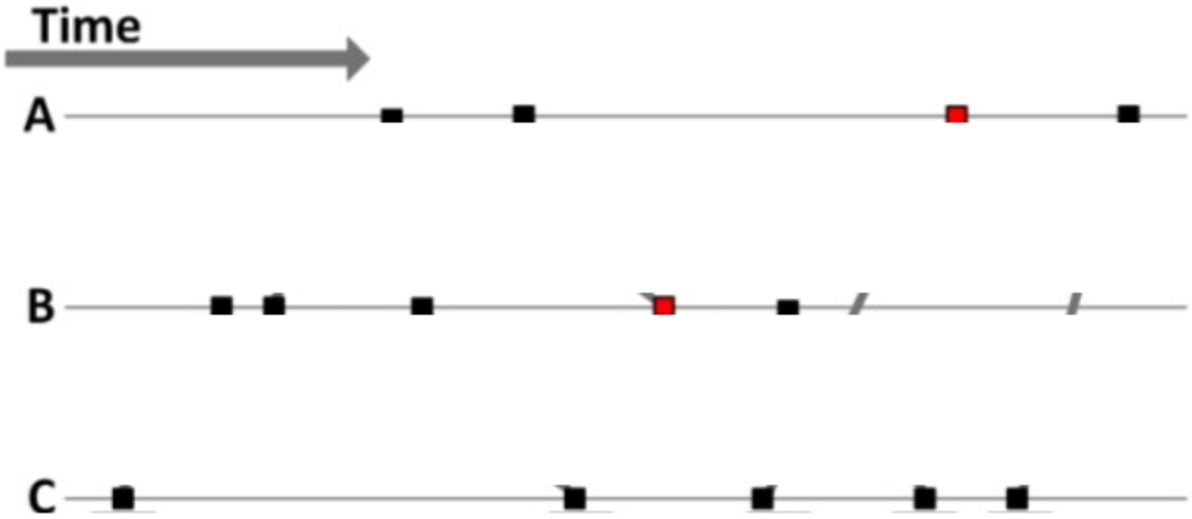
Concurrency Continued: RaceTrack

Emmett Witchel

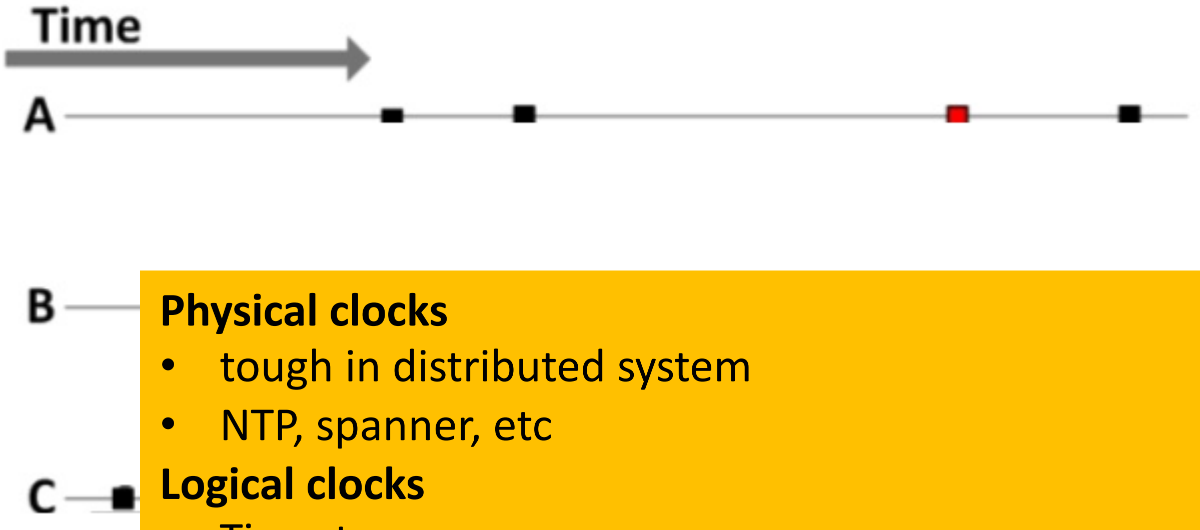
CS380L

Ordering and Causality



- A, B, C have local orders
- Why do we care about total order across all?
- Why is it hard to define such an order?
- What is causality?
- How does causality inform order?

Ordering and Causality



Physical clocks
- tough in distributed system
- NTP, spanner, etc

Logical clocks
- Timestamps
- conservatively respect causality
- A's timestamp is later than any event A knows about

Vector clocks
- O (N) timestamps that say what A knows about events elsewhere

Matrix clocks
- O(N^2) timestamps showing pairwise knowledge of event orders

- A, B, C have local orders
- Why do we care about
otal order across all?
hy is it hard to define
uch an order?
hat is causality?
ow does causality
form order?

Causality

- Need to maintain *causality*
  - If a -> b then a is casually related to b
  - *Causal delivery*:If send(m) -> send(n) => deliver(m) -> deliver(n)
  - Capture causal relationships between groups of processes
  - Need a time-stamping mechanism such that:
    - If $T(A) < T(B)$ then *A* should have causally preceded *B*

Logical Clocks

Logical Clocks

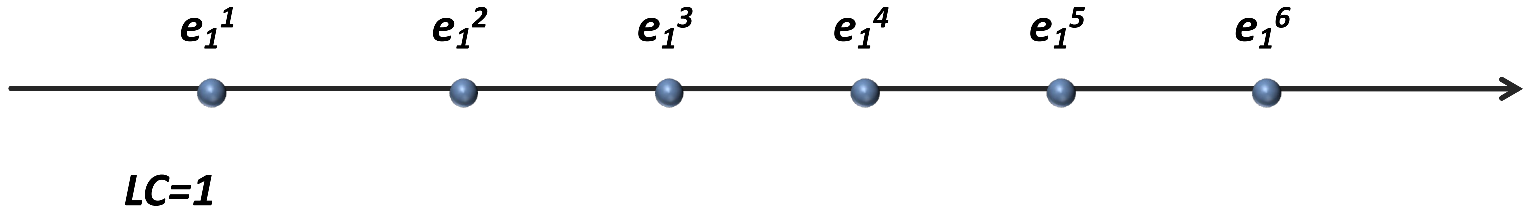
- Each process maintains a local value of a logical clock ***LC***

Logical Clocks

- Each process maintains a local value of a logical clock ***LC***

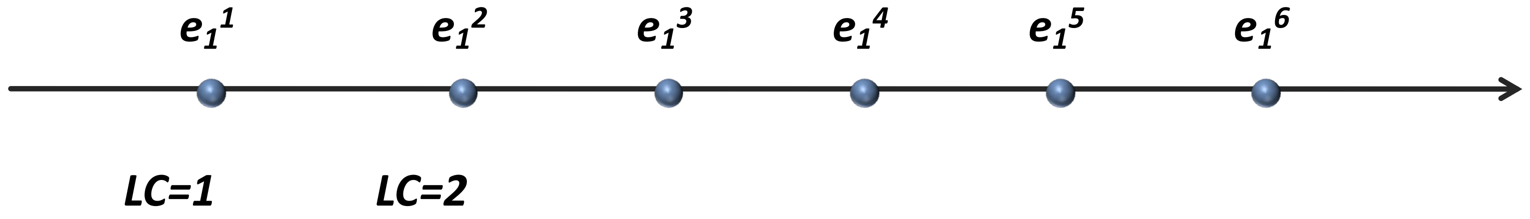
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***



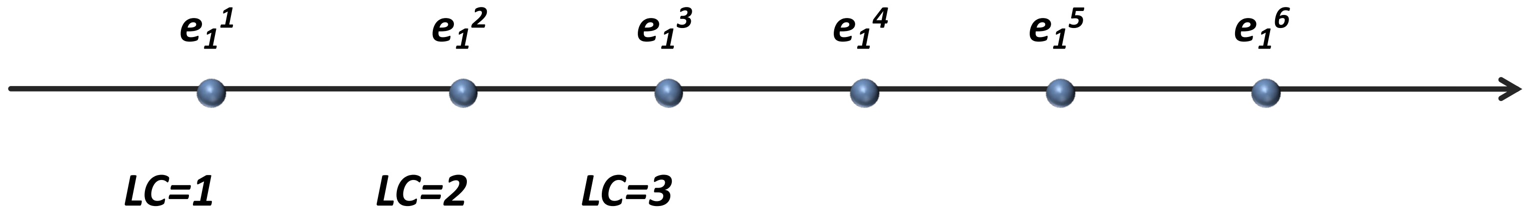
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***



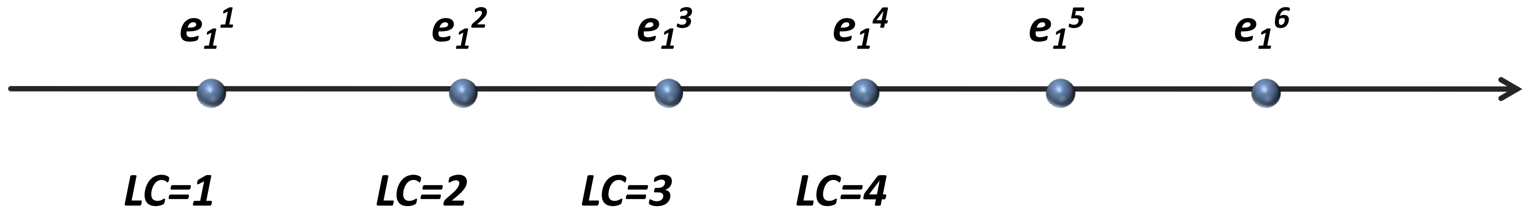
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***



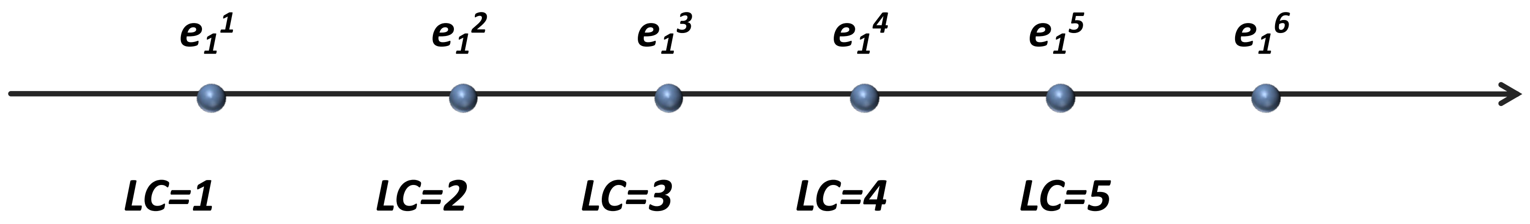
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***



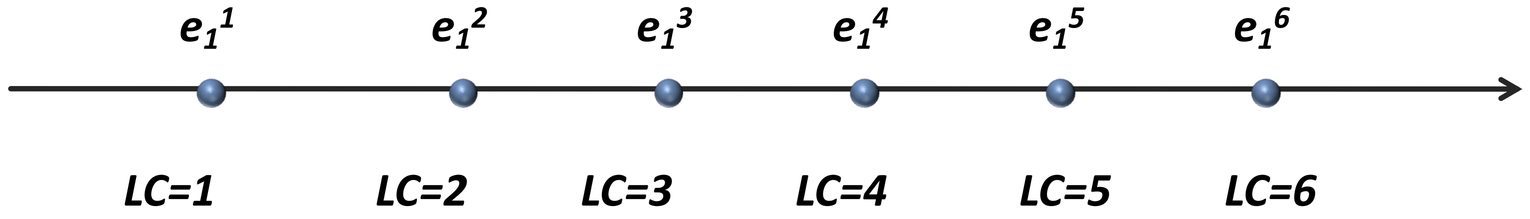
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***



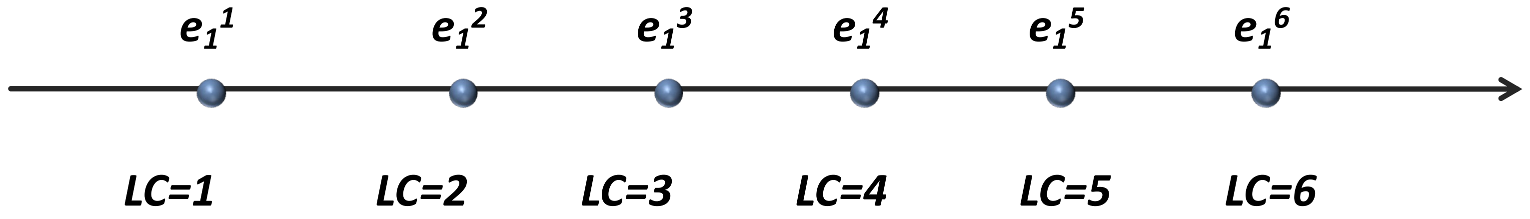
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***



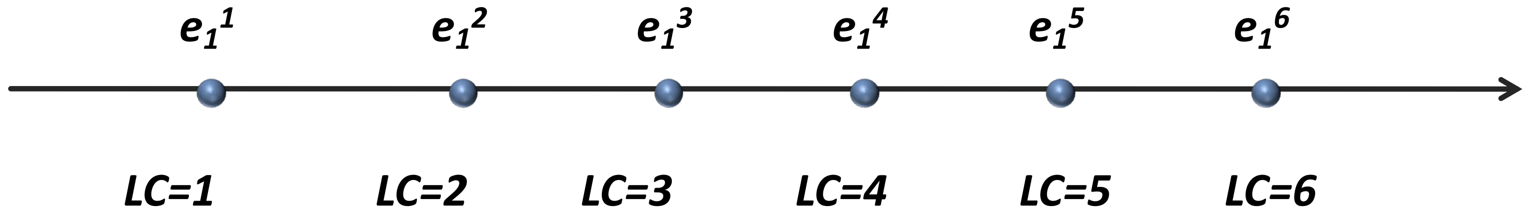
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of *p* → **how many events causally preceded the current event at *p***
  - (including the current event).
  - *Conservative approximation: why?*



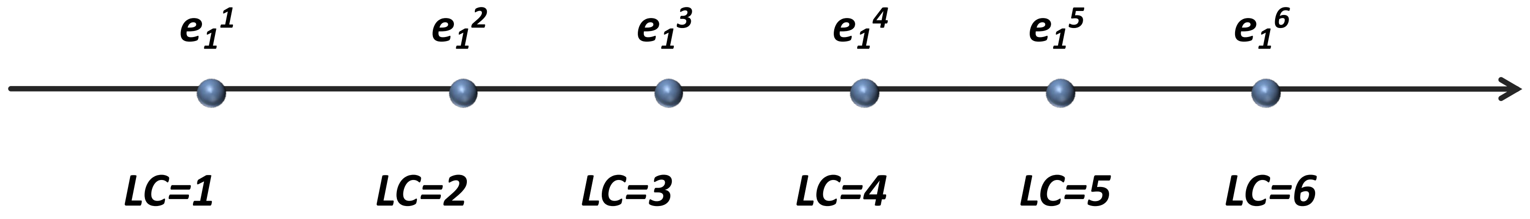
Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of *p* → **how many events causally preceded the current event at *p***
  - (including the current event).
  - *Conservative approximation: why?*
- $LC(e_i)$ – the logical clock value at process p_i at event e_i



Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of *p* → **how many events causally preceded the current event at *p***
  - (including the current event).
  - *Conservative approximation: why?*
- ***LC(e_i)*** – the logical clock value at process p_i at event e_i
- Each message m that is sent contains a timestamp TS(m)



Logical Clocks

- Each process maintains a local value of a logical clock ***LC***
- Logical clock of *p* → **how many events causally preceded the current event at *p***
  - (including the current event).
  - *Conservative approximation: why?*
- ***LC(e$_i$)*** – the logical clock value at process p_i at event e_i
- Each message m that is sent contains a timestamp TS(m)
- Update rules:
  - Send: TS(m) (logical clock value at process) sending event at the sending process
  - Recv: process receives message m, it updates *its* logical clock to: $max\{LC, TS(m)\} + 1$

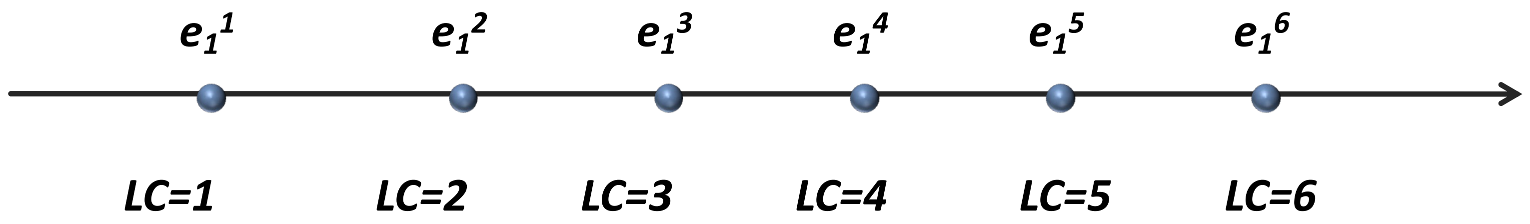
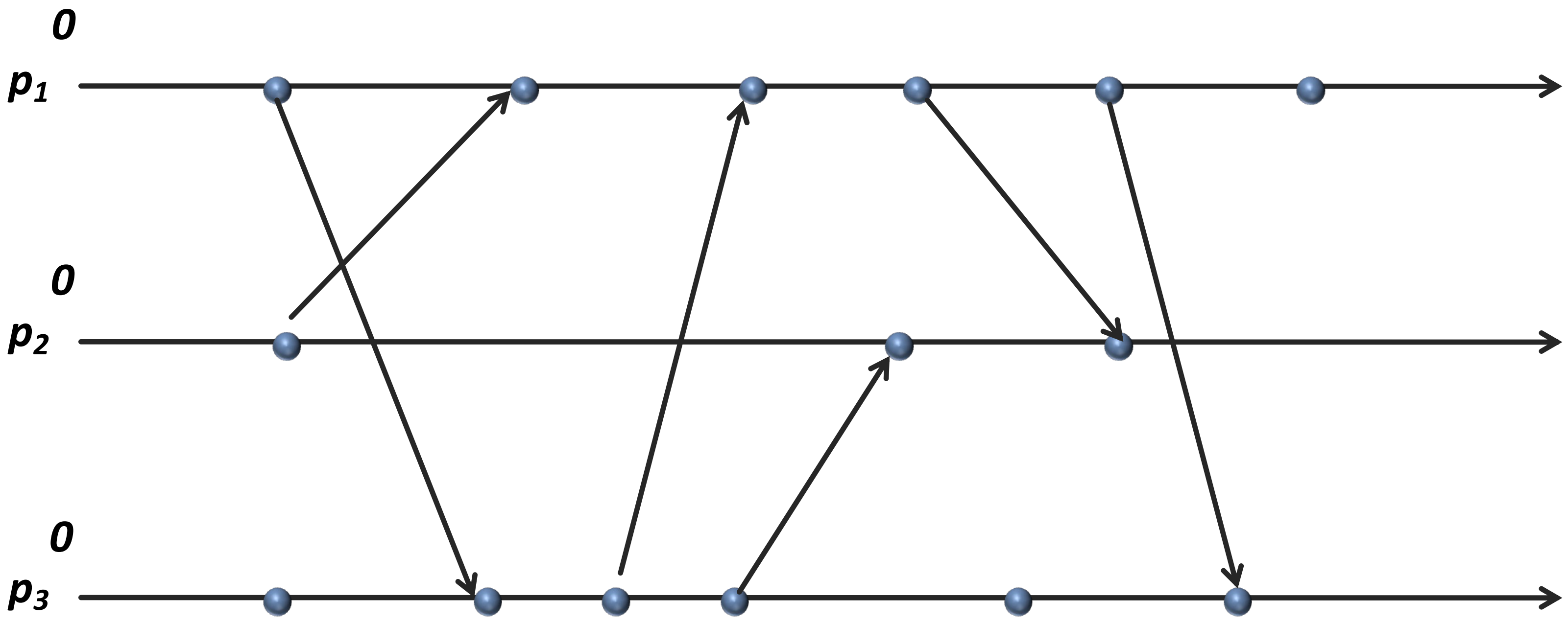


