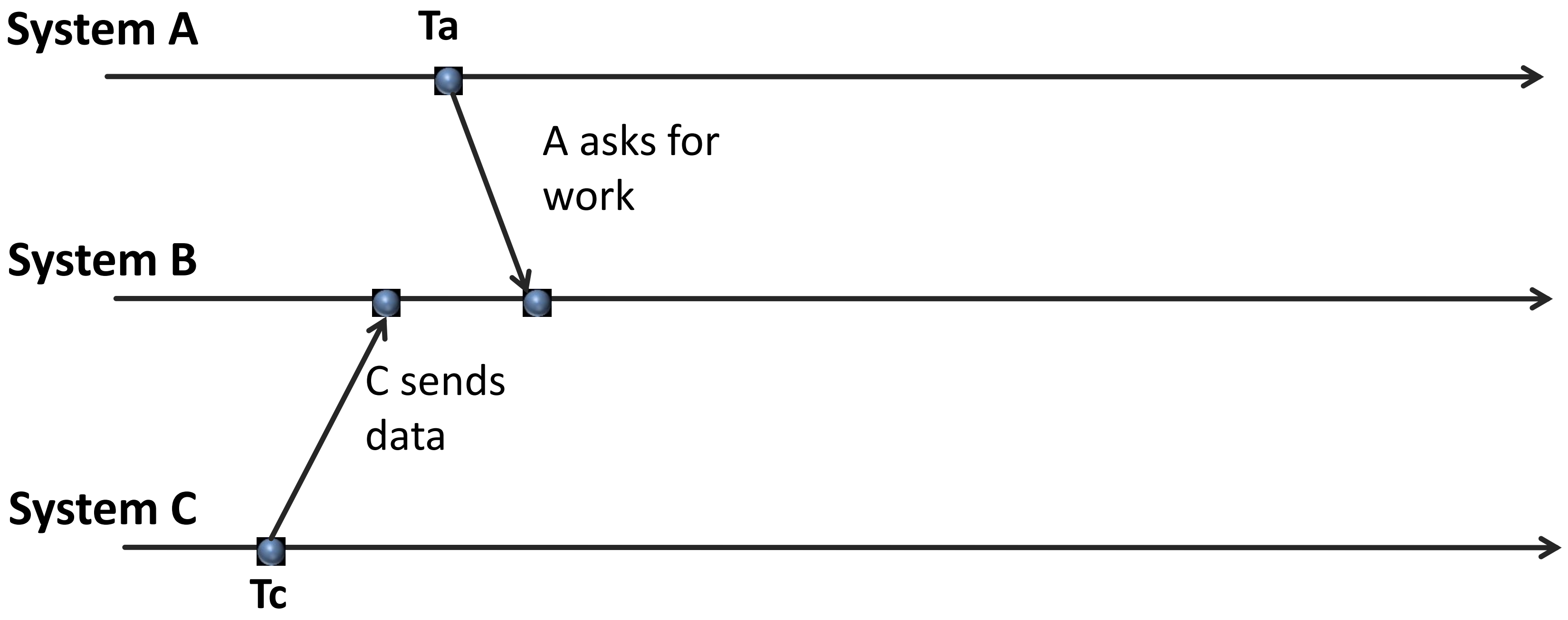
Strawperson Approach

- *Ideally*, construct real order from local timestamps
- Thus, detect *actual* dependency chain Tc→Ta→Tb:



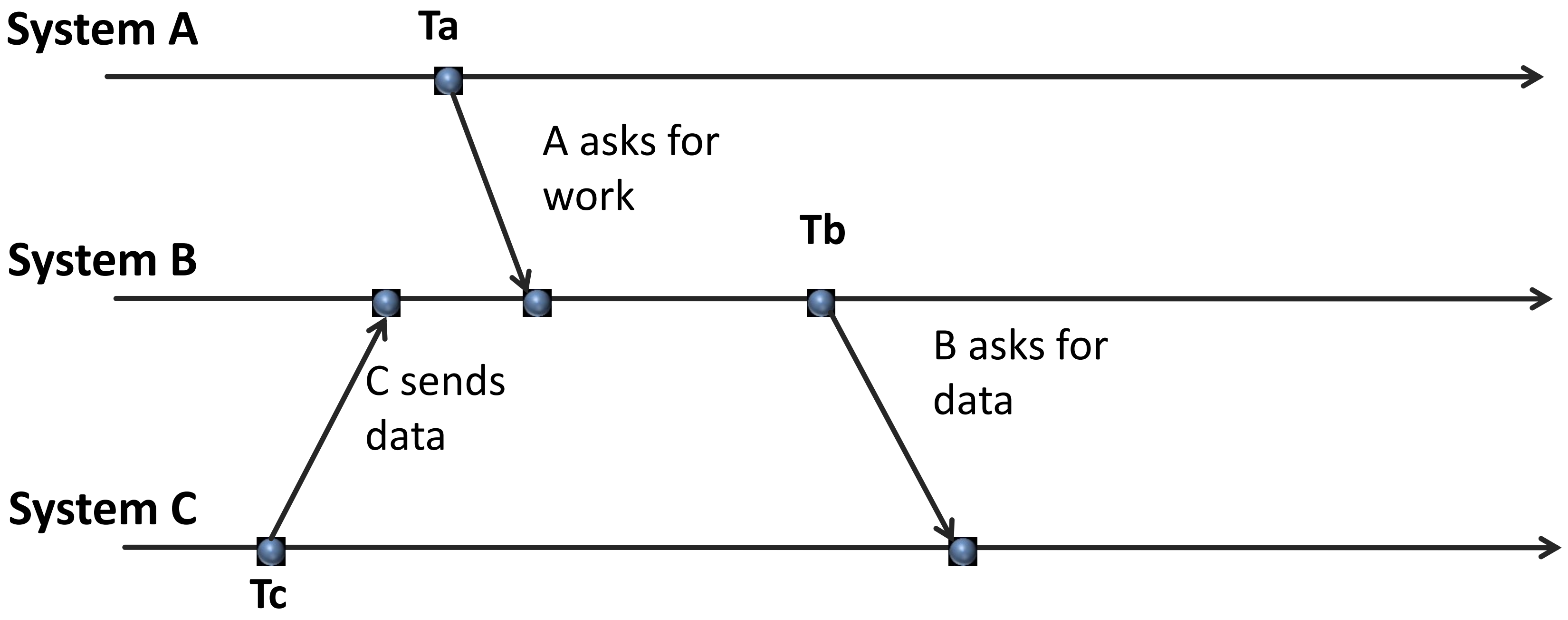
Strawperson Approach

- *Ideally*, construct real order from local timestamps
- Thus, detect *actual* dependency chain Tc→Ta→Tb:



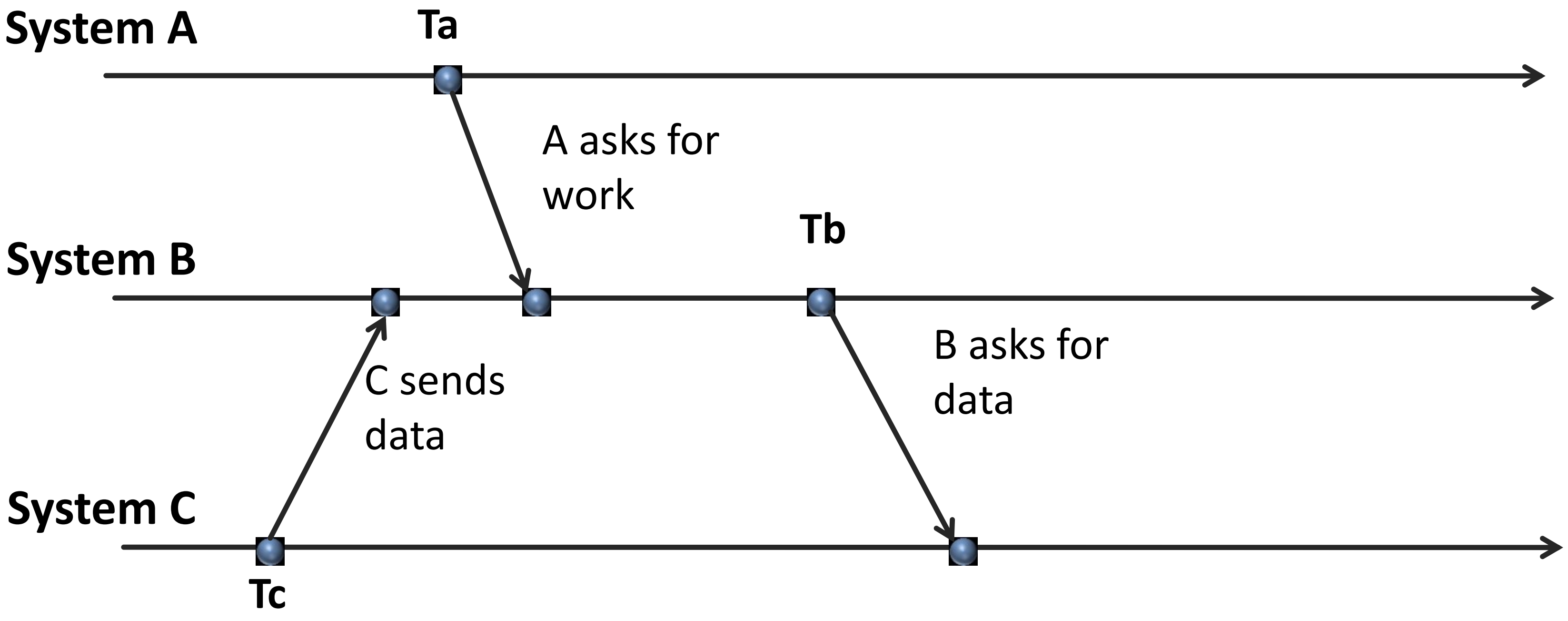
Strawperson Approach

- *Ideally*, construct real order from local timestamps
- Thus, detect *actual* dependency chain Tc→Ta→Tb:



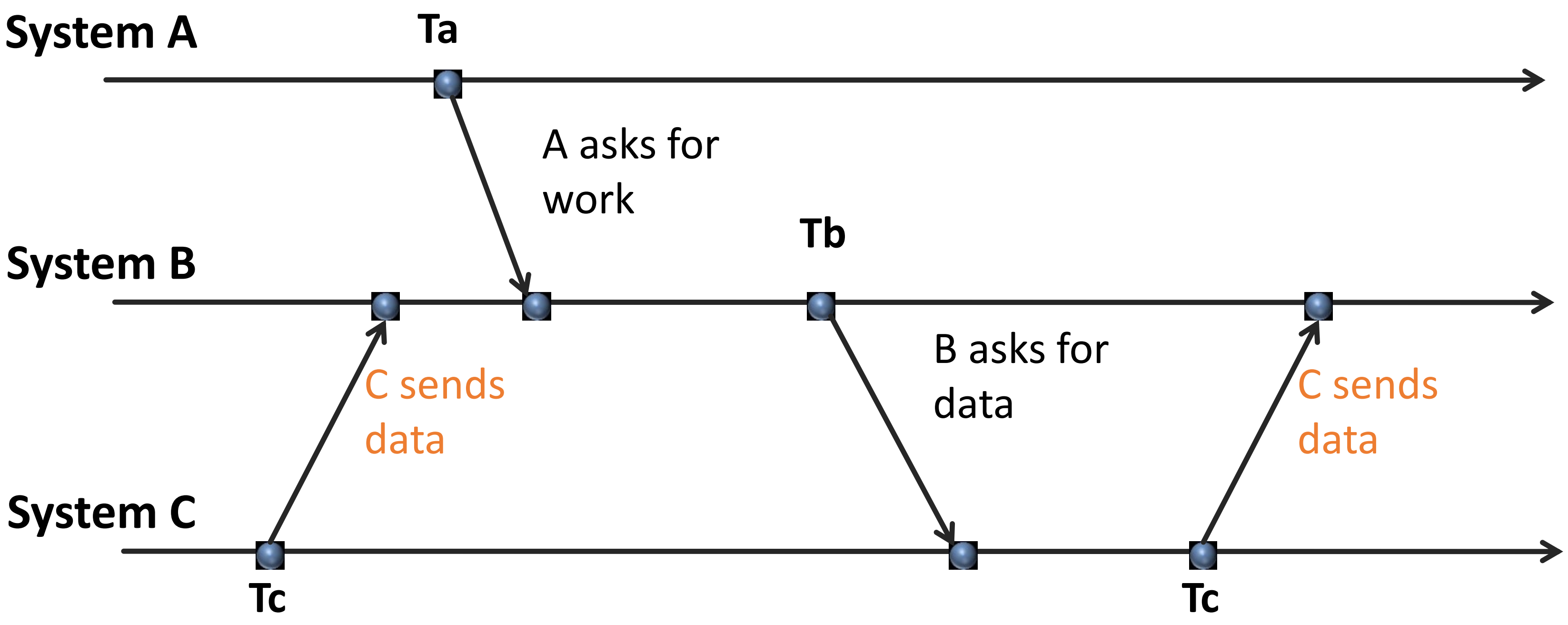
Strawperson Approach

- In reality, we do not know if Tc occurred **before** Ta and Tb. Why?
- In an asynchronous system **clocks are not synchronized**!



Strawperson Approach

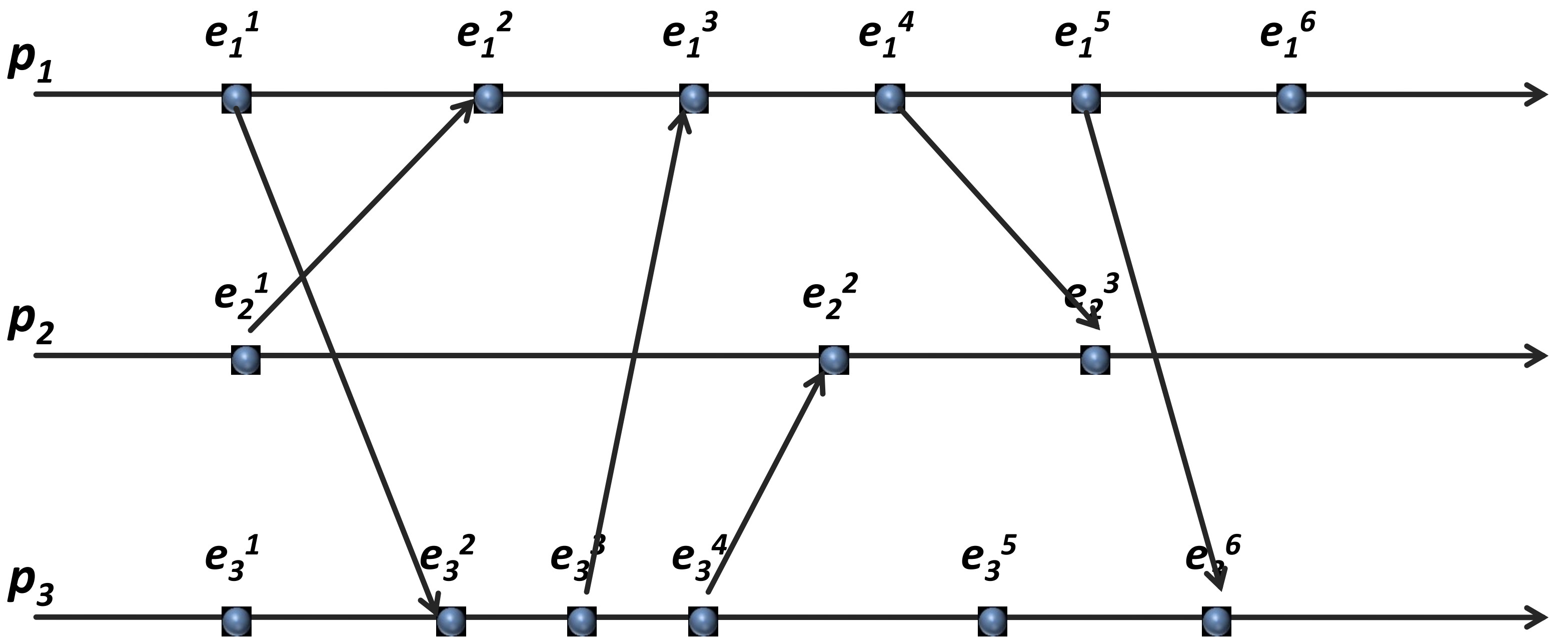
- In reality, we do not know if Tc occurred **before** Ta and Tb. Why?
- In an asynchronous system **clocks are not synchronized**!



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 $e_i^k \rightarrow e_i^m$
- Send of message always precedes receipt :
  - 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



local events precede one another → precede one another globally:

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

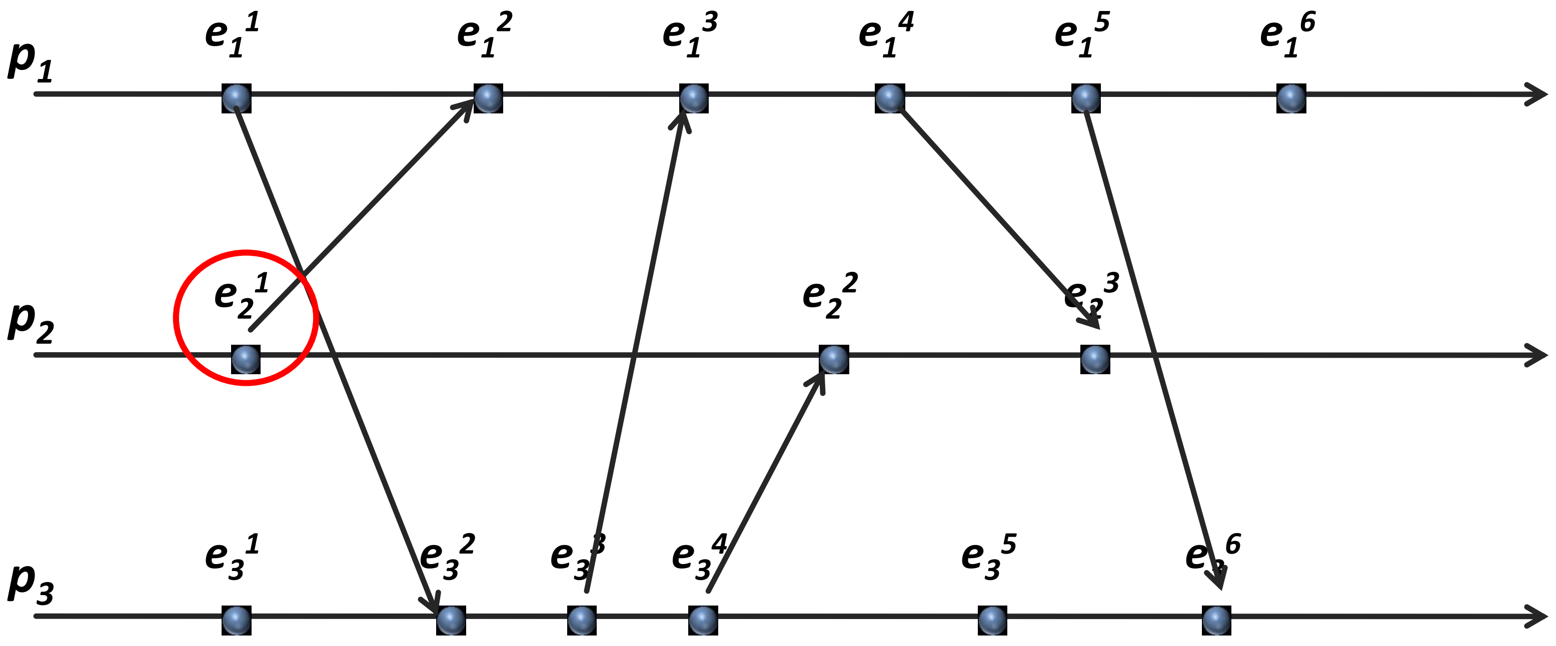
Sending a message always precedes receipt of that message:

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



local events precede one another → precede one another globally:

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

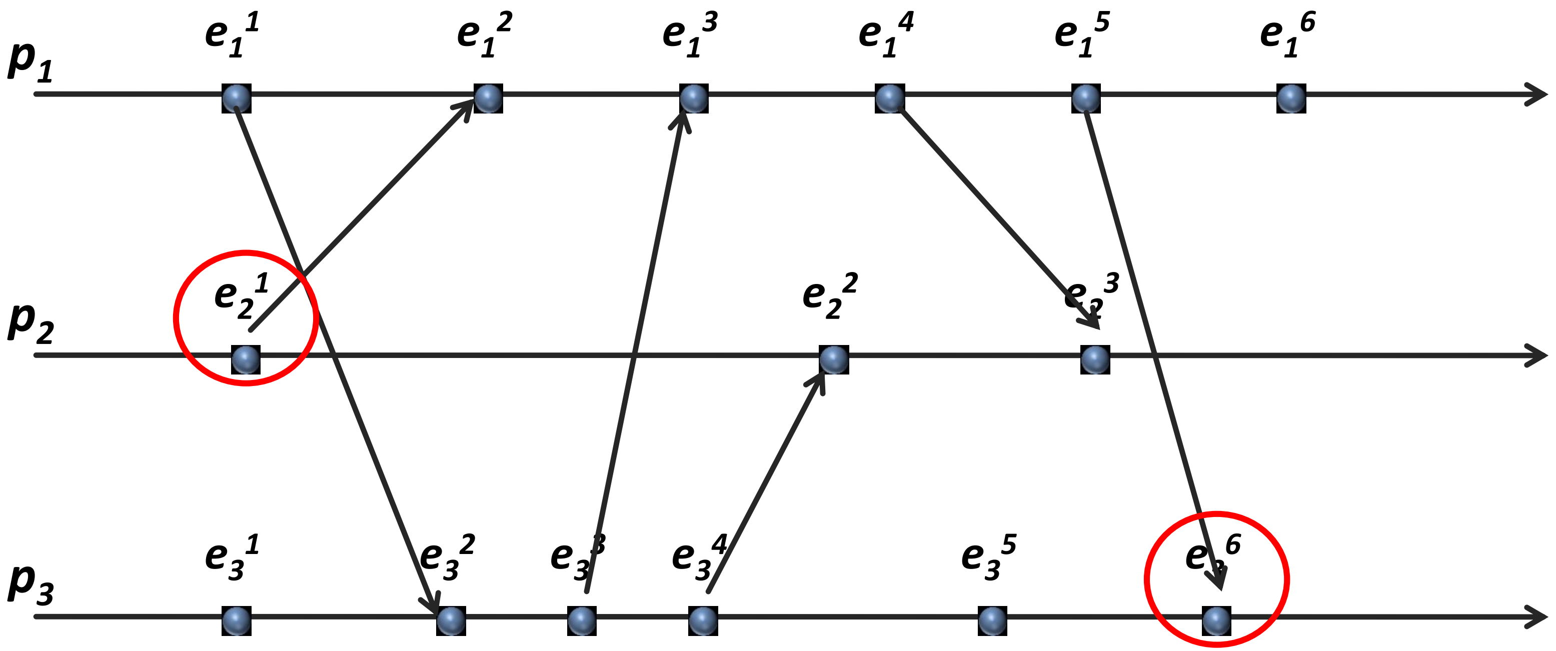
Sending a message always precedes receipt of that message:

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



local events precede one another → precede one another globally:

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

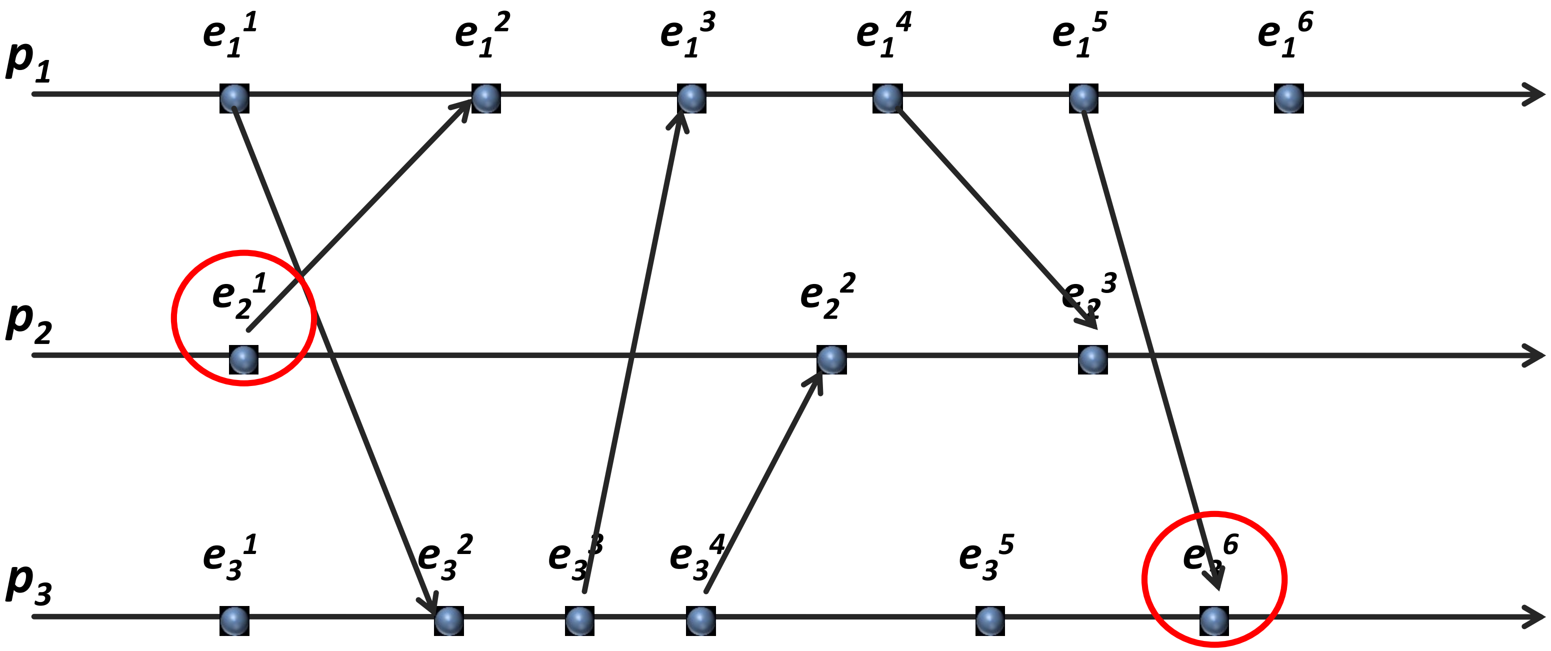
Sending a message always precedes receipt of that message:

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



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 $e_i^k \rightarrow e_i^m$

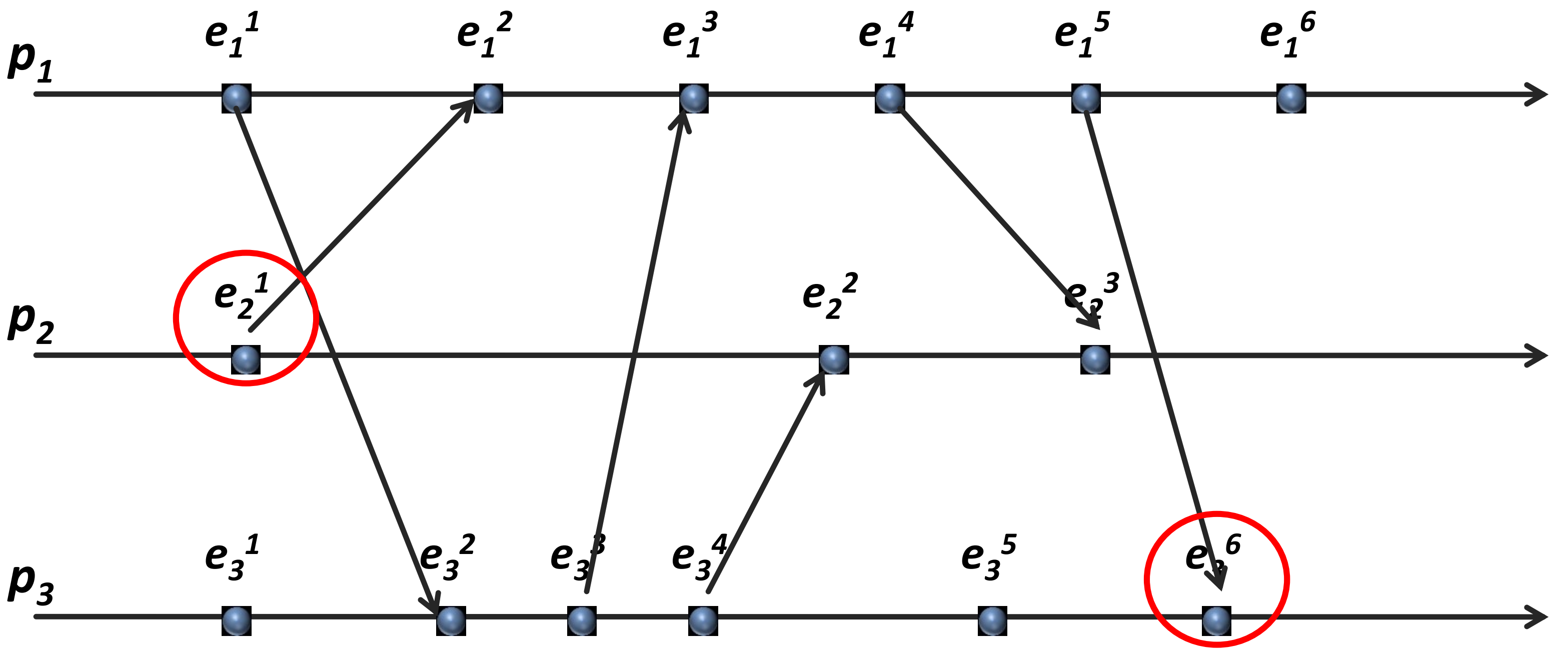
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

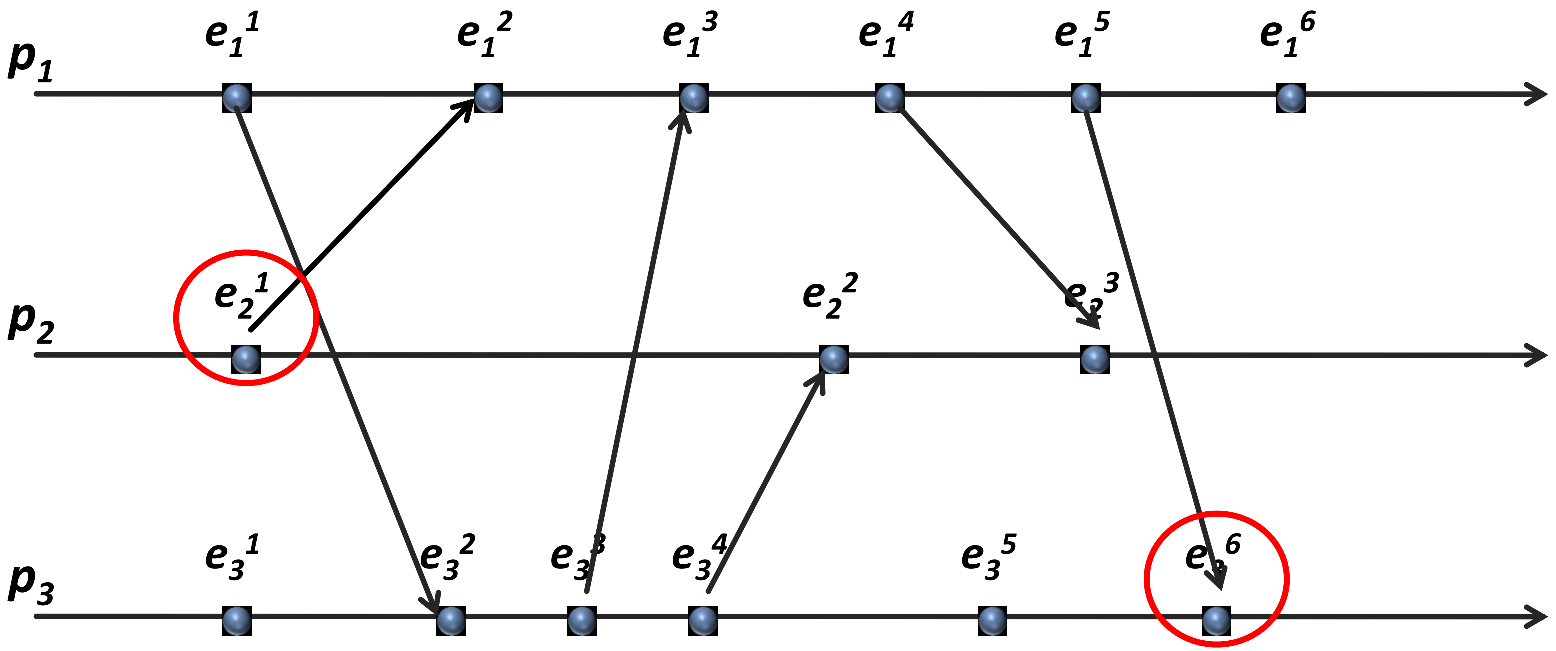
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

