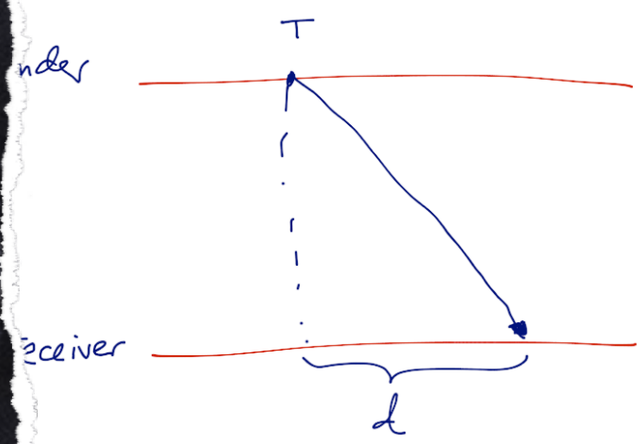
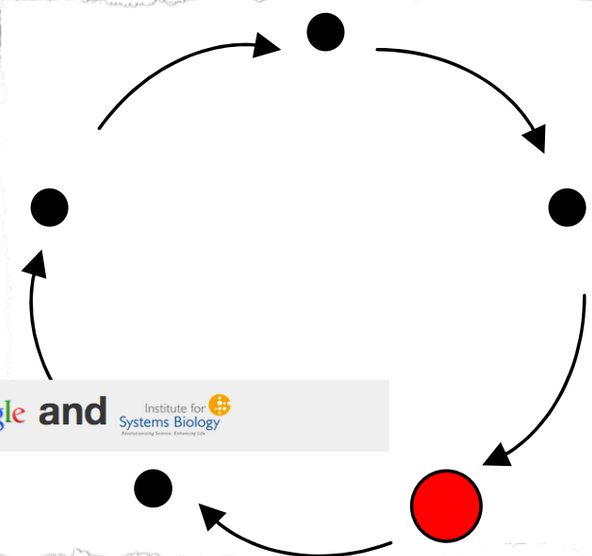
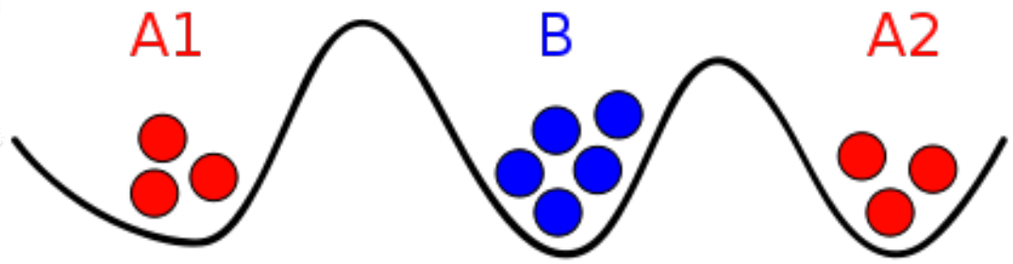