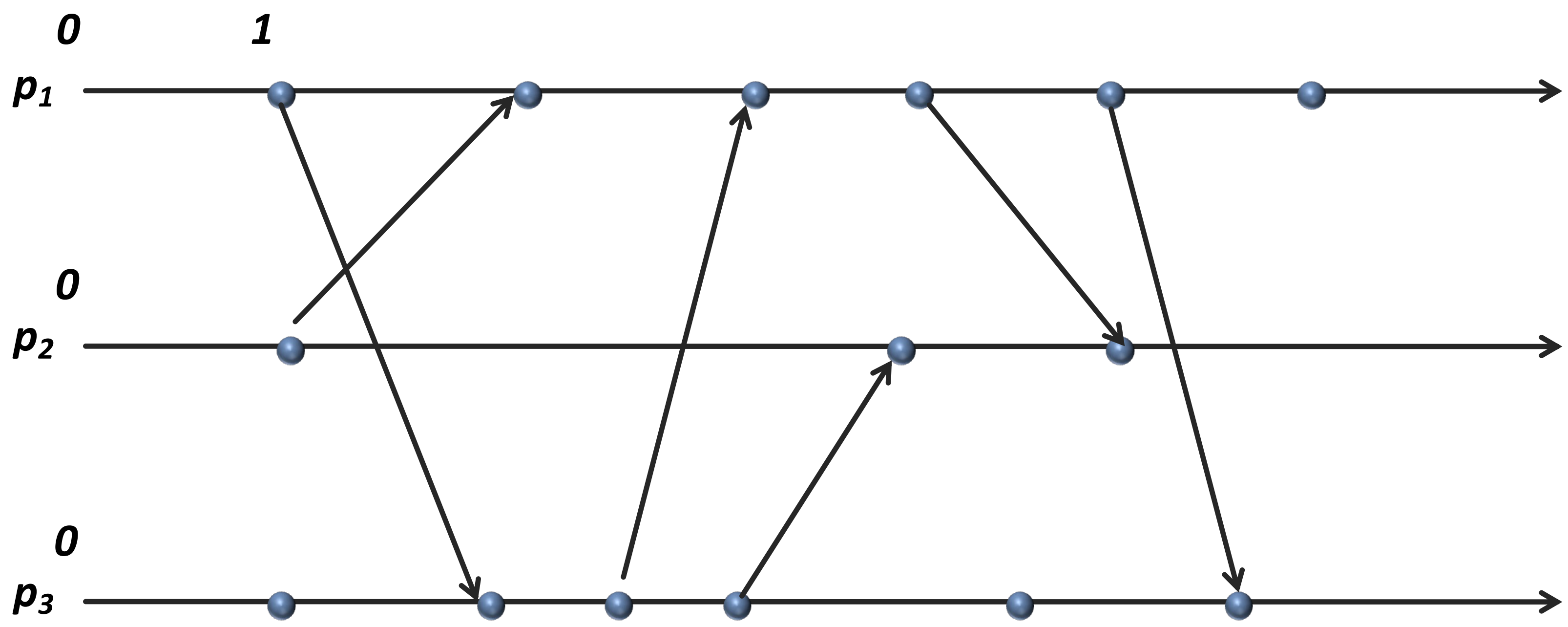
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

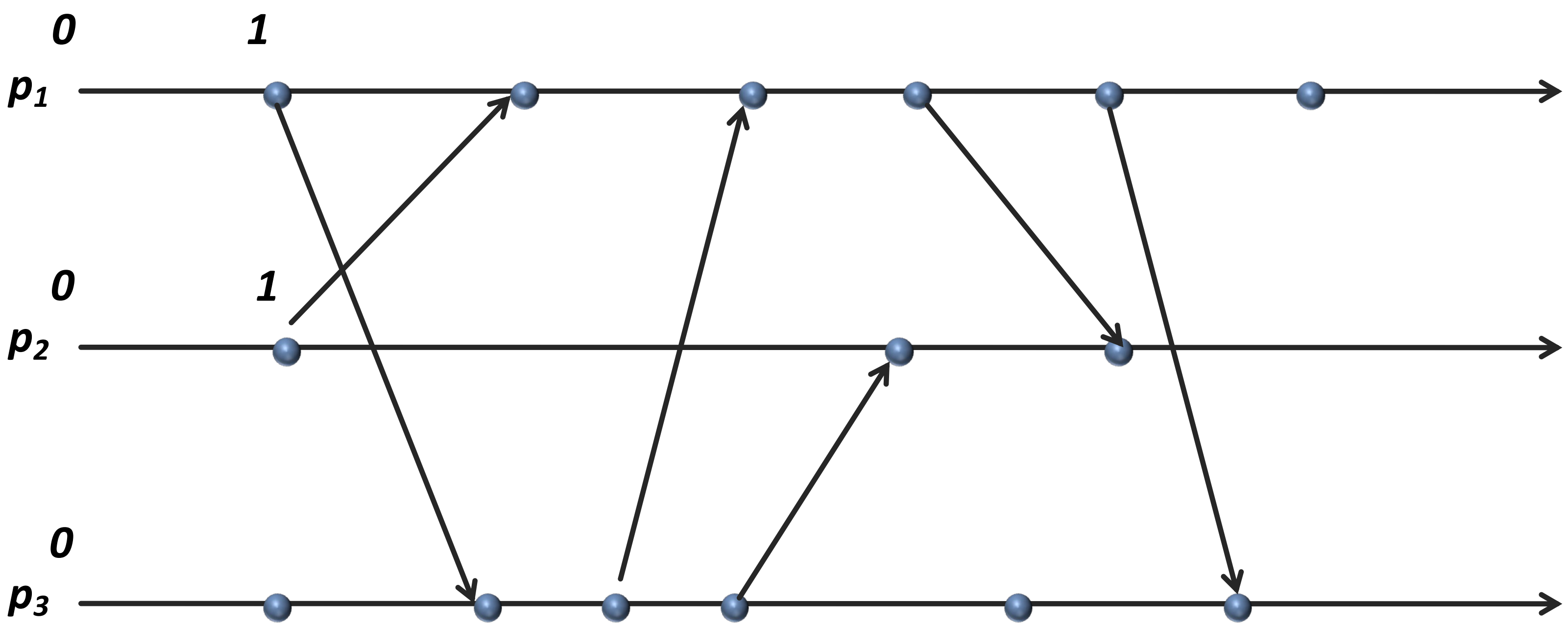
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

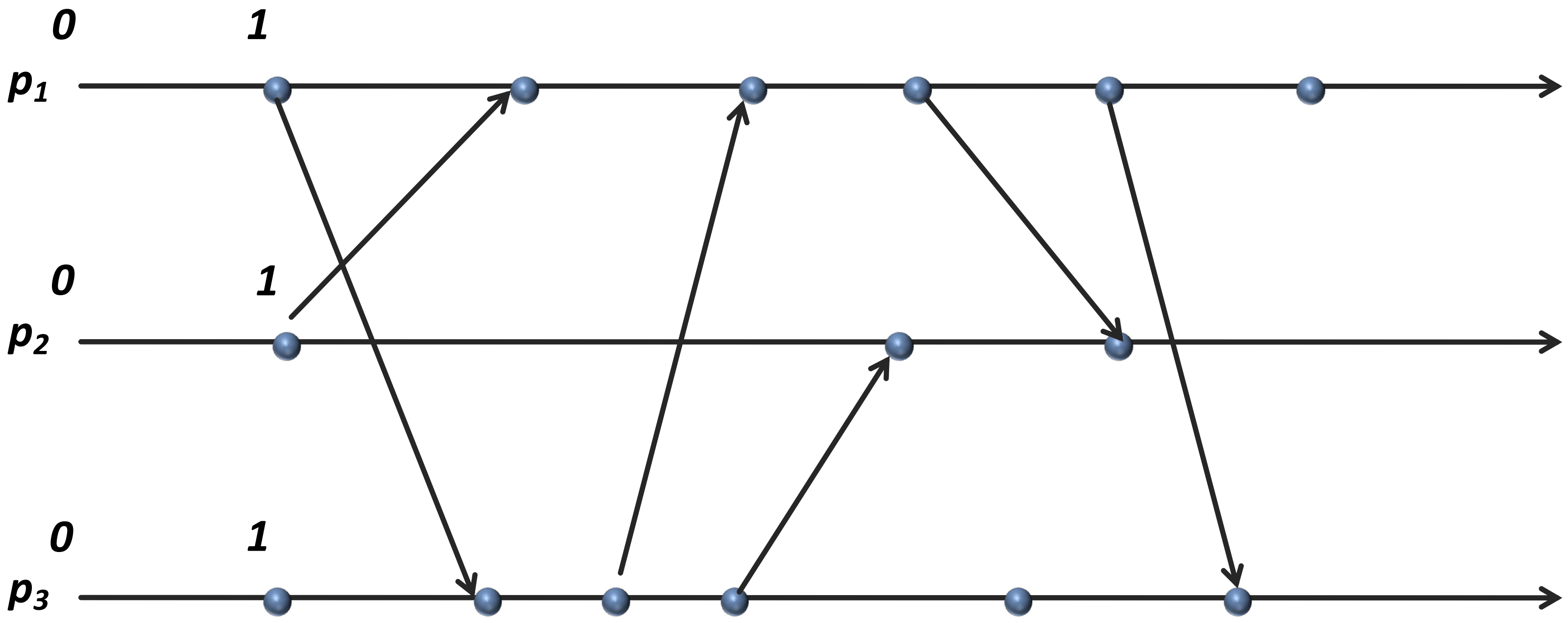
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

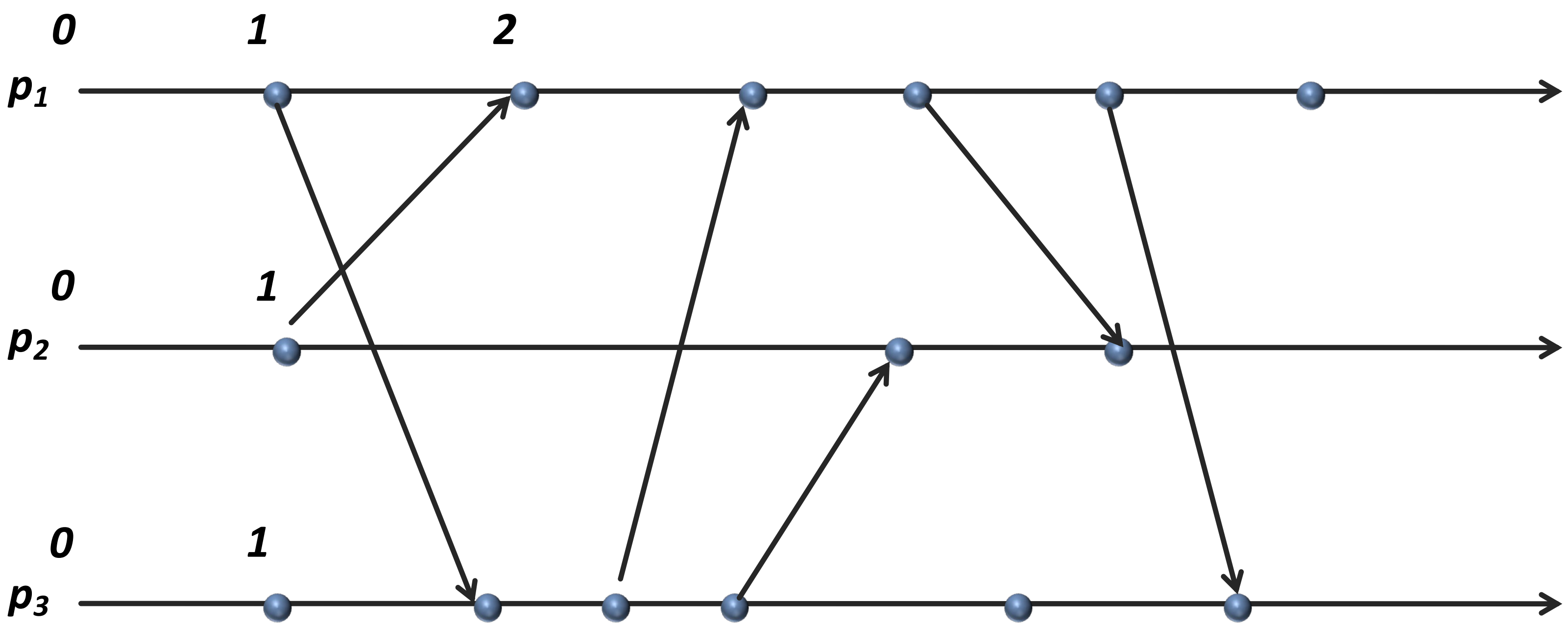
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

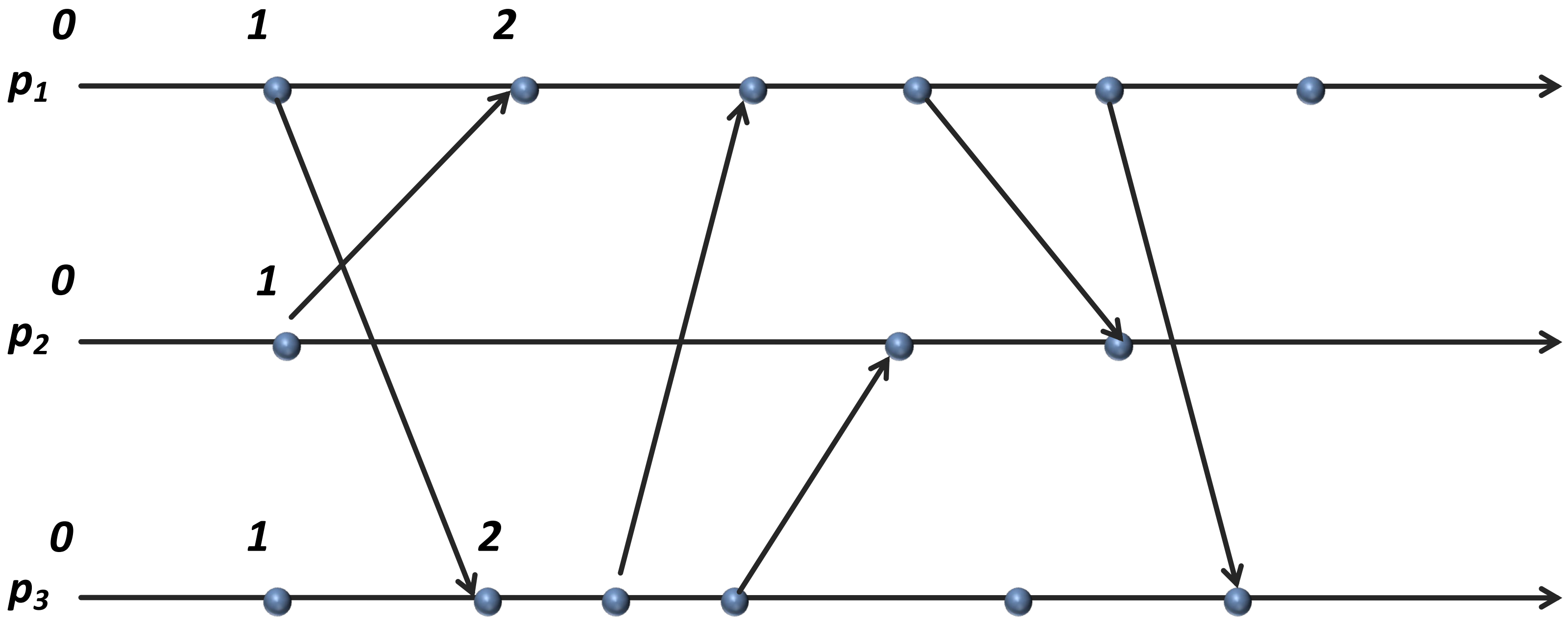
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

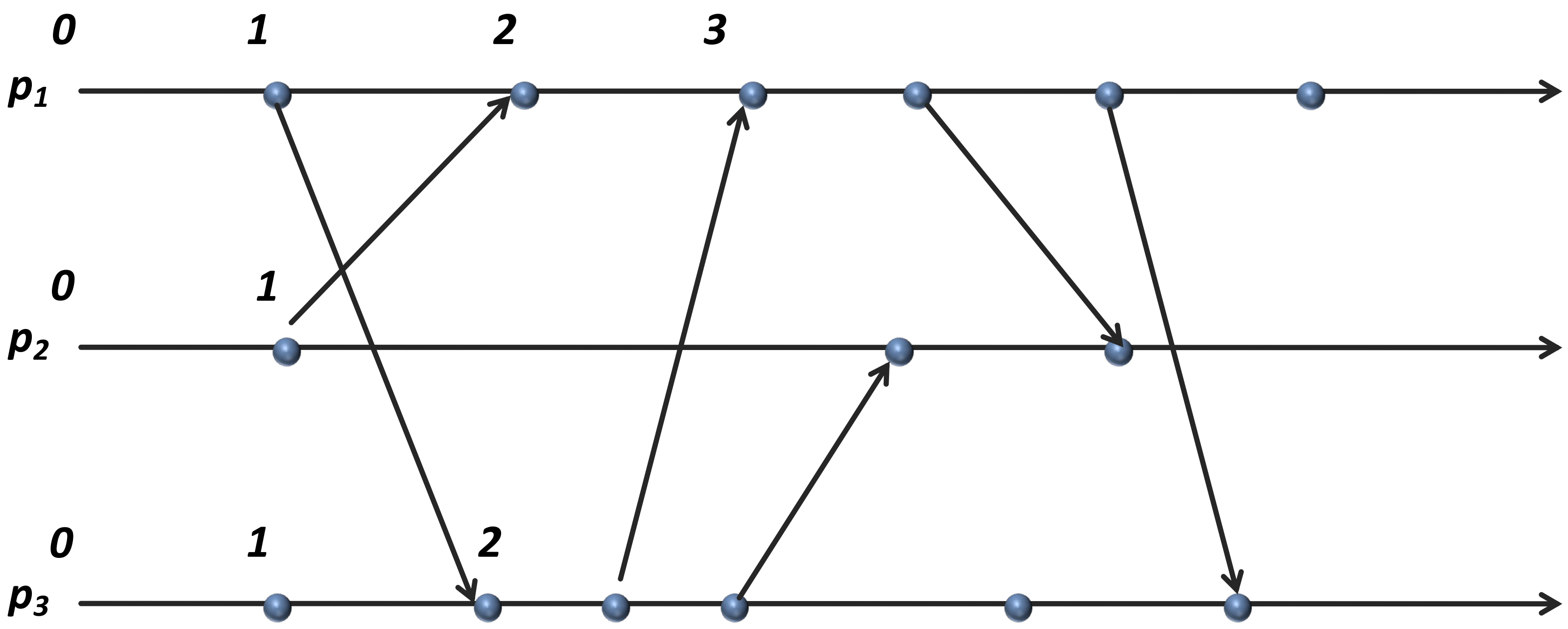
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