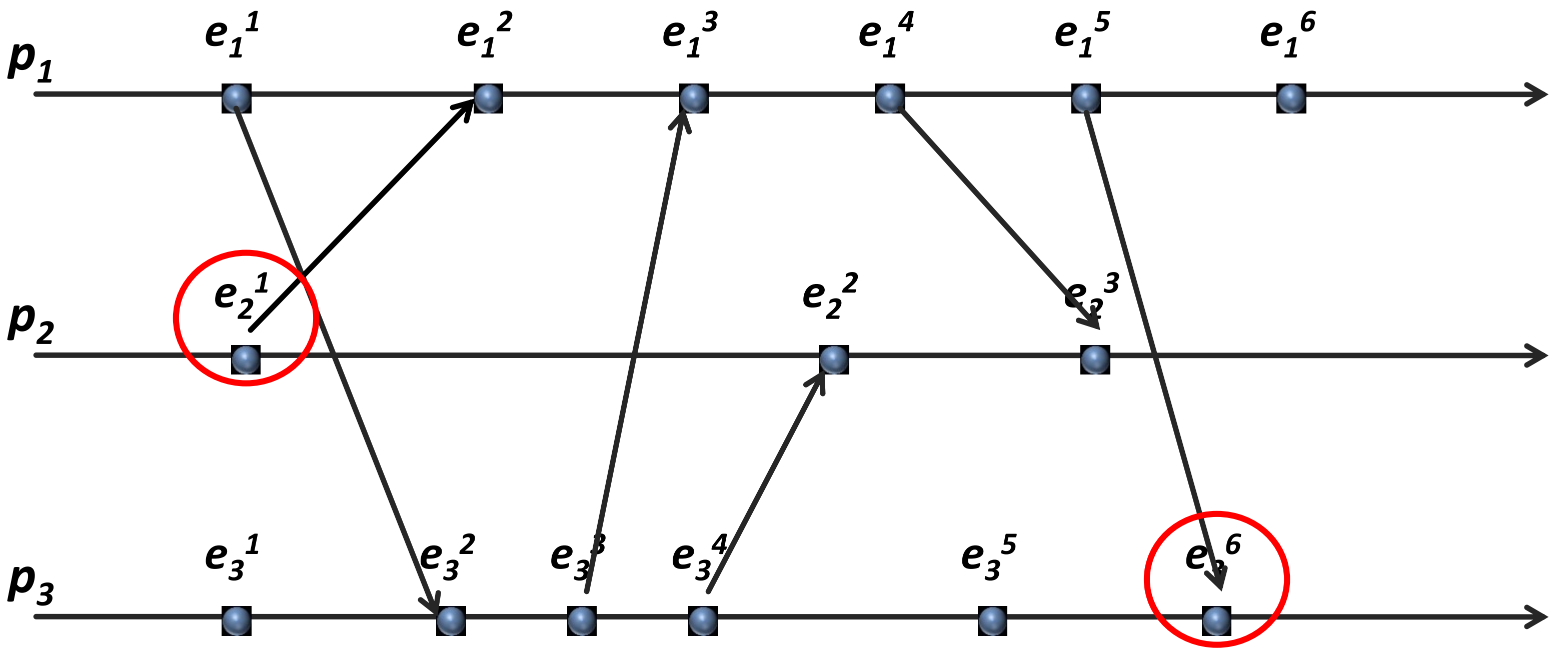
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

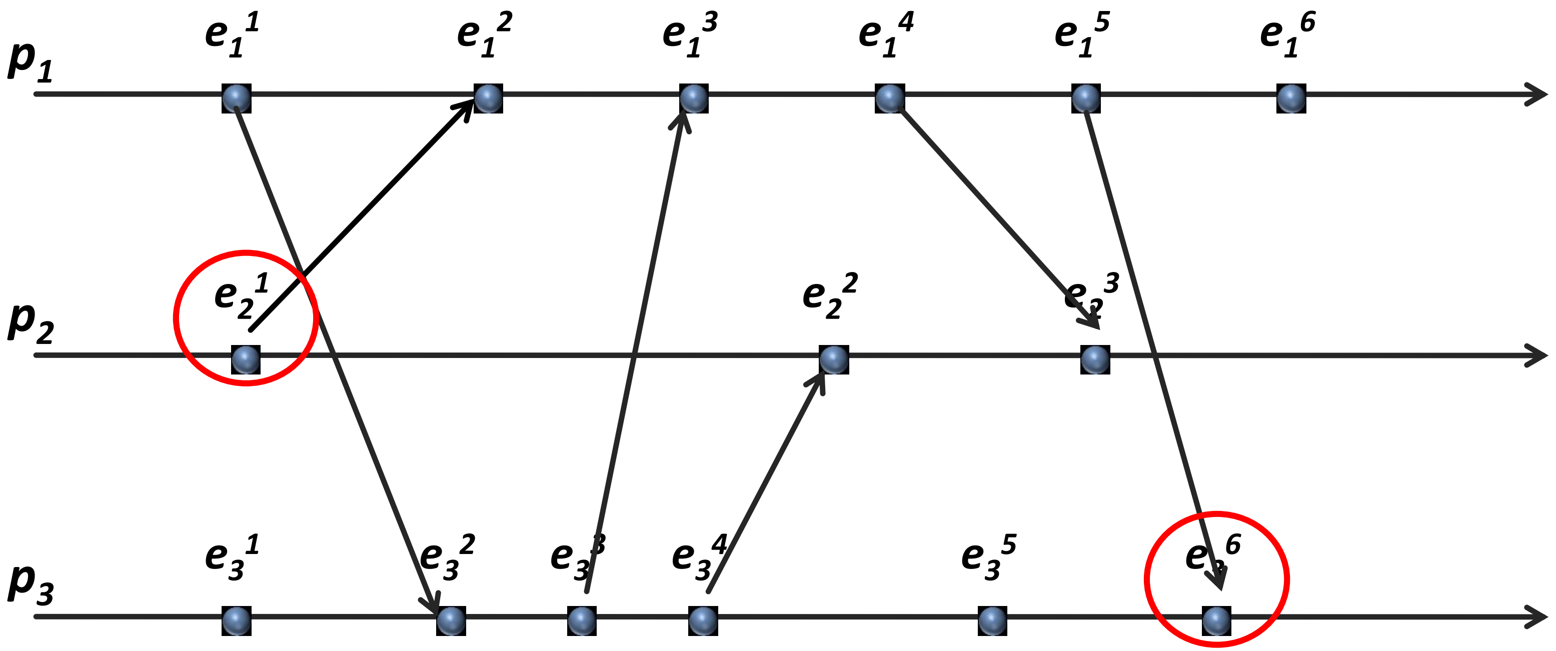
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

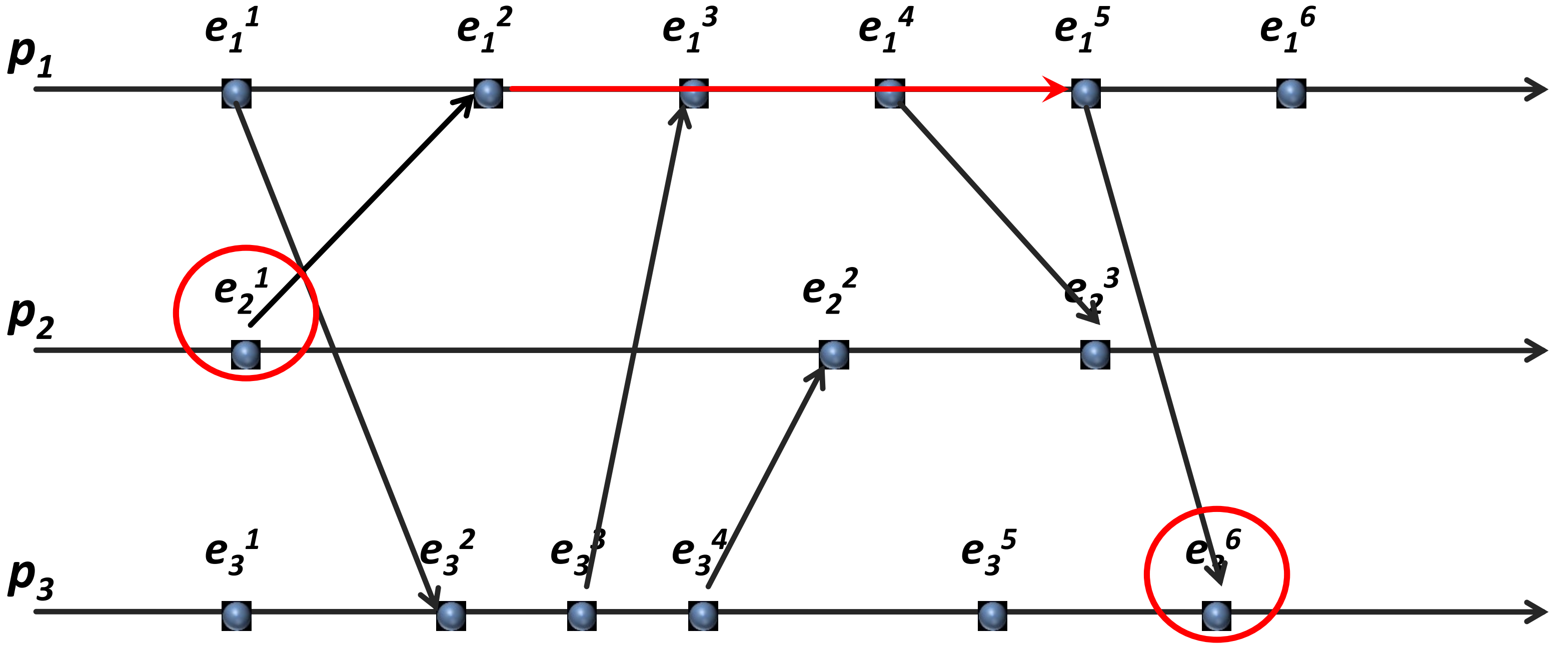
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

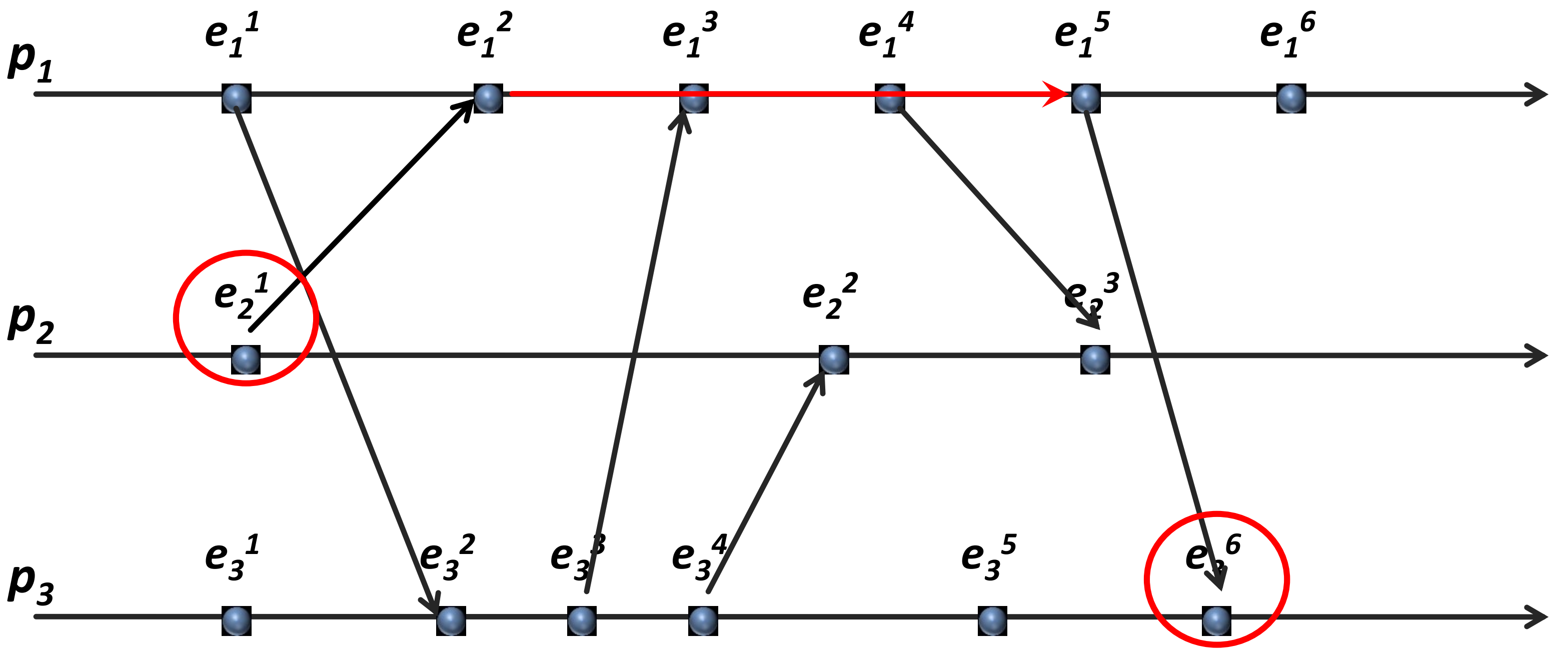
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

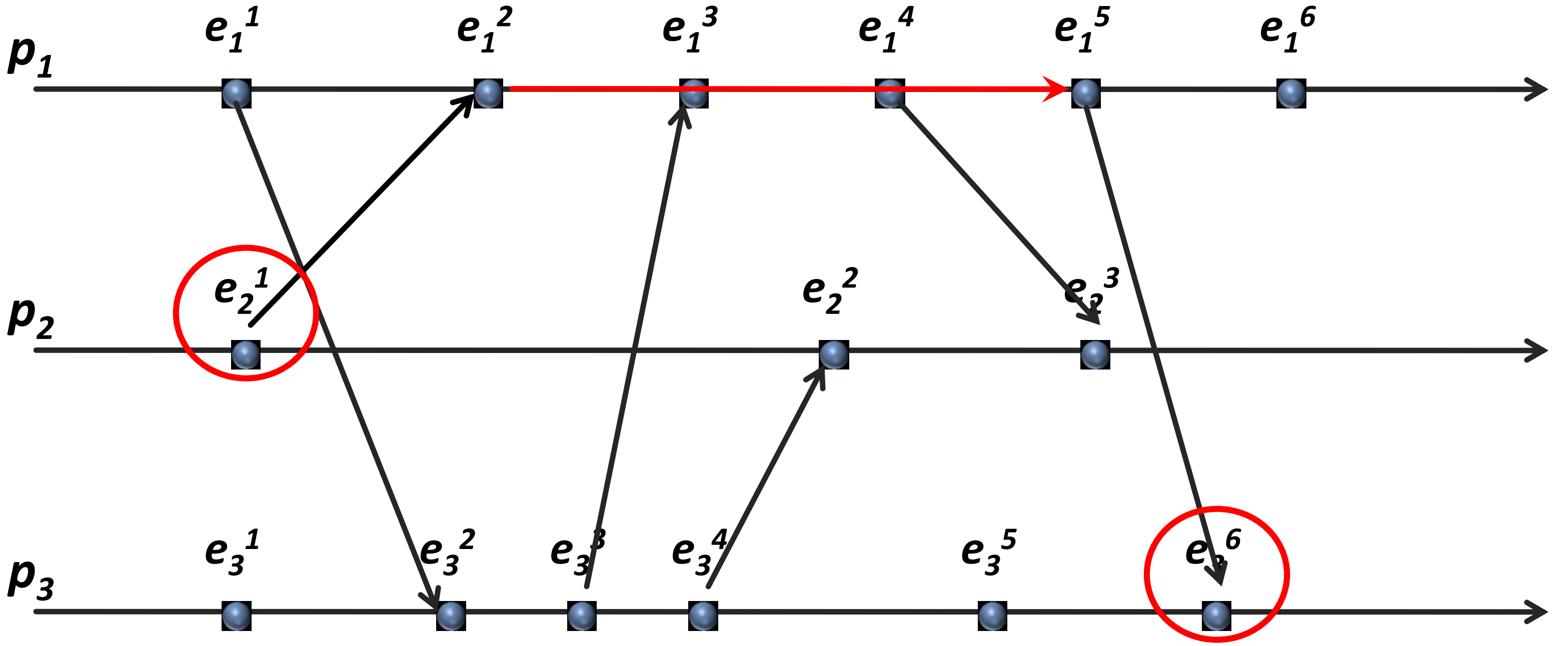
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

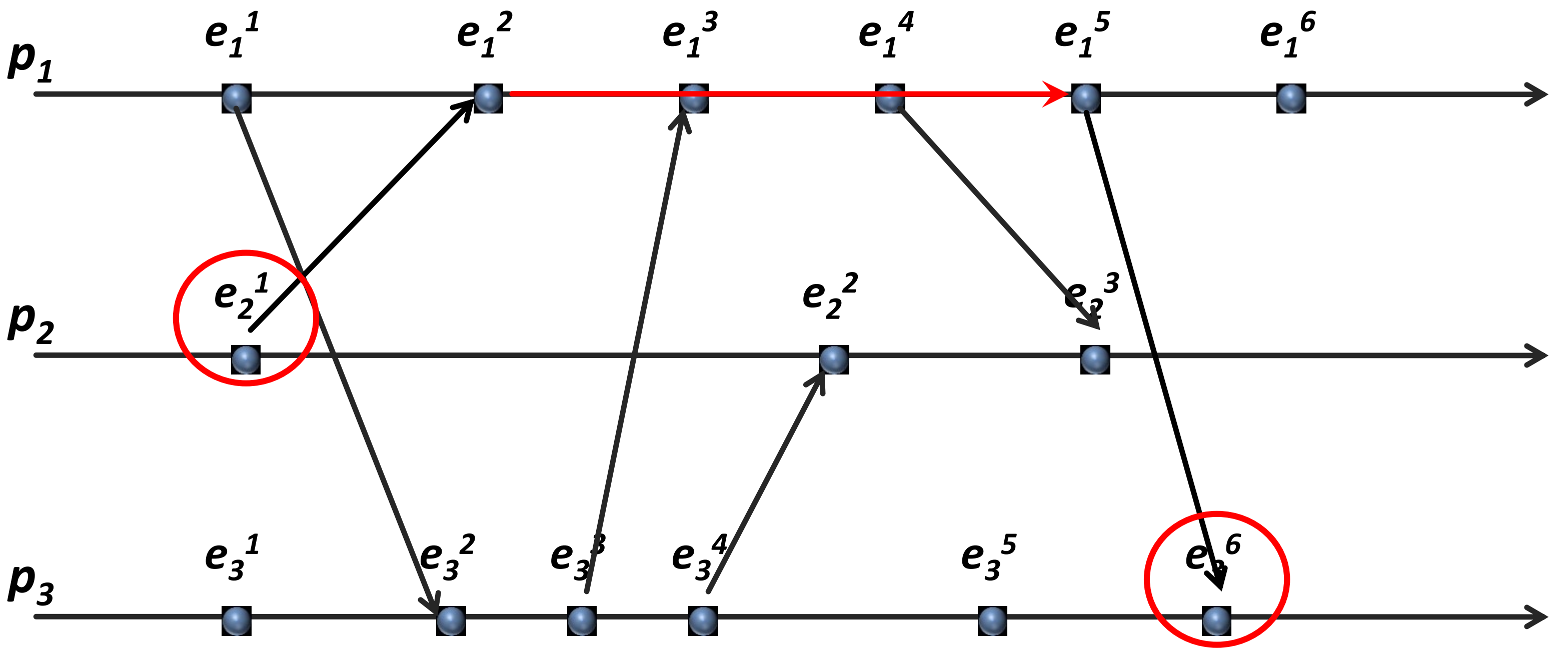
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

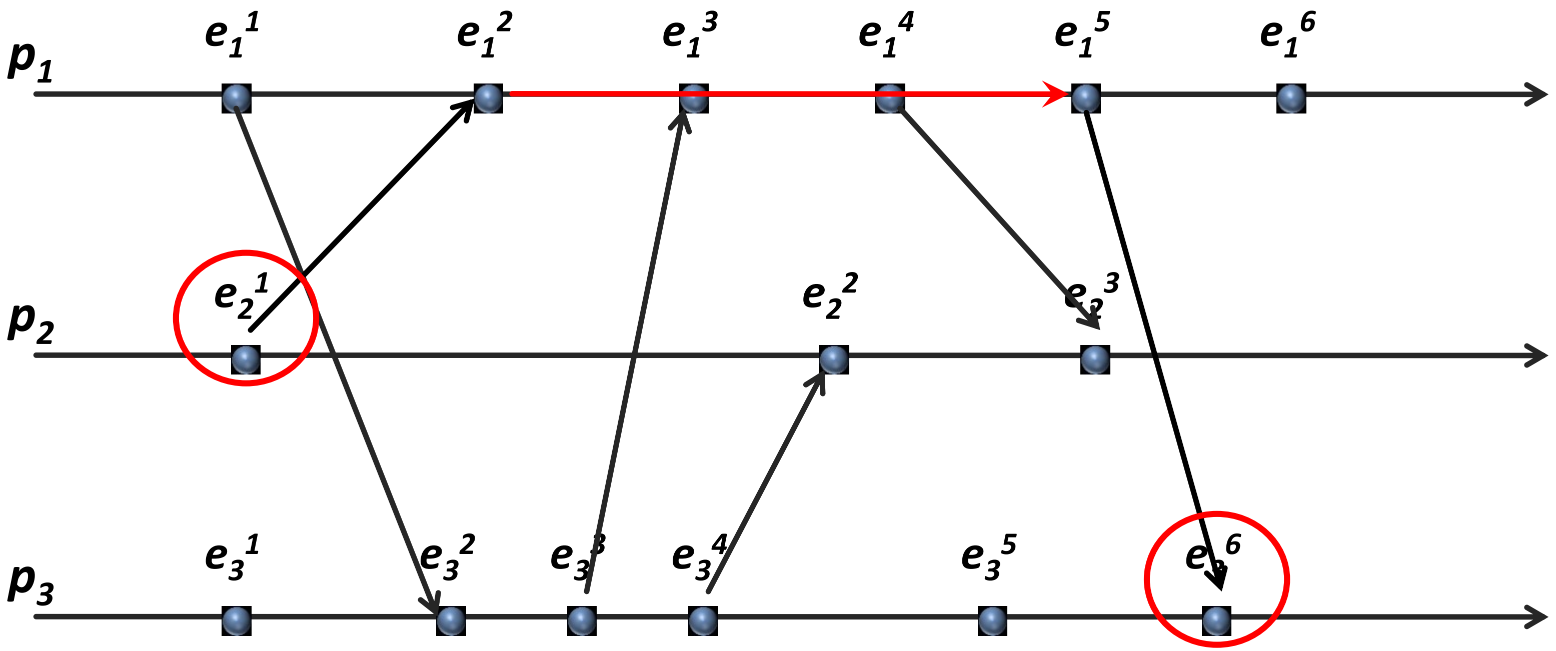
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

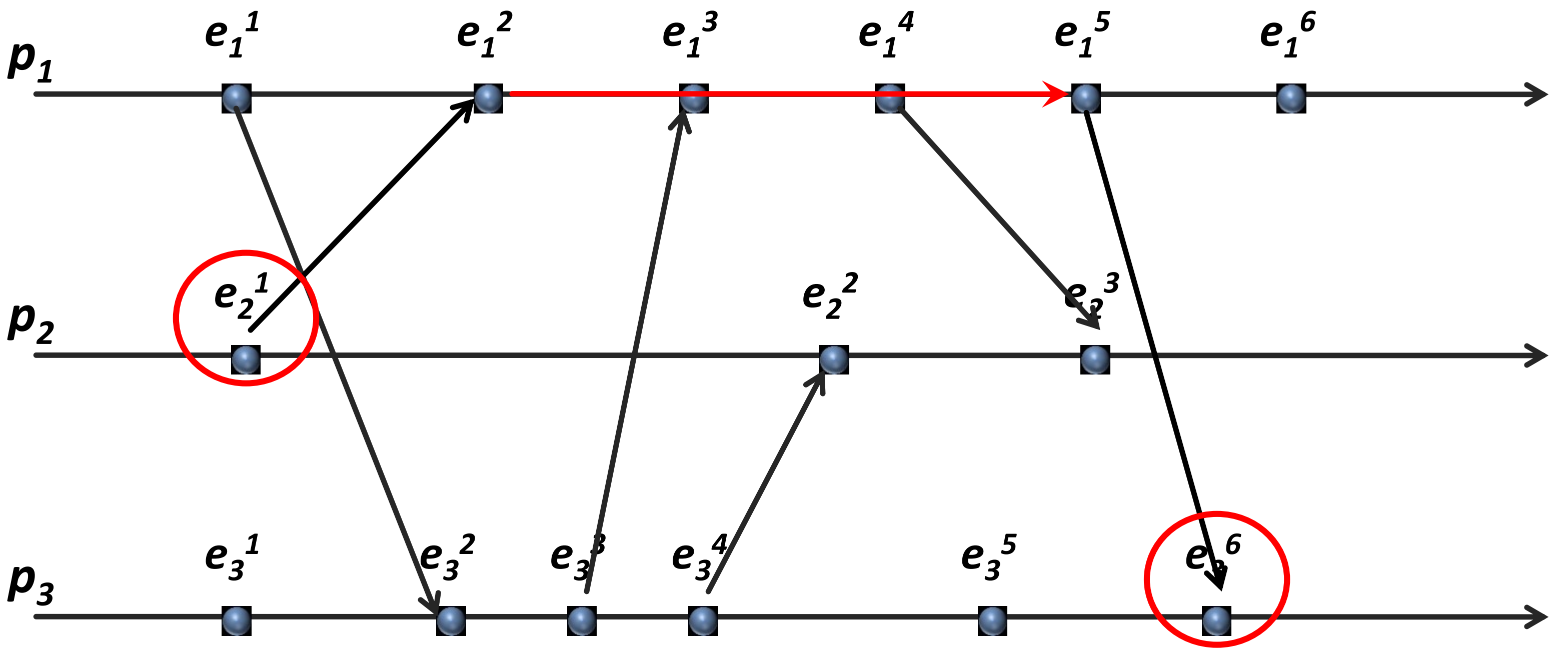
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

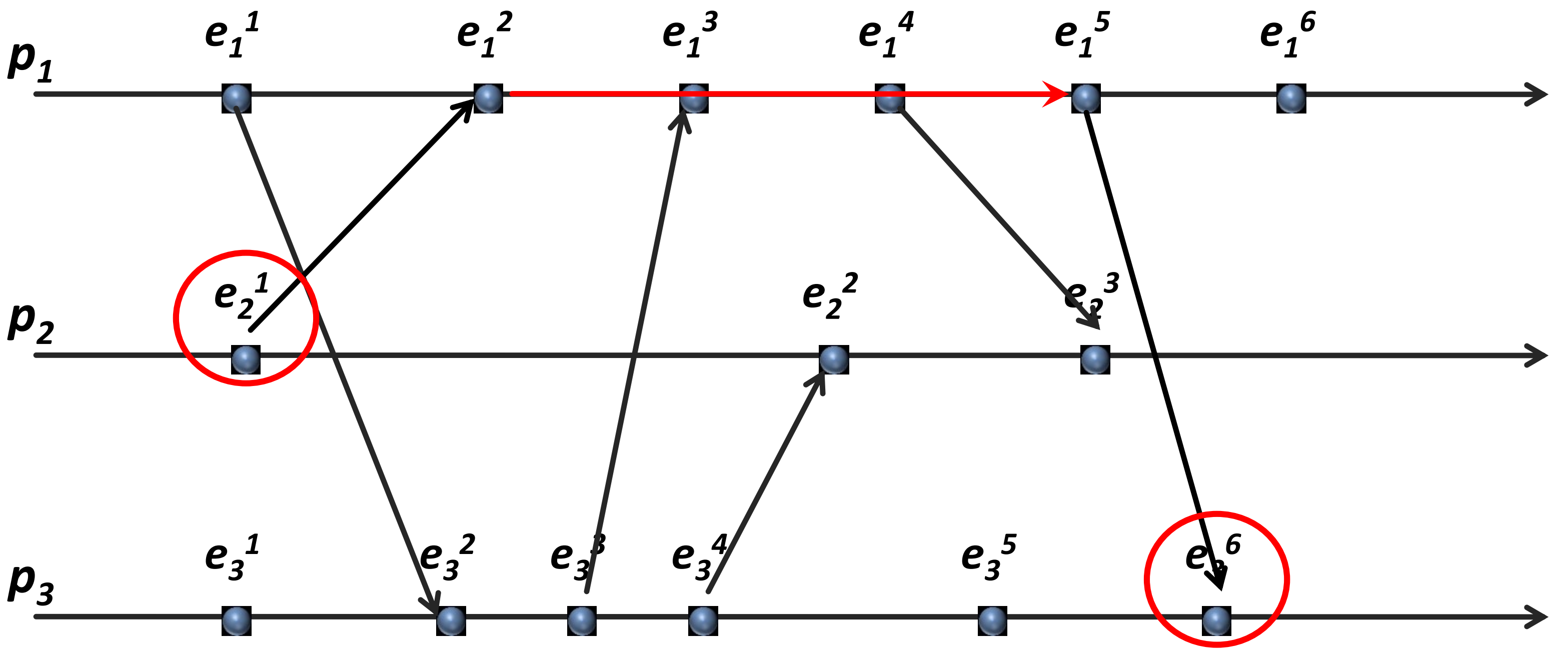
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