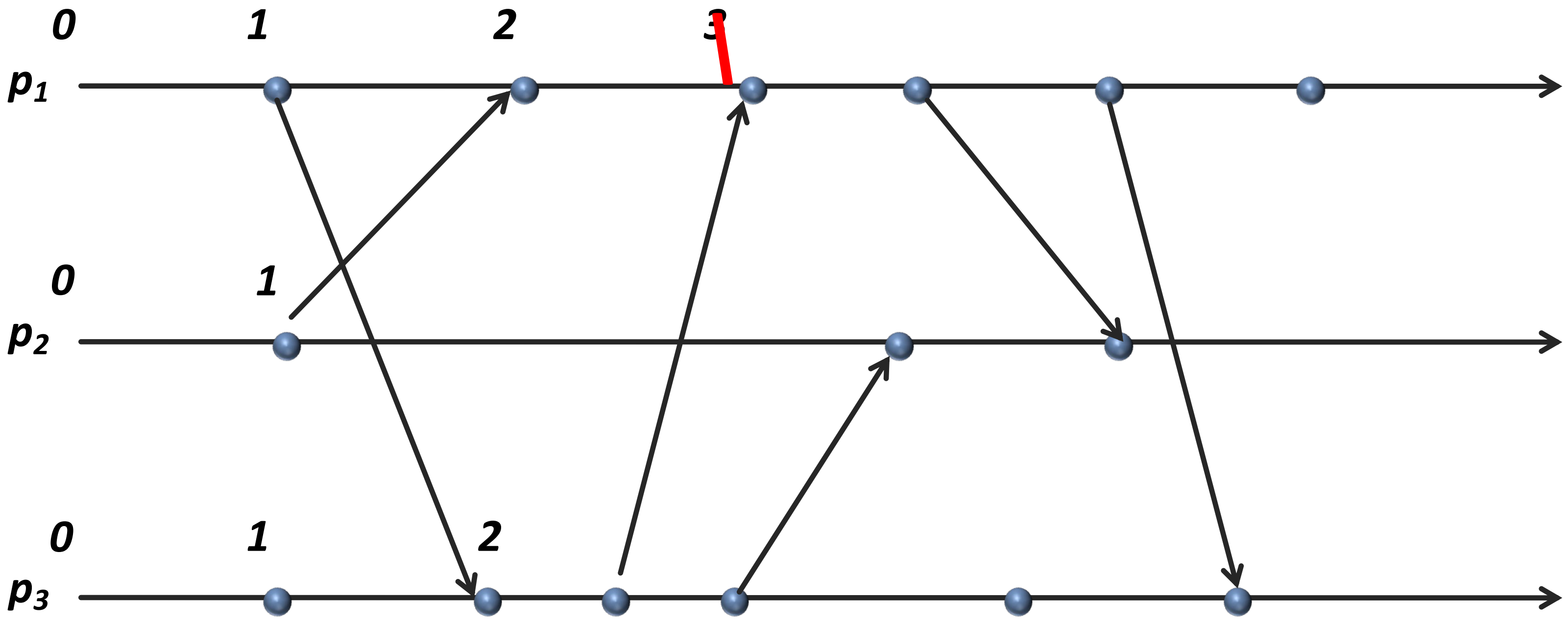
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

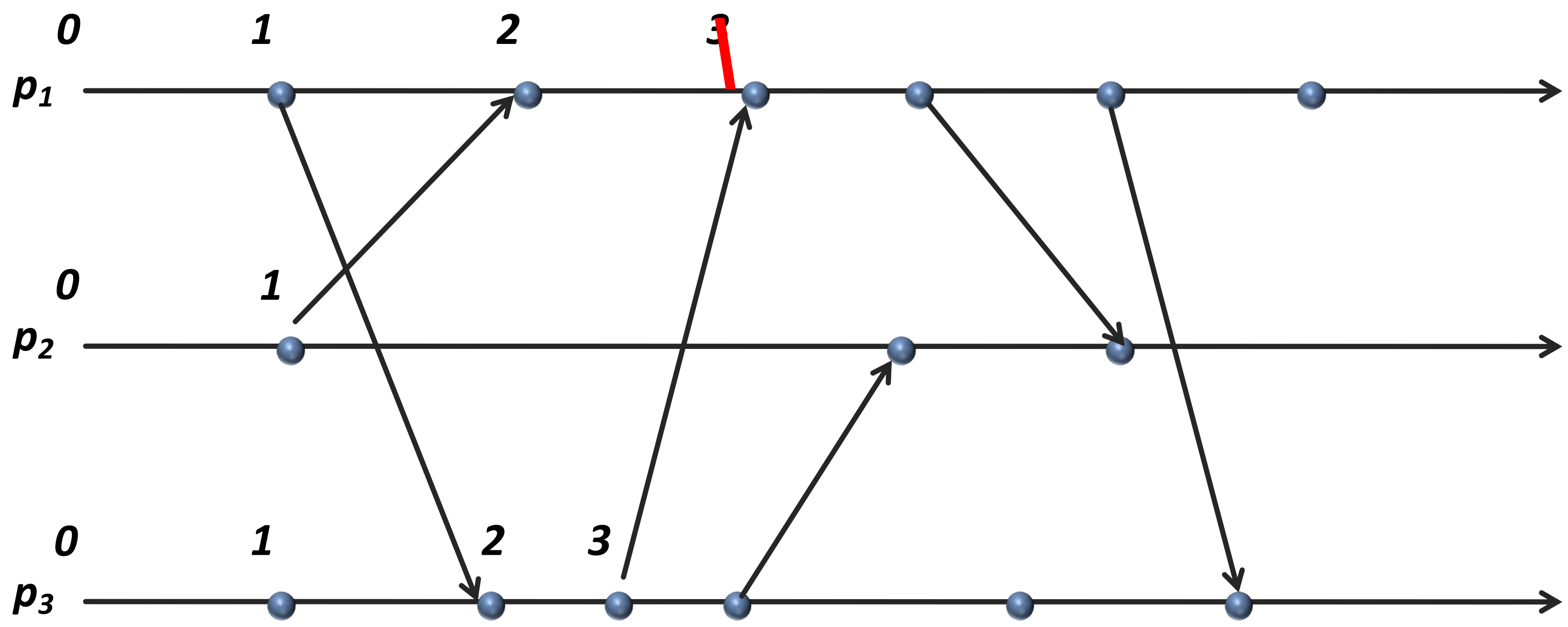
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

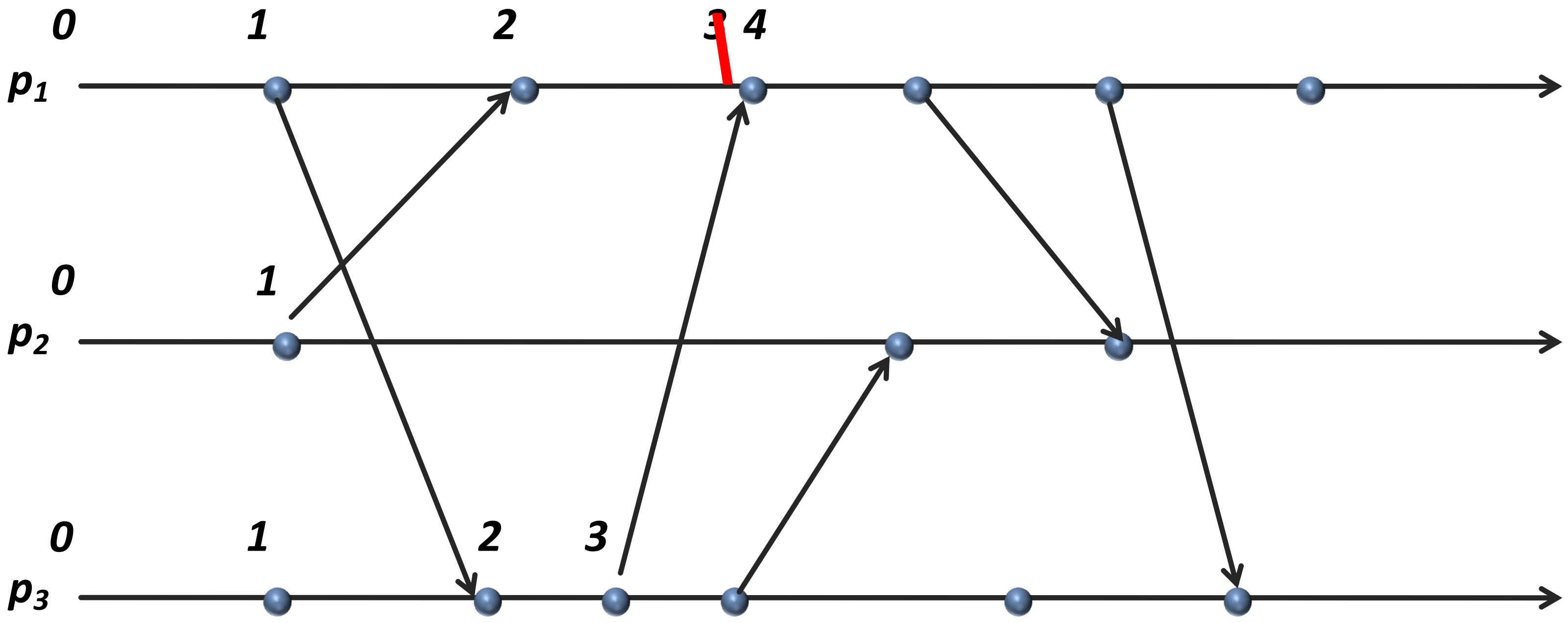
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

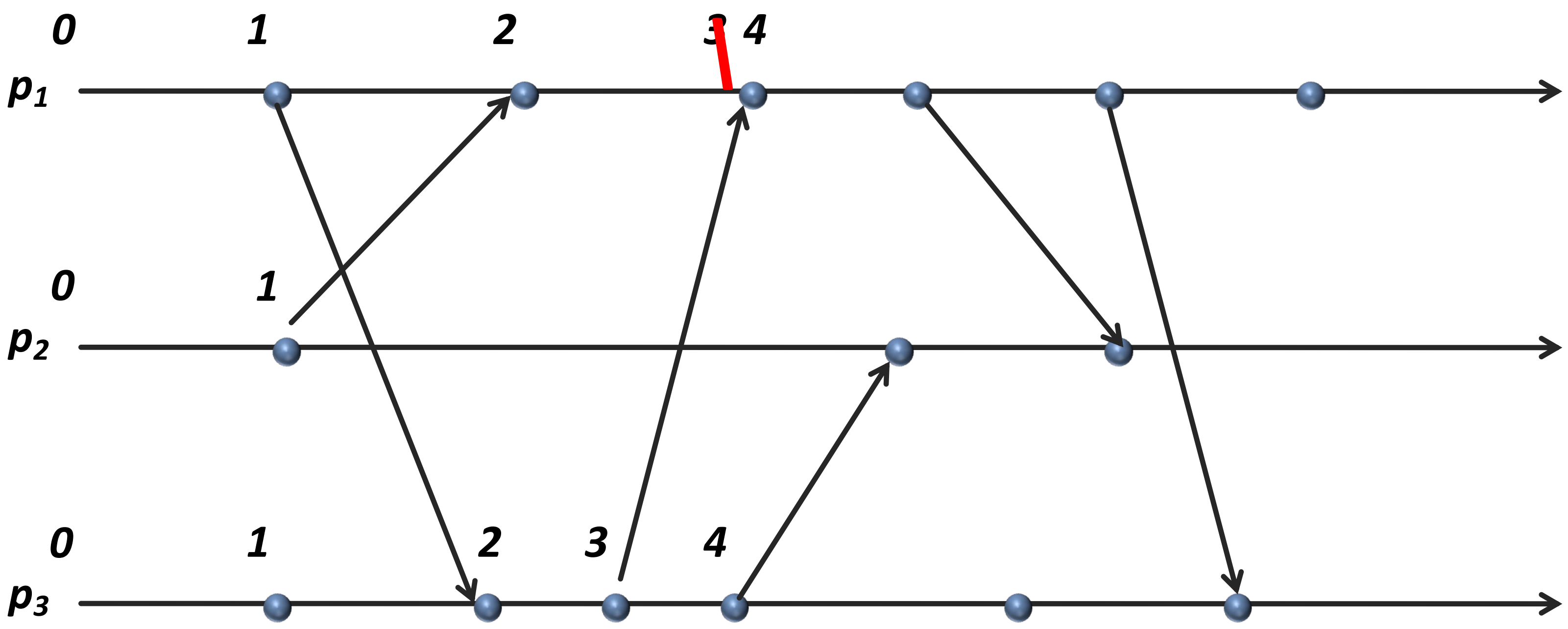
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

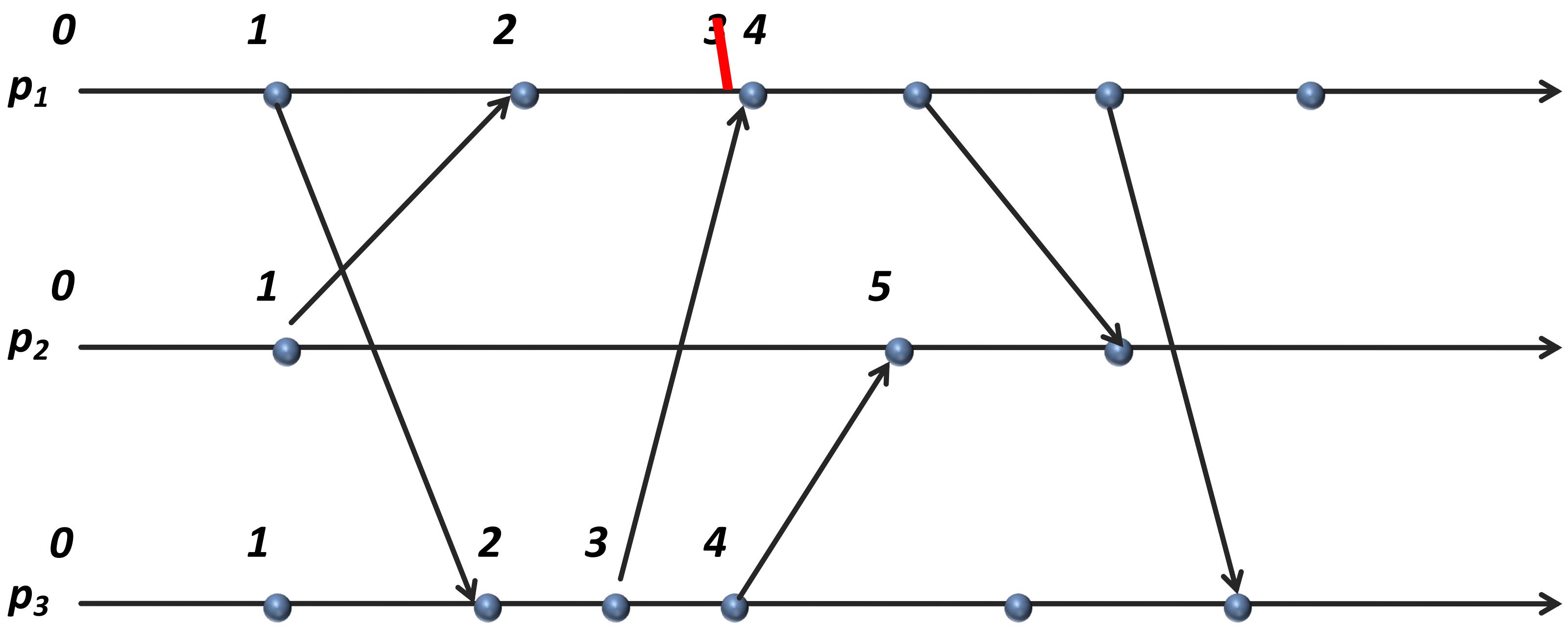
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

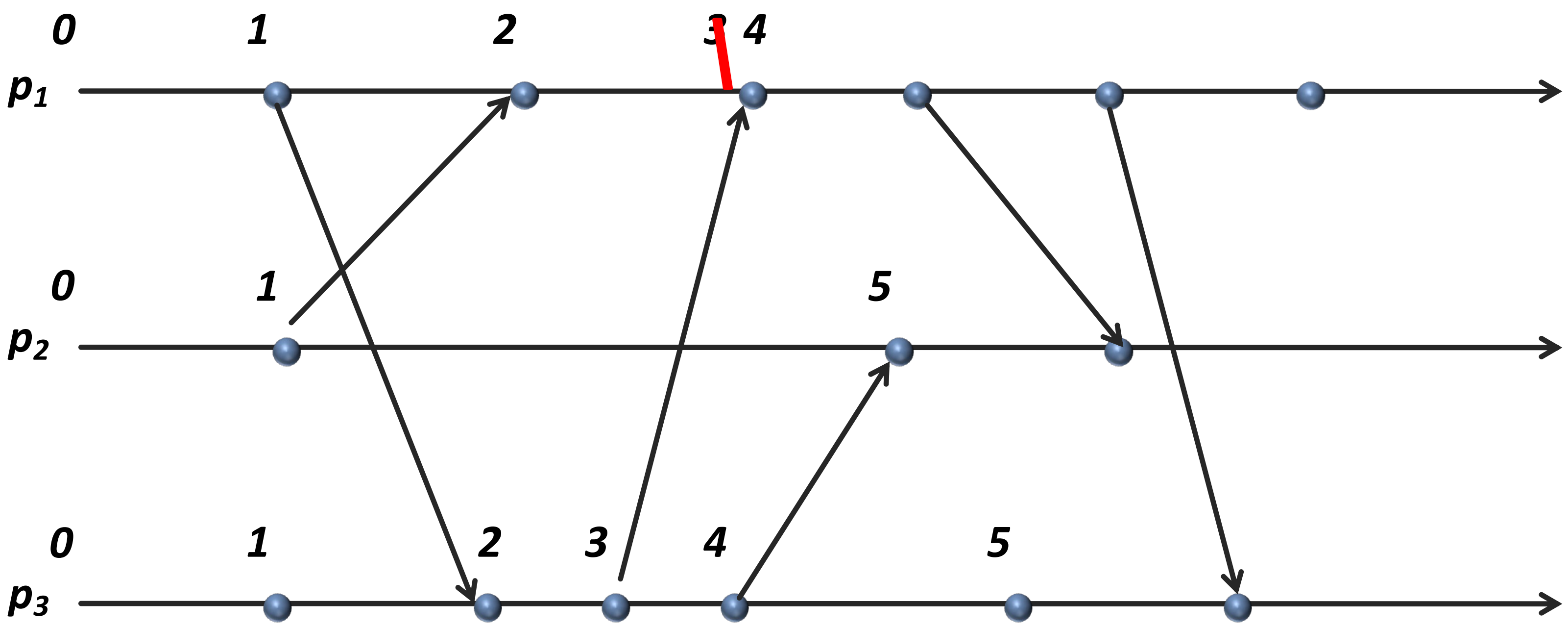
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