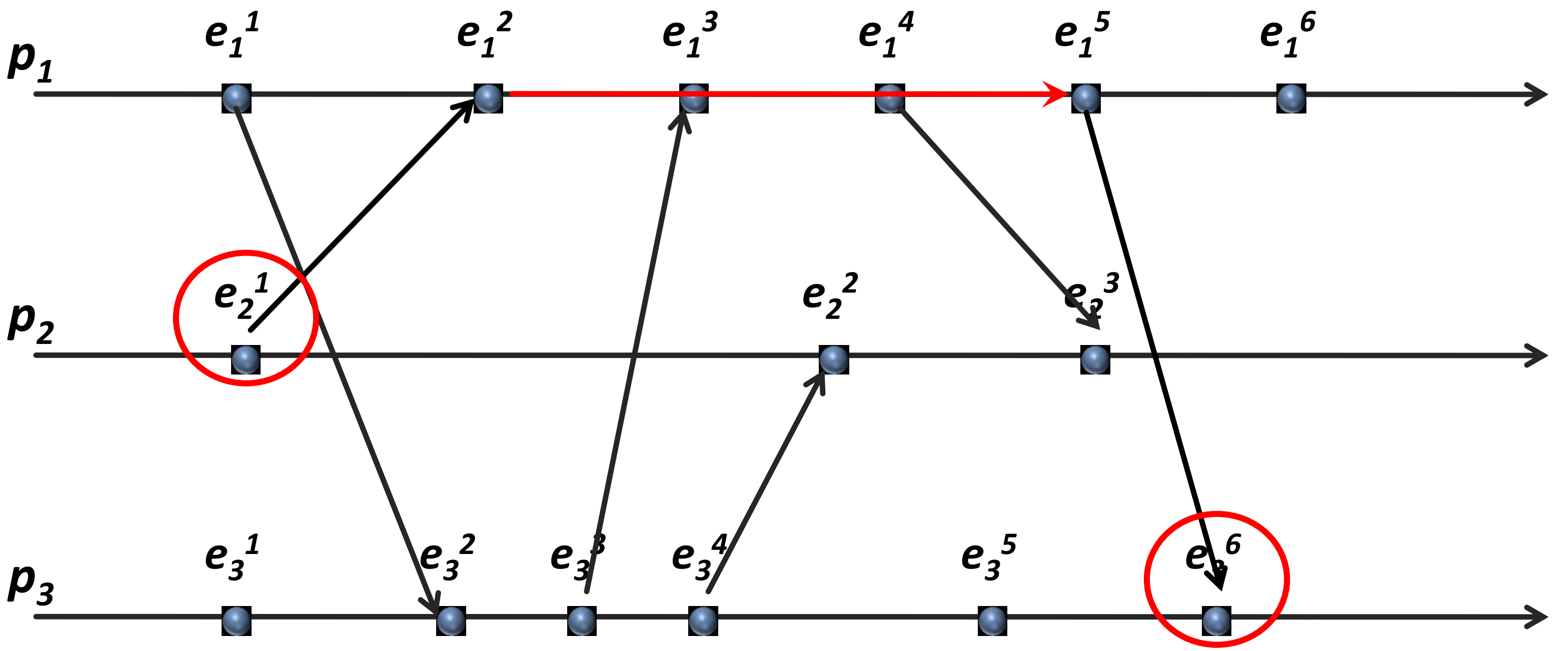
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

local events precede one another → precede one another globally:

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

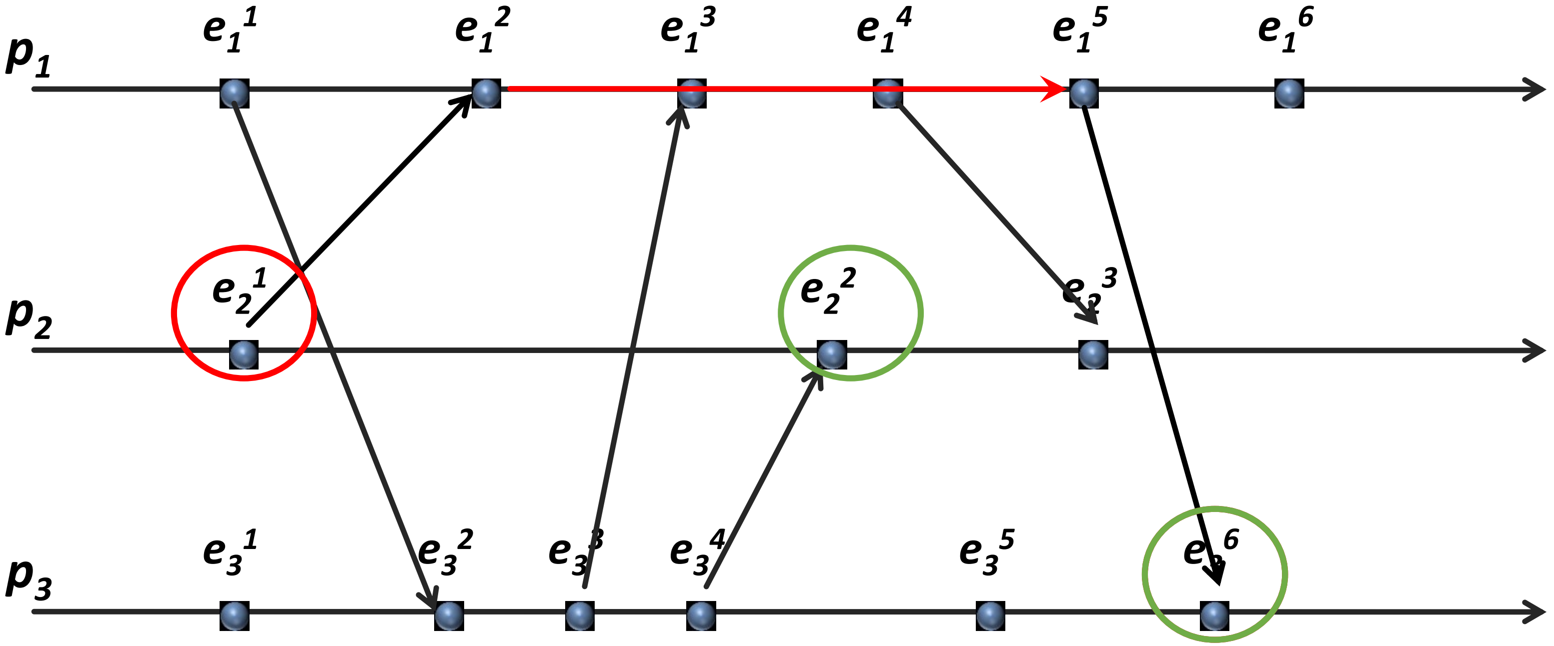
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

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* $e_i^k \rightarrow e_i^m$

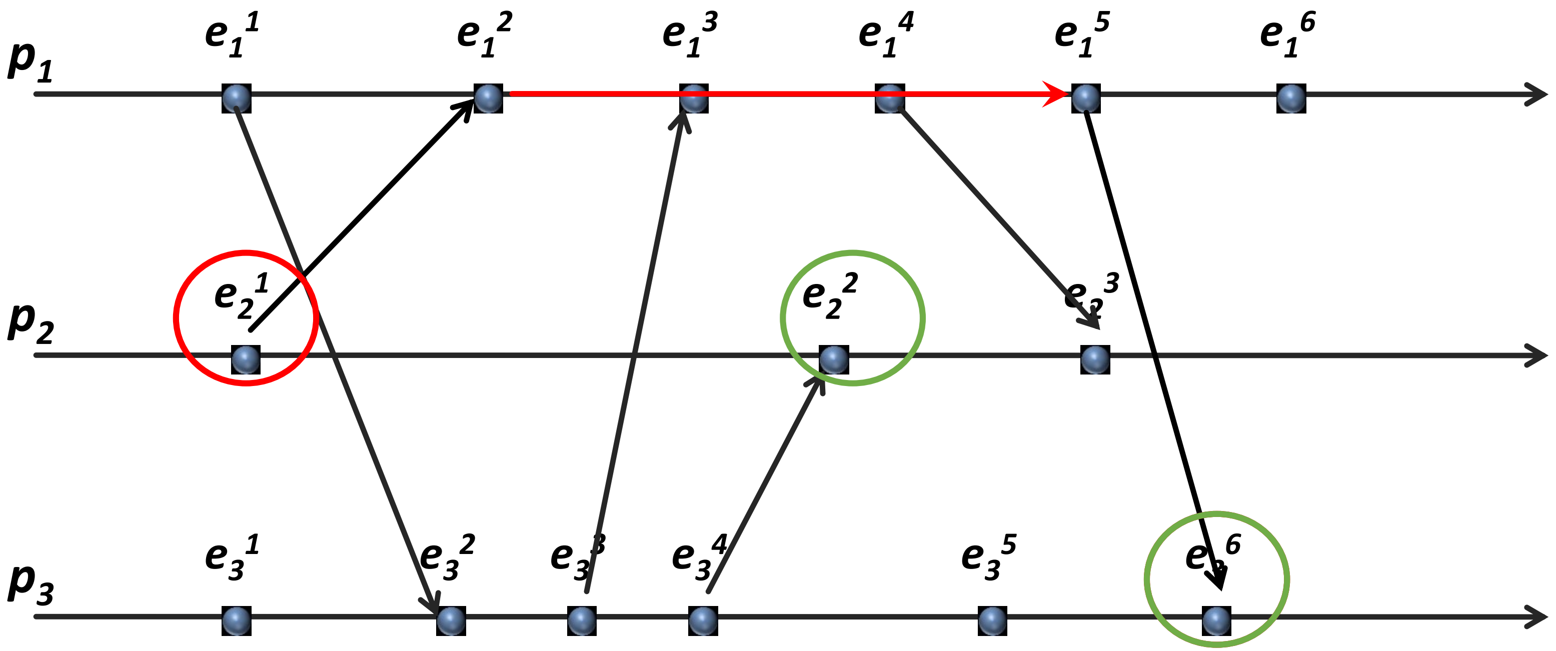
Sending a message always precedes receipt of that message:

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

Event ordering is transitive:

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

Space-time Diagram



$e_2^1 \rightarrow e_3^6$

$e_2^2 \,||\, e_3^6$

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 transitive:

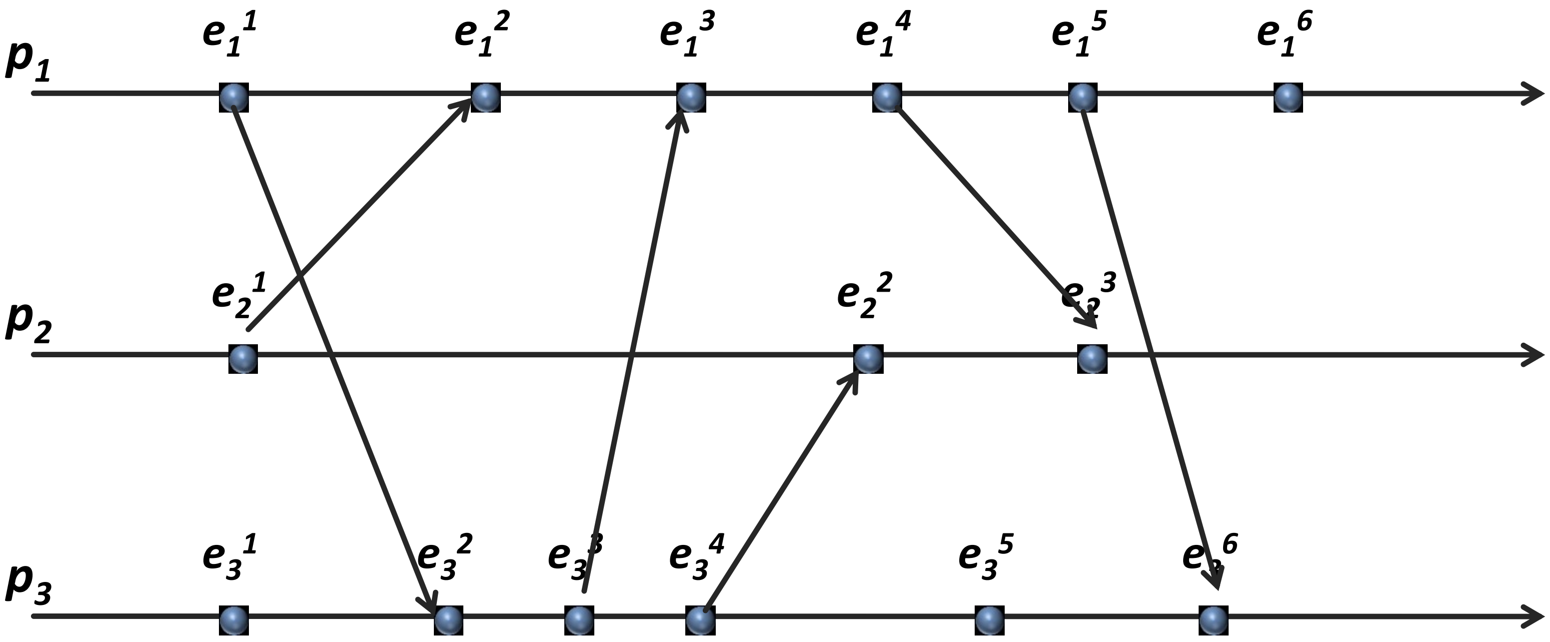
If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Cuts of an Asynchronous Computation

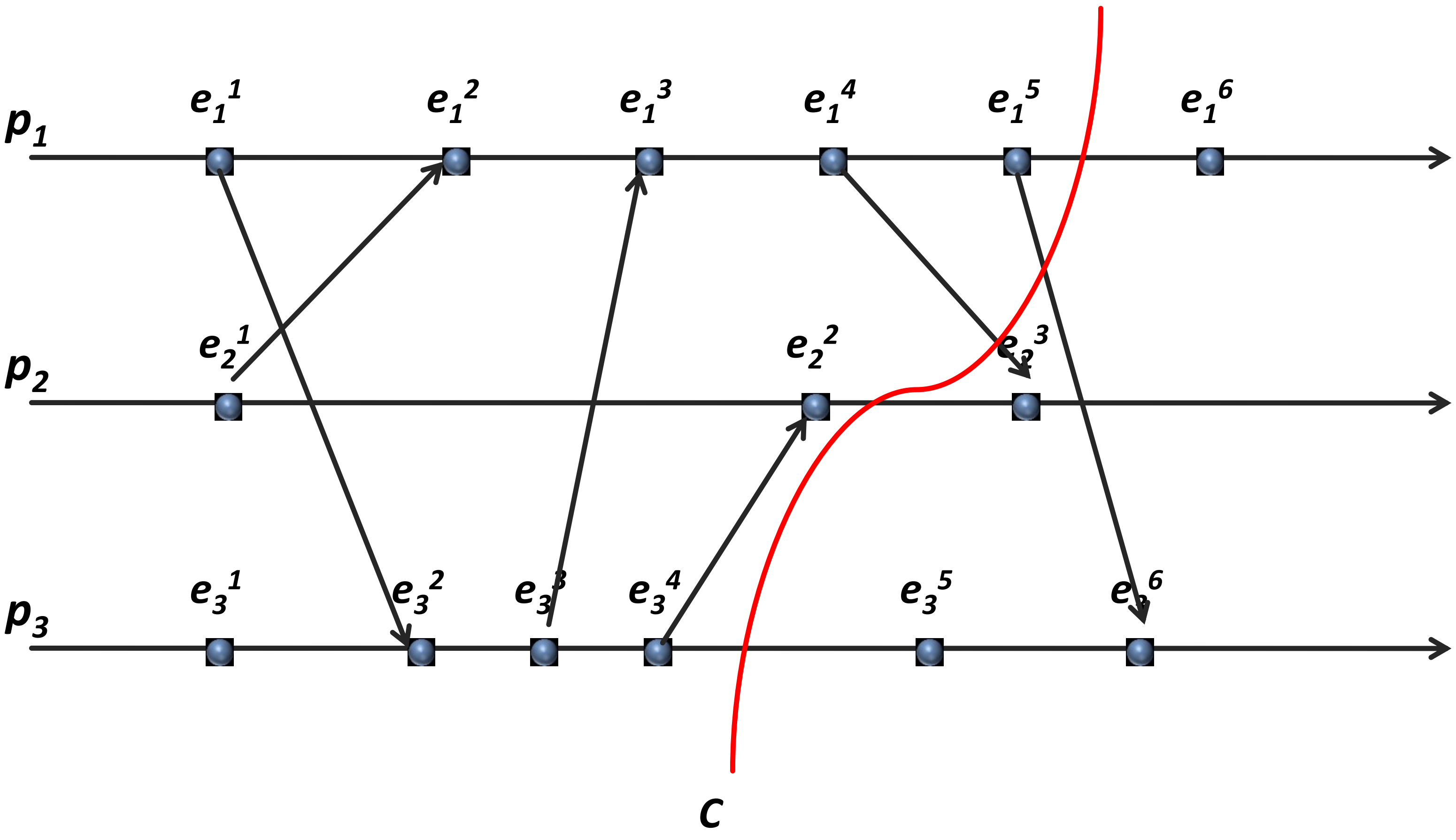
- Suppose there is an ***external monitor*** process
- External monitor constructs a global state:
  - Asks processes to send it local history
- Global state constructed from these local histories is:

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

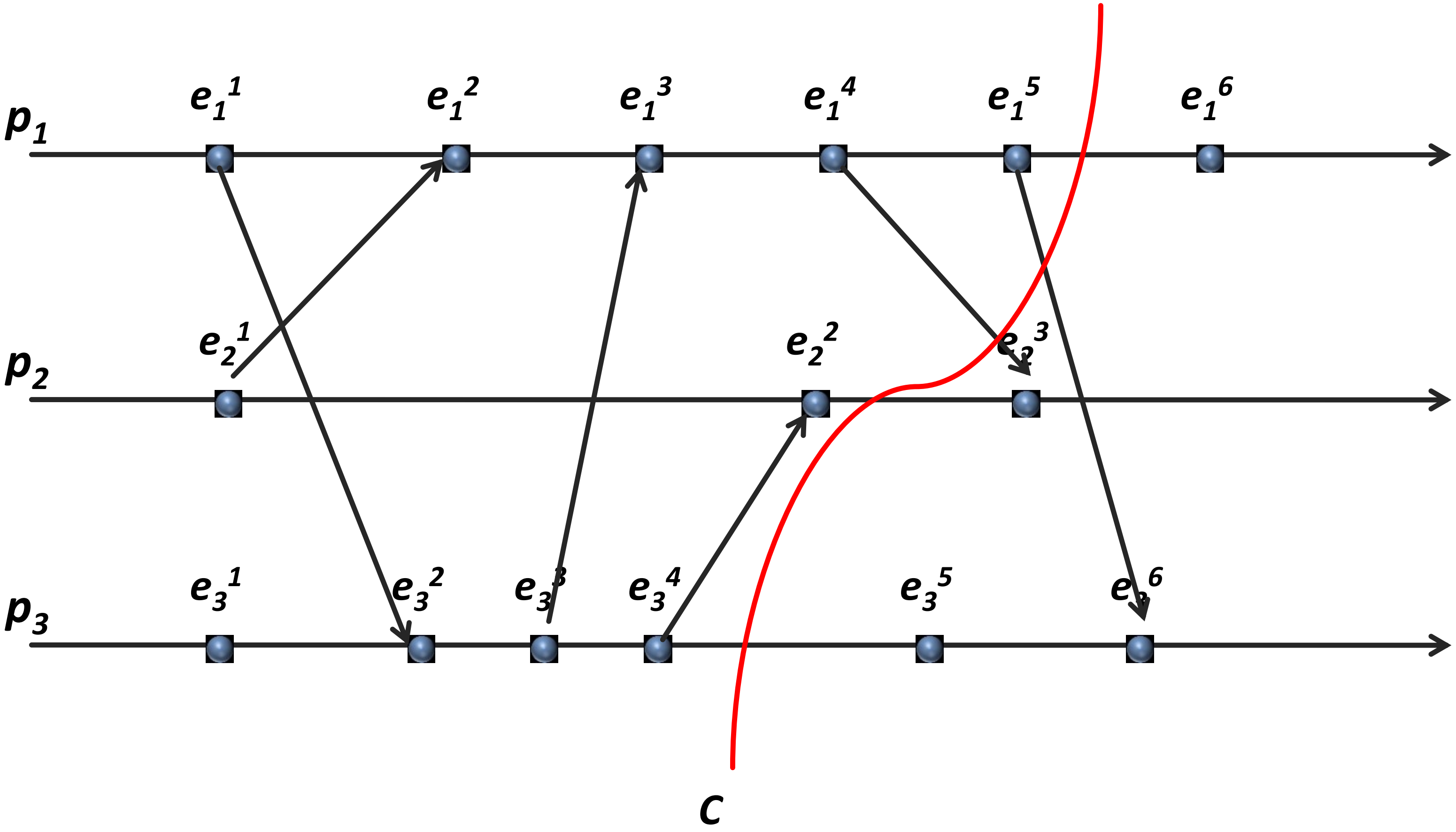
Example Cuts



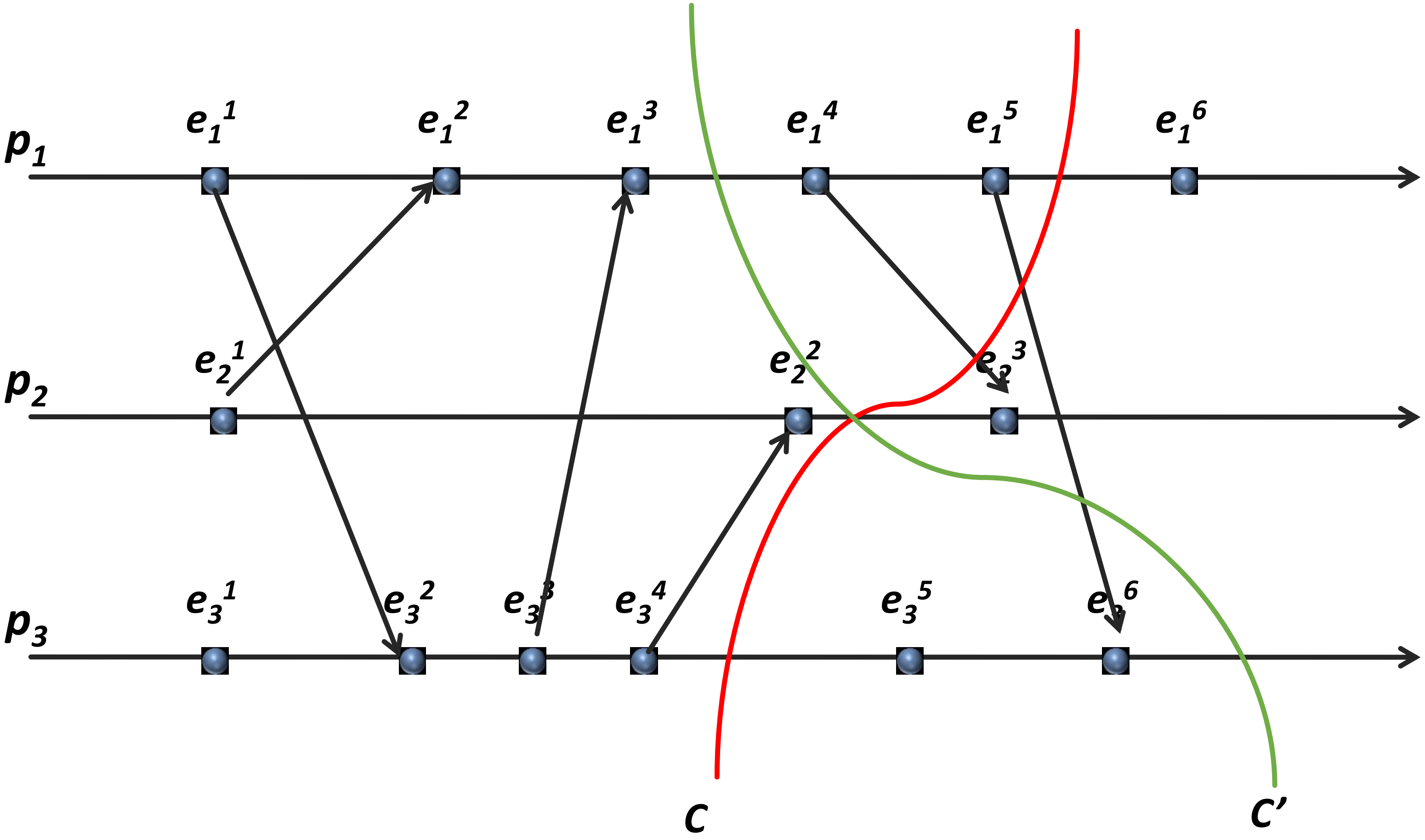
Example Cuts



Example Cuts



Example Cuts

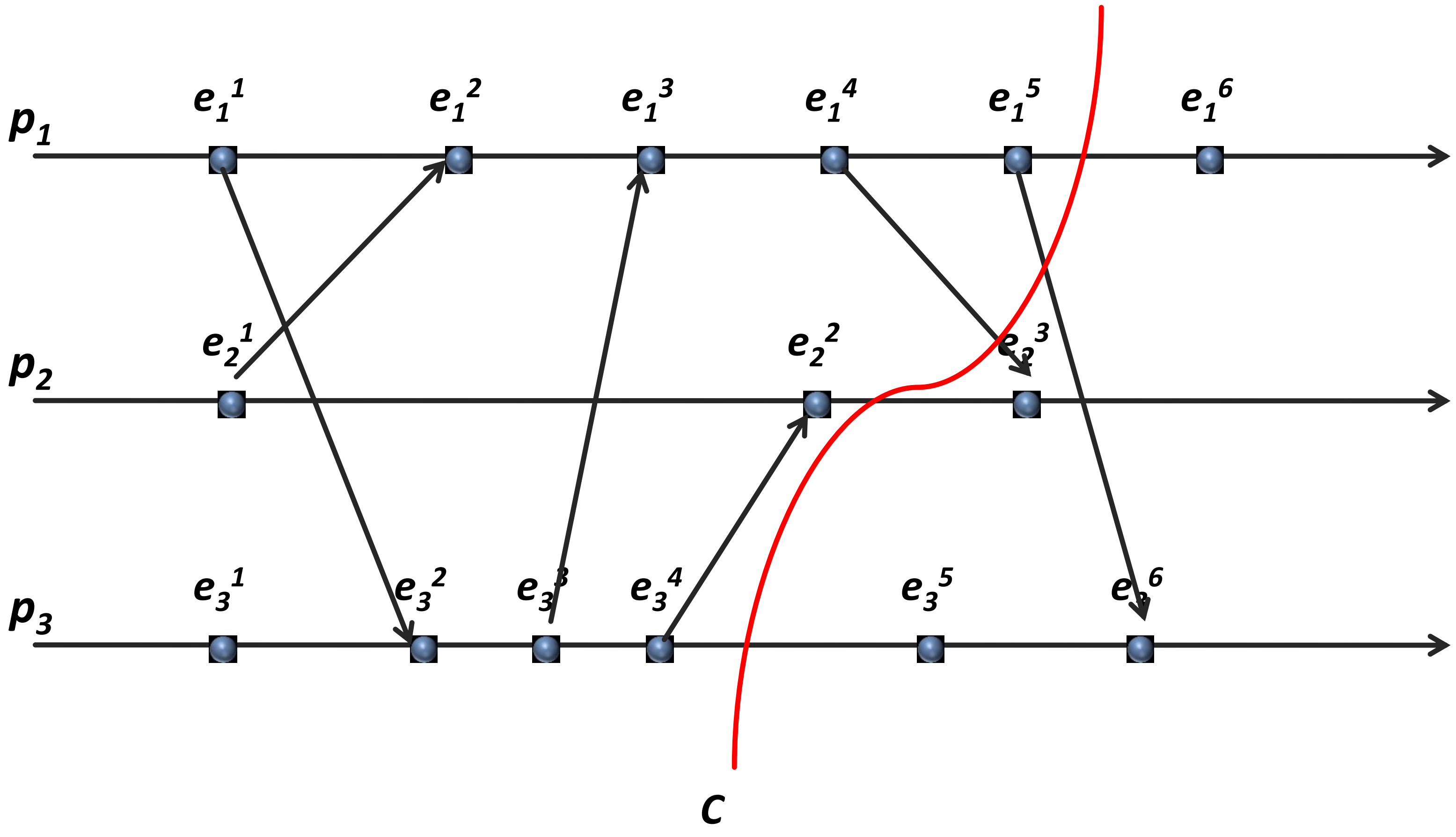


Consistent vs. Inconsistent Cuts

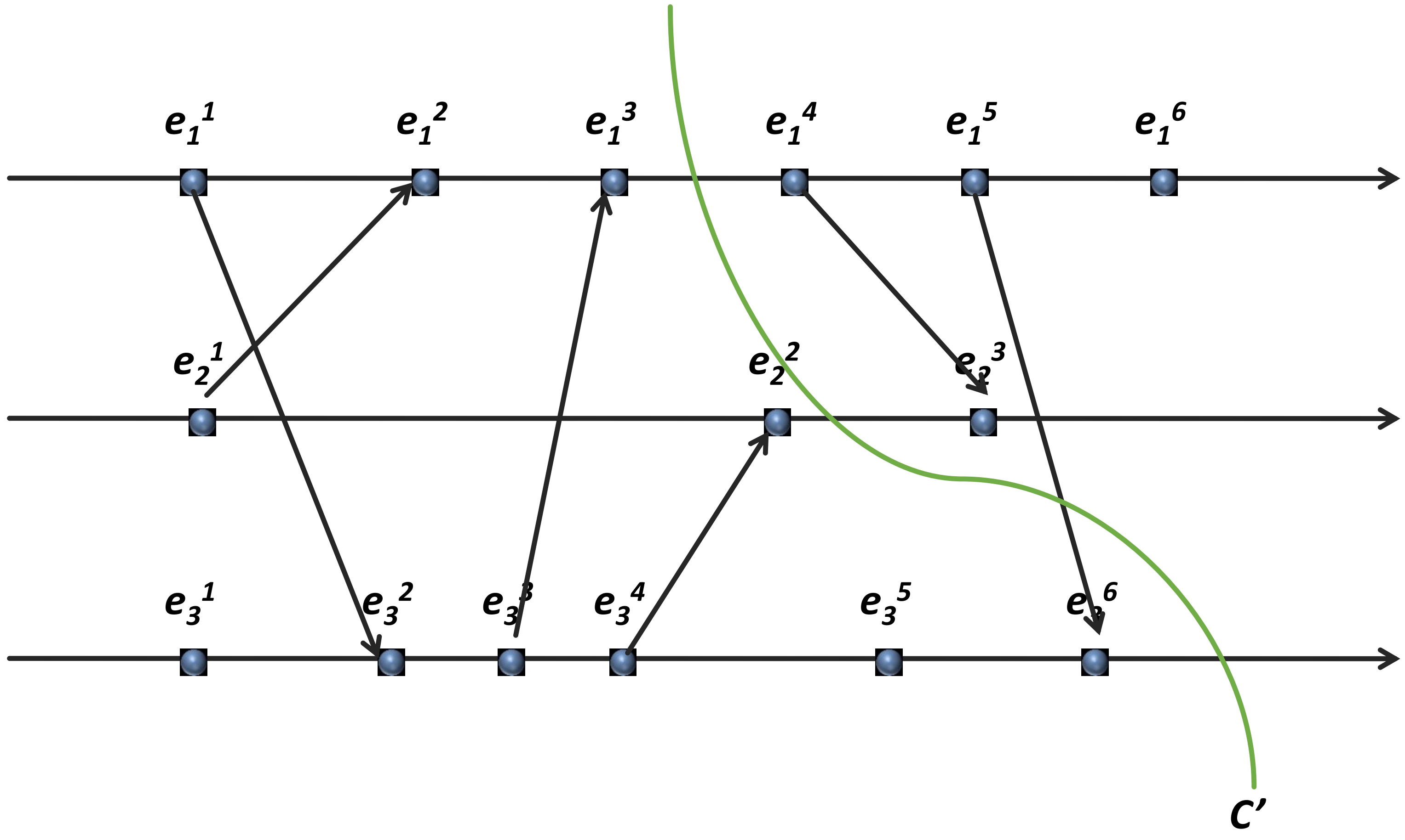
- A cut is consistent if
  - for any event *e* included in the cut
  - any *e'* that causally precedes *e* is also in the cut
- For cut *C:*

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

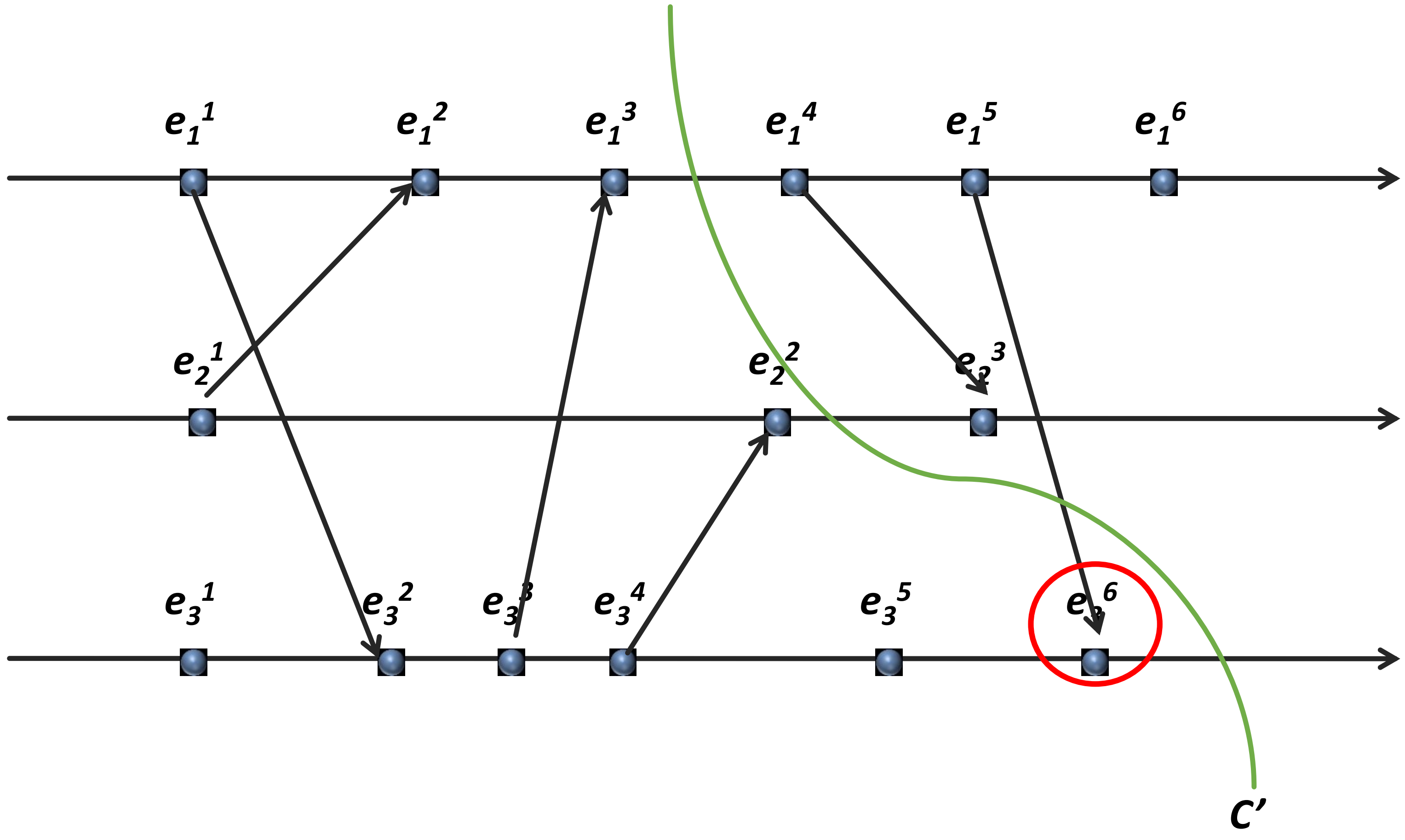
Are These Cuts Consistent?