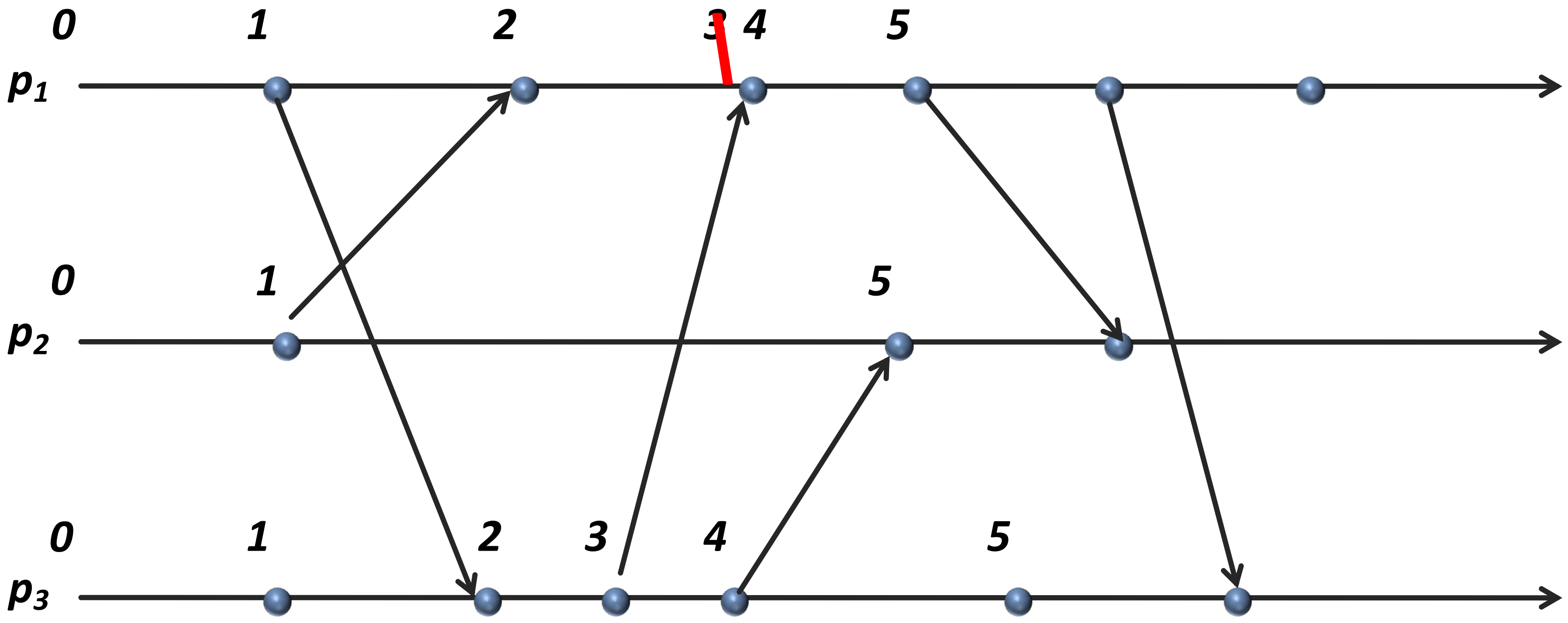
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

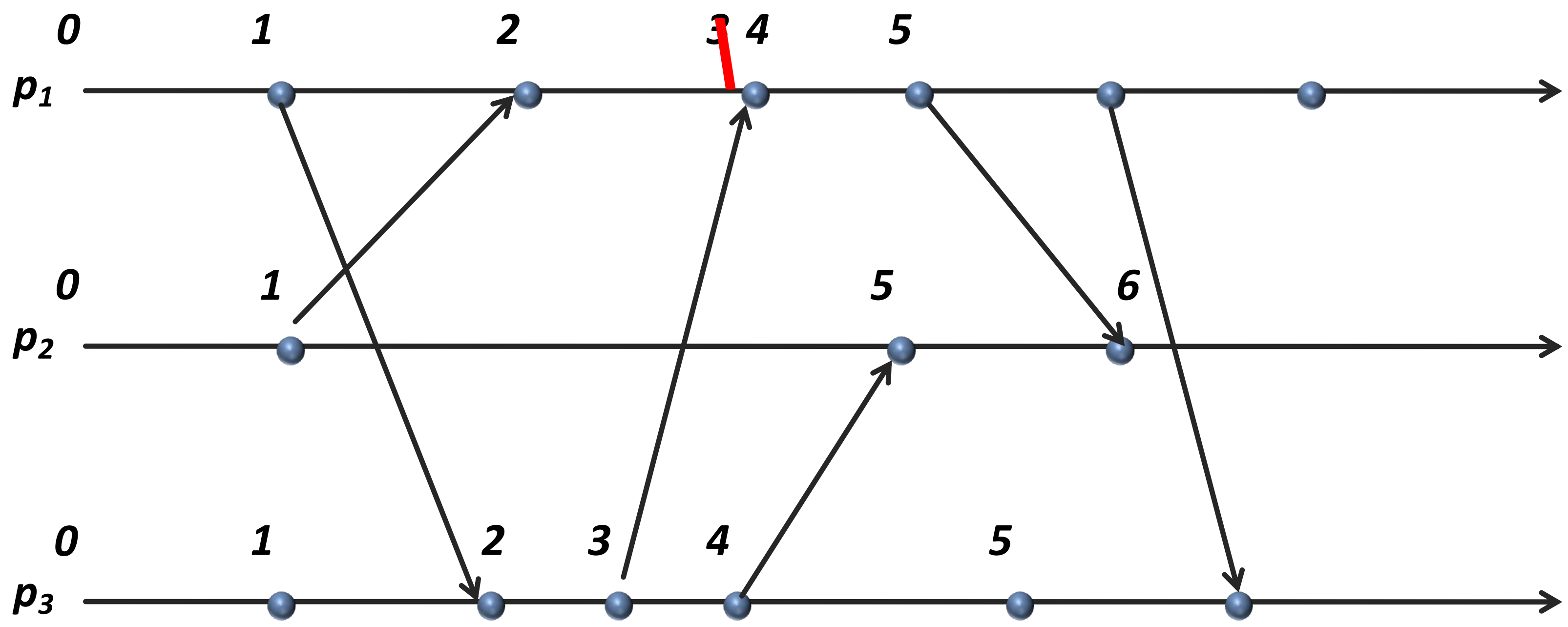
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

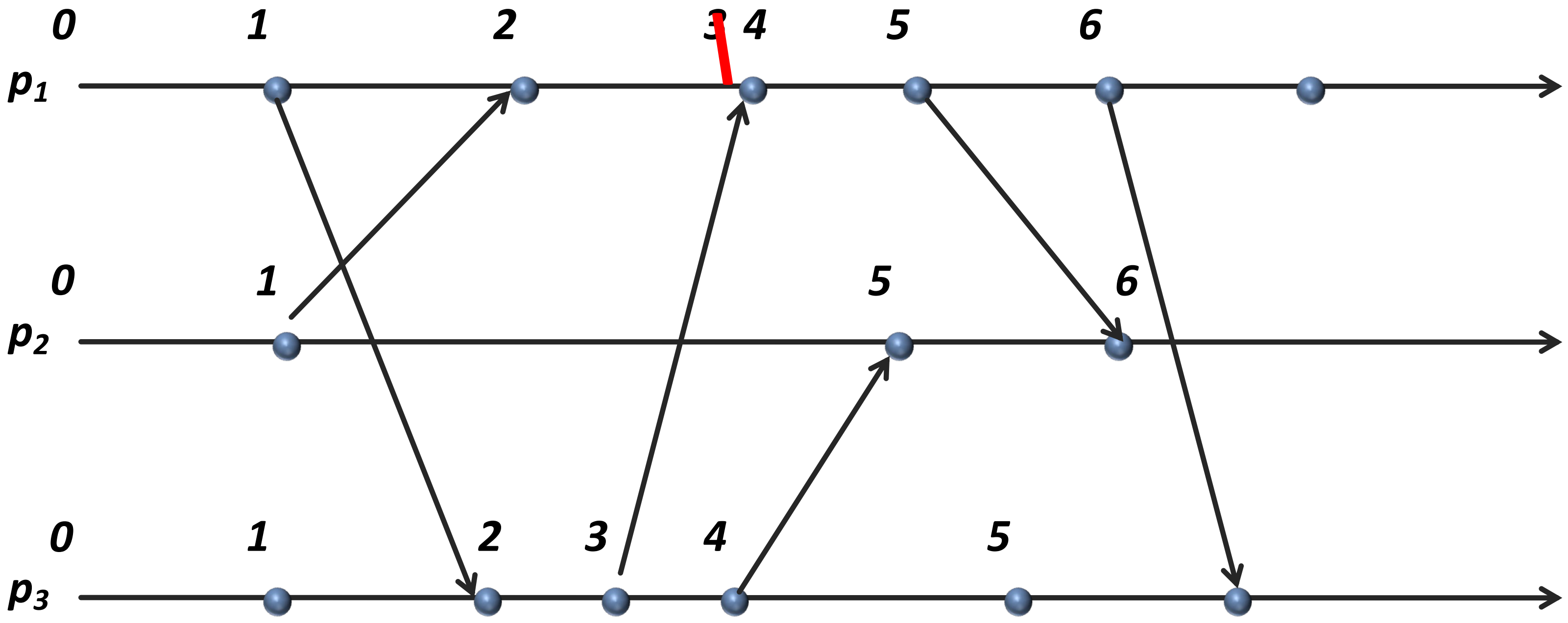
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

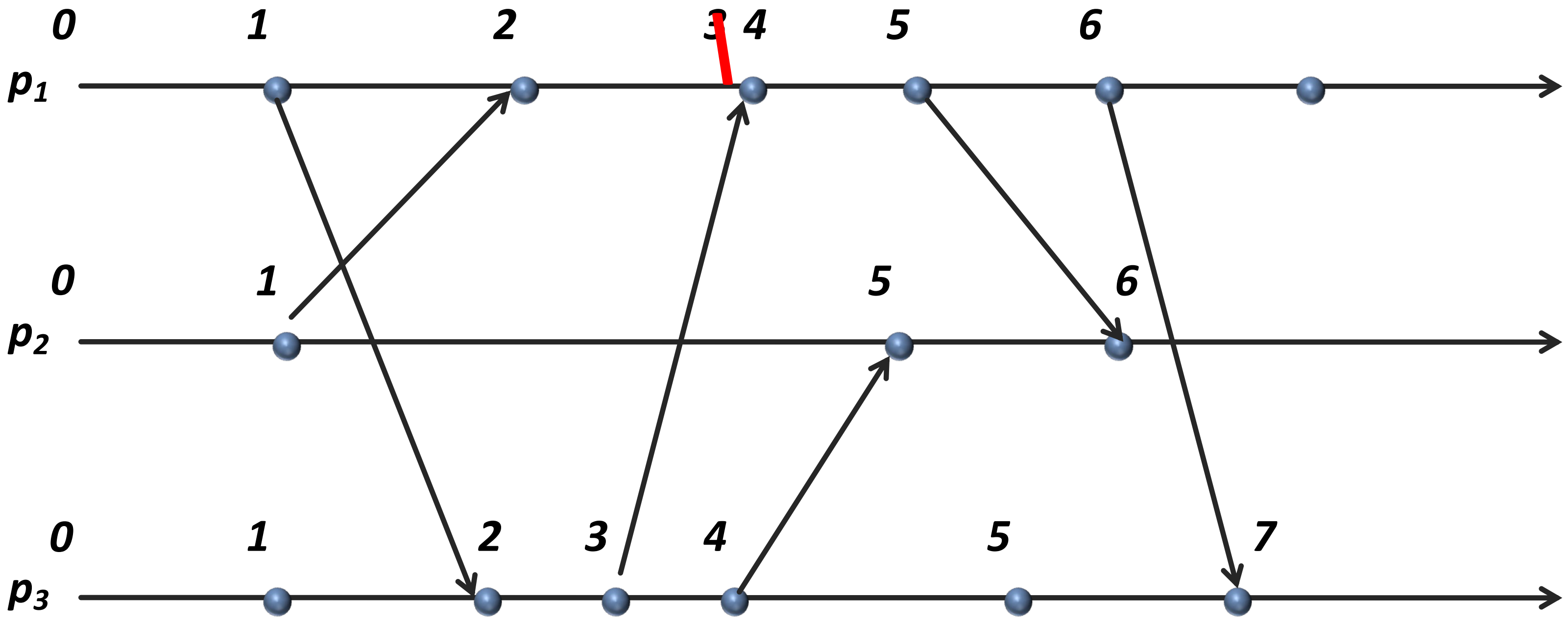
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

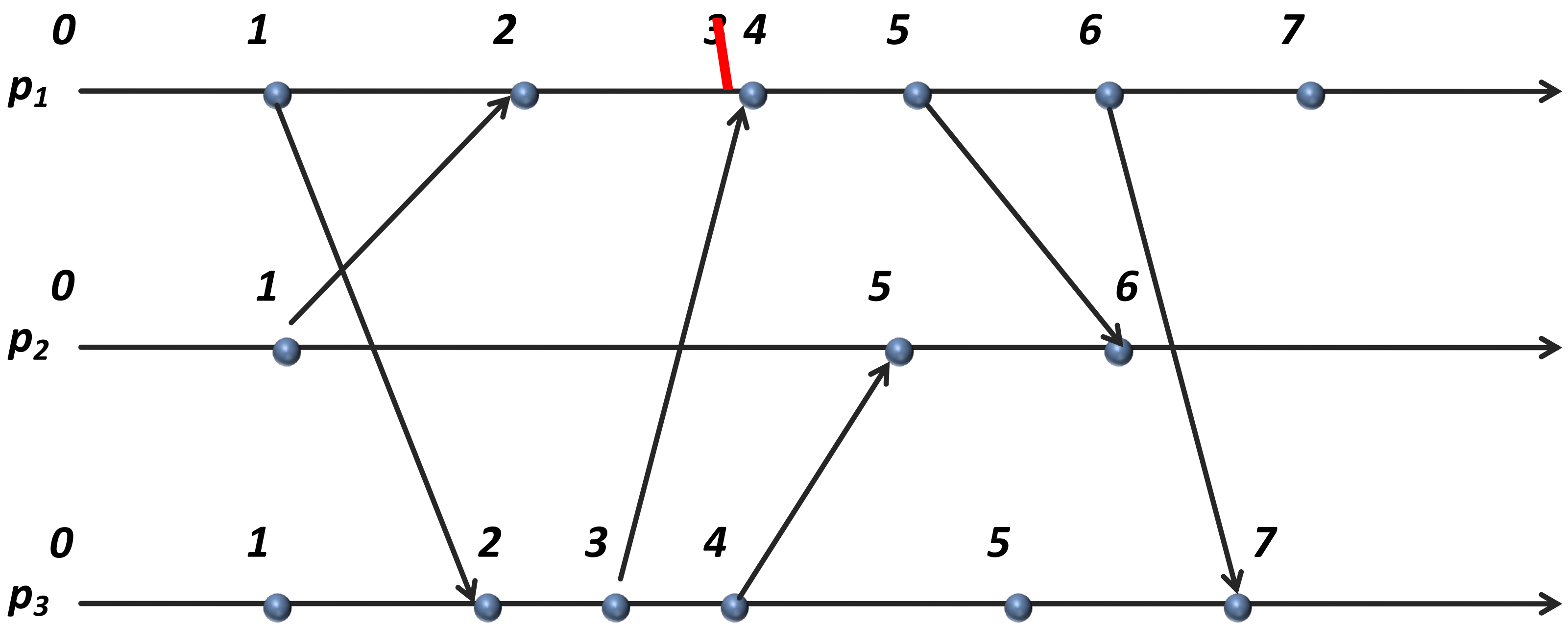
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

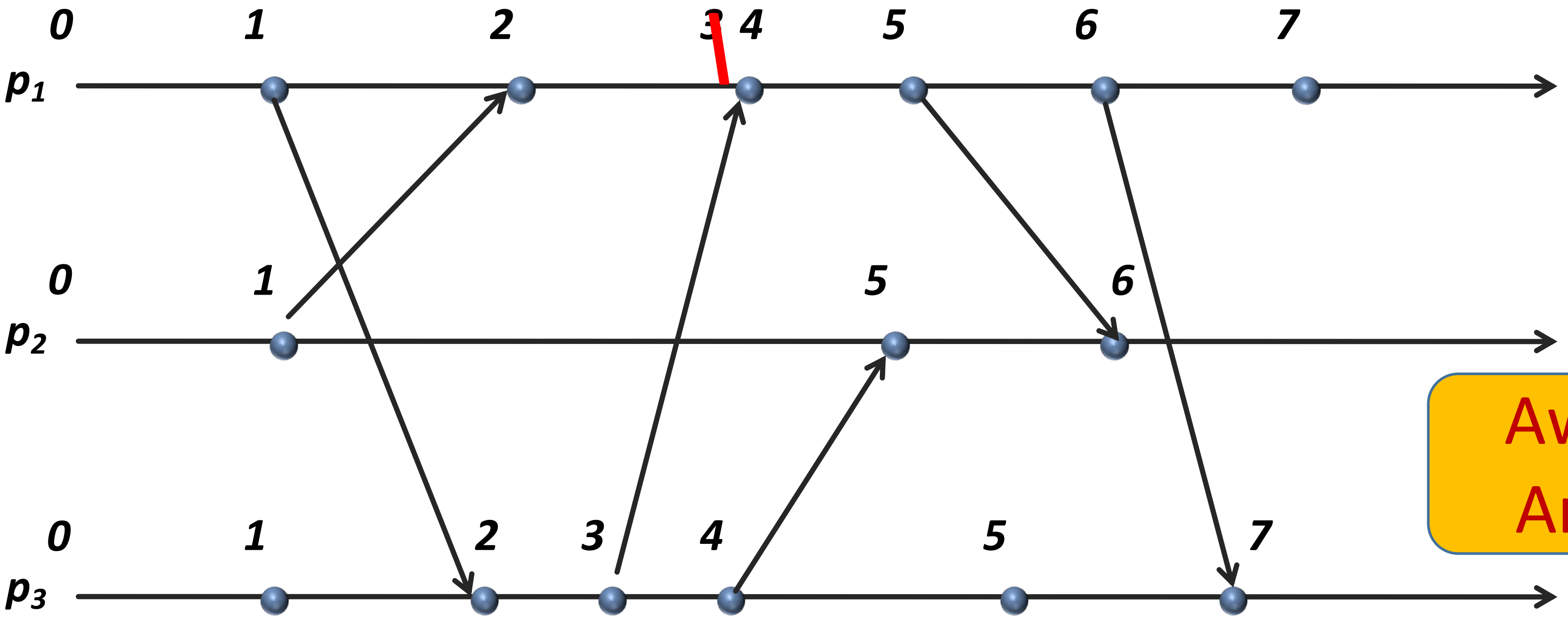
Illustration of a Logical Clock



Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

Illustration of a Logical Clock

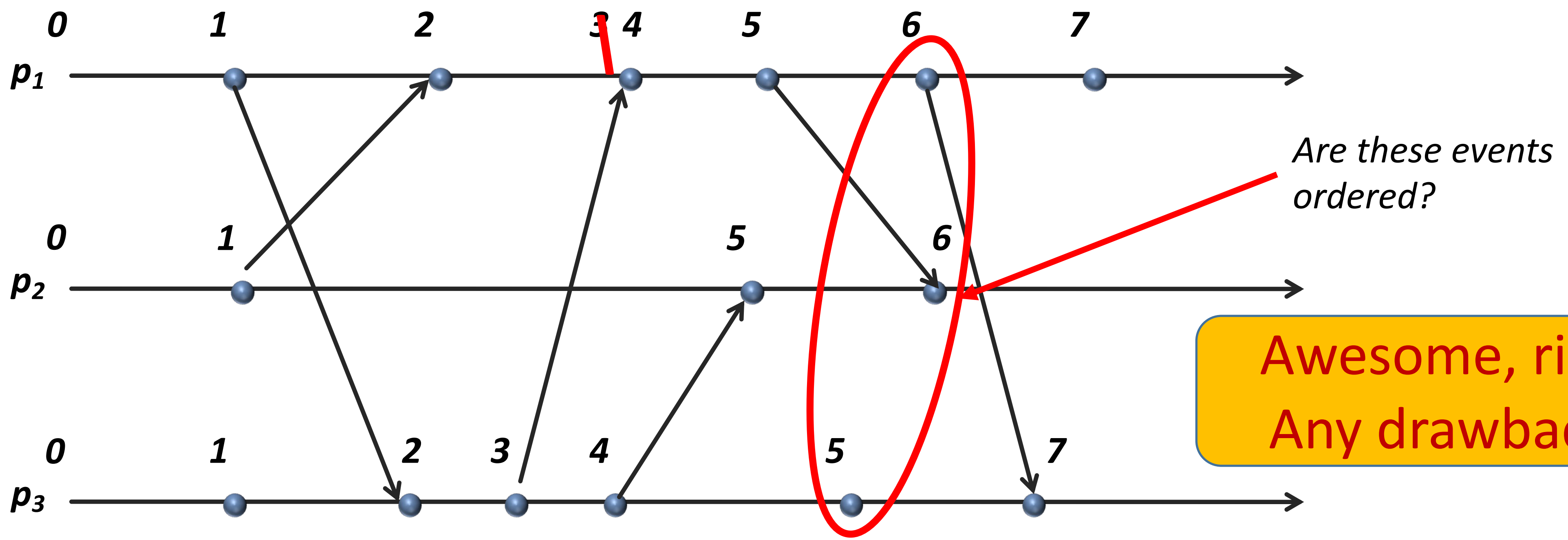


Awesome, right?
Any drawbacks?

Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

Illustration of a Logical Clock



Awesome, right?
Any drawbacks?

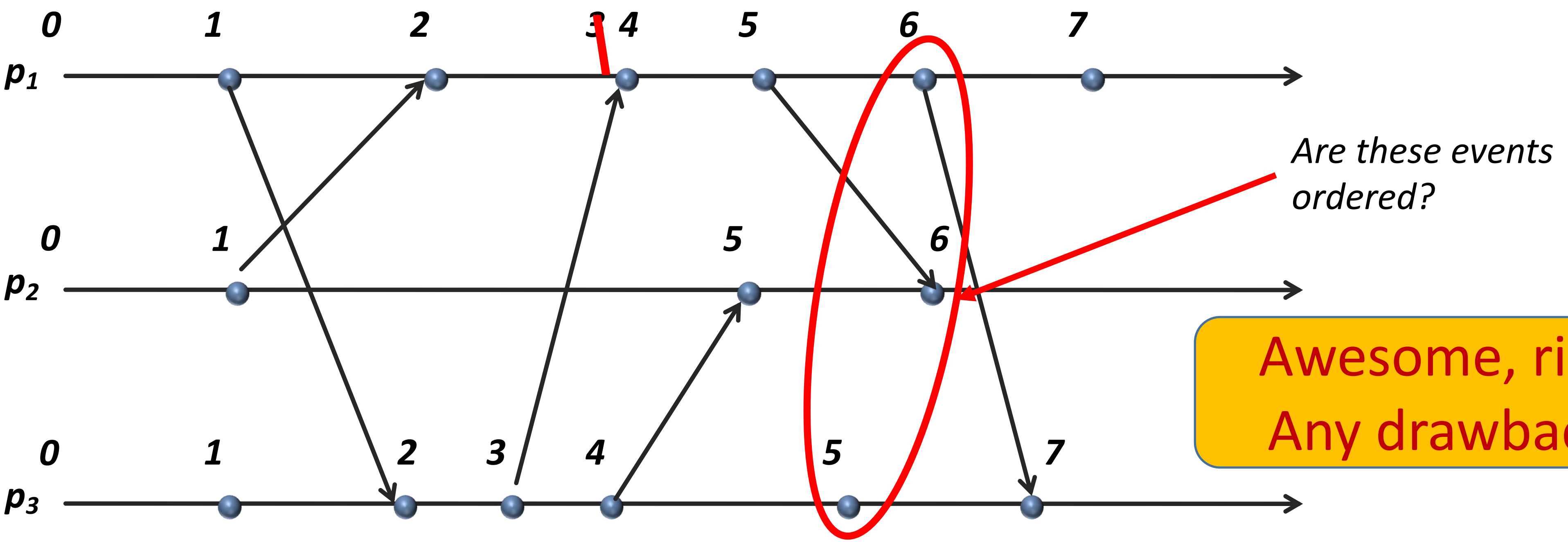
Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

Illustration of a Logical Clock

Guarantees: a < b → TS(a) < TS(b)

Does *not* guarantee: TS(a) < TS(b) → a < b



Awesome, right?
Any drawbacks?

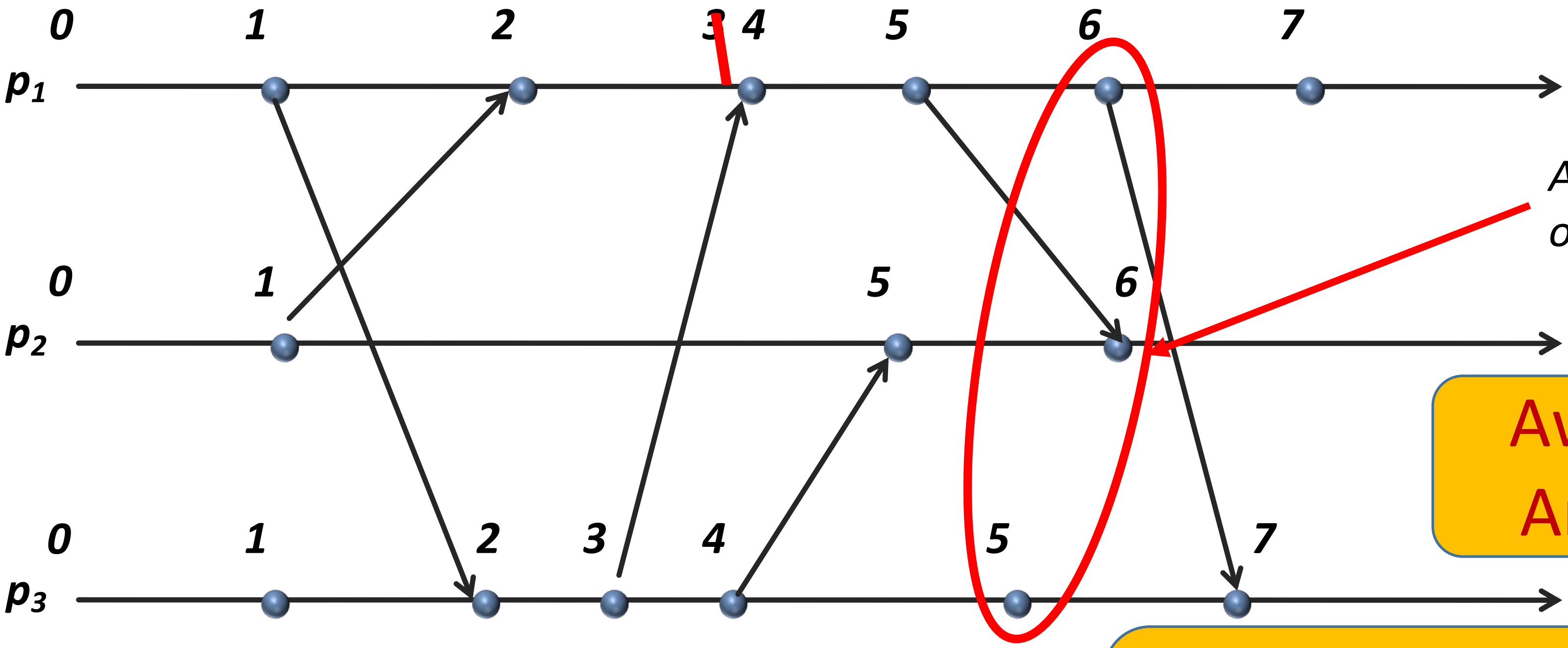
Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

Illustration of a Logical Clock

Guarantees: a < b →TS(a) < TS(b)

Does *not* guarantee: TS(a) < TS(b) → a < b



Are these events ordered?

Awesome, right?
Any drawbacks?

Not strong enough…so
Replace Single Logical value with Vector!

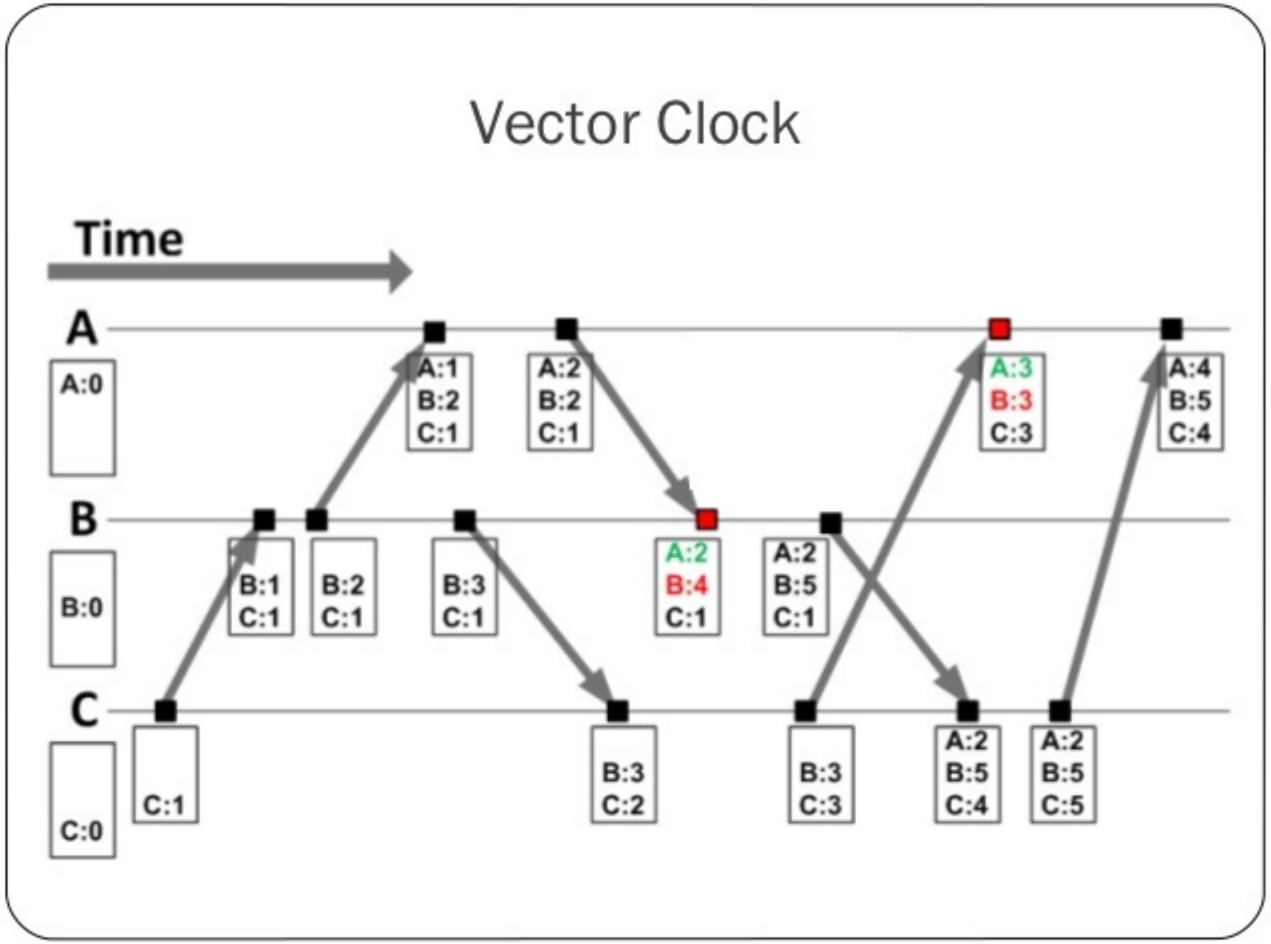
Update Rules:

- on-send: LC++
- on-recv: LC = max(LC, TS(m))+1

Vector Clocks

- 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 vector clocks:
  - 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
- *Exercise:* prove that if *V(A)<V(B),* then *A* causally precedes *B* and the other way around.
  - *Under what conditions are V(A) and V(B) not ordered (concurrent)?*