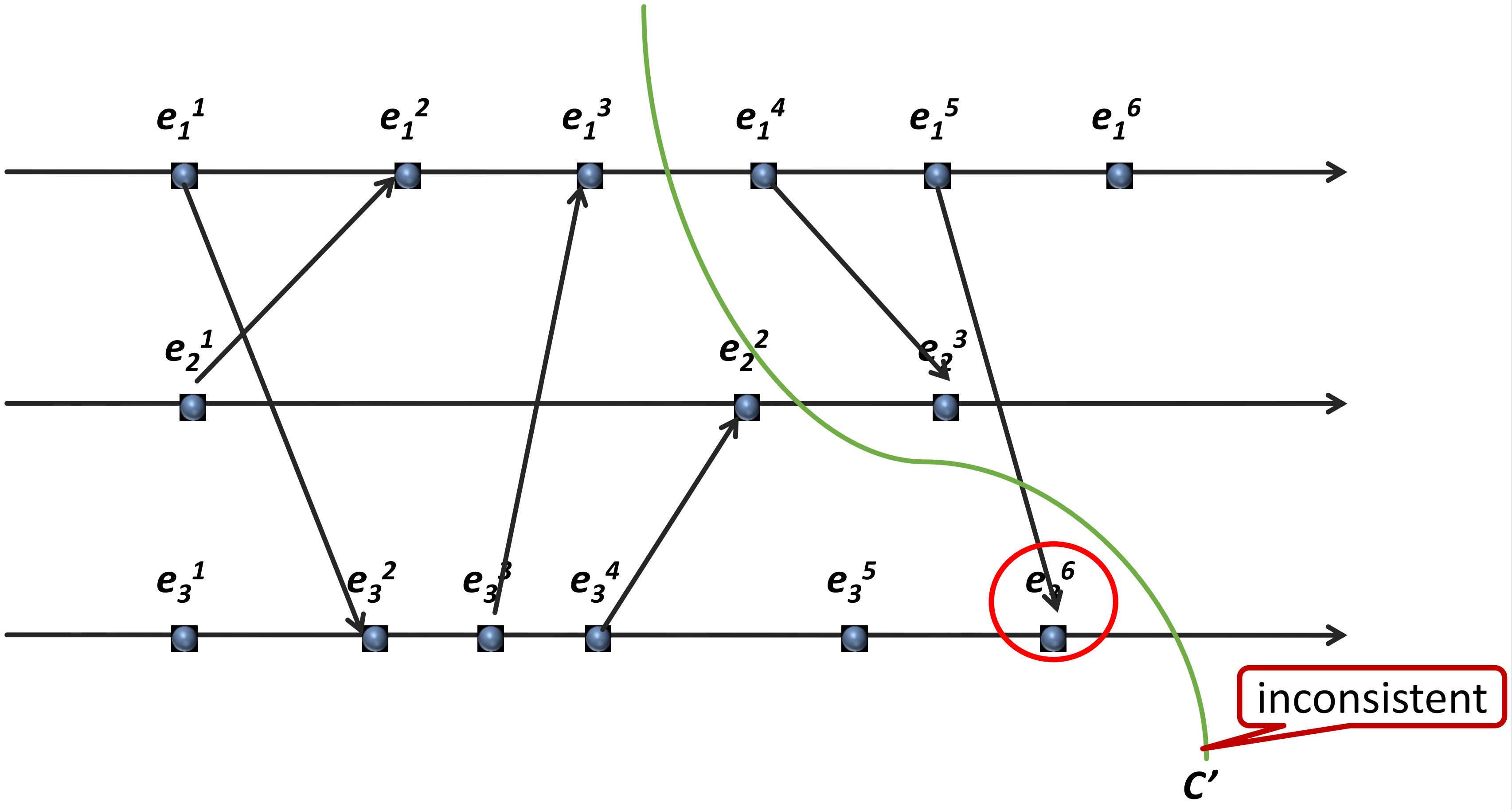
Are These Cuts Consistent?



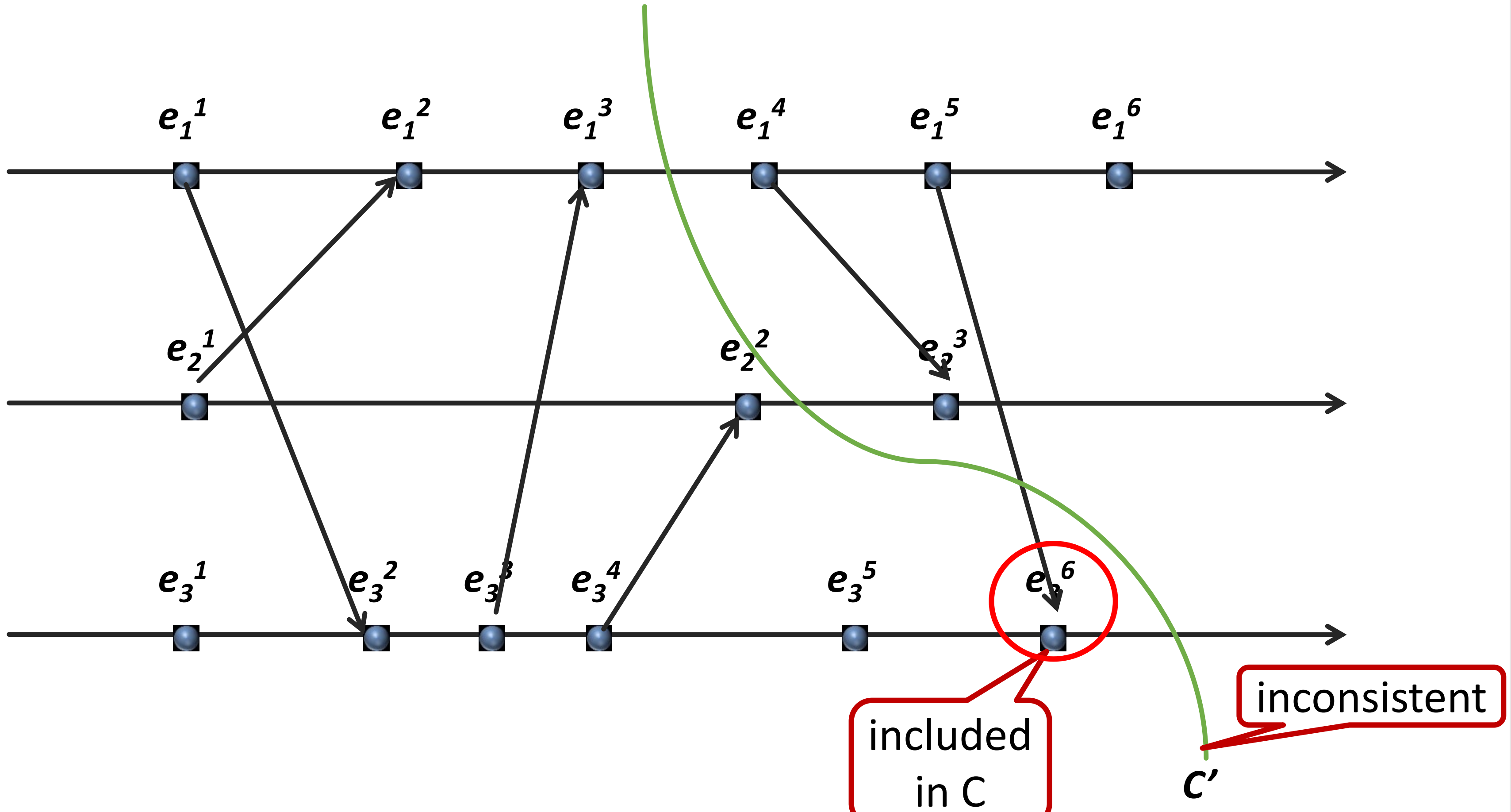
Are These Cuts Consistent?



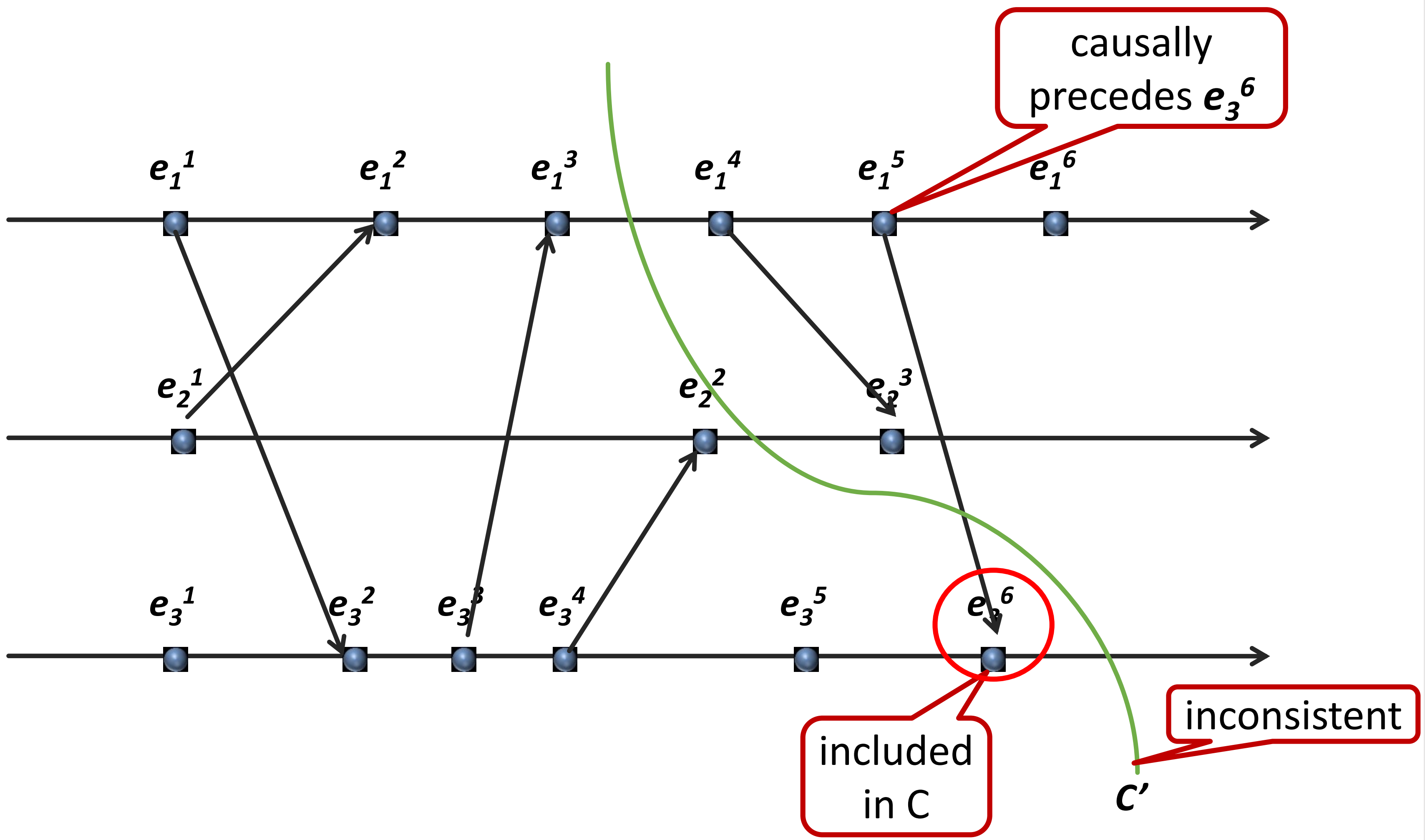
Are These Cuts Consistent?



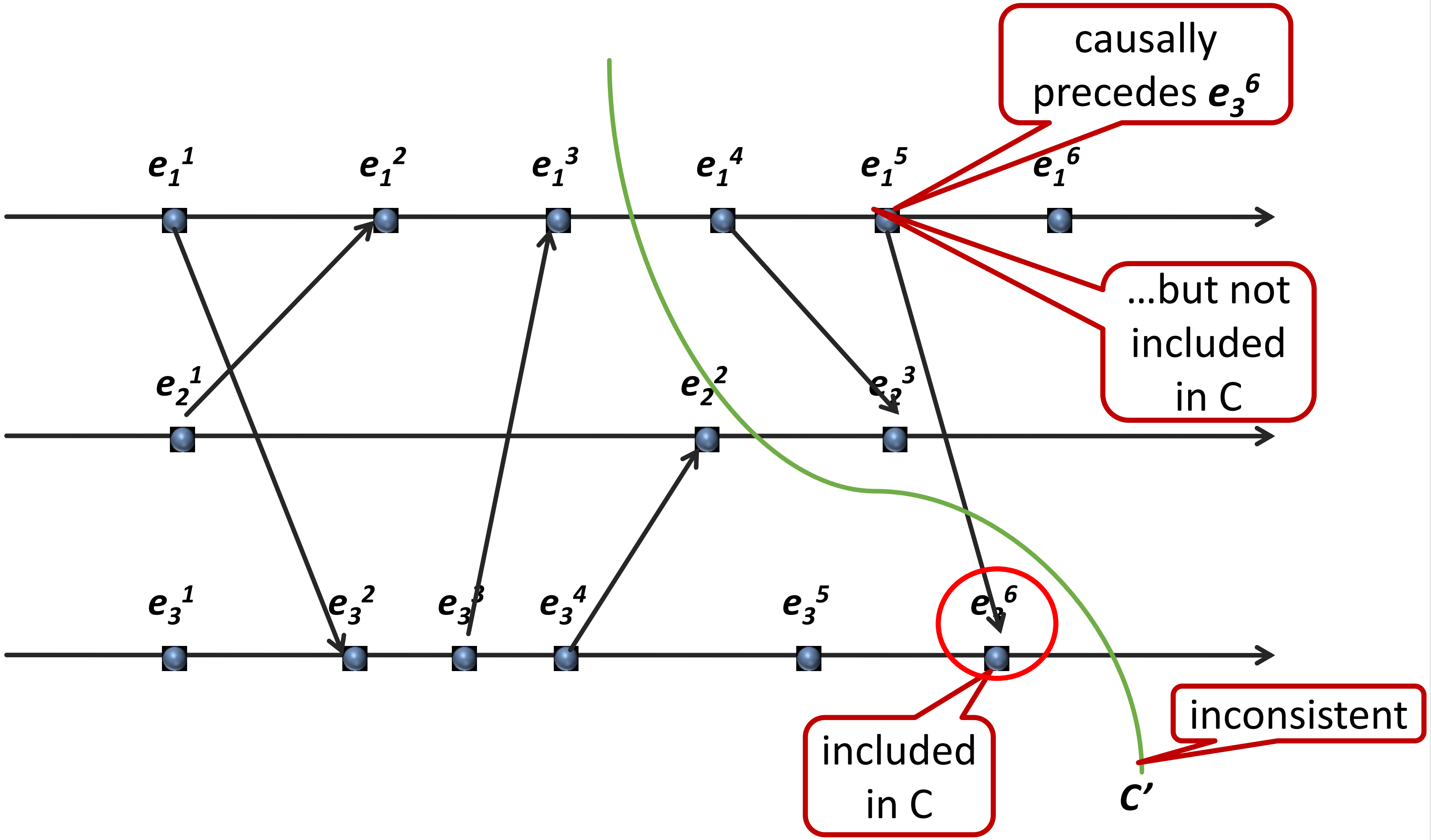
Are These Cuts Consistent?



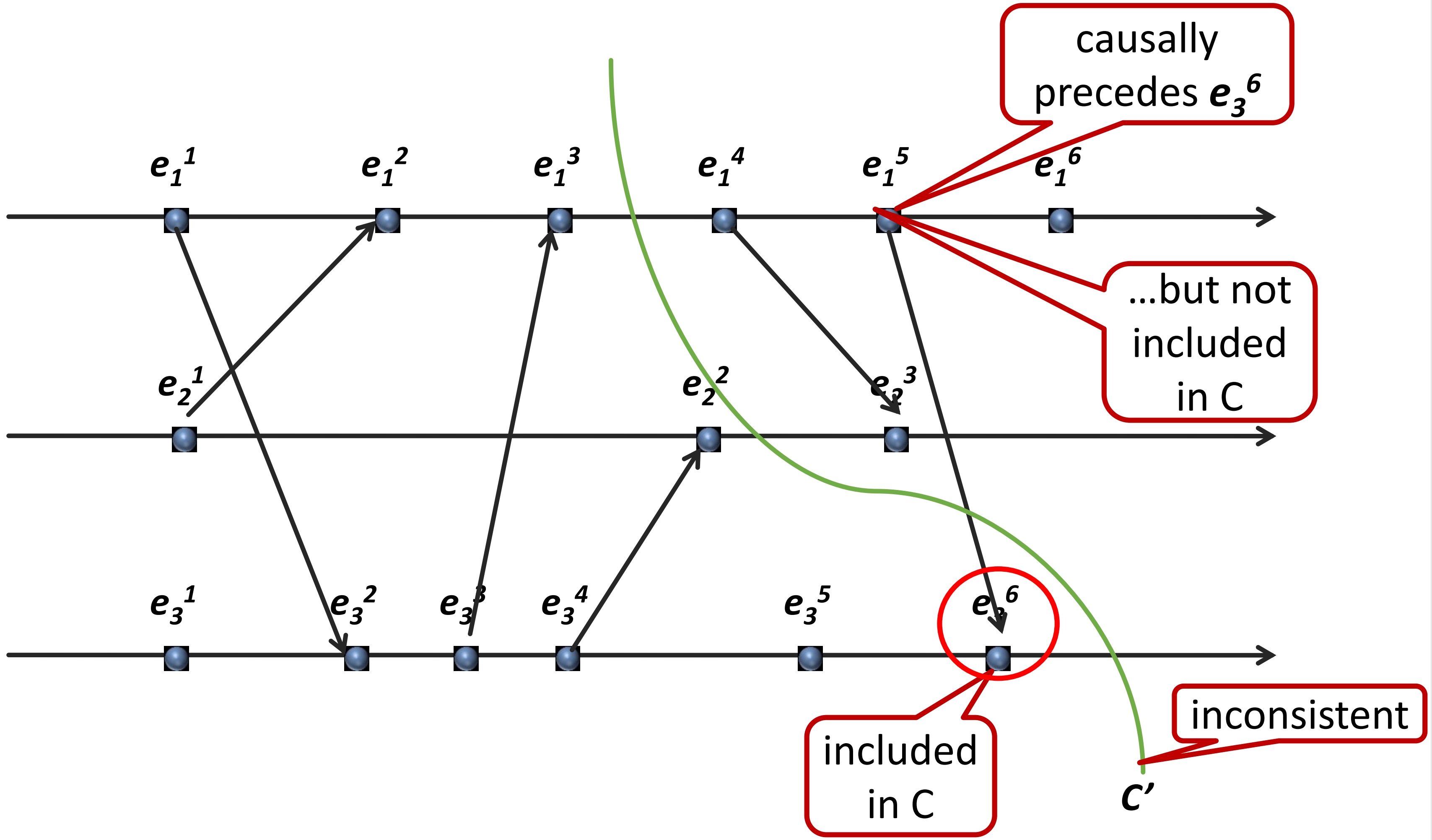
Are These Cuts Consistent?



Are These Cuts Consistent?

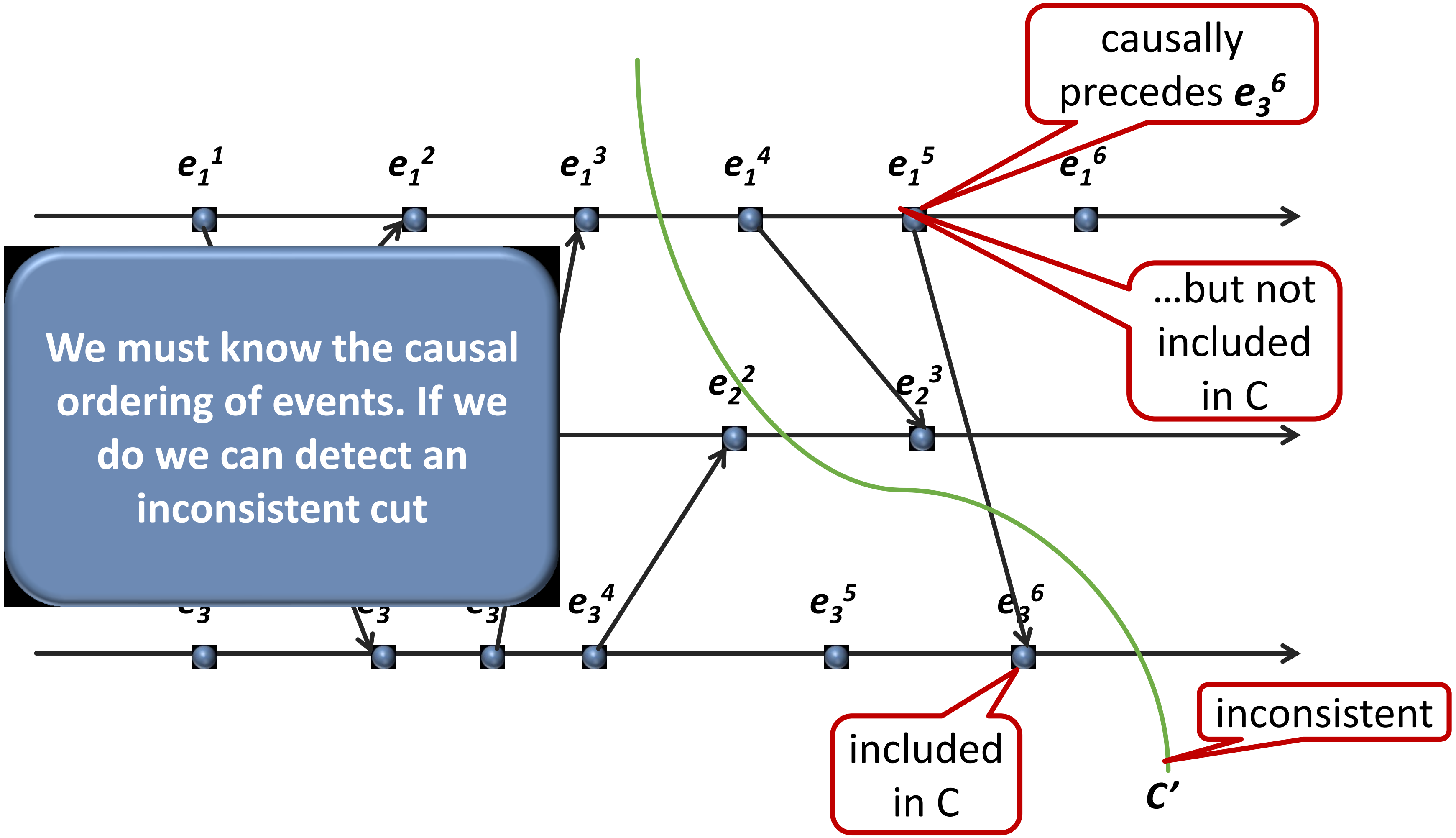


Are These Cuts Consistent?



A consistent cut corresponds to a consistent global state

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



Logical Clocks

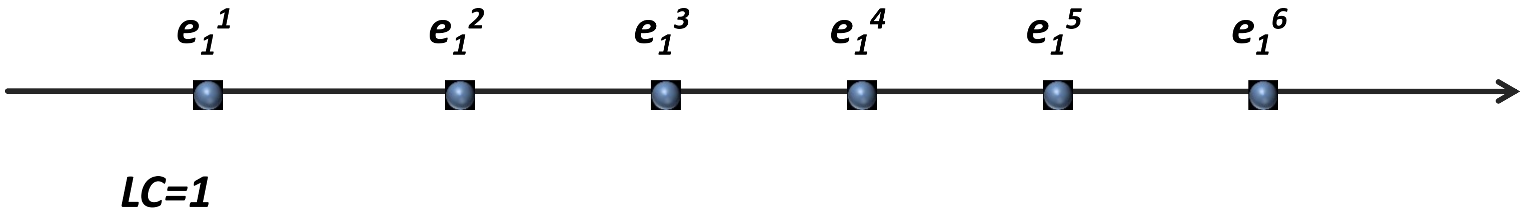
- Each process maintains a local value of a logical clock ***LC***
- LC for process *p* counts **how many events causally preceded the current event at *p*** (including the current event).
- $LC(e_i)$ – the logical clock value at process p_i at event e_i
- Suppose we had only a single process:

Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- LC for process *p* counts **how many events 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 only a single process:

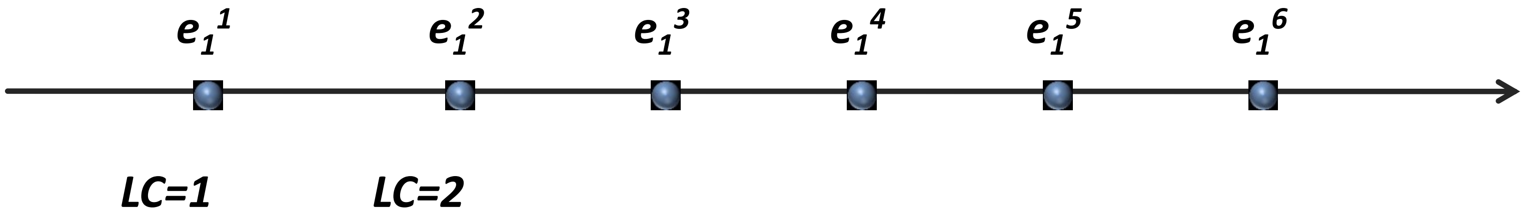
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- LC for process *p* counts **how many events 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 only a single process:



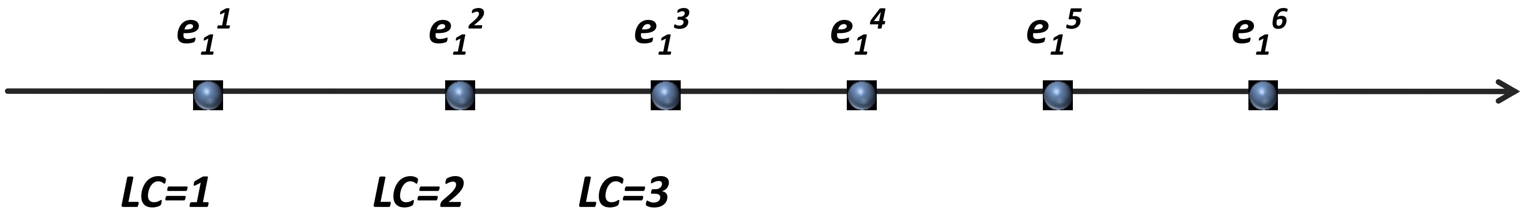
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- LC for process *p* counts **how many events 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 only a single process:



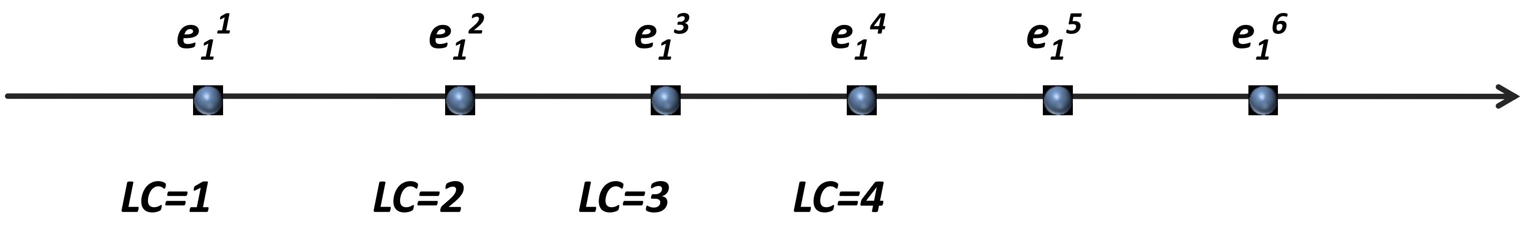
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- LC for process *p* counts **how many events 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 only a single process:



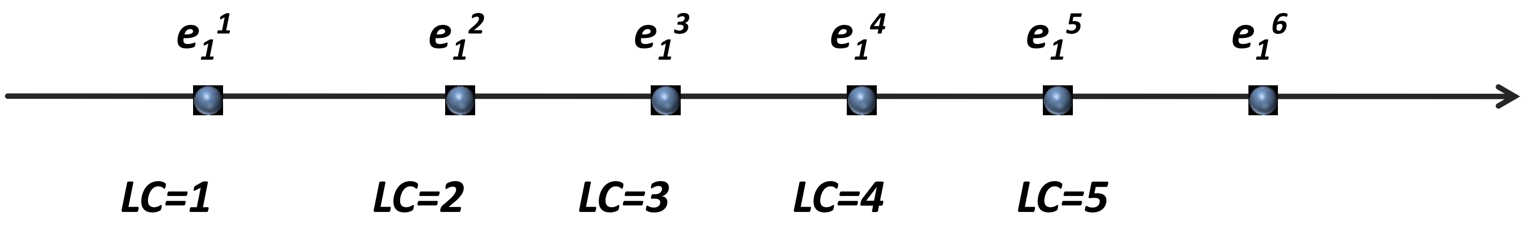
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- LC for process *p* counts **how many events 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 only a single process:



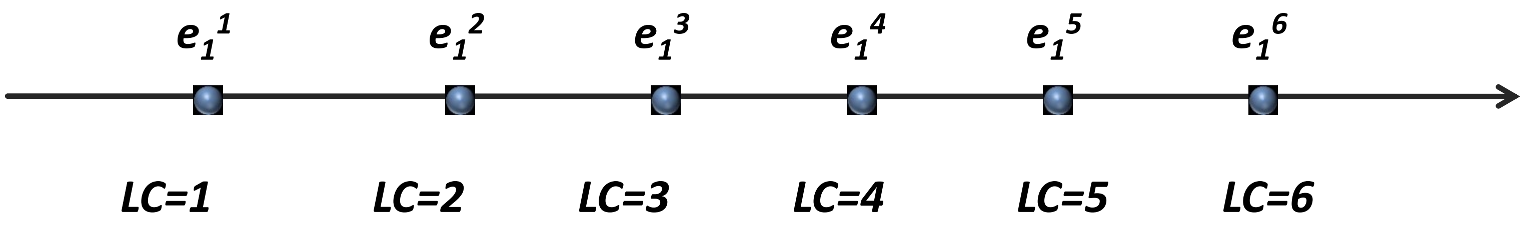
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- LC for process *p* counts **how many events 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 only a single process:



Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- LC for process *p* counts **how many events 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 only a single process:



Logical Clocks (cont.)

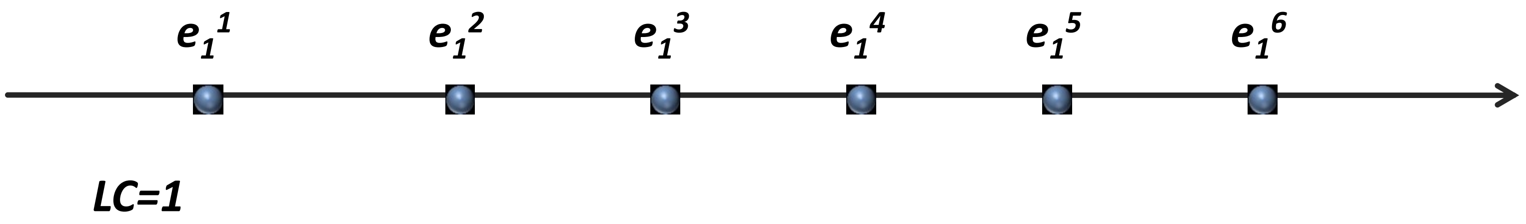
With >1 process (thread) logical clocks updated:

- Each message m sent contains a timestamp TS(m)
- TS(m) is the logical clock value associated with sending event at the sending process

Logical Clocks (cont.)

With >1 process (thread) logical clocks updated:

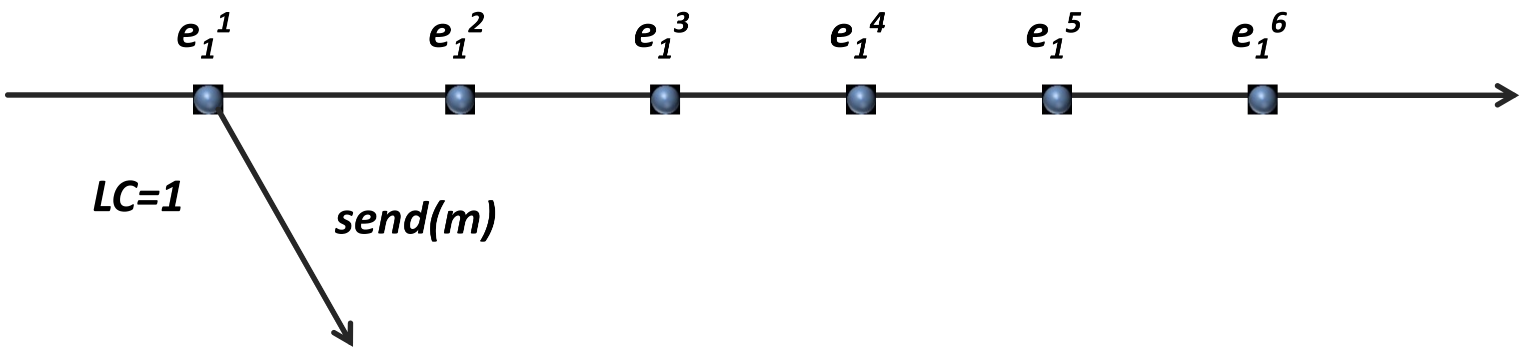
- Each message m sent contains a timestamp TS(m)
- TS(m) is the logical clock value associated with sending event at the sending process



Logical Clocks (cont.)

With >1 process (thread) logical clocks updated:

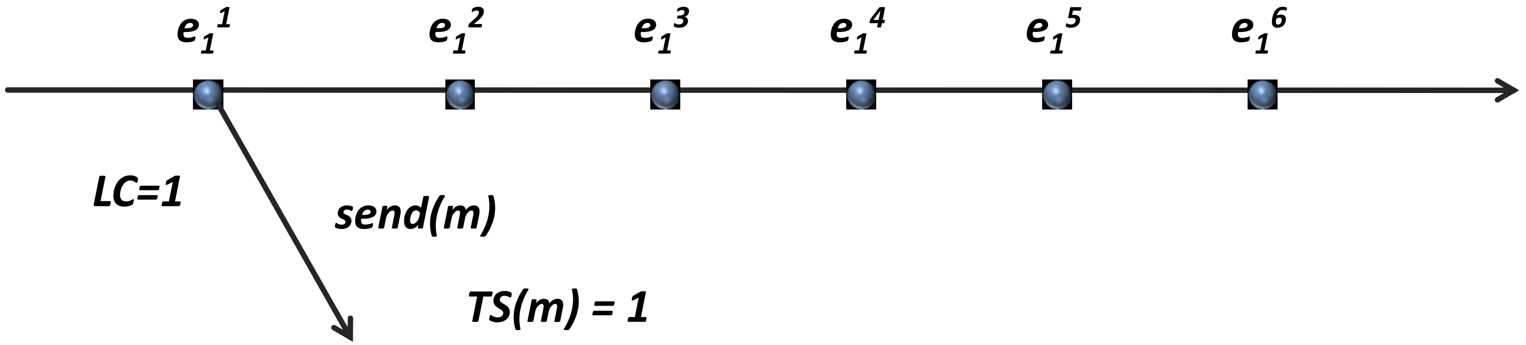
- Each message m sent contains a timestamp TS(m)
- TS(m) is the logical clock value associated with sending event at the sending process



Logical Clocks (cont.)

With >1 process (thread) logical clocks updated:

- Each message m sent contains a timestamp TS(m)
- TS(m) is the logical clock value associated with sending event at the sending process



Logical Clocks (cont)

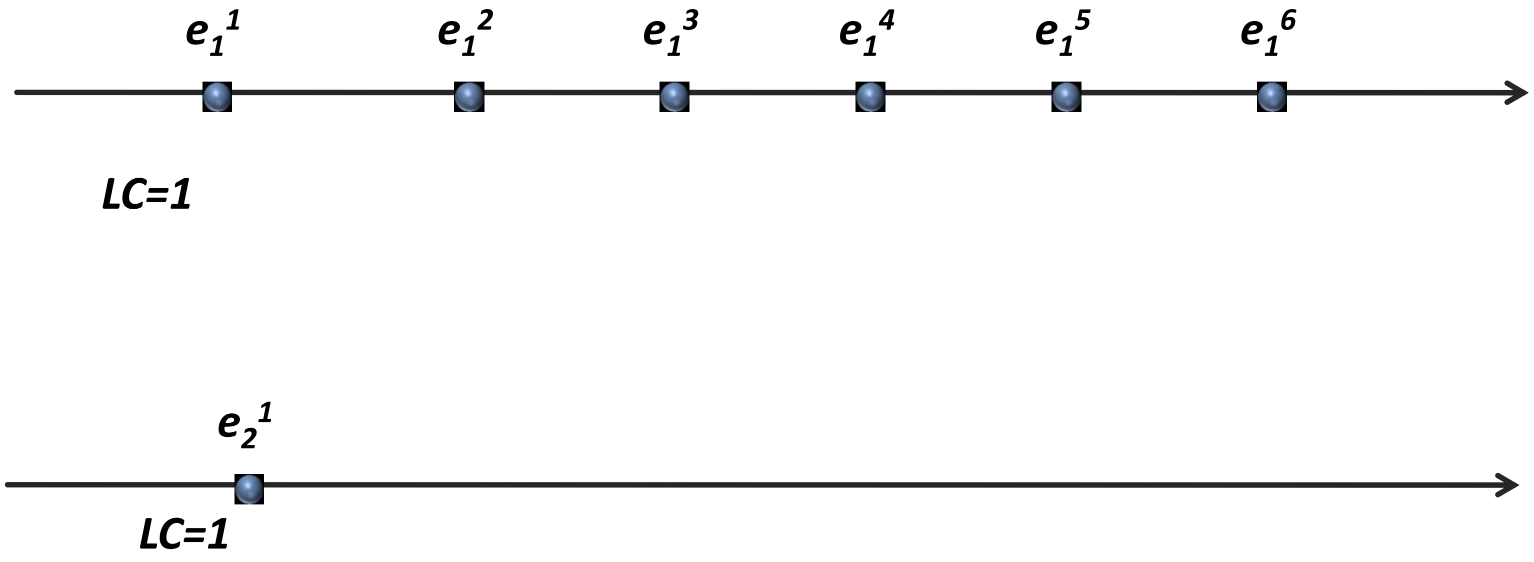
- When process receives m, update logical clock to:

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

Logical Clocks (cont)

- When process receives m, update logical clock to:

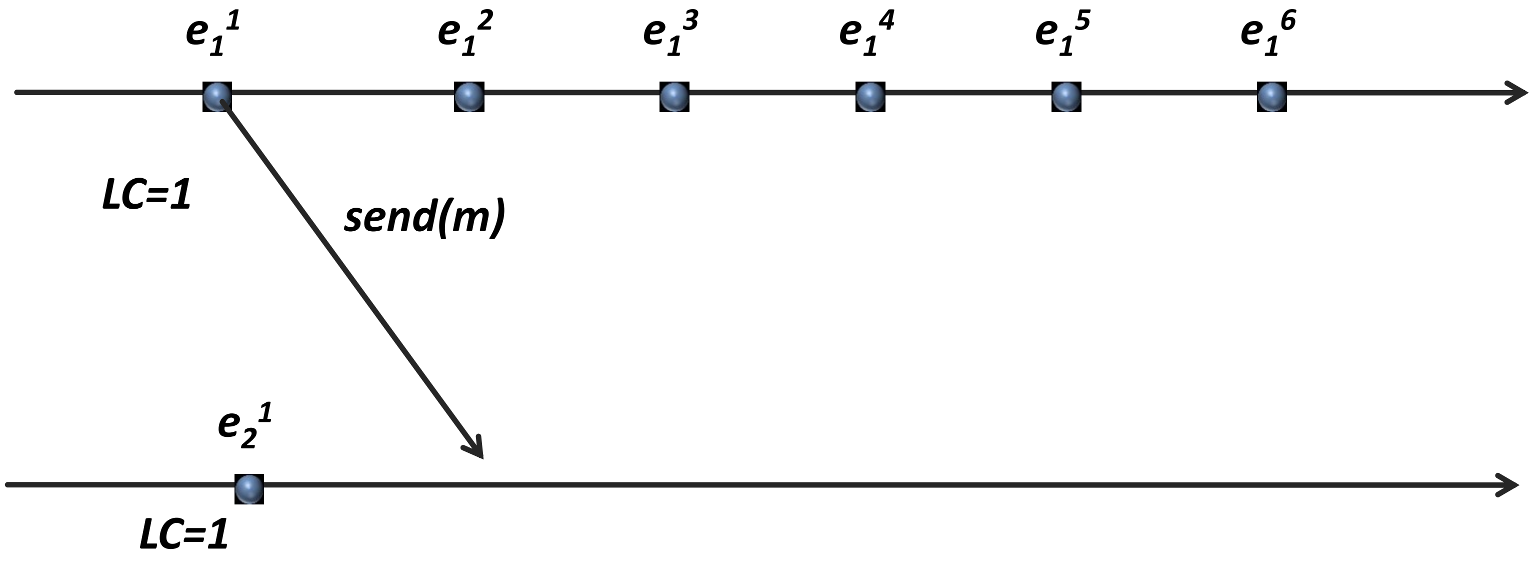
$$max\{LC, TS(m)\} + 1$$



Logical Clocks (cont)

- When process receives m, update logical clock to:

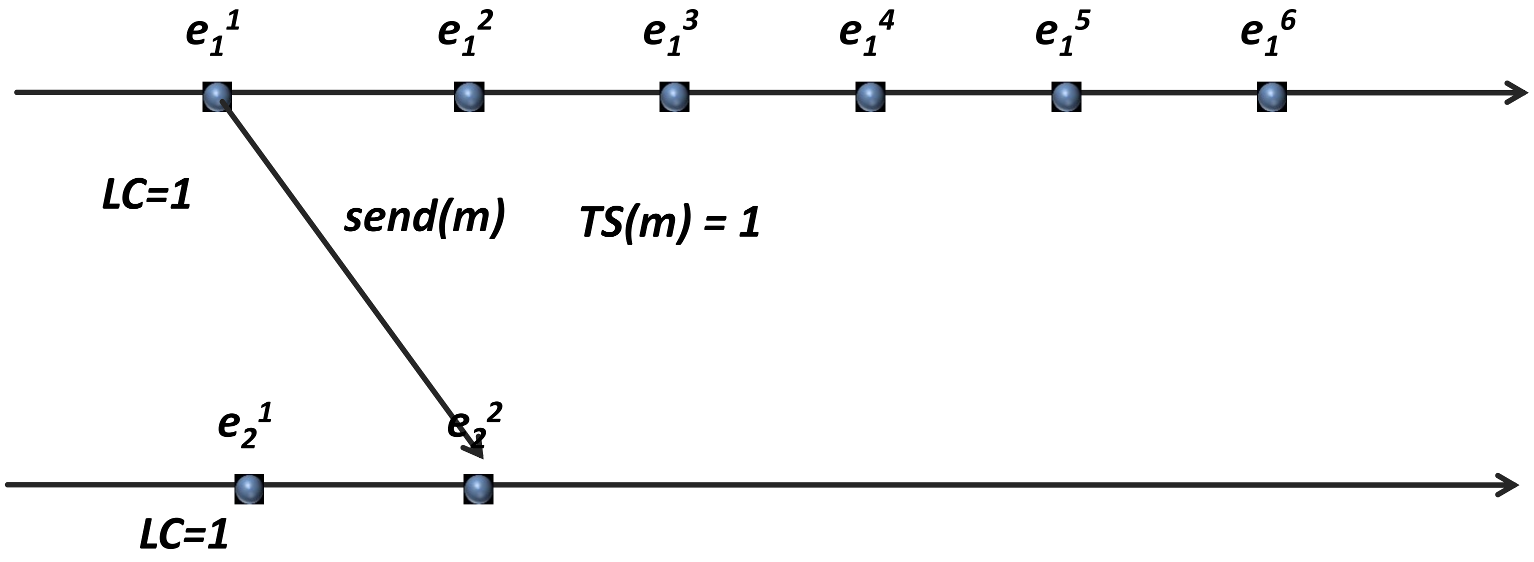
  $$max\{LC, TS(m)\} + 1$$



Logical Clocks (cont)

- When process receives m, update logical clock to:

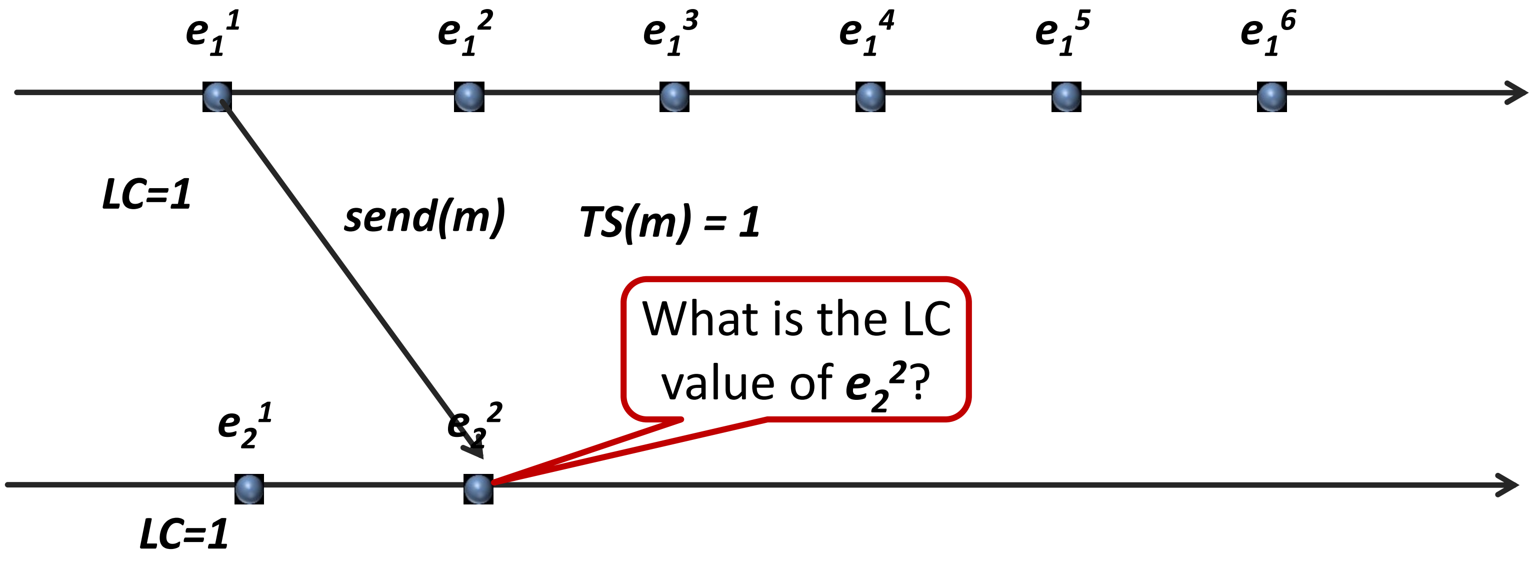
$$max\{LC, TS(m)\} + 1$$



Logical Clocks (cont)

- When process receives m, update logical clock to:

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



Logical Clocks (cont)

- When process receives m, update logical clock to:

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

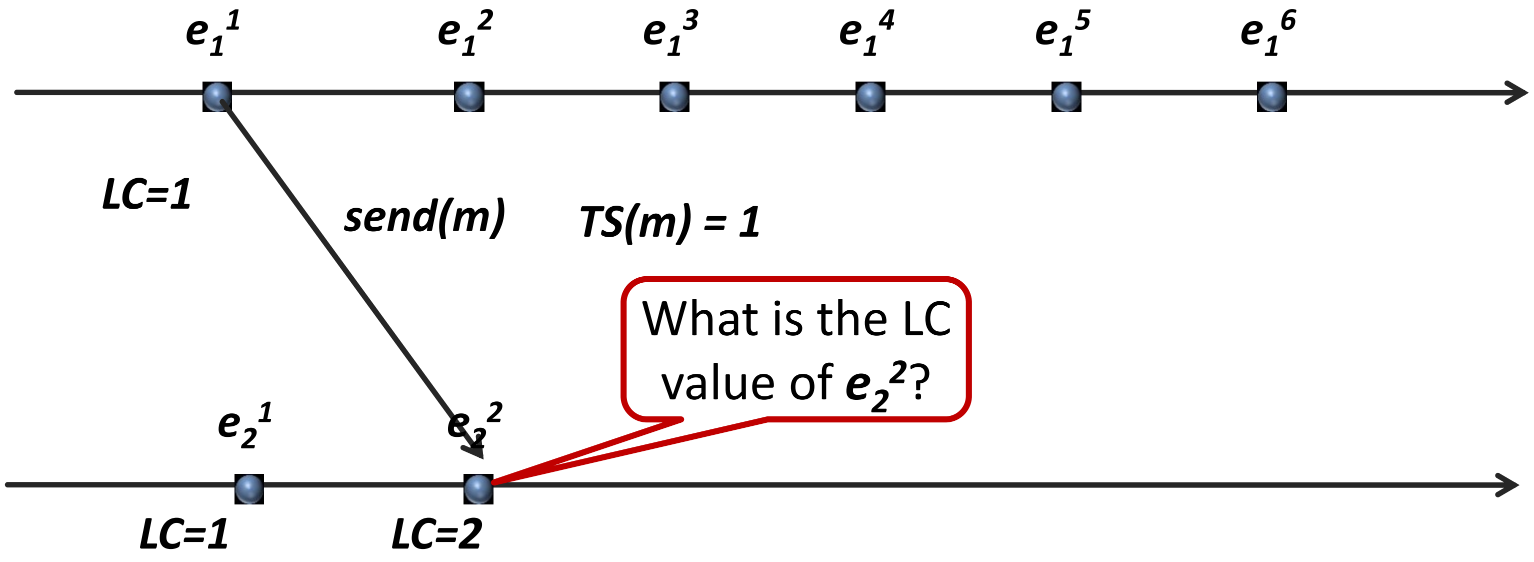


Illustration of a Logical Clock

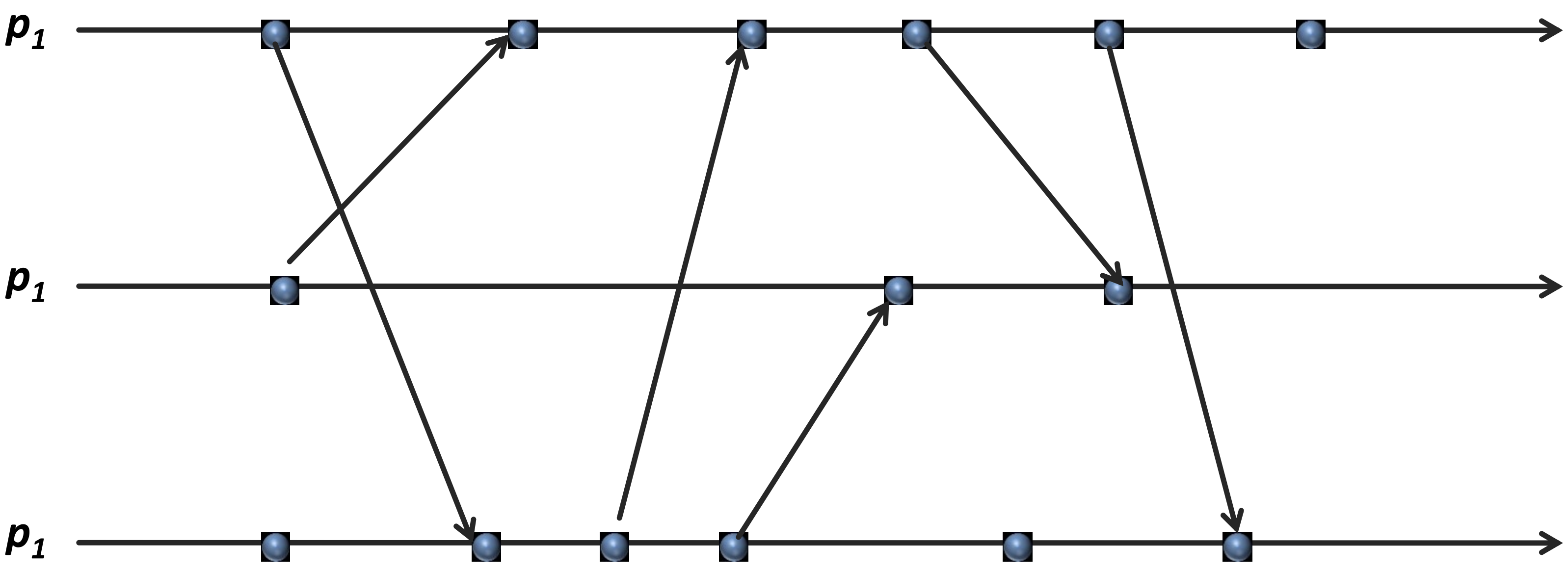


Illustration of a Logical Clock

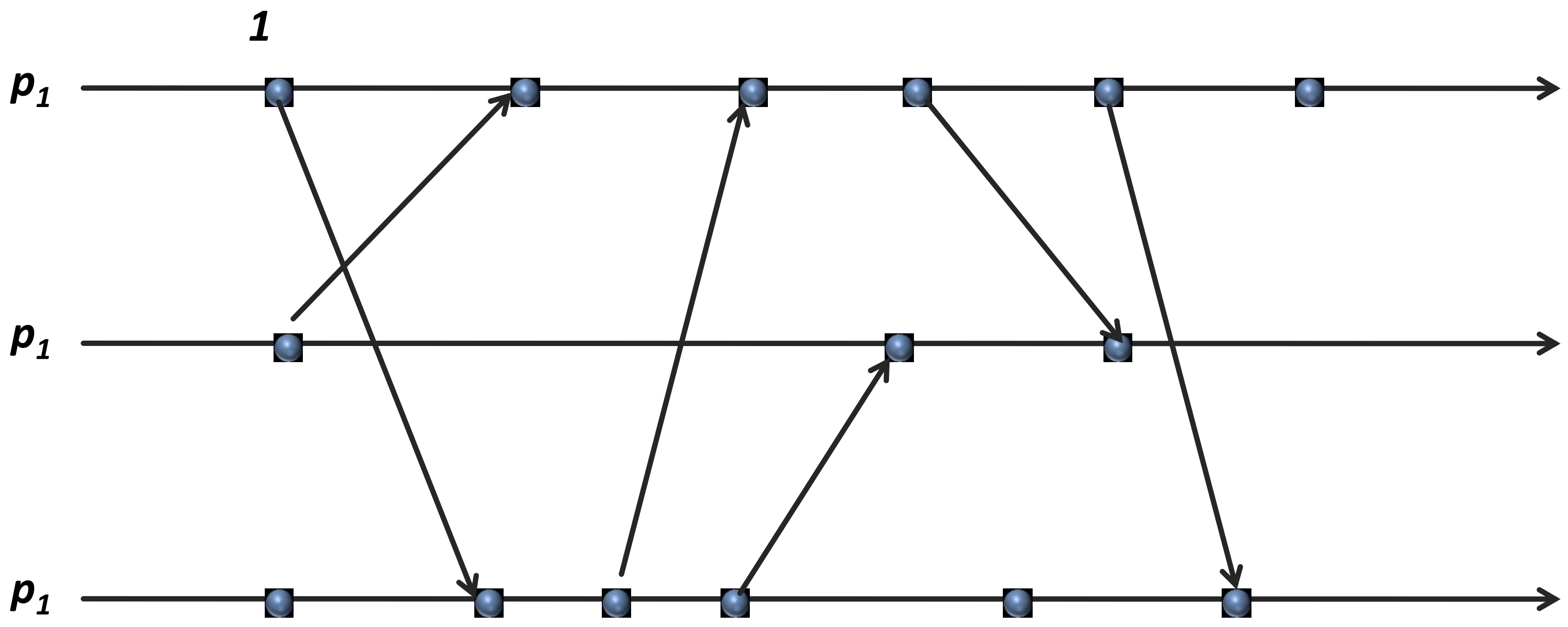


Illustration of a Logical Clock

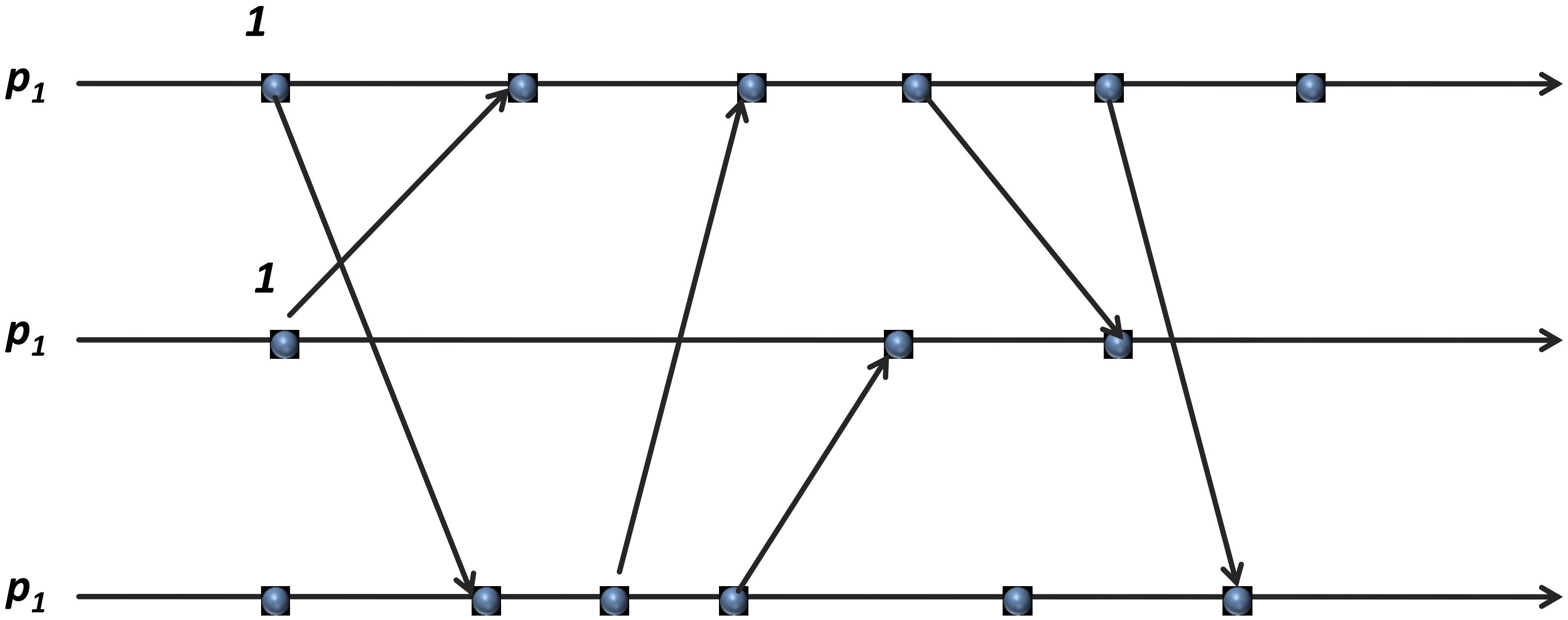


Illustration of a Logical Clock

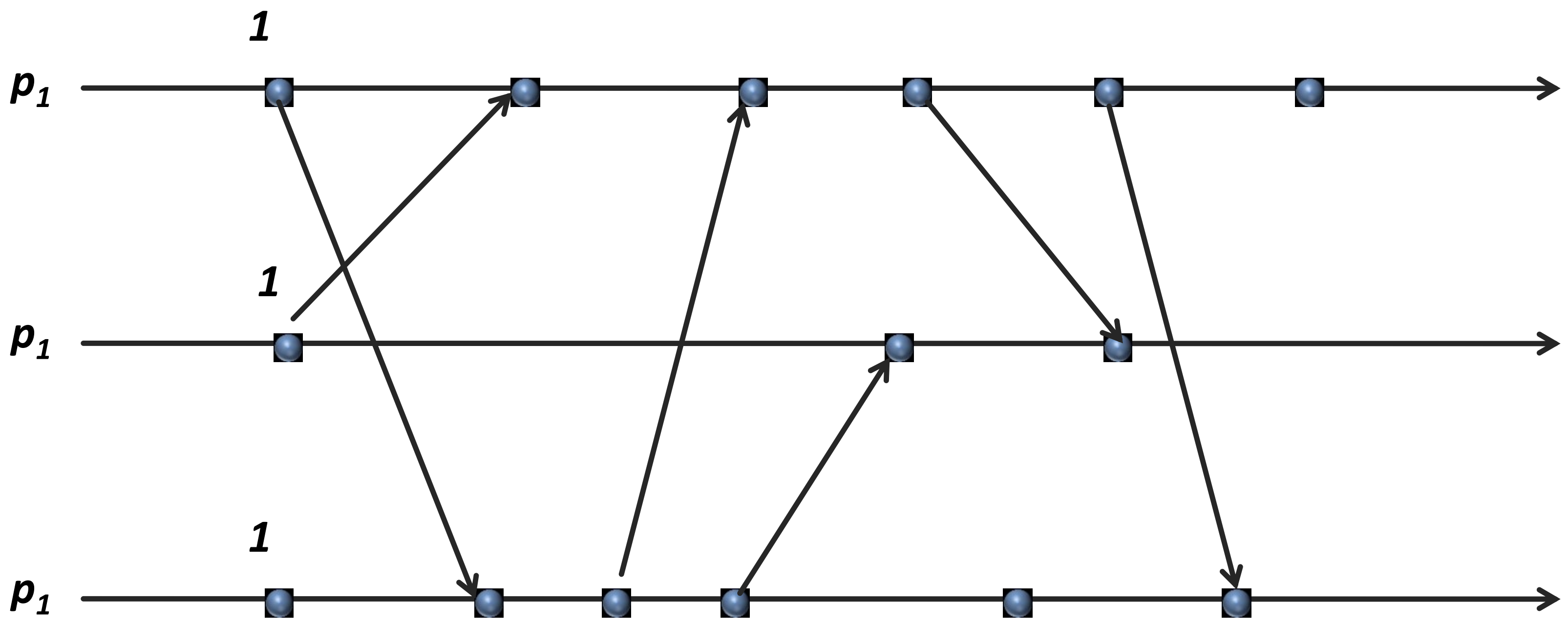


Illustration of a Logical Clock

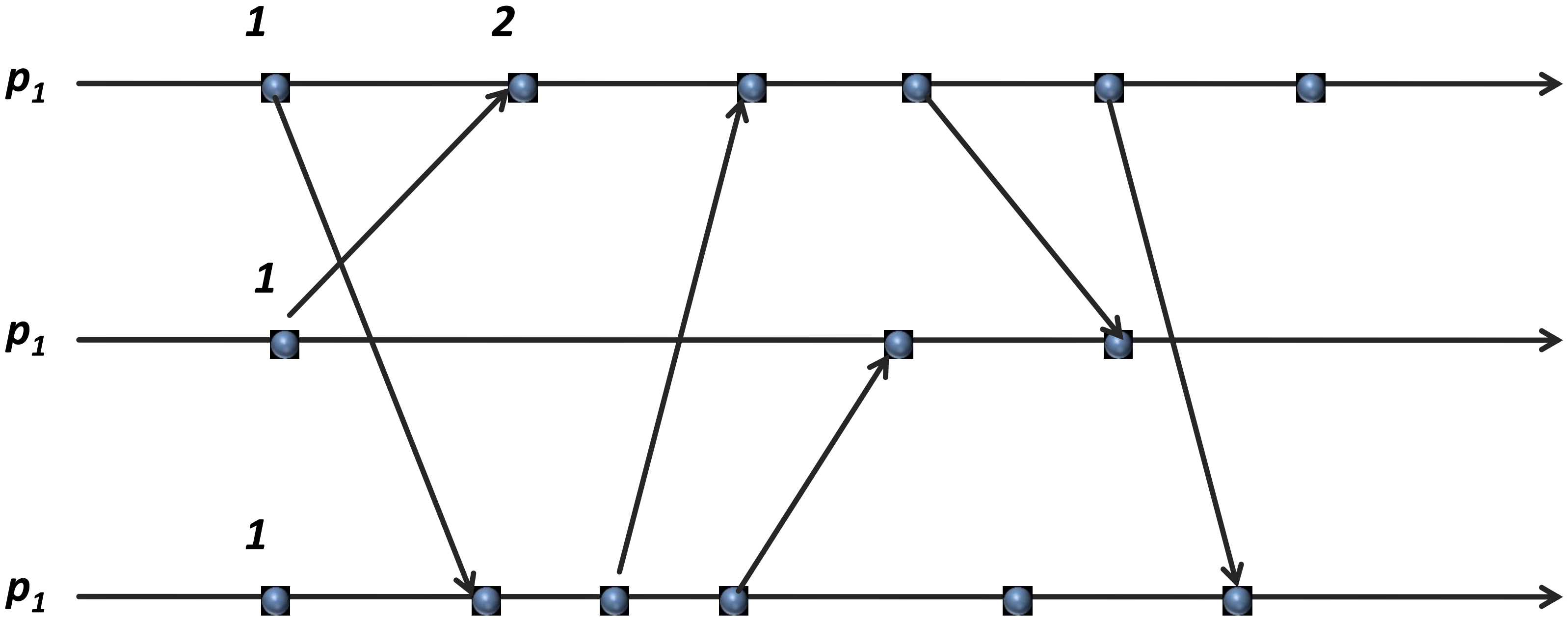


Illustration of a Logical Clock

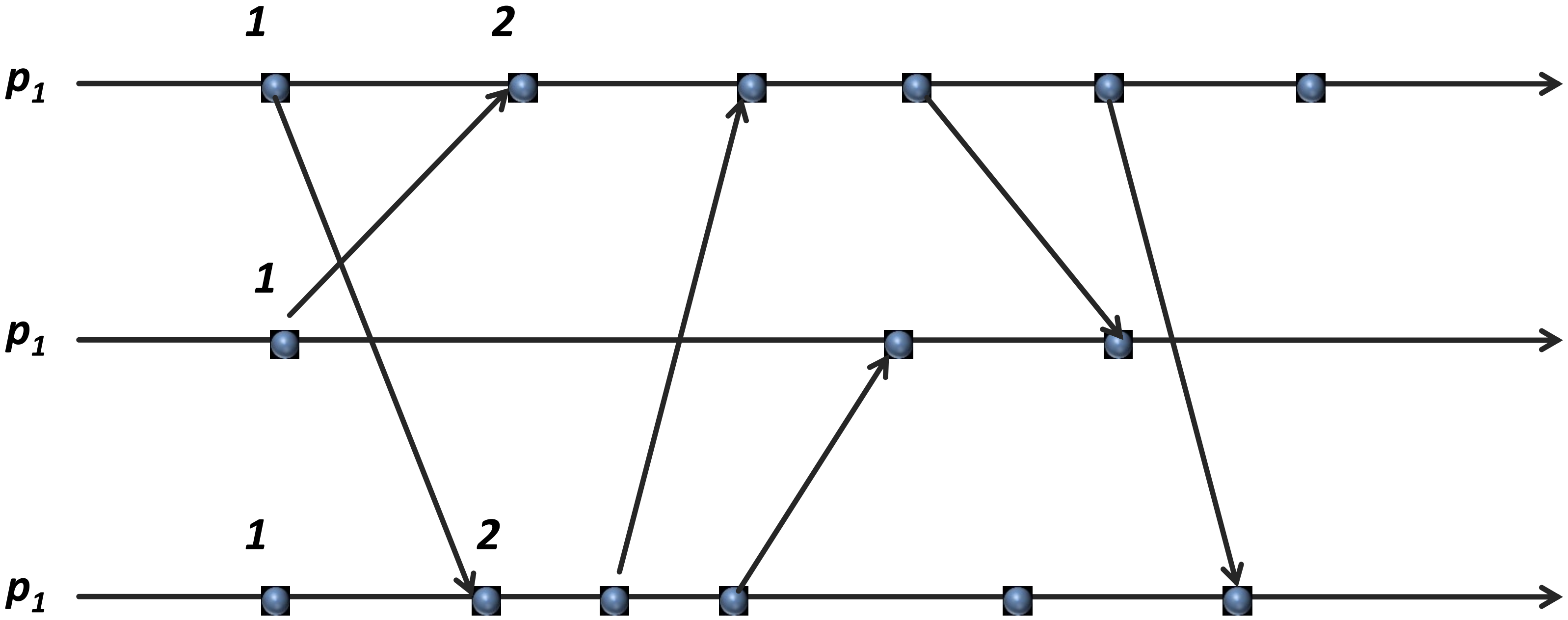


Illustration of a Logical Clock

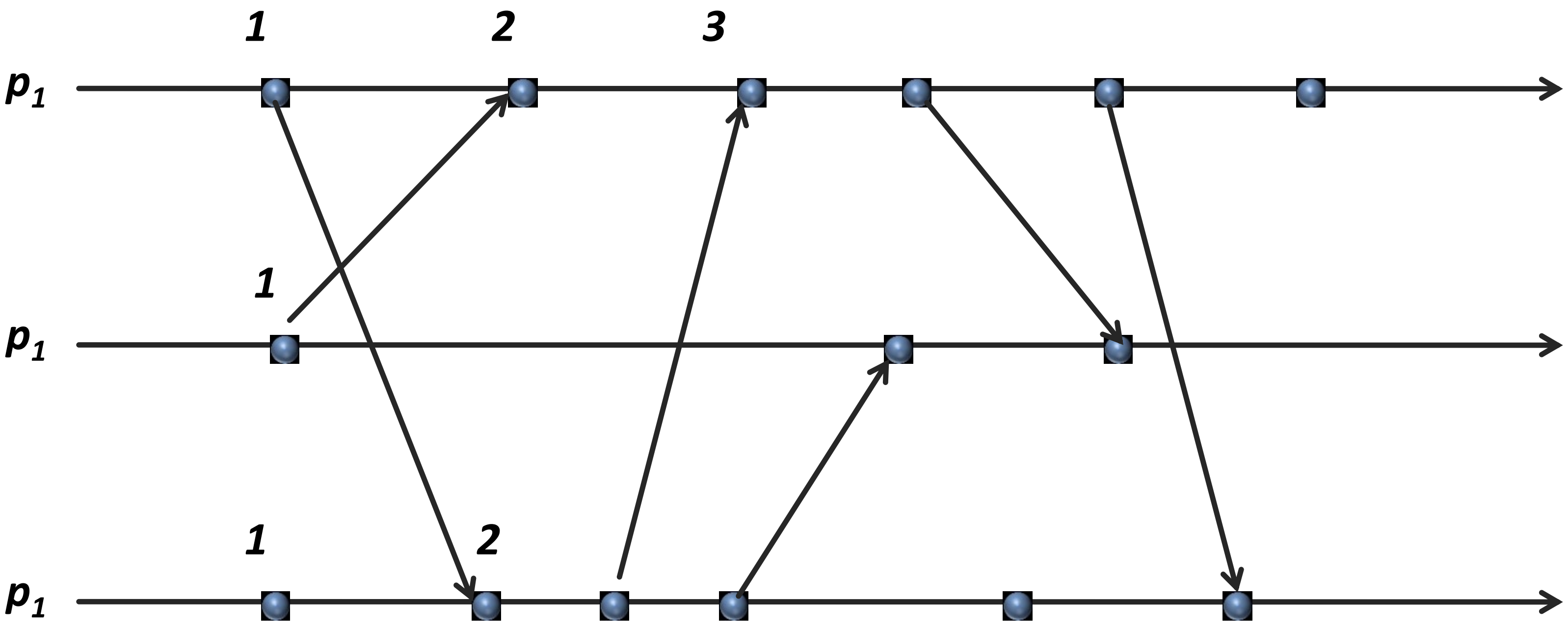


Illustration of a Logical Clock

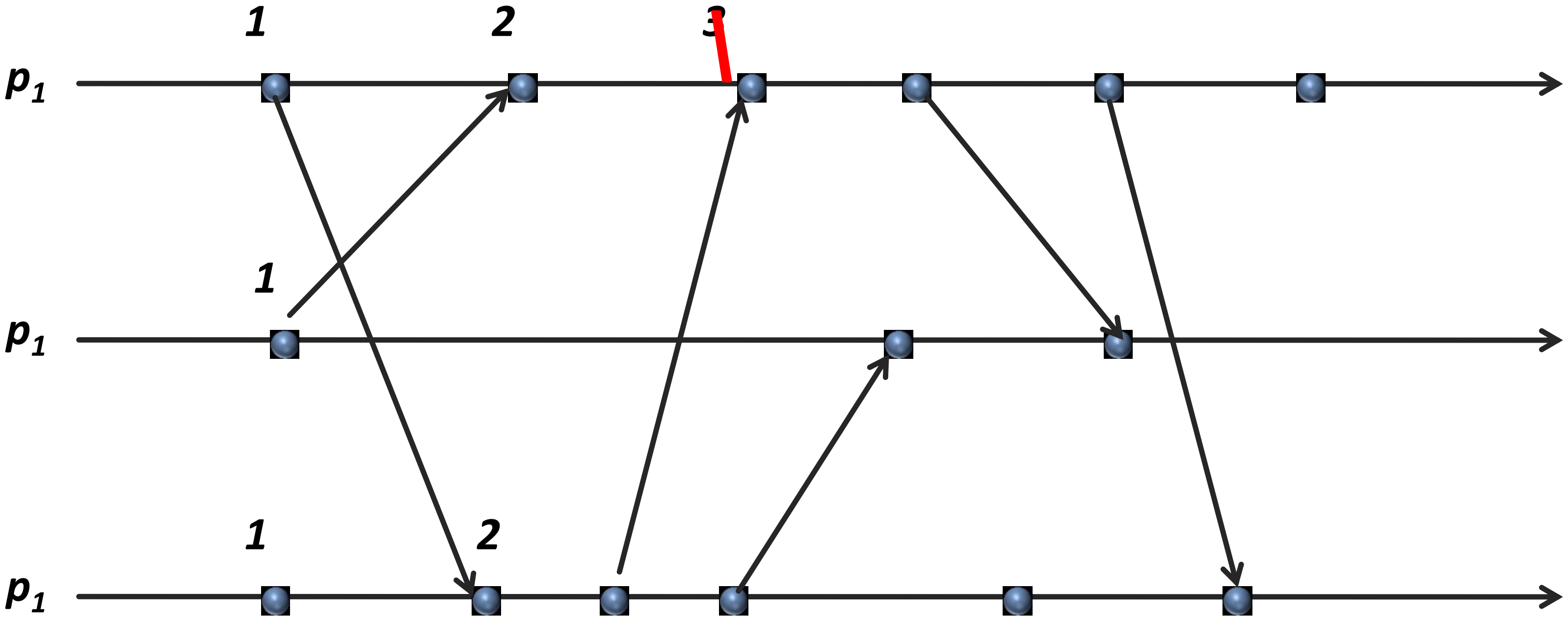


Illustration of a Logical Clock

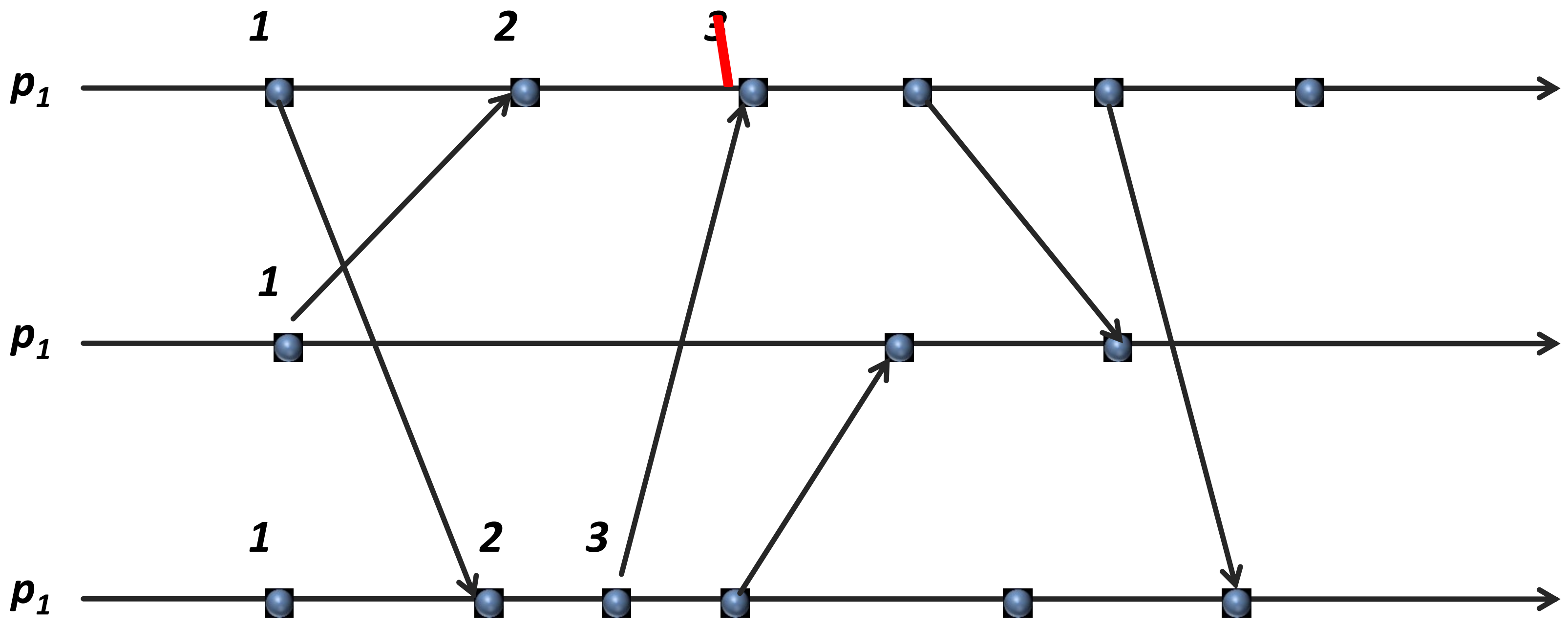


Illustration of a Logical Clock

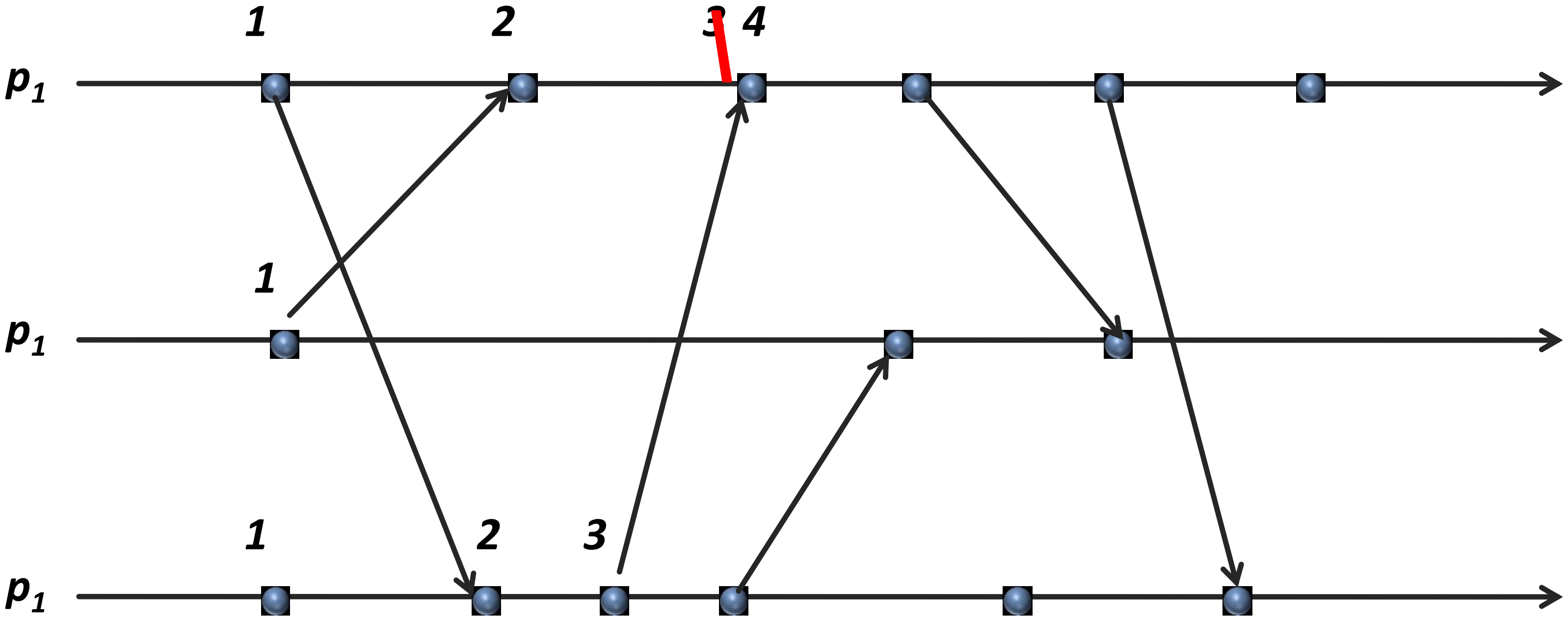


Illustration of a Logical Clock

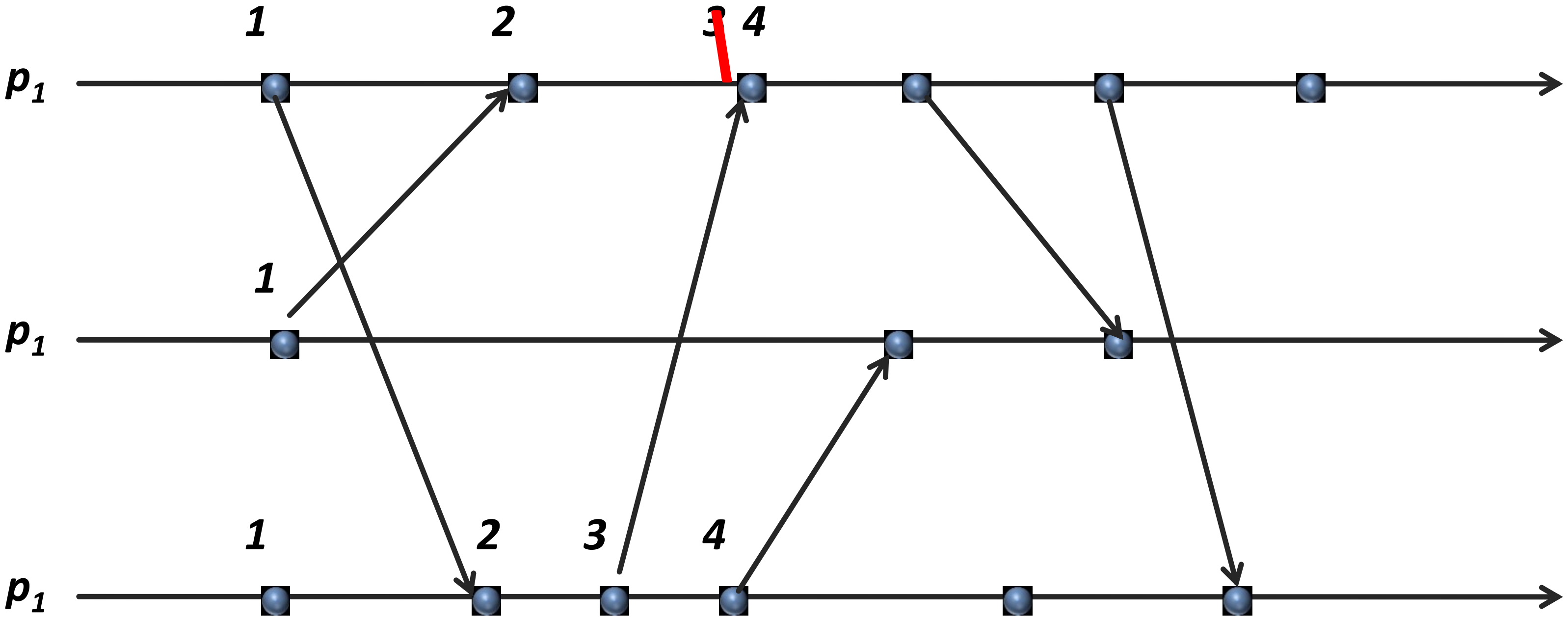


Illustration of a Logical Clock

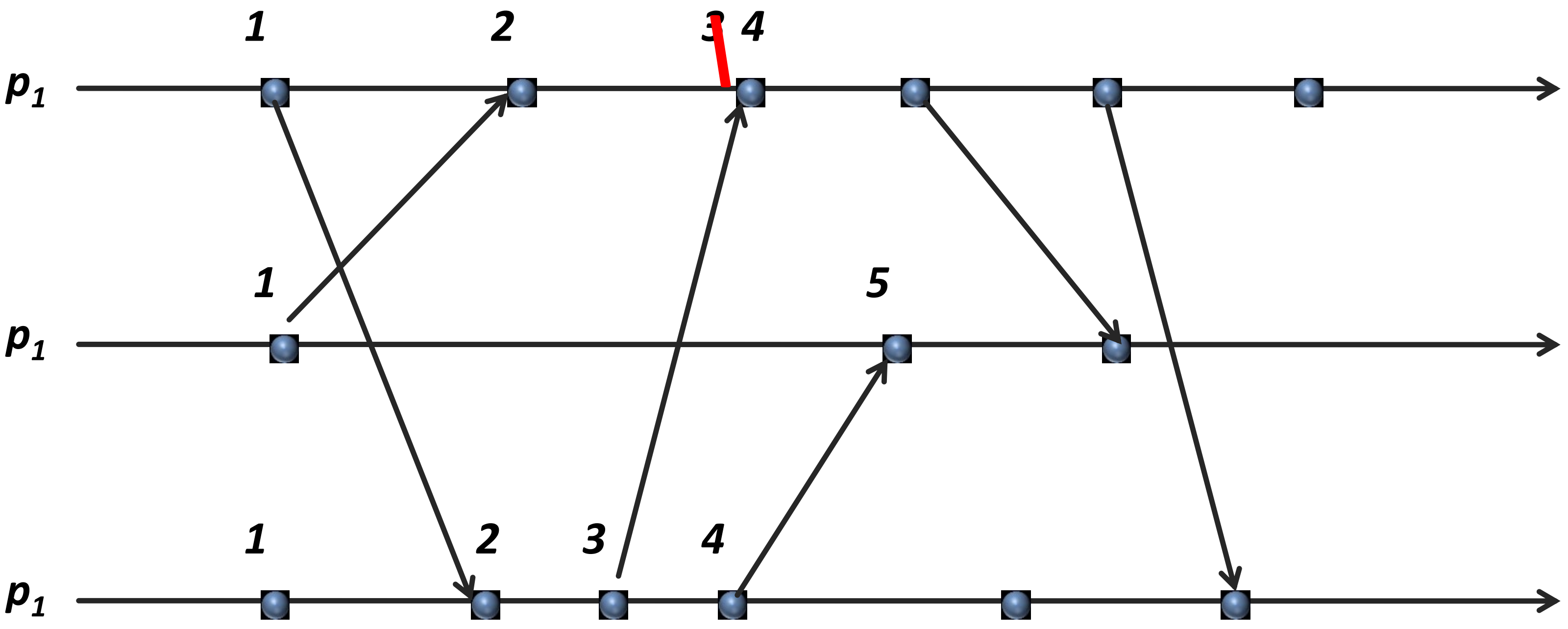


Illustration of a Logical Clock

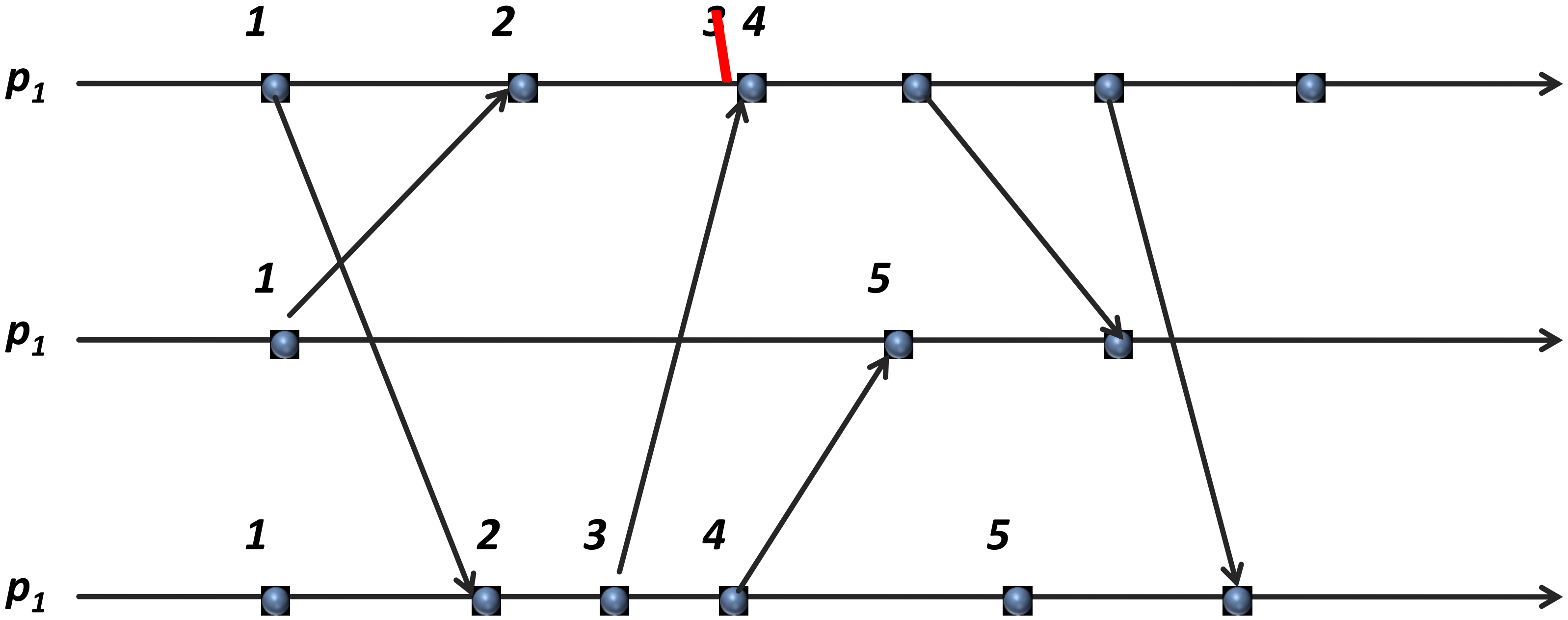


Illustration of a Logical Clock

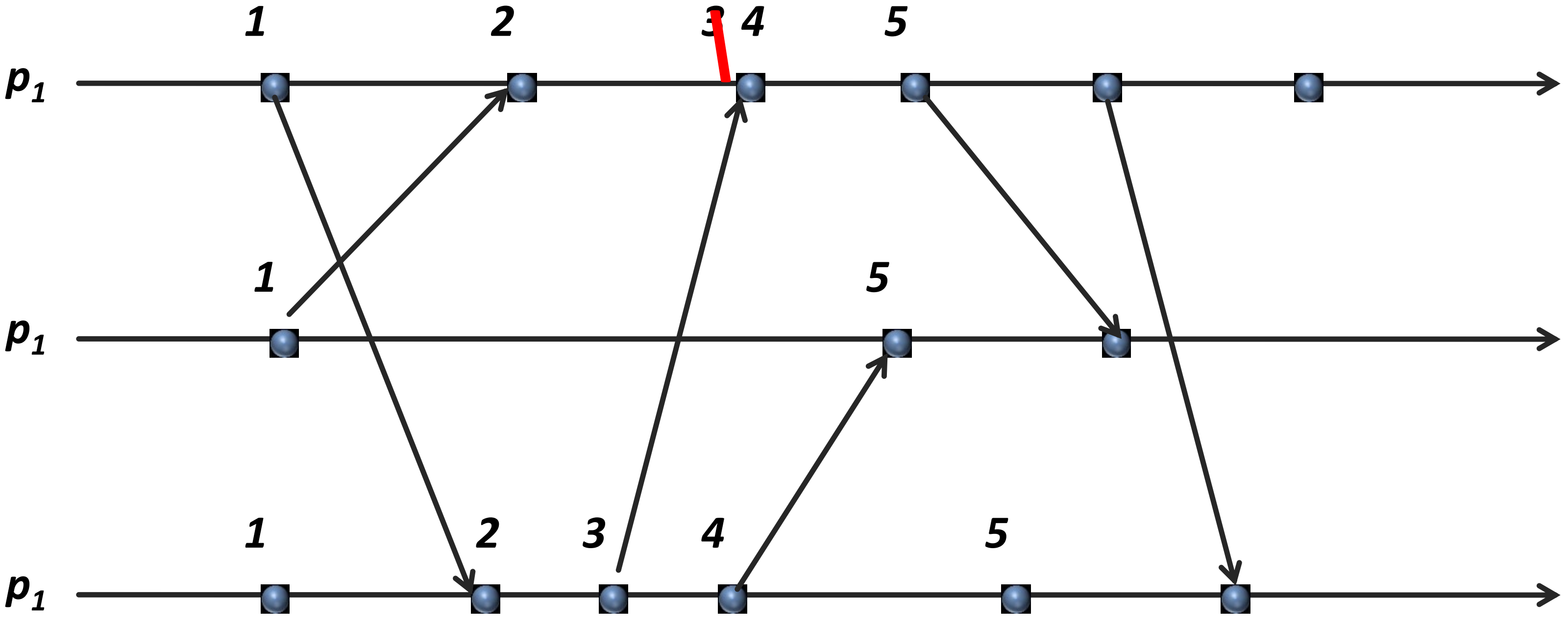


Illustration of a Logical Clock

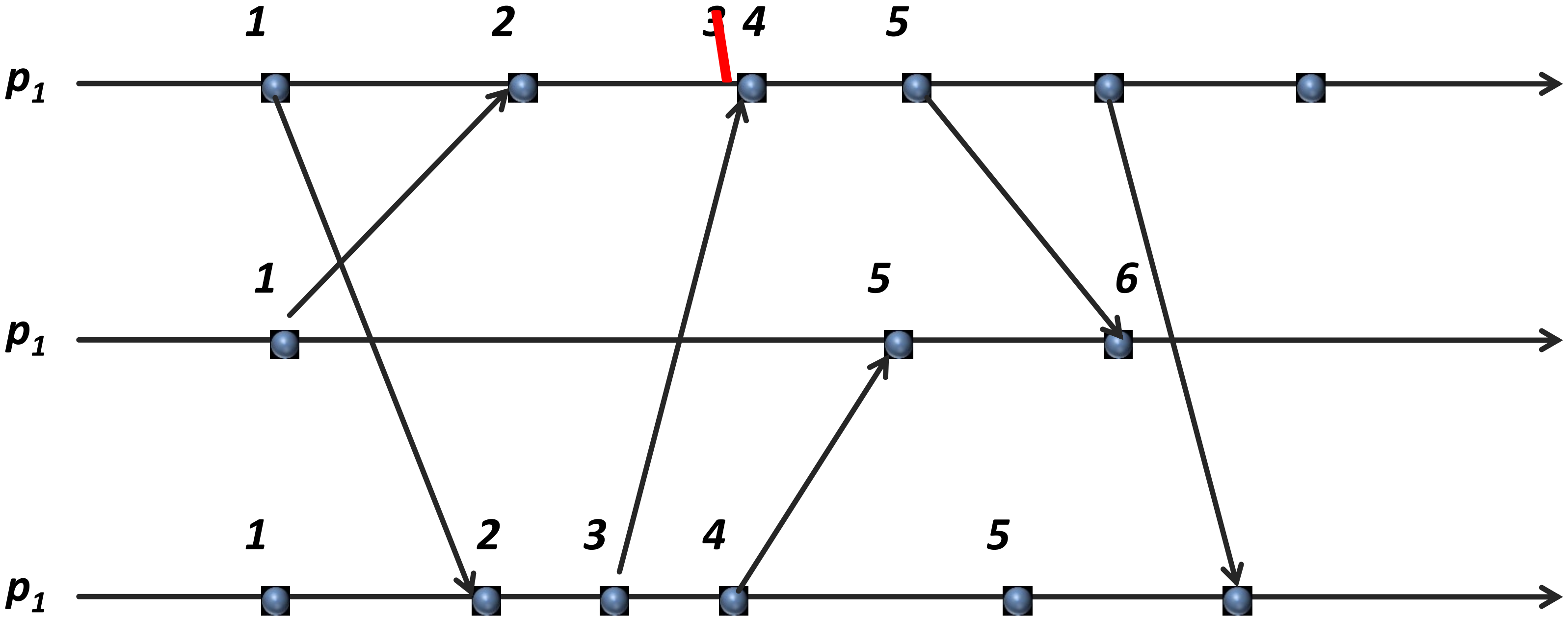


Illustration of a Logical Clock

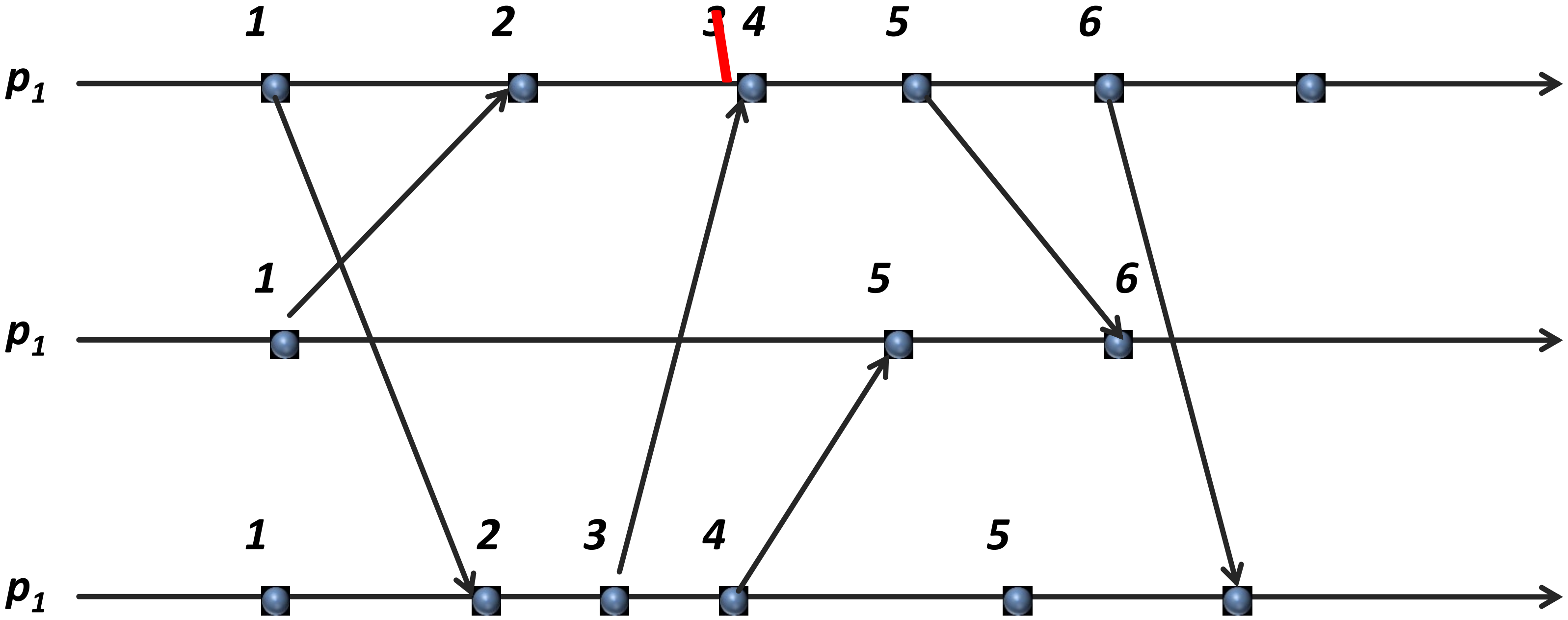


Illustration of a Logical Clock

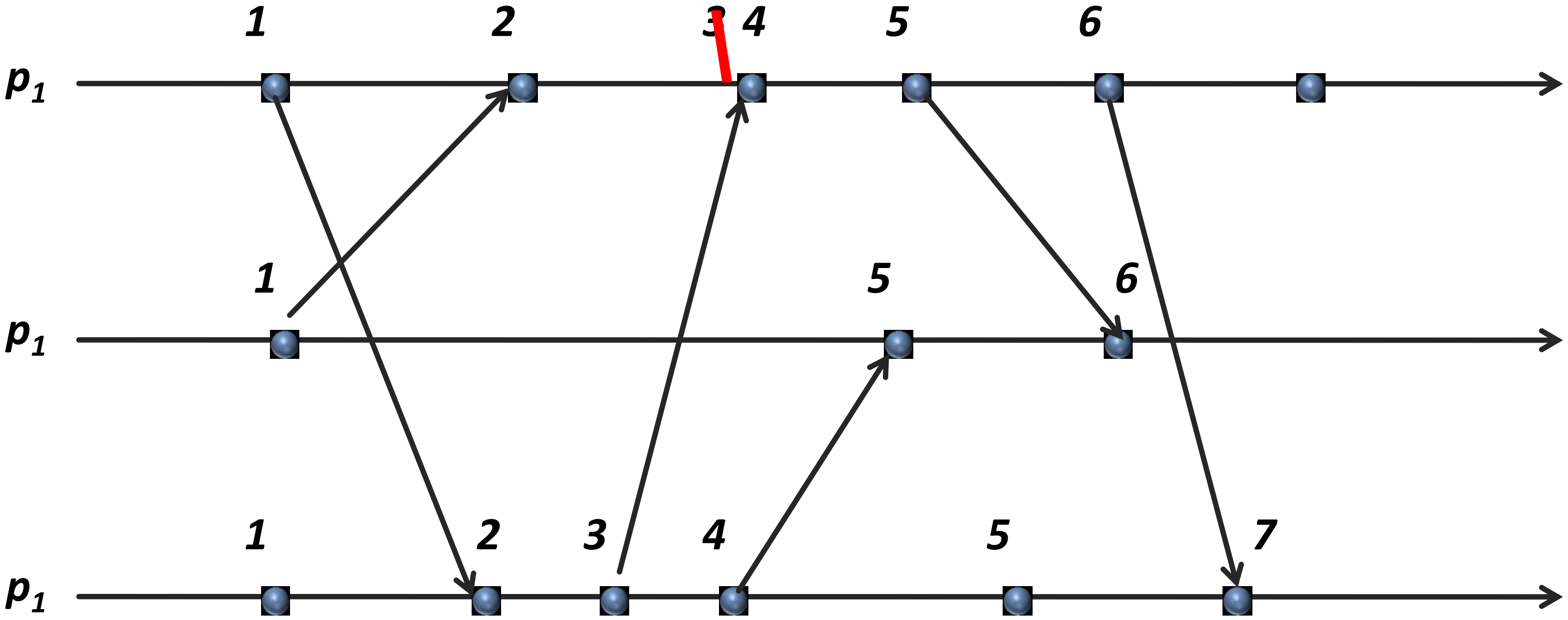


Illustration of a Logical Clock

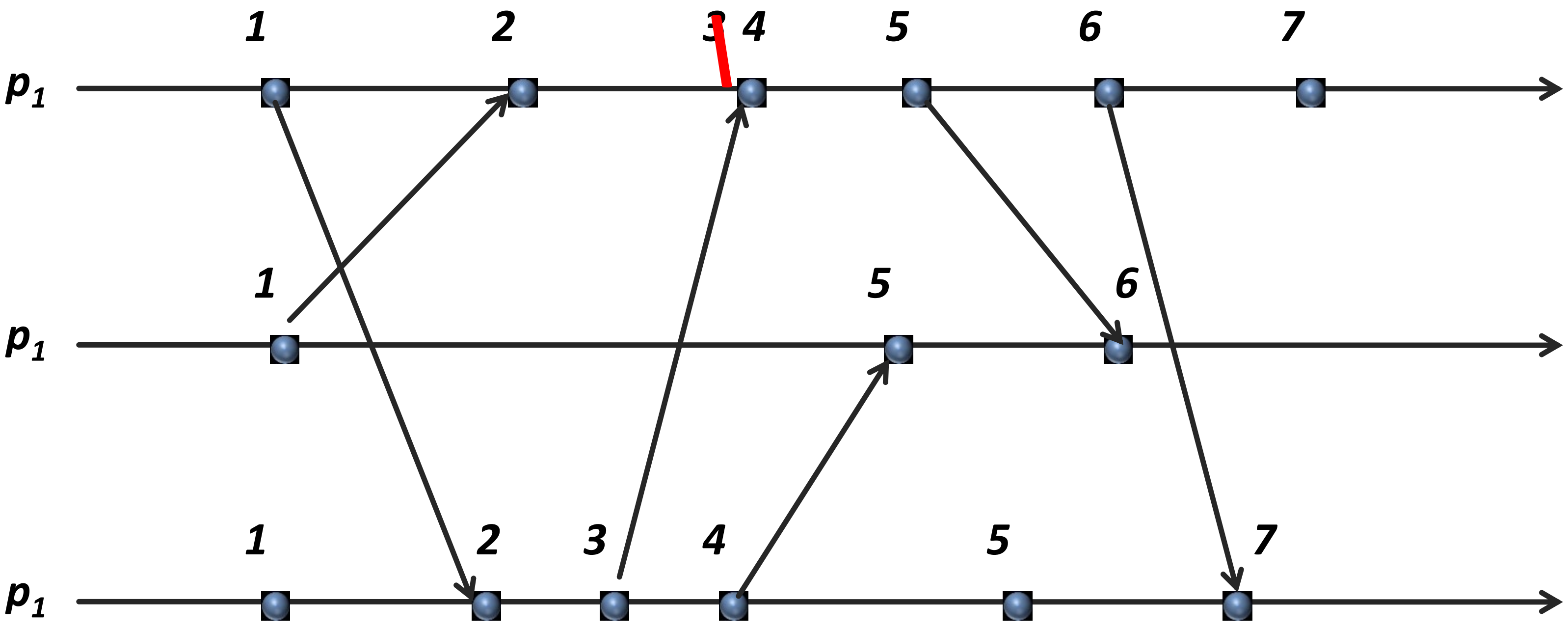