Vector Clock Example



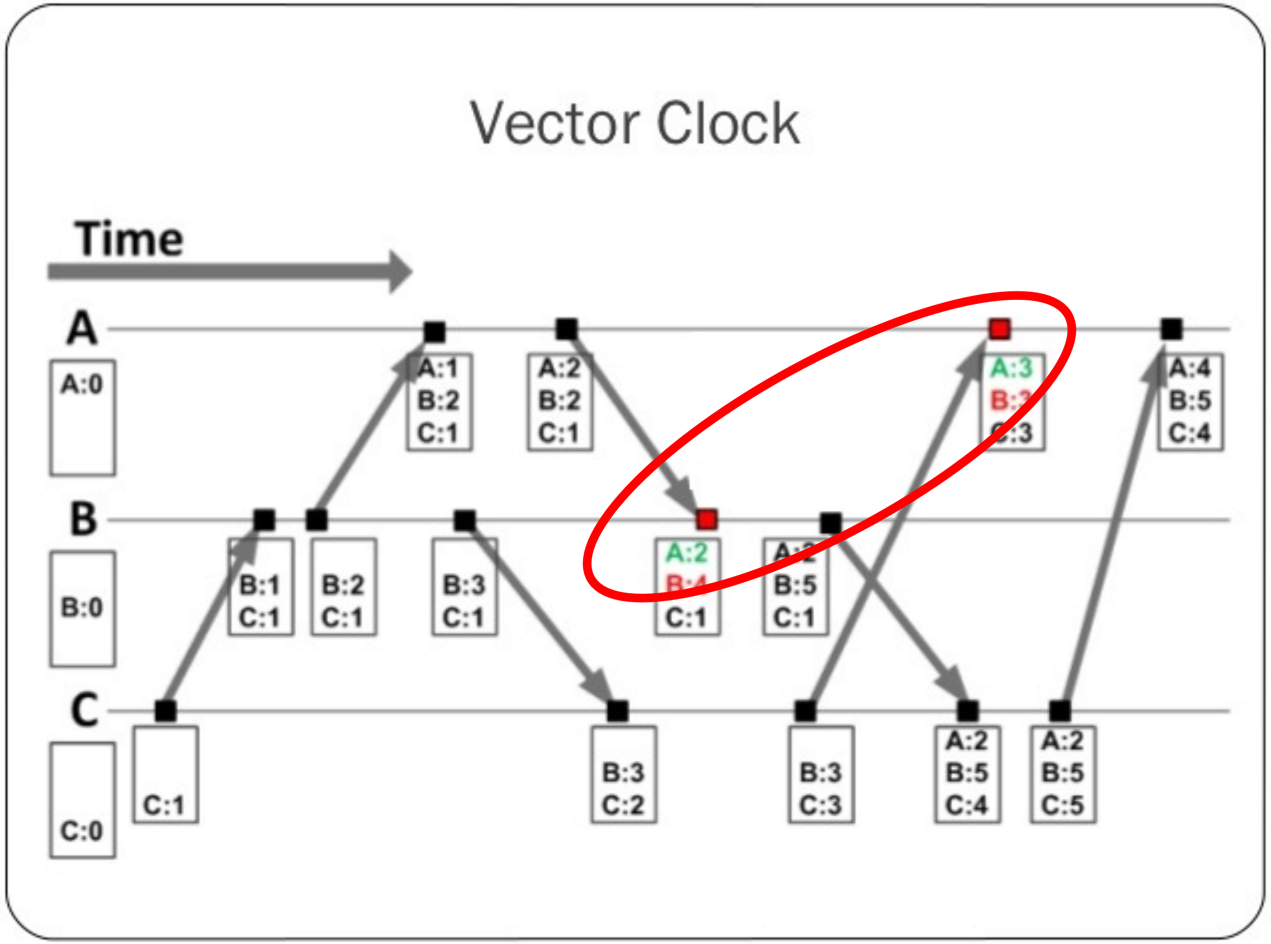
$V_i[i]$: #events occurred at i

$V_i[j]$: #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



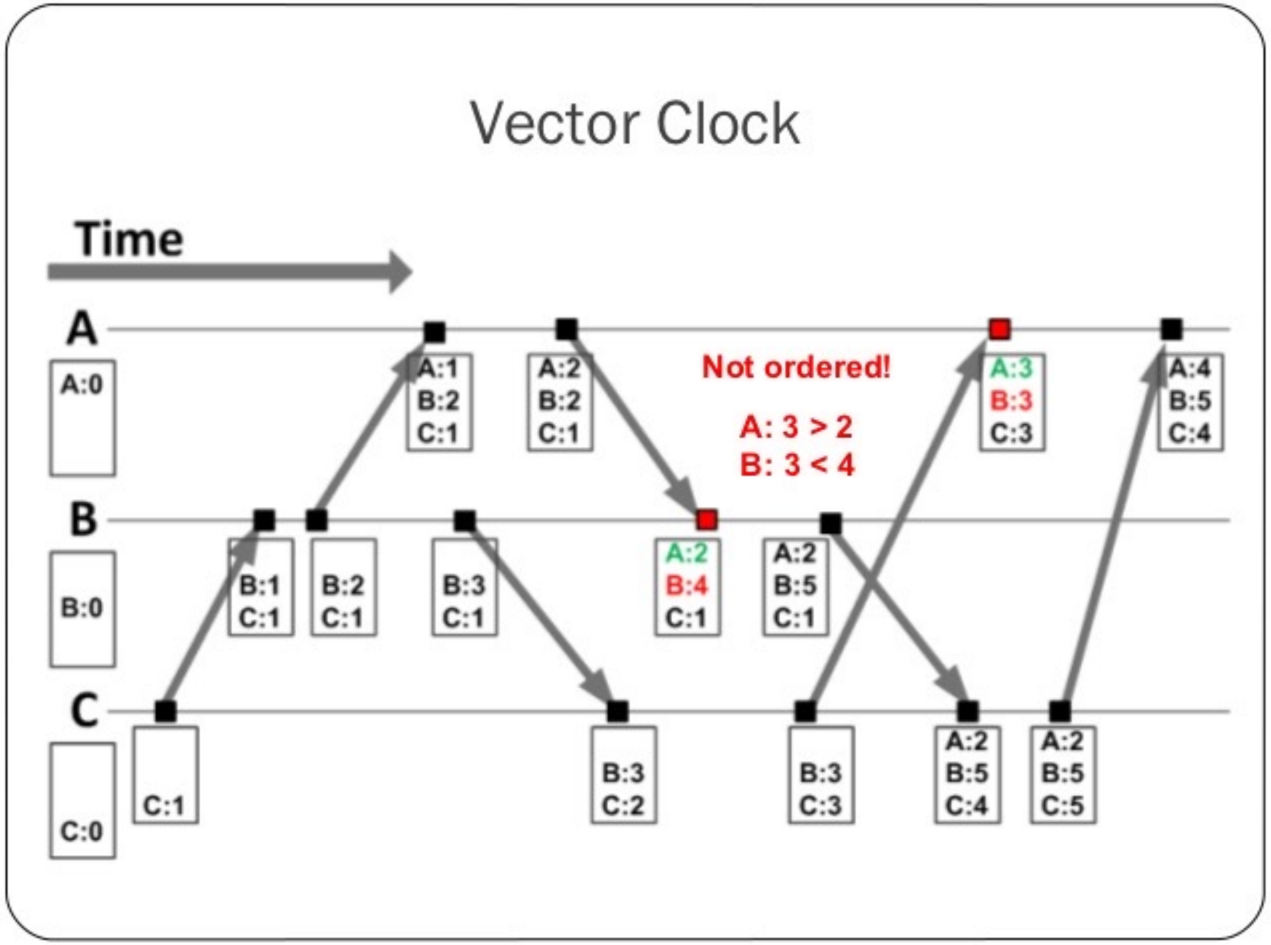
$V_i[i]$: #events occurred at i

$V_i[j]$: #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



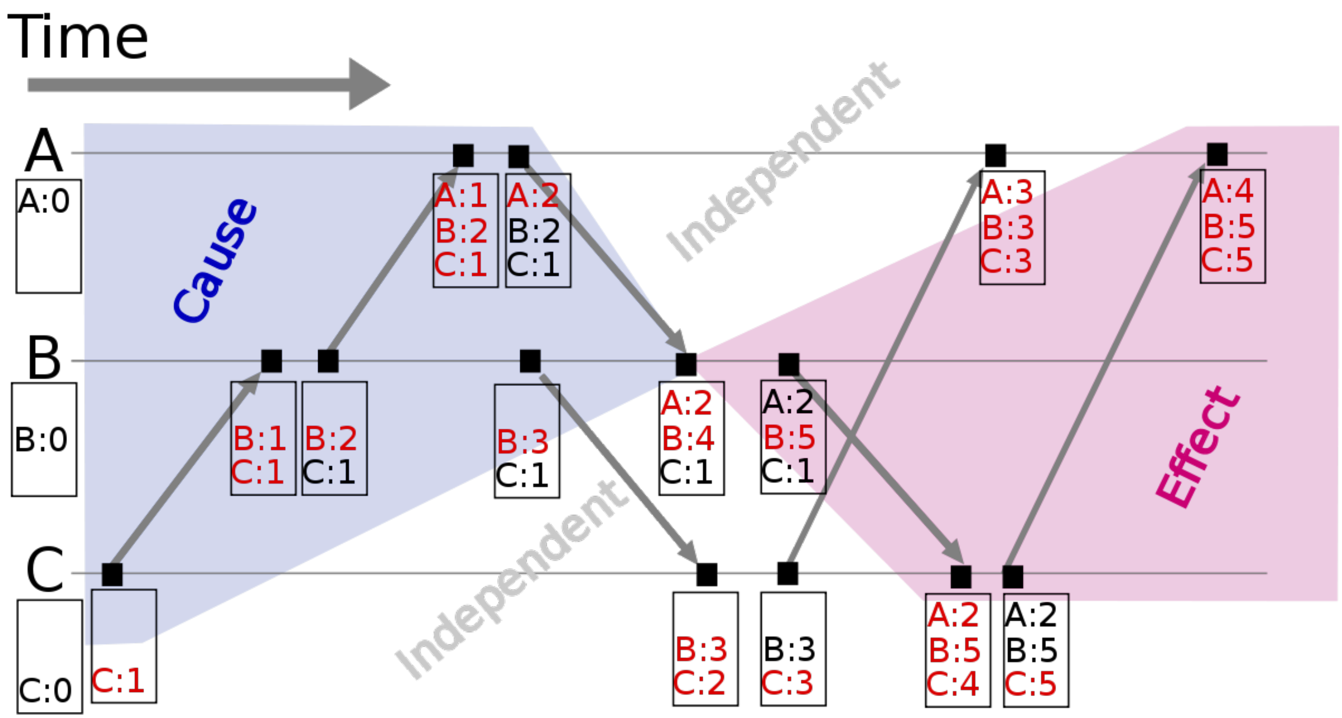
$V_i[i]$: #events occurred at i

$V_i[j]$: #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



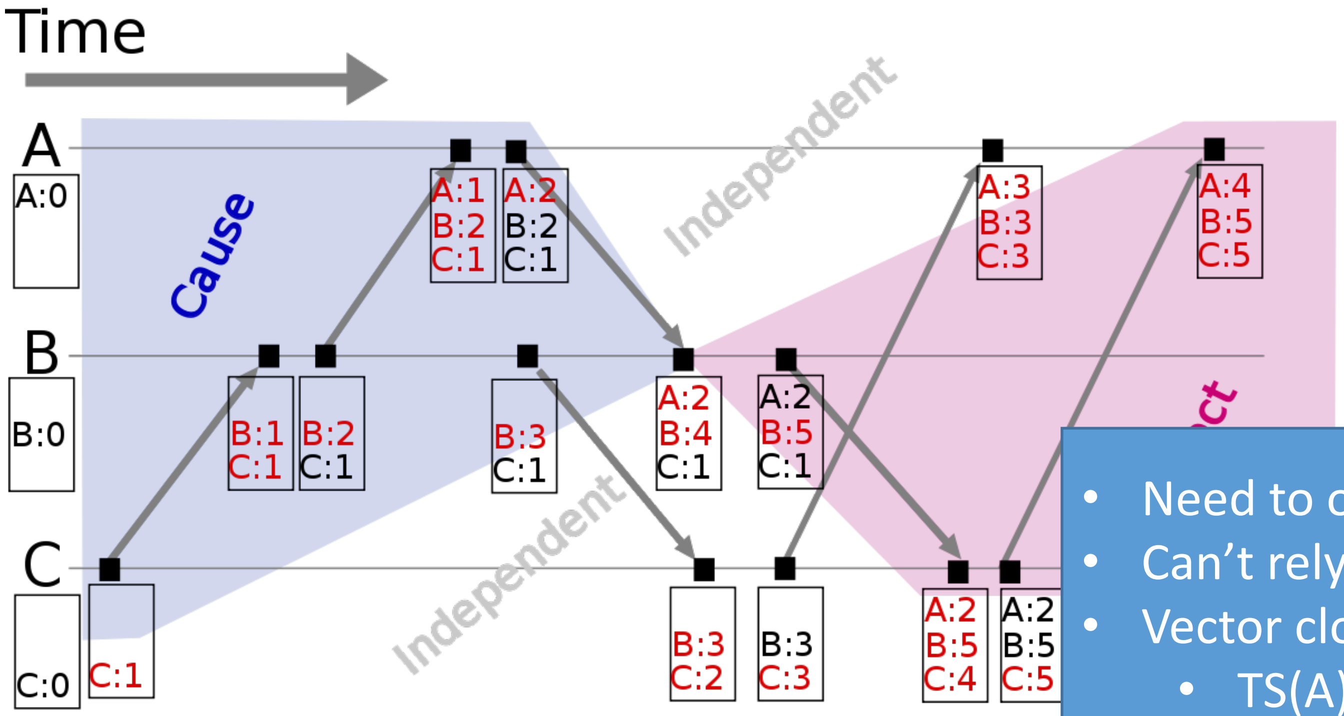
Each process *i* maintains a vector V_i

- $V_i[i]$: 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

$V_i[k]$)

ts 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?

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 synchronization
- Formally:
  - >1 threads access same item
  - No intervening synchronization
  - At least one access is a write

Races

Thread 1

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

Thread 2

```
thread-proc() {

 read-write(X);

}
```

Is there a race here?

How can a race detector tell?

Races

Thread 1

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

Thread 2

```
thread-proc() {

 read-write(X);

}
```

Is there a race here?

How can a race detector tell?

Races

Thread 1

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

Thread 2

```
thread-proc() {

  read-write(X);

}
```

Is there a race here?

How can a race detector tell?

Unsynchronized access can be

Races

Thread 1

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

Thread 2

```
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

Thread 1

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

Thread 2

```
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

Thread 1

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

Thread 2

```
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

Thread 1

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

Thread 2

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

Lockset Algorithm

- Locking discipline
  - Every shared 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

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

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}

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++;`		{} 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}	
v++;		{}
unlock(lockB);	{}	

ACK! race

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

Group activity

- Analyze figure 3 to determine why the lockset algorithm would report a spurious race

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?

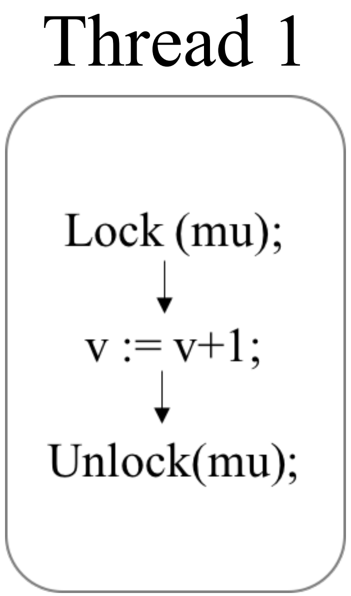
- A-1 happens before B-3
- B-3 happens before A-6
- Insight: races occur when “happens-before” cannot be known

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

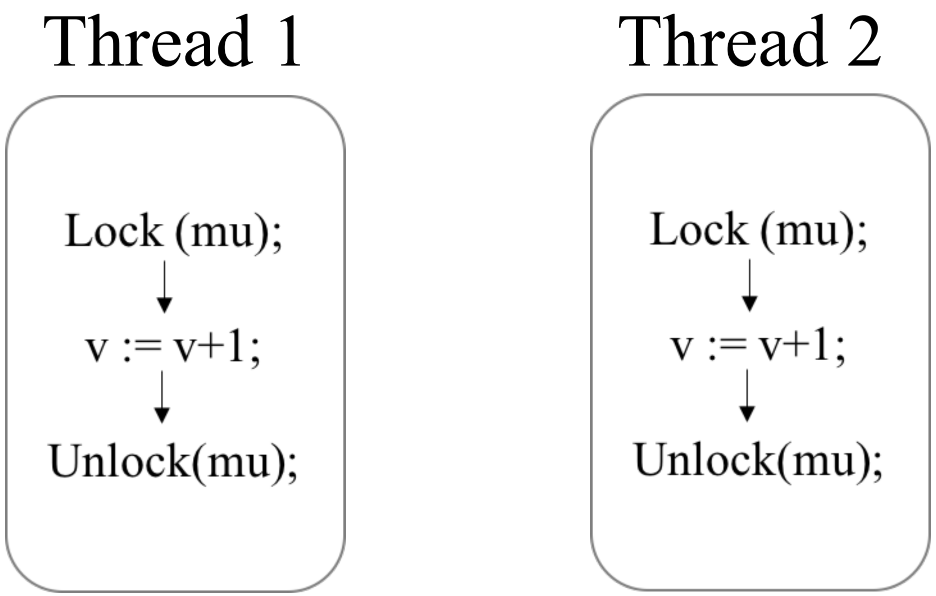
Happens-before

- *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



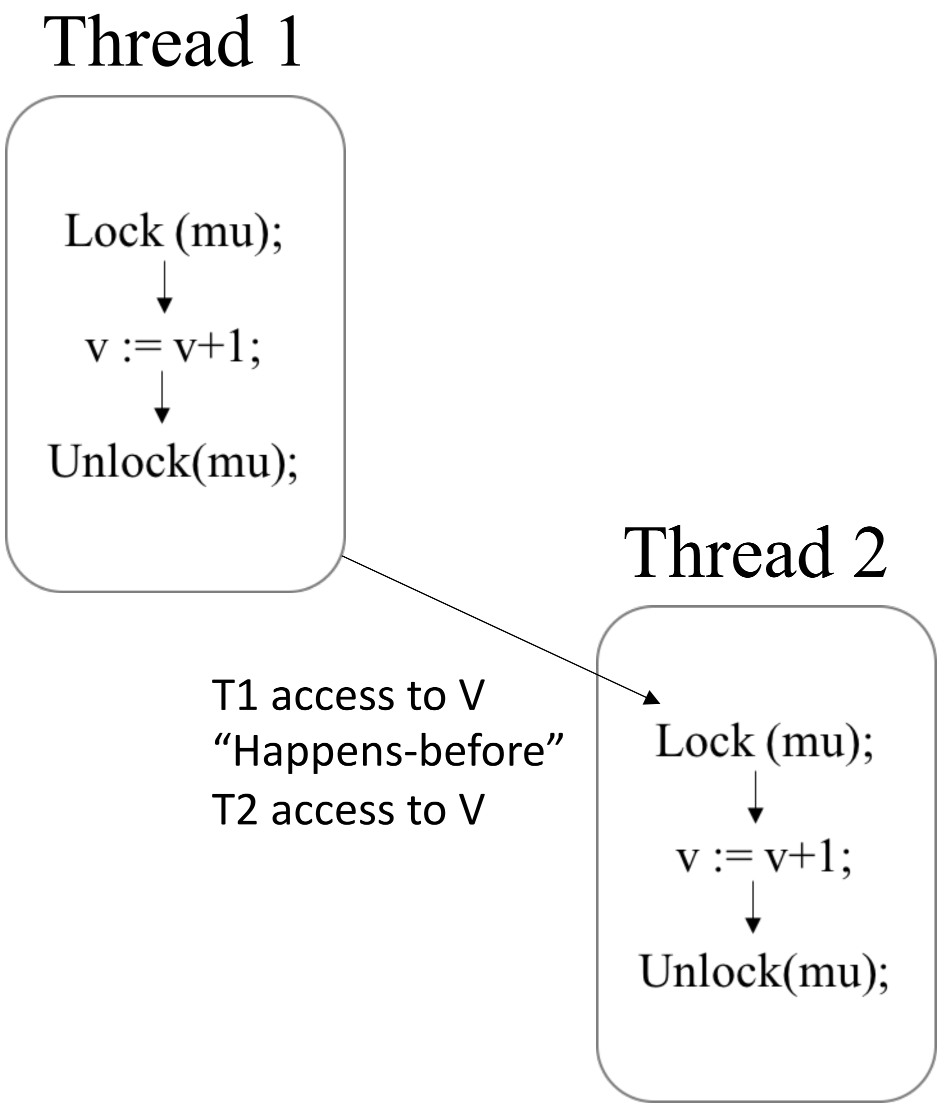
Happens-before

- *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

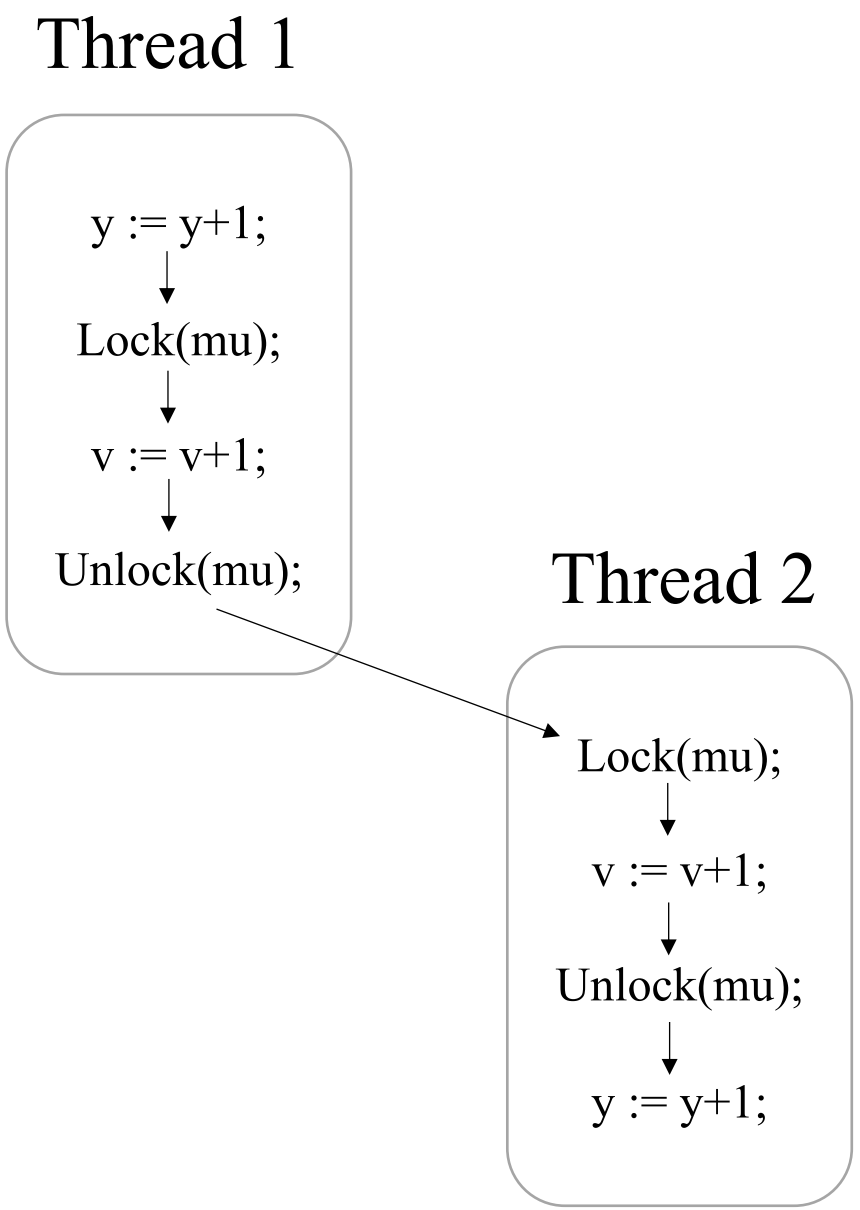


Flaws of *Happens-before*

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

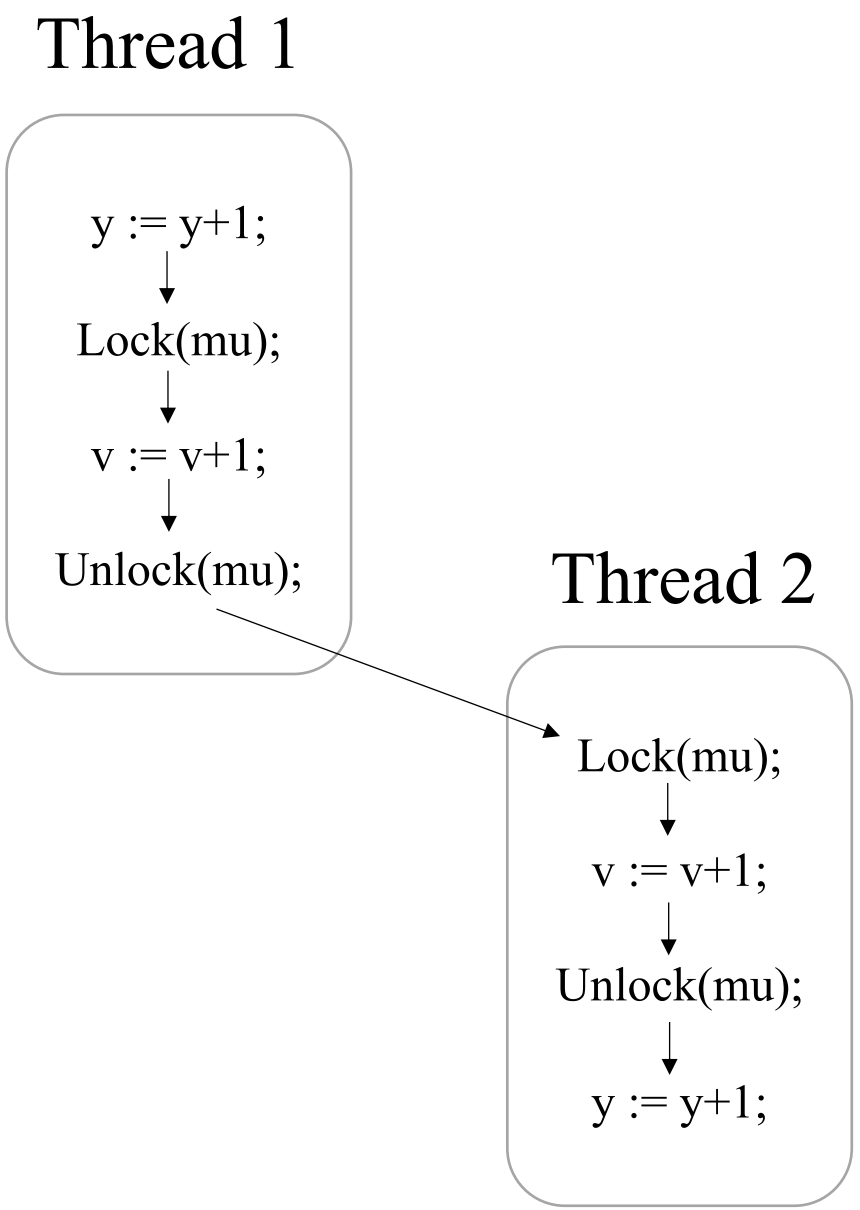
Flaws of *Happens-before*

- 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

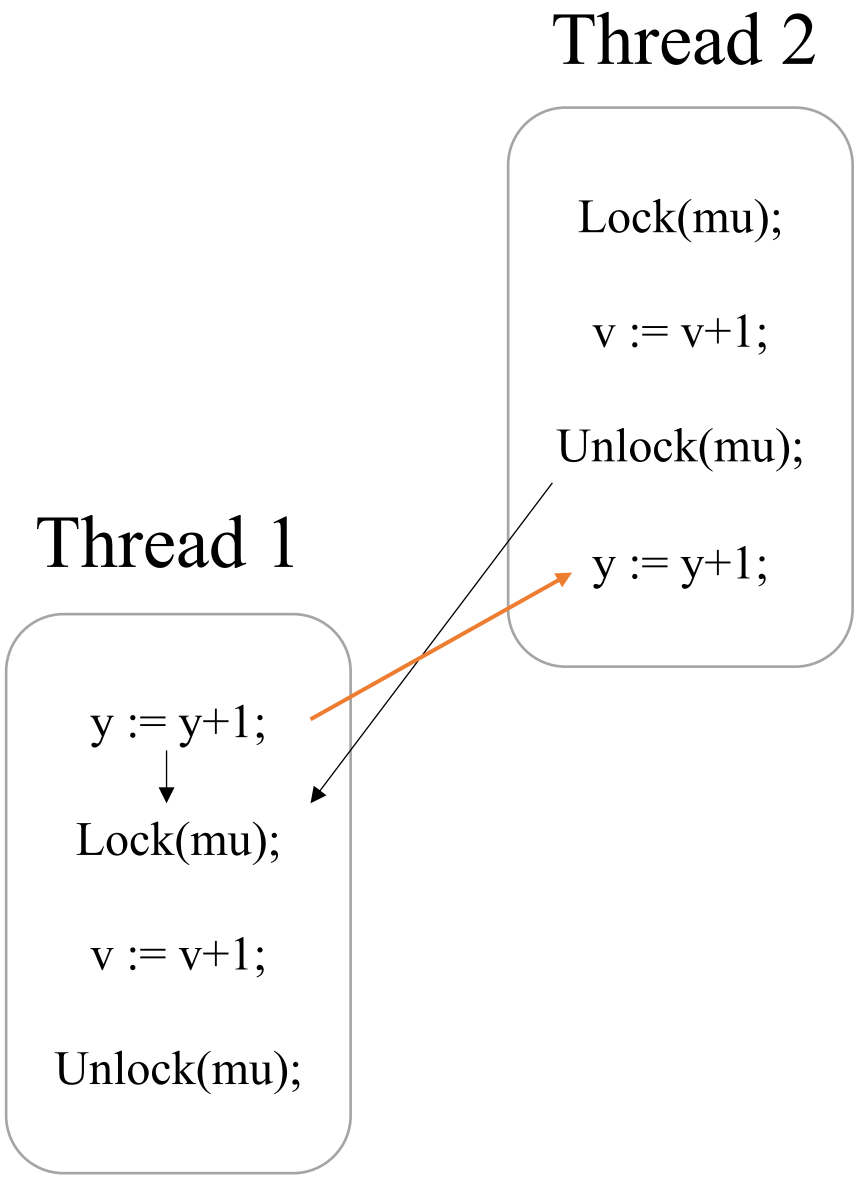


Flaws of *Happens-before*

- 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

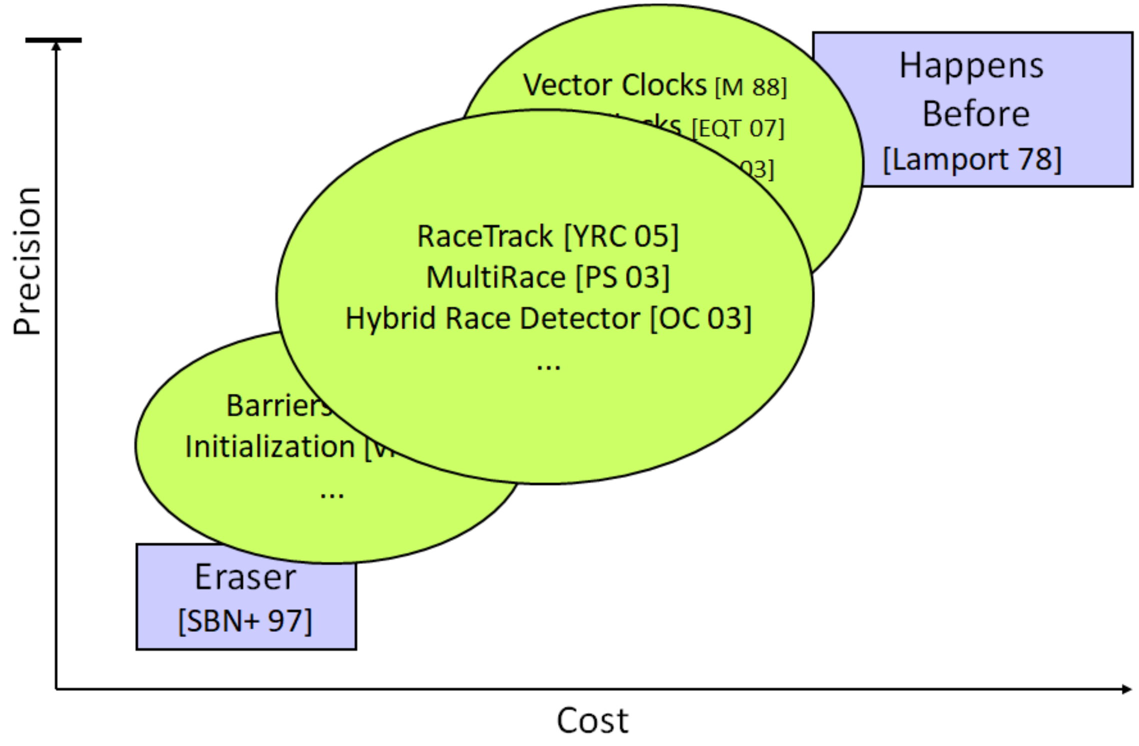
Flaws of *Happens-before*

- 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



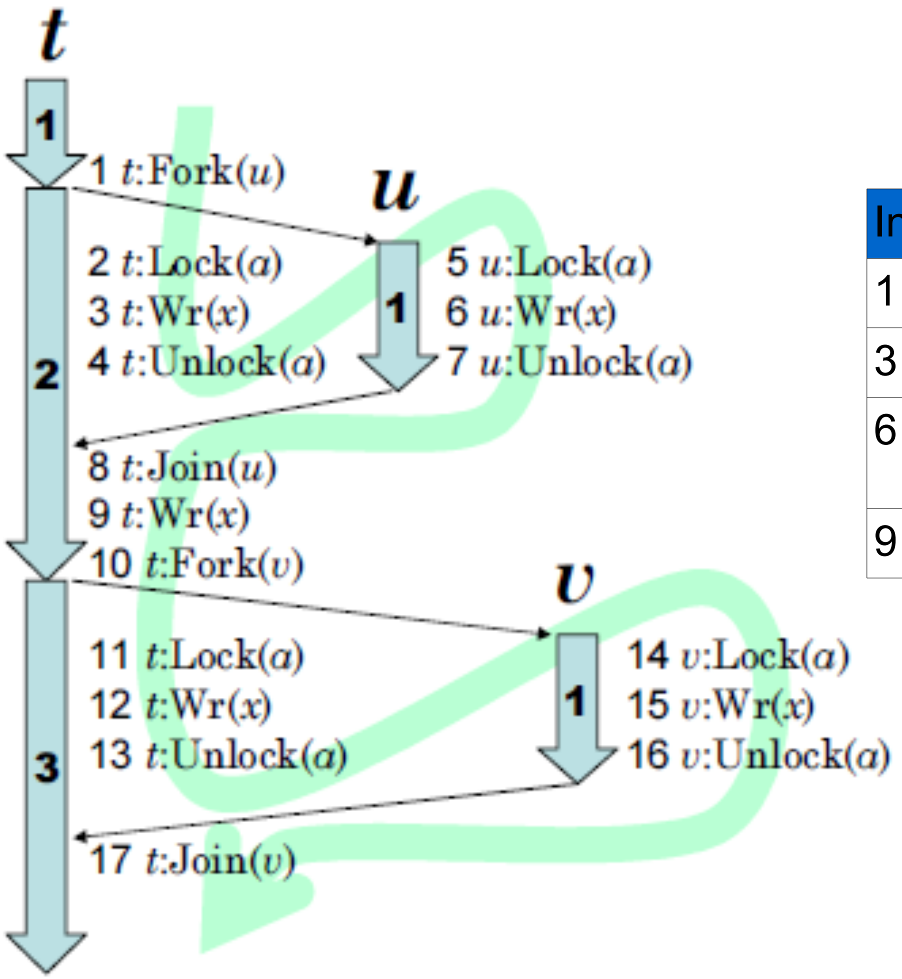
Dynamic Race Detection Summary

- 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):
    $L_t \leftarrow L_t \cup \{l\}$

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

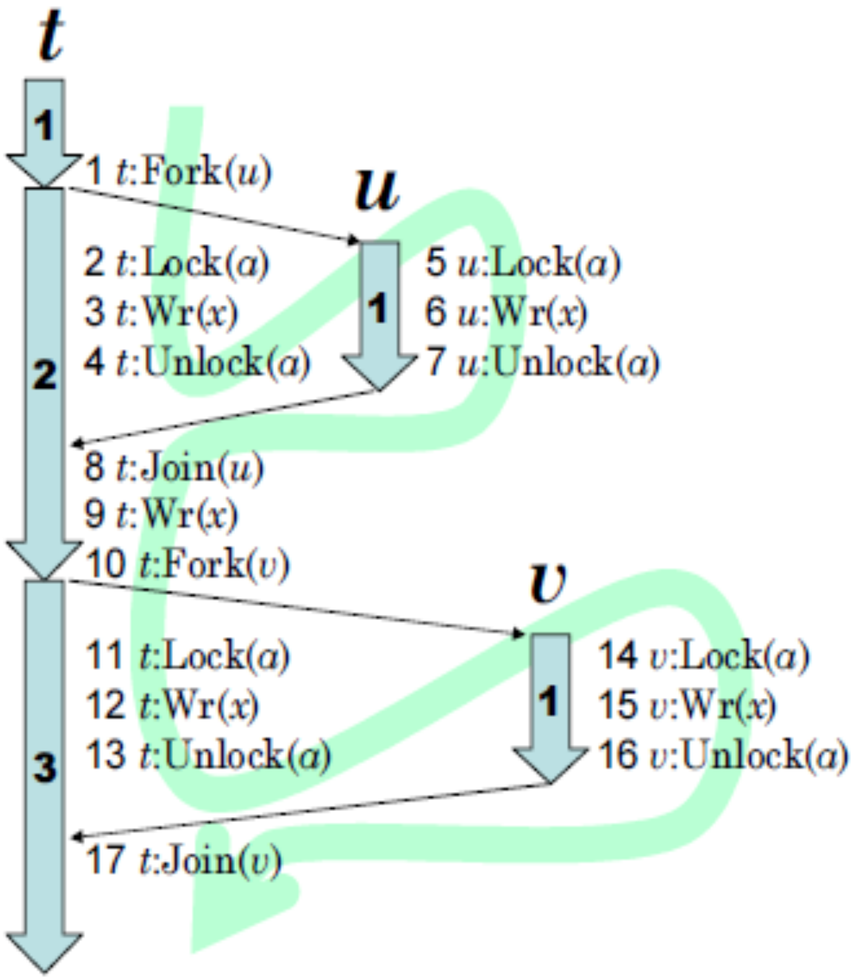
At t:Fork(u):
    $L_u \leftarrow \{\}$
    $B_u \leftarrow Merge(\{\langle u, 1\rangle\}, B_t)$
    $B_t \leftarrow Inc(B_t, t)$

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

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

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

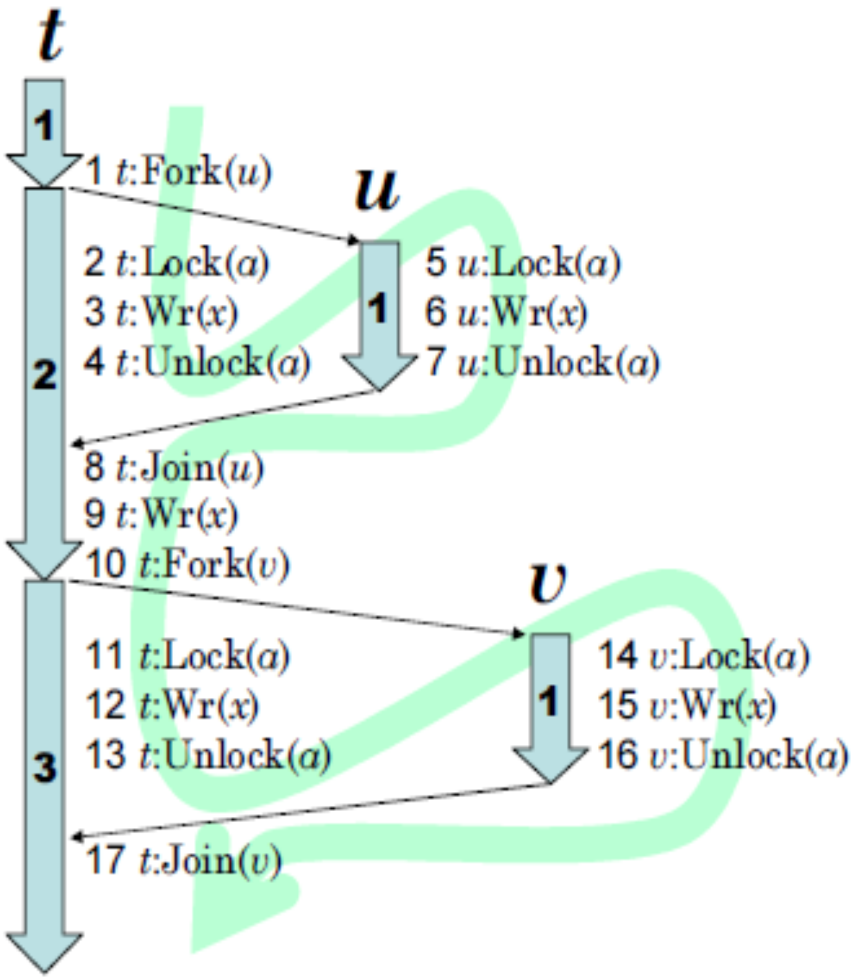
Avoiding Lockset's false positive (1)



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)

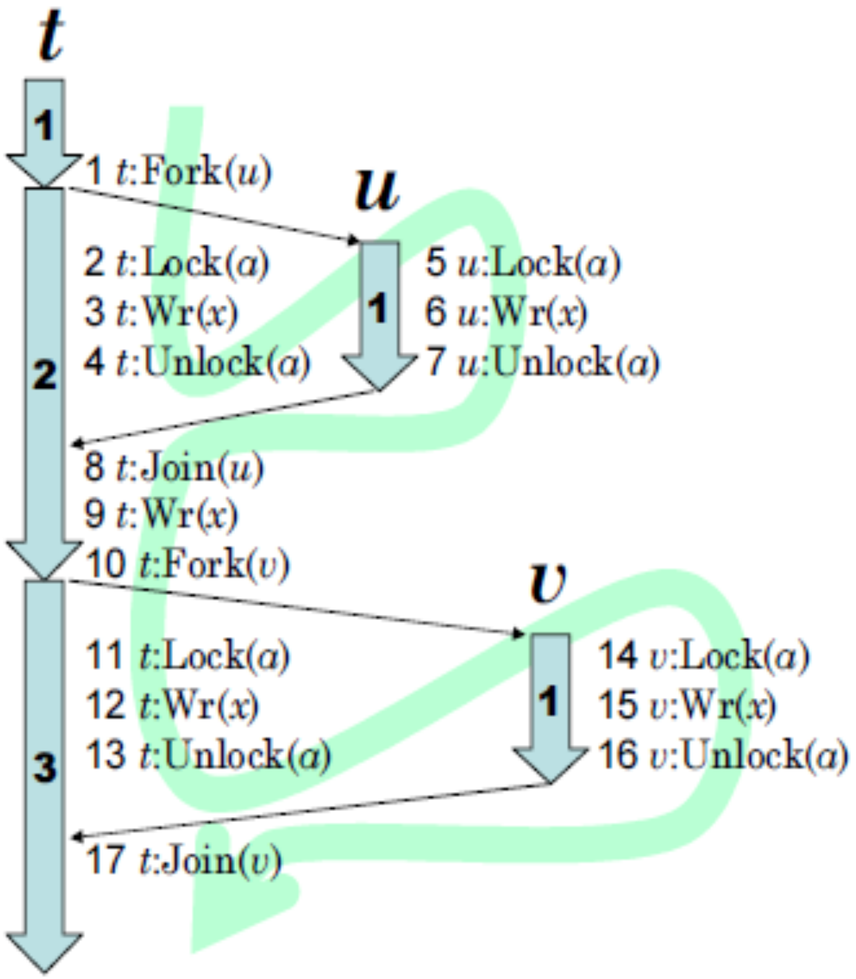
Notation	Meaning
L_t	Lockset of thread **t**
C_x	Lockset of memory **x**
B_t	Vector clock of thread **t**
S_x	Threadset of memory **x**