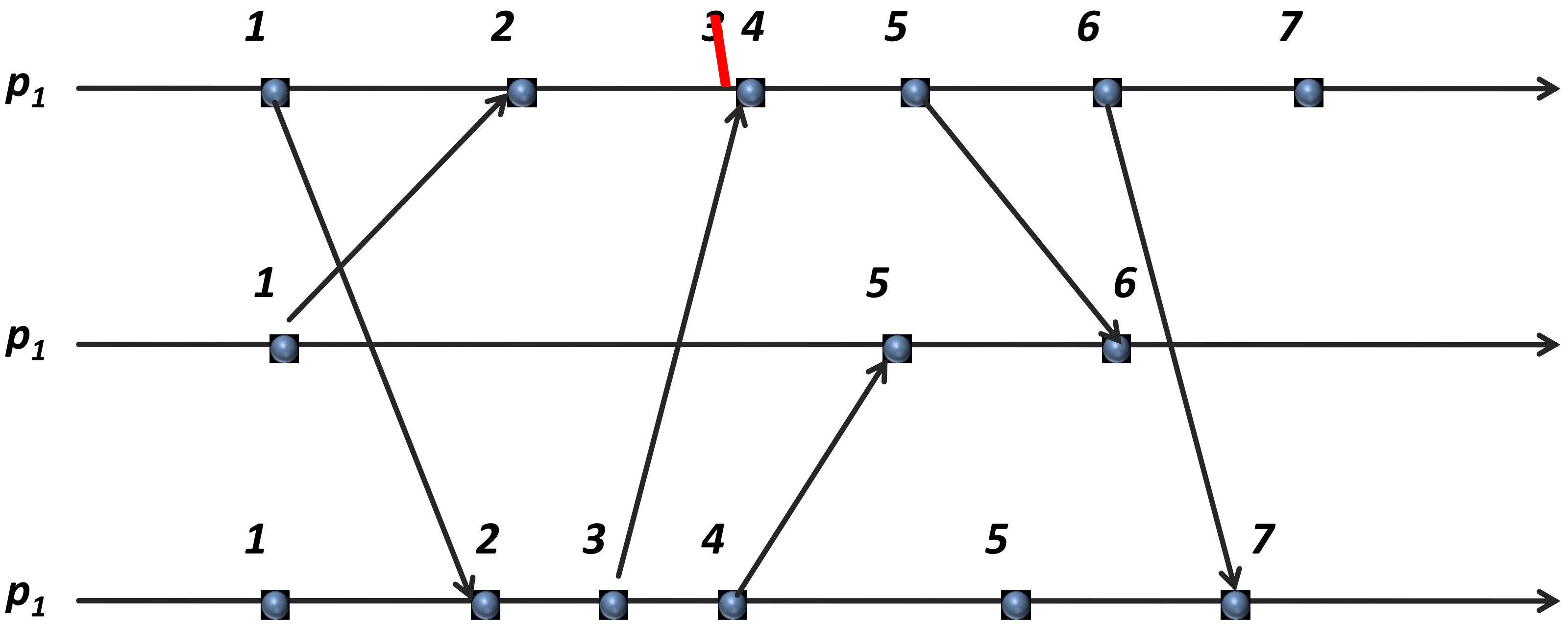
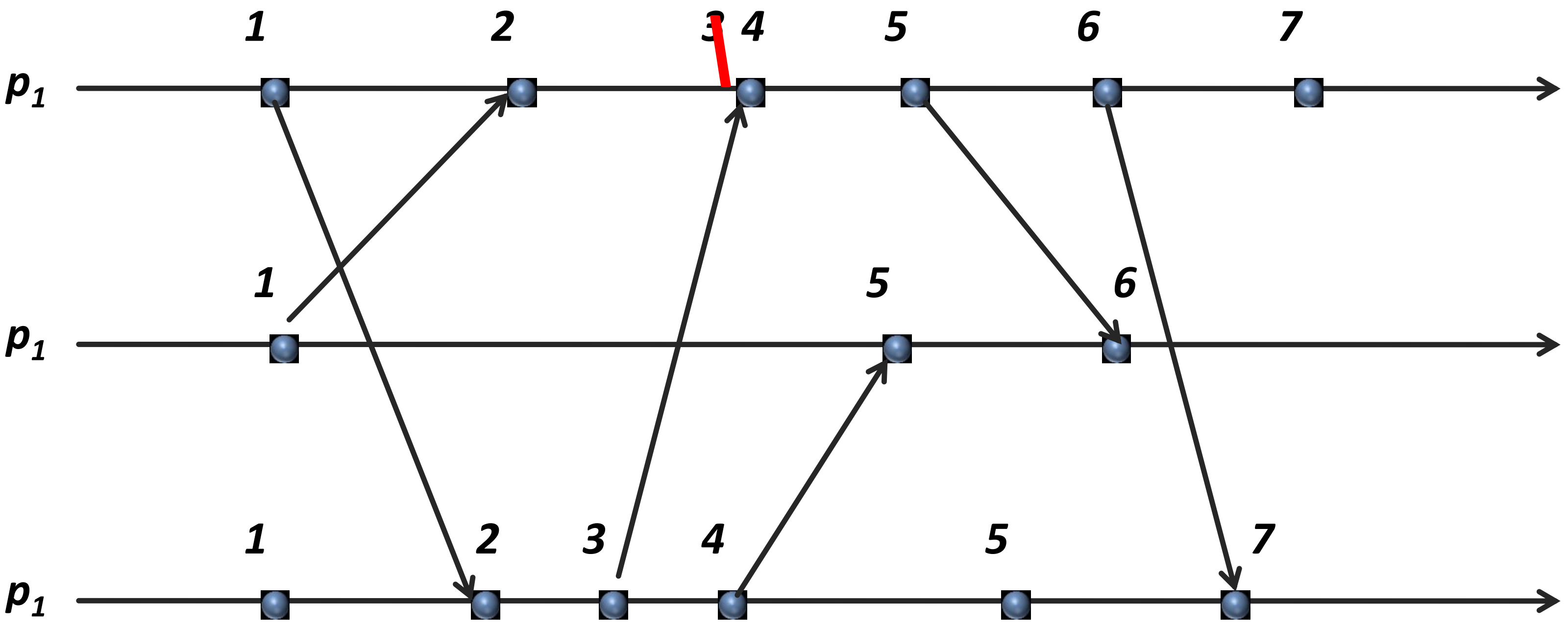


Illustration of a Logical Clock



Awesome, right?
Any drawbacks?

Illustration of a Logical Clock



Awesome, right?
Any drawbacks?

Total vs Partial Order

Vector Clock

Vector Clock

Replace Logical scalar with Vector!

Vector Clock

Replace Logical scalar with Vector!

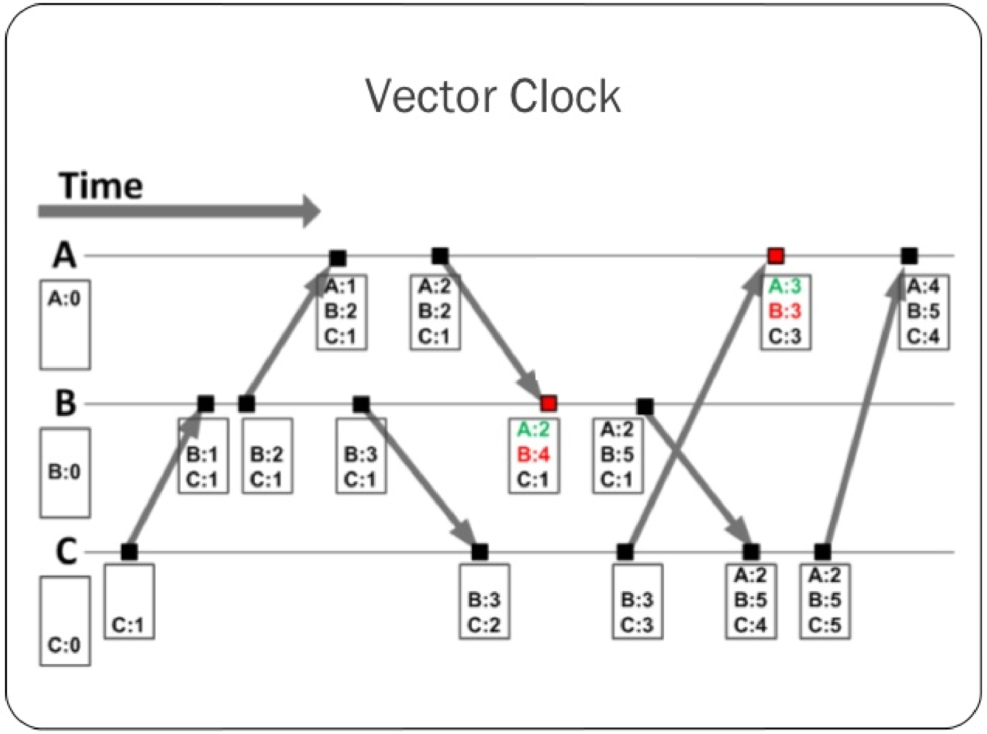
$V_i[i]$: #events occurred at i

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

Update

- On local-event: increment $V_i[I]$
- On send: 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 Logical scalar with Vector!

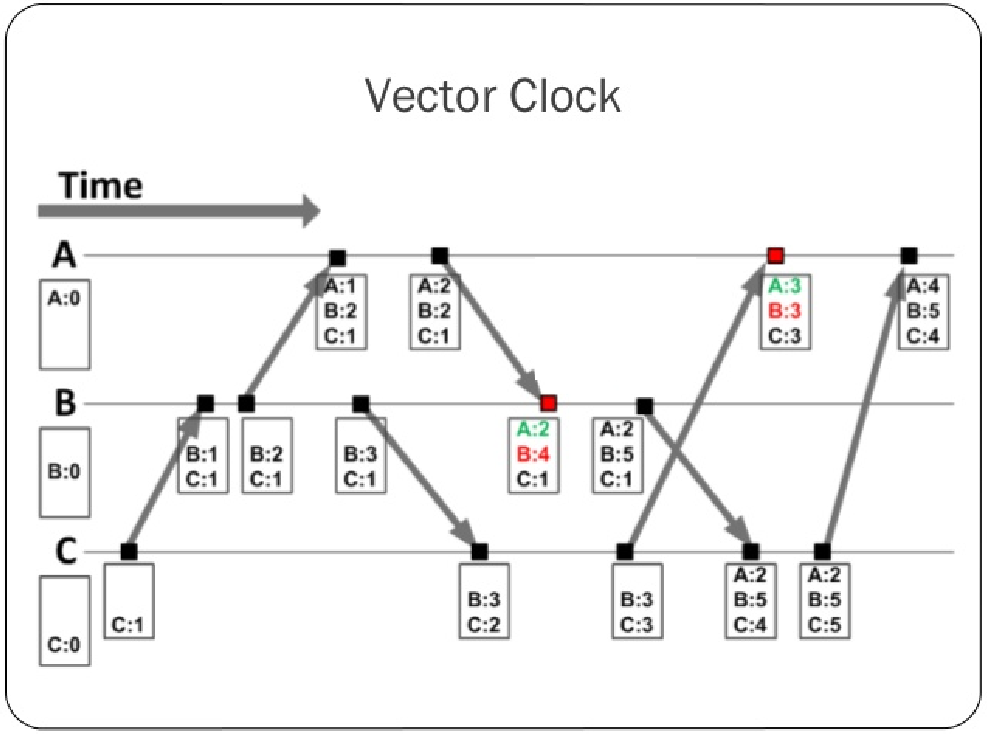
$V_i[i]$: #events occurred at i

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

Update

- On local-event: increment $V_i[I]$
- On send: 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 Logical scalar with Vector!

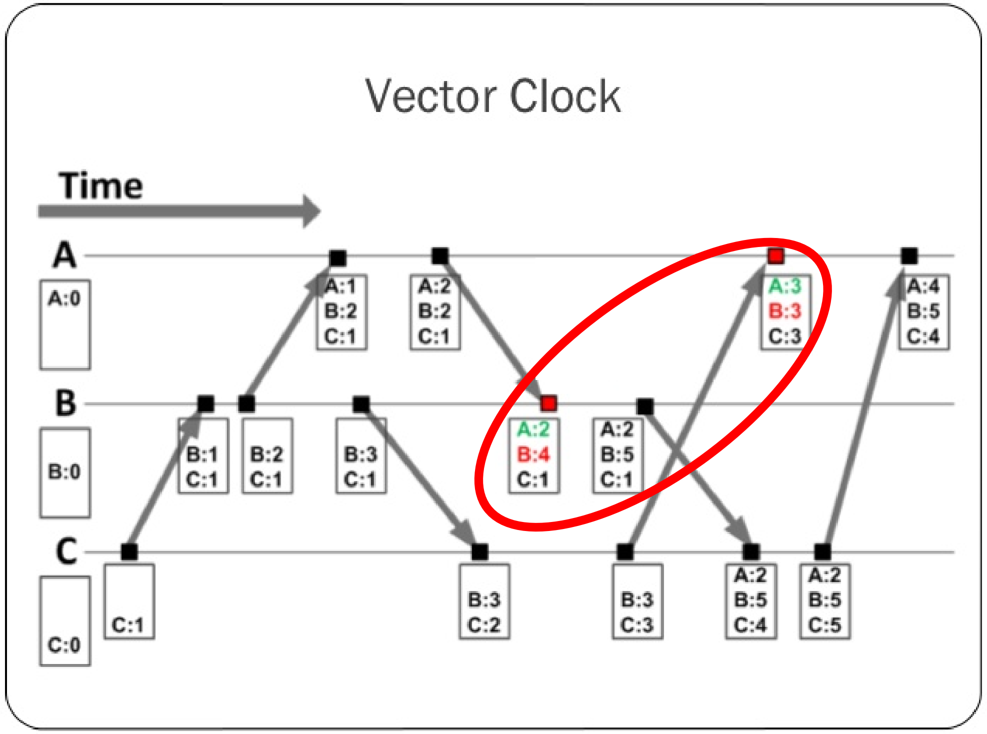
$V_i[i]$: #events occurred at i

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

Update

- On local-event: increment $V_i[I]$
- On send: 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 Logical scalar with Vector!

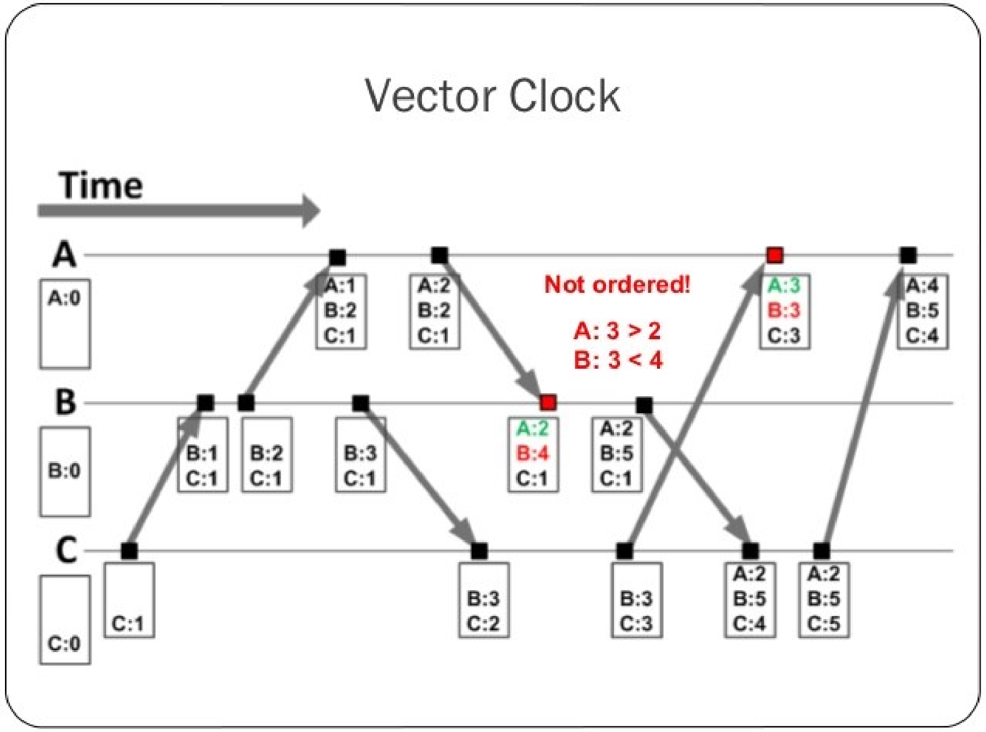
$V_i[i]$: #events occurred at i

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

Update

- On local-event: increment $V_i[I]$
- On send: 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 Logical scalar with Vector!