$\mathbf{t_1}$	Thread **t** at clock time 1



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**
$\mathbf{t_1}$	Thread **t** at clock time 1

Avoiding Lockset's false positive (2)

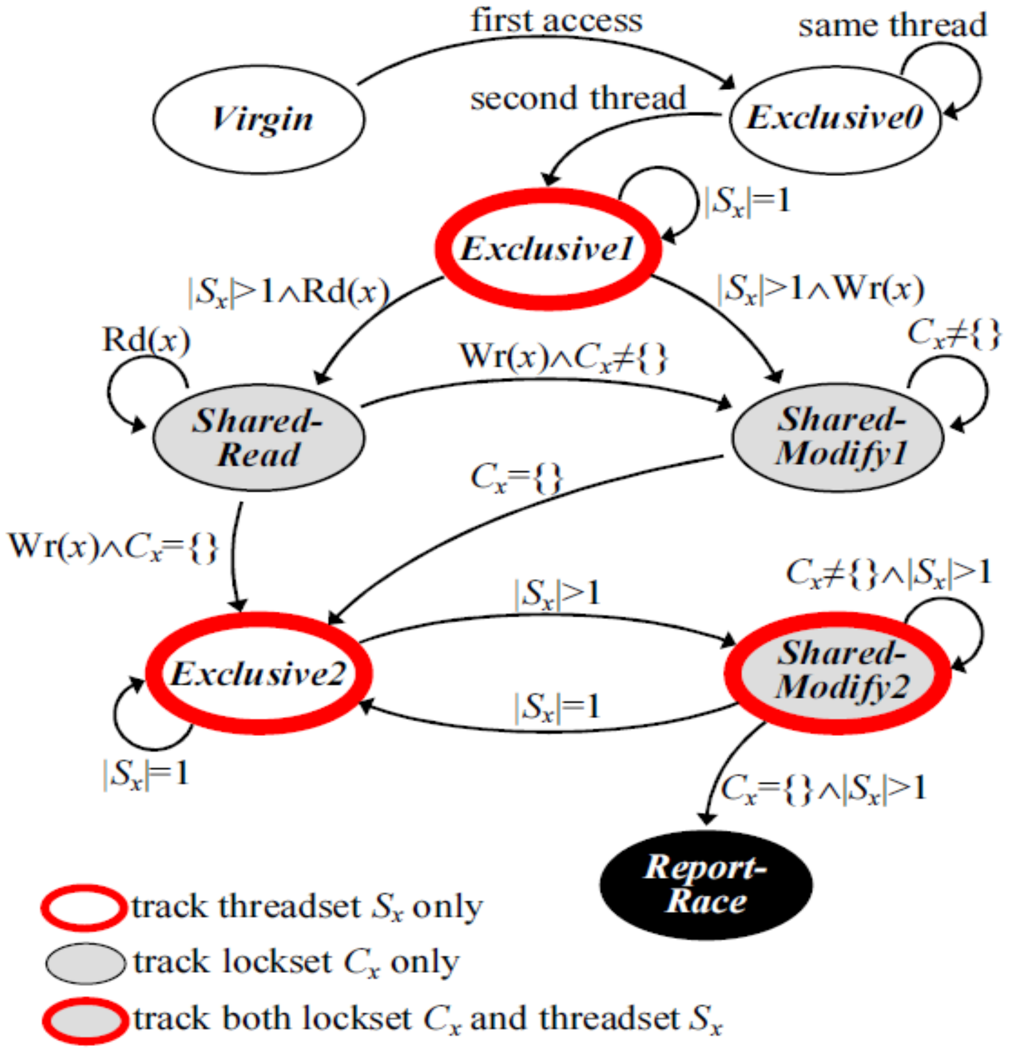


Inst	C_x	S_x	L_t	B_t	L_v	B_v
8	{a}	$\{t_2,u_1\}$	{ }	$\{t_2,u_1\}$	-	-
9	{ }	$\{t_2\}$				
10				$\{t_3,u_1\}$	{ }	$\{t_2,v_1\}$
11			{a}			
12	{a}	$\{t_3\}$				
13			{ }			
14					{a}	
15		$\{t_3,v_1\}$				
16					{ }	

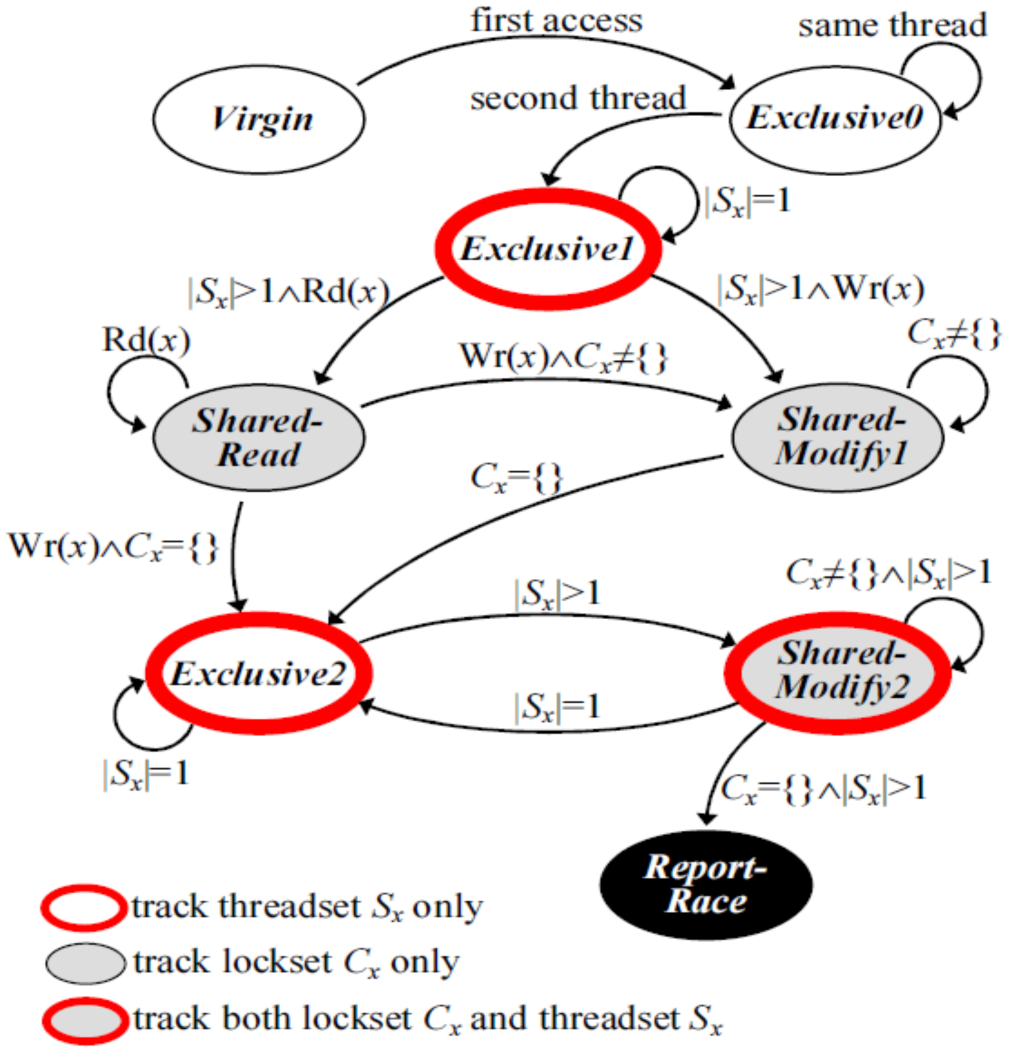
Only one thread!

Are we done?

RaceTrack's state machine



RaceTrack's state machine



Deal with outrageous proliferation of mechanism with adaptivity

Performance & Conclusions

program	Boxwood				SATsolver				SpecJBB				Crawler				Vclient			
lines of code	8579				10,883				31,405				7246				165,192			
active threads	10				1				various				19				69			
	slowdown (sec)	ratio	memory (MB)	ratio	slowdown (sec)	ratio	memory (MB)	ratio	slowdown (ops/s)	ratio	memory (MB)	ratio	slowdown (pages)	ratio	memory (MB)	ratio	slowdown (%cpu)	ratio	memory (MB)	ratio
no RaceTrack	312	1.00	11.5	1.00	713	1.00	102	1.00	19174	1.00	373	1.00	2364	1.00	63.9	1.00	6.4	1.00	63.9	1.00
lockset	366	1.17	15.4	1.34	1974	2.77	170	1.67	6732	2.85	655	1.76	2189	1.08	84.8	1.33	12.5	1.95	74.4	1.17
+threadset	407	1.30	16.5	1.43	2123	2.98	222	2.18	6678	2.87	752	2.02	2214	1.07	108.0	1.69	12.8	2.00	75.6	1.19
+granularity	378	1.21	11.9	1.03	1822	2.56	155	1.52	6029	3.18	441	1.18	2212	1.07	65.0	1.02	12.8	2.00	74.7	1.18

- 3X slowdown on memory intensive programs
  - < 2X on other programs
- 1.2X memory usage

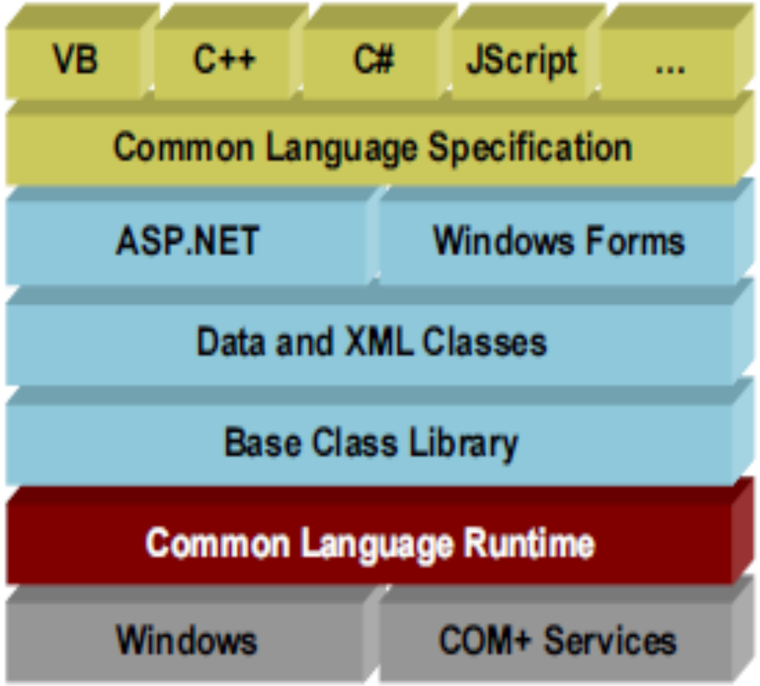
Key ideas recap

- Eliminate Lockset false positives using happens-before
- Refine state machine based on common coding style
- Trade off accuracy for performance/scalability
- *Detail slides moved to end*

Additional ideas from paper

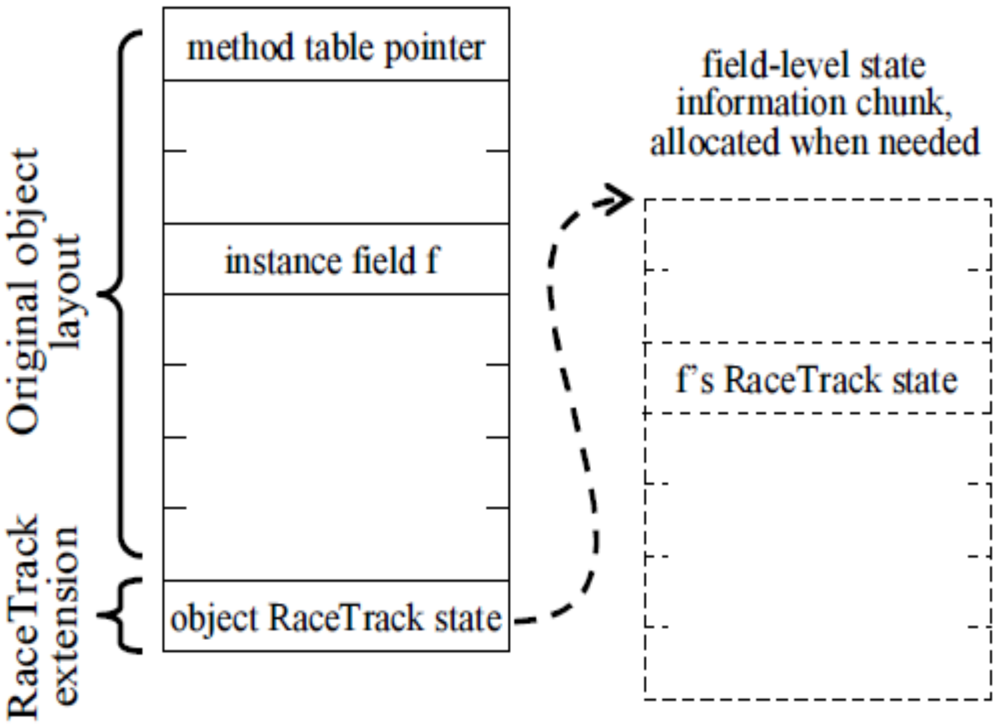
- Accuracy vs performance & scalability tradeoff
  - Object granularity tracking
  - Track subset of (array) objects
  - Prune vector clock
- Annotations to eliminate false positives
- Warnings report analysis
  - Ranking and classification
  - Multiple stack traces

Microsoft CLR Implementation

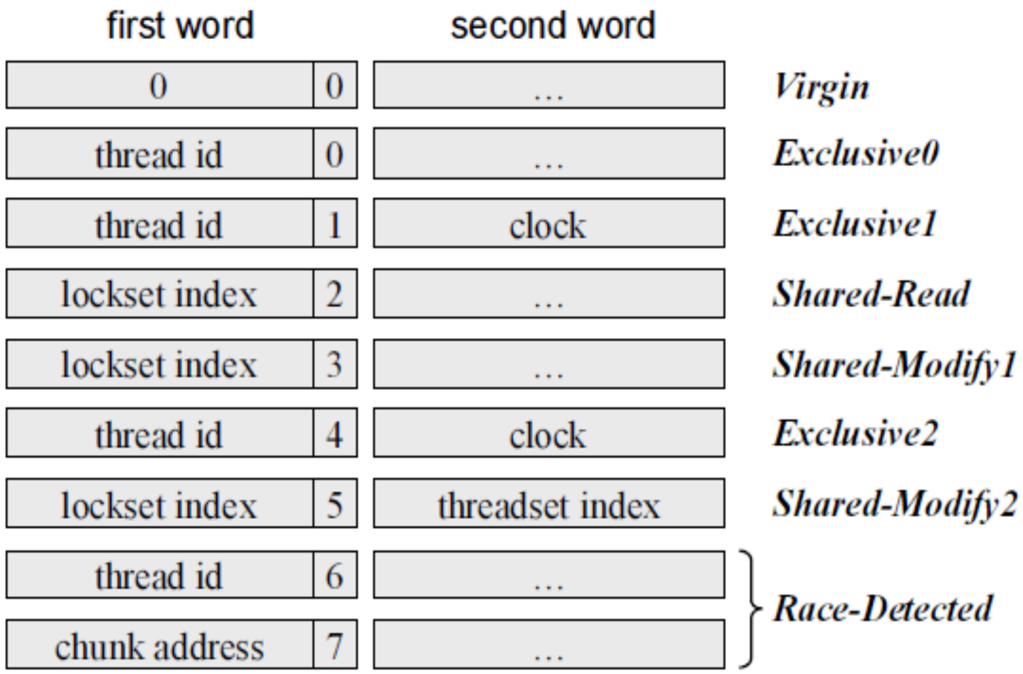


- .NET framework

RaceTrack Object Layout



RaceTrack Encodings



Evaluation

- CLR Regression tests
  - 2122 tests (0.5 MLOC)
  - 48 warnings

#	Category	
6	A.	false alarm - fork/join
2	B.	false alarm - user defined locking
5	C.	performance counter / inconsequential
4	D.	locked atomic mutate, unlocked read
4	E.	double-checked locking
2	F.	cache semantics
2	G.	other correct lock-free code
7	H.	unlocked lazy initialization
8	I.	too complicated to figure out
8	J.	potentially serious bug
48	total	

- Performance
  - 5 real world programs