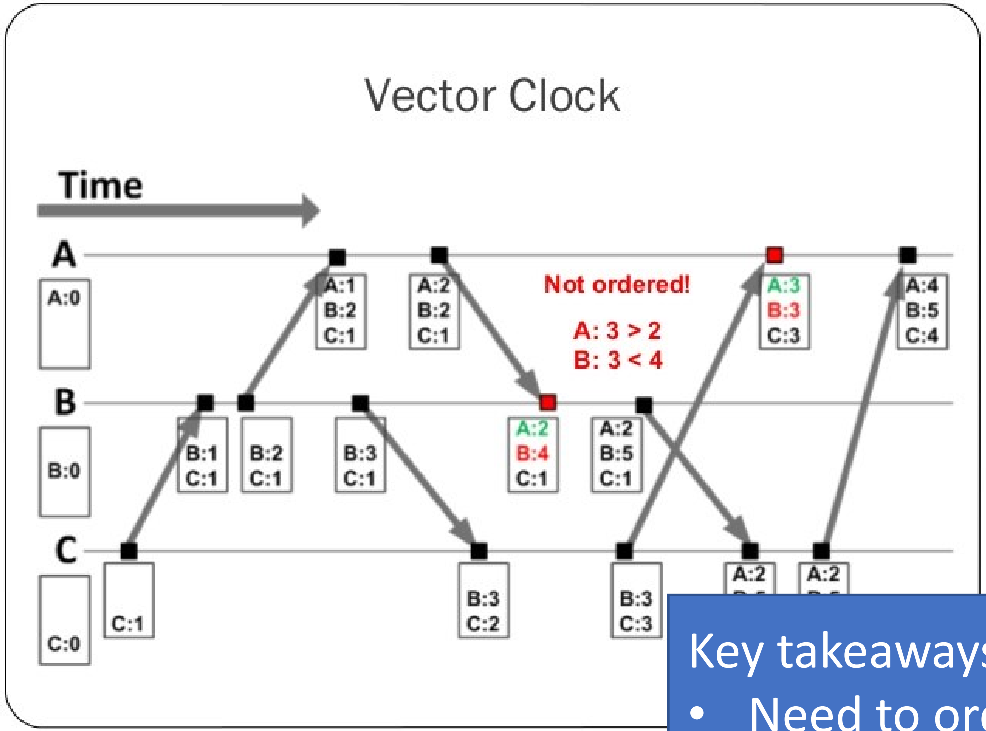
$V_i[i]$: #events occurred at i

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

Update

- On local-event: increment $V_i[I]$
- On send: 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 Logical scalar 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: 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)*

Key takeaways:

- 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
- *Good primitive for tracking happens-before*

Happens-before

Happens-before

- Difficult to implement
  - Need logical/vector clocks!
  - Requires per-thread information

Happens-before

- Difficult to implement
  - Need logical/vector clocks!
  - Requires per-thread information

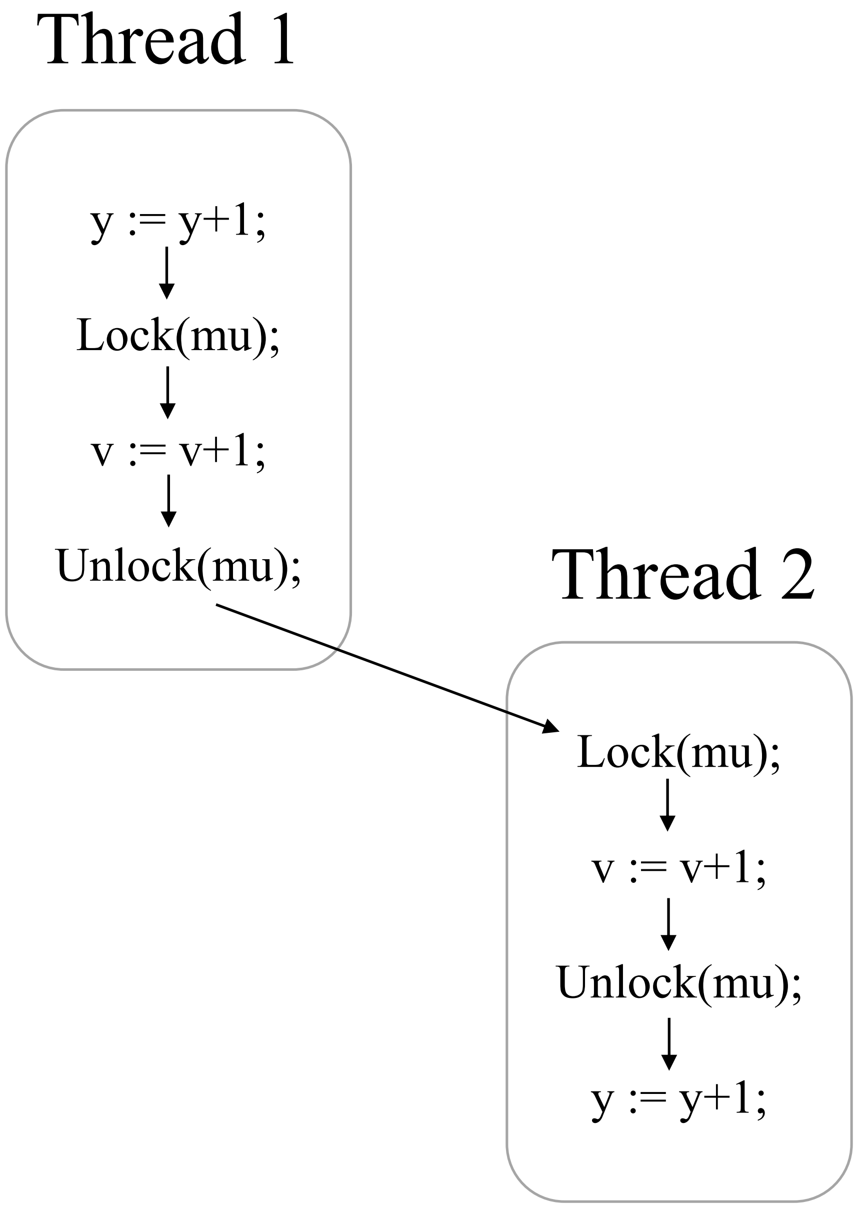
- Dependent on the interleaving produced by the scheduler

Happens-before

- Difficult to implement
  - Need logical/vector clocks!
  - Requires per-thread information
- Dependent on the interleaving produced by the scheduler
- Example

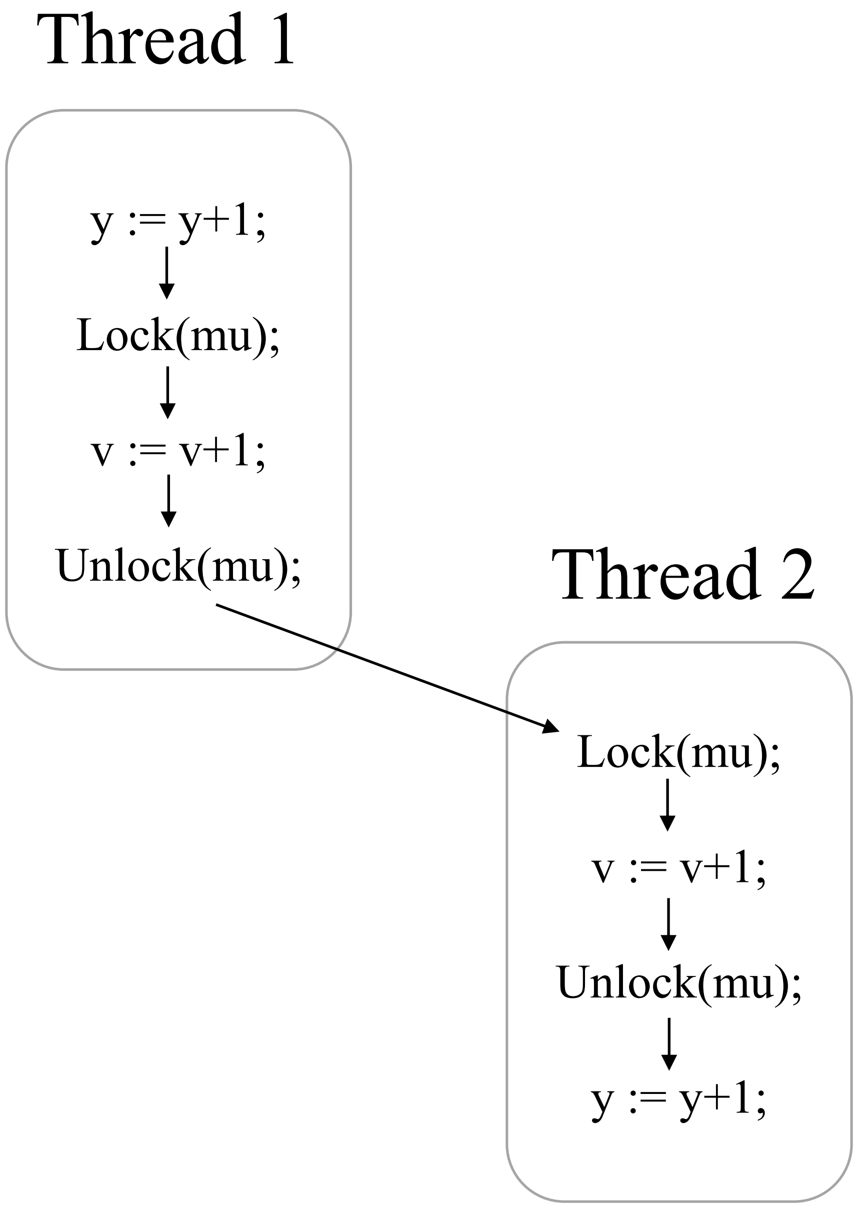
Happens-before

- Difficult to implement
  - Need logical/vector clocks!
  - Requires per-thread information
- Dependent on the interleaving produced by the scheduler
- Example



Happens-before

- Difficult to implement
  - Need logical/vector clocks!
  - 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



Happens-before

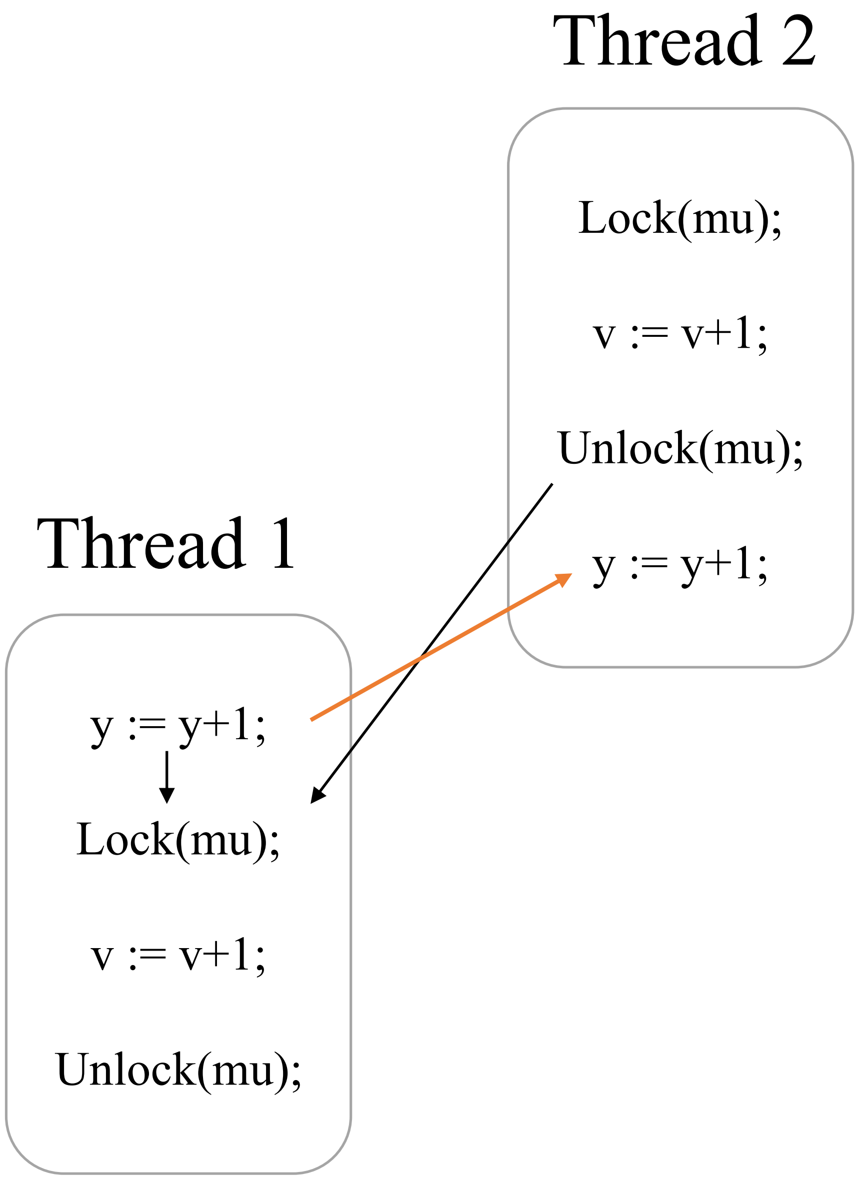
- Difficult to implement
  - Need logical/vector clocks!
  - Requires per-thread information
- Dependent on the interleaving produced by the scheduler
- Example
  - T1-acc(v) happens before T2-acc(v)
  - T1-acc(y) happens before T1-acc(v)
  - T2-acc(v) happens before T2-acc(y)
  - Conclusion: no race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

Happens-before

- Difficult to implement
  - Need logical/vector clocks!
  - 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



Better Dynamic Race Detection

- 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

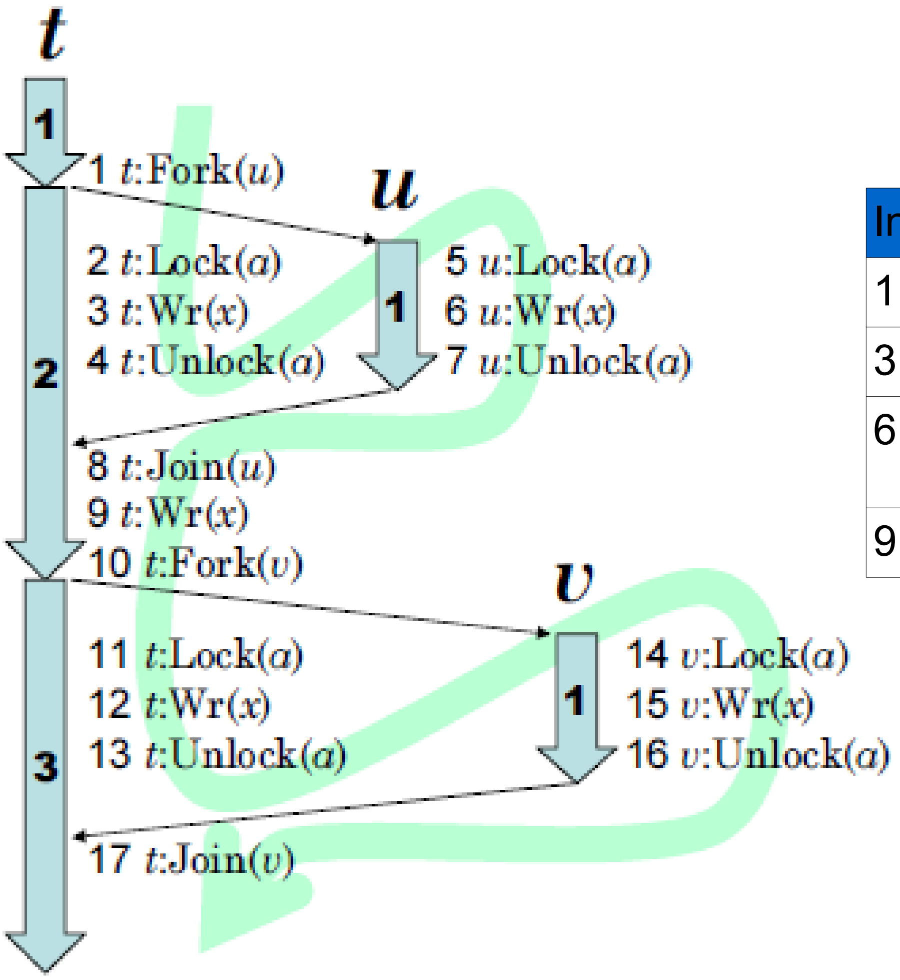
RaceTrack = Lockset + Happens-before

Summary

Race detection

- Static vs Dynamic
- Lock set vs. Happens-Before
- Lots of really interesting related work
- Lots of increasingly practical tools

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

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

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

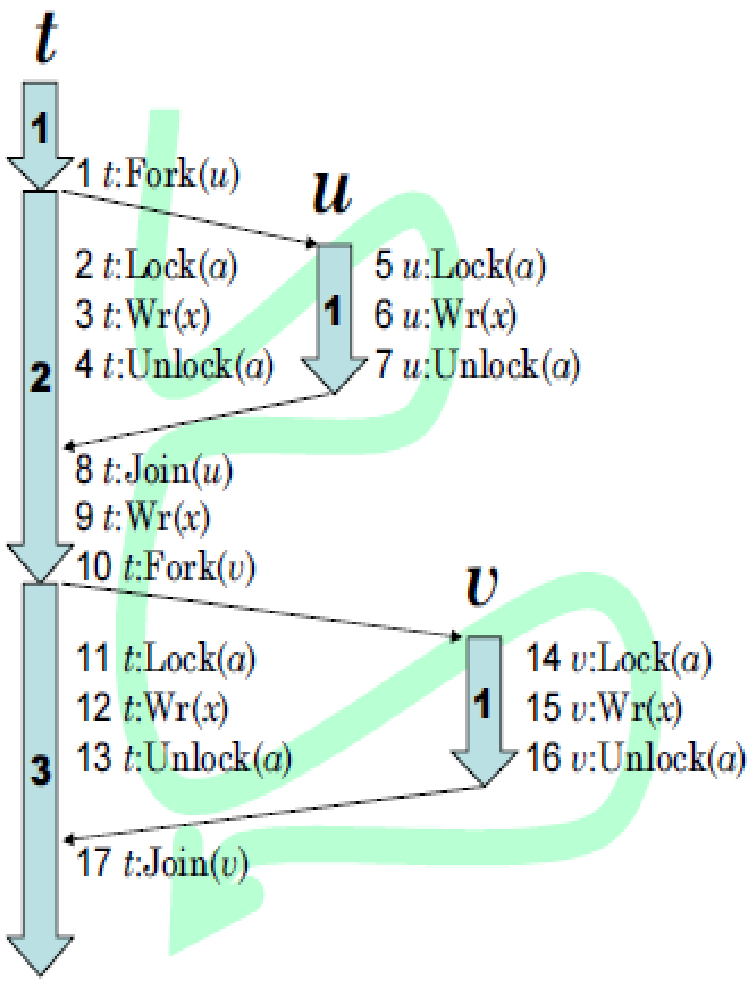
At t:Unlock(l):
$\quad L_t \leftarrow L_t - \{l\}$

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):
$\quad B_t \leftarrow Merge(B_t, B_u)$

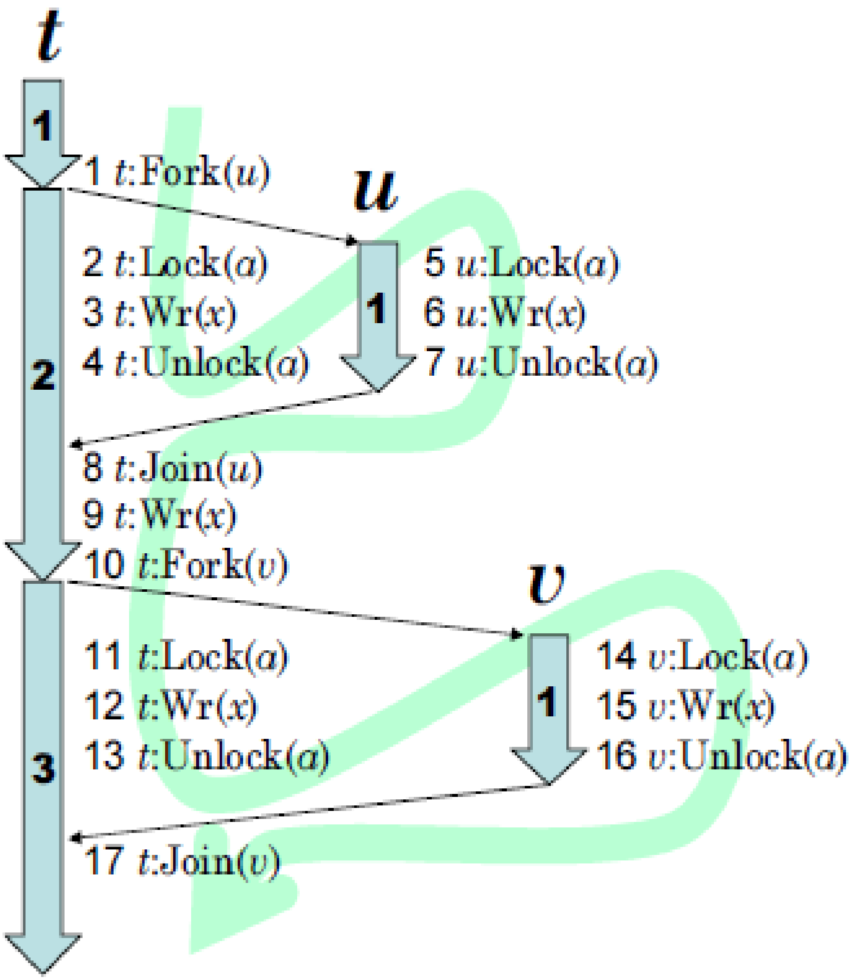
At t:Rd(x) or t:Wr(x):
$\quad S_x \leftarrow Merge(Remove(S_x, B_t), \{\langle t, B_t(t)\rangle\})$
$\quad$ if $|S_x| > 1$
$\quad\quad$ then $C_x \leftarrow C_x \cap L_t$
$\quad\quad$ else $C_x \leftarrow L_t$
$\quad$ if $|S_x| > 1 \wedge C_x = \{\}$ then report race

Avoiding Lockset's false positive (1)



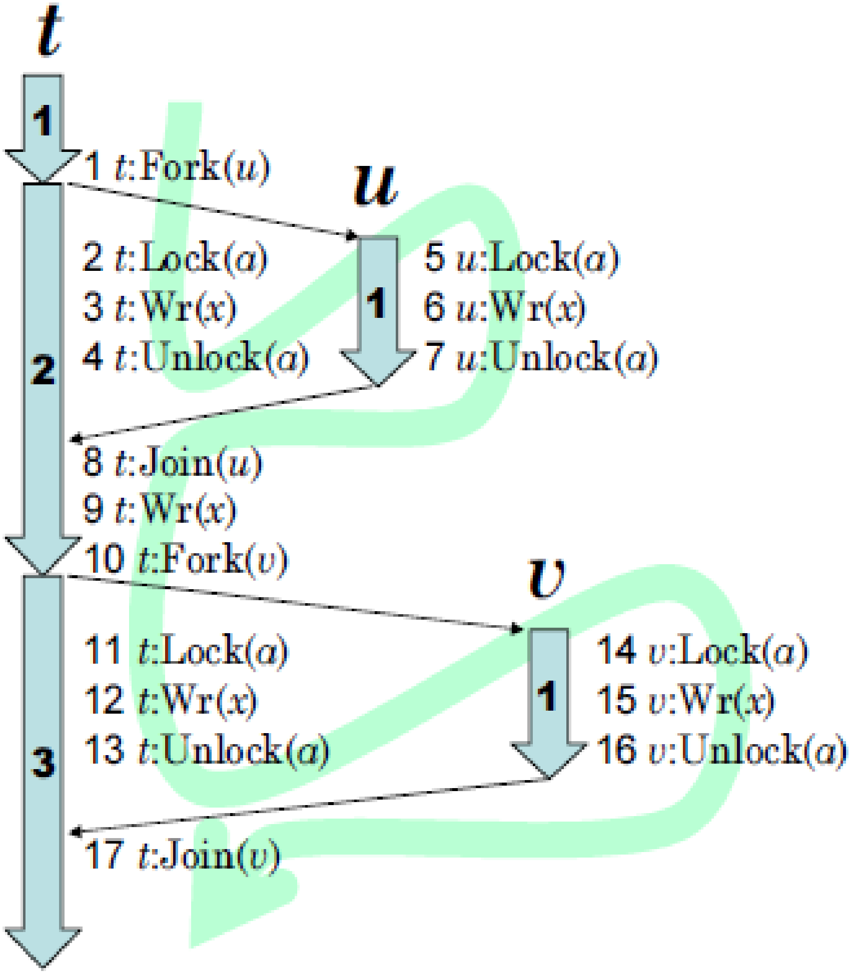
Inst	C_x	S_x	L_t	B_t	L_u	B_u
0	All	{ }	{ }	$\{t_1\}$	-	-
1				$\{t_2\}$	{ }	$\{ t_1,u_1 \}$
2			{a}			
3	{a}	$\{t_2\}$				
4			{ }			
5					{a}	
6		$\{t_2,u_1\}$				
7					{ }	
8				$\{t_2,u_1\}$	-	-

Avoiding Lockset's false positive (2)



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					{ }	

Avoiding Lockset's false positive (2)

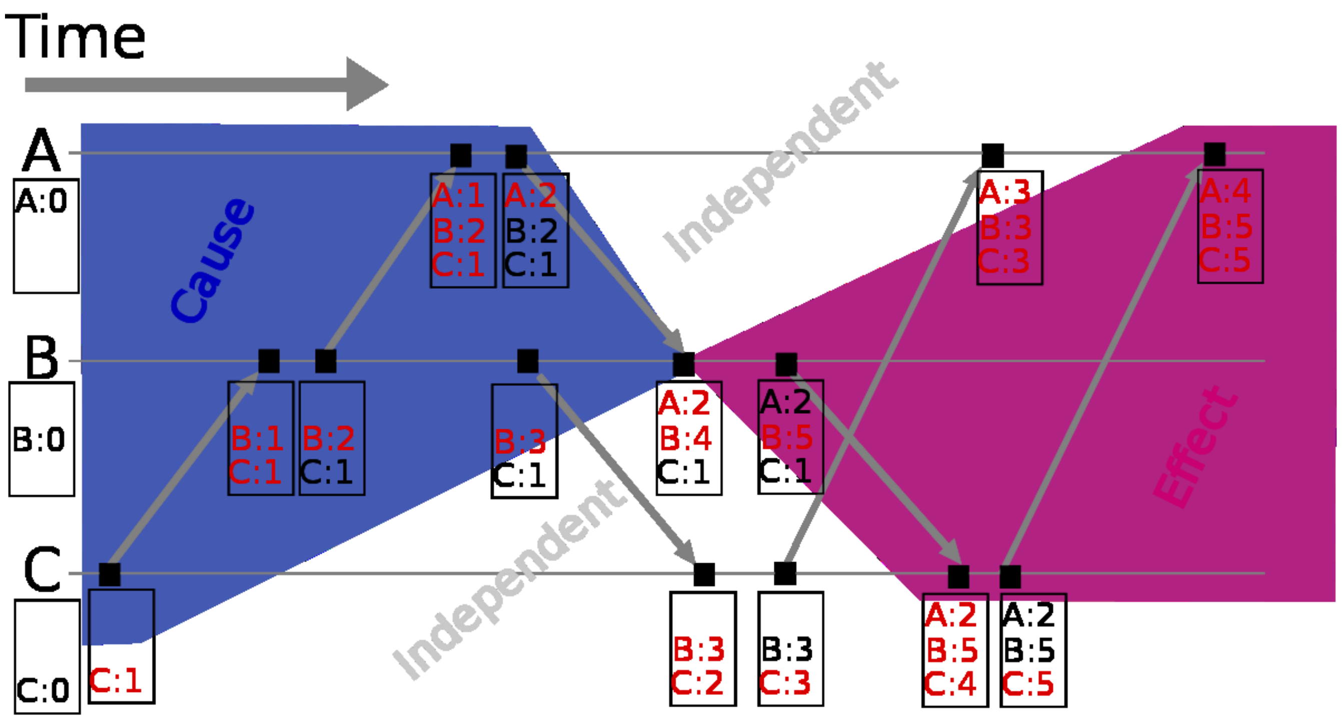


Inst	C_x	S_x	L_t	B_t	L_v	B_v
8	{a}	$\{t_2,u_1\}$	{ }	$\{t_2,u_1\}$	-	-
9	{ }	$\{t_2\}$				
10				$\{t_3,u_1\}$	{ }	$\{t_2,v_1\}$
11			{a}			
12	{a}	$\{t_3\}$				
13			{ }			
14					{a}	
15		$\{t_3,v_1\}$				
16					{ }	

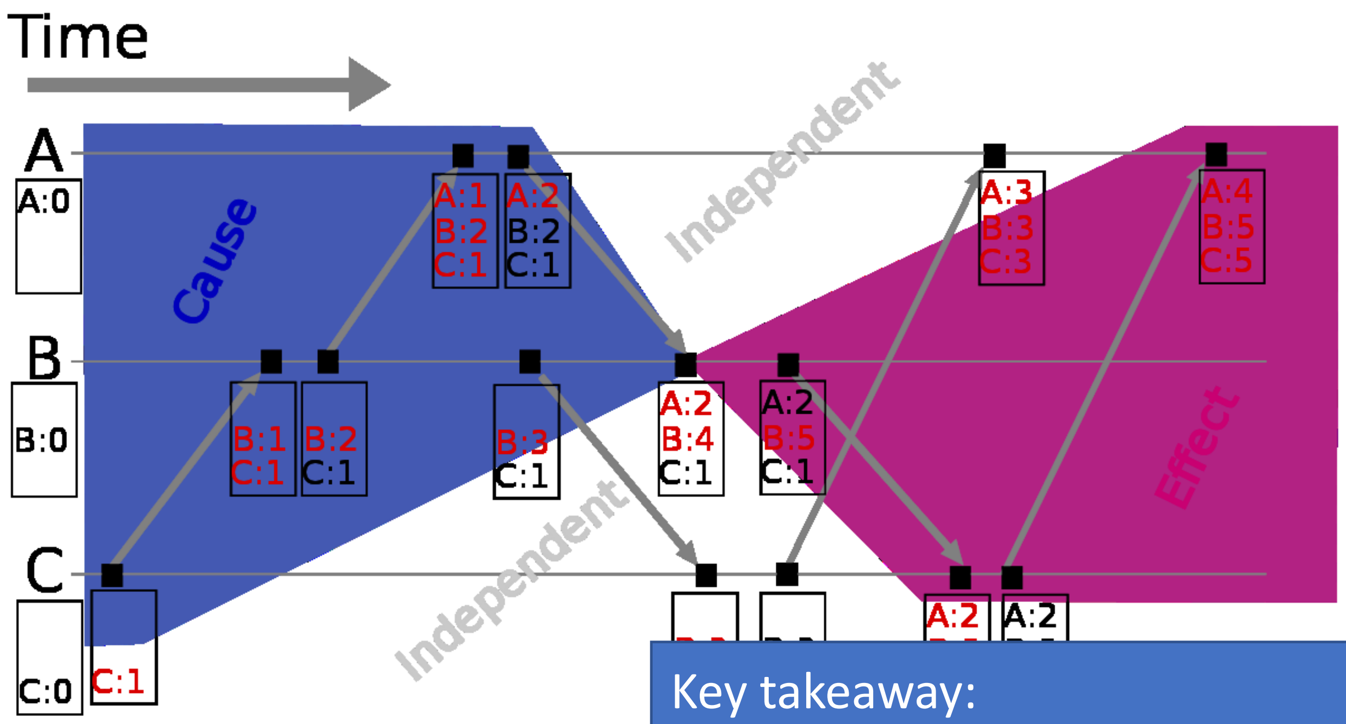
Only one thread!

Are we done?

Vector Clock Example



Vector Clock Example



Key takeaway:

- 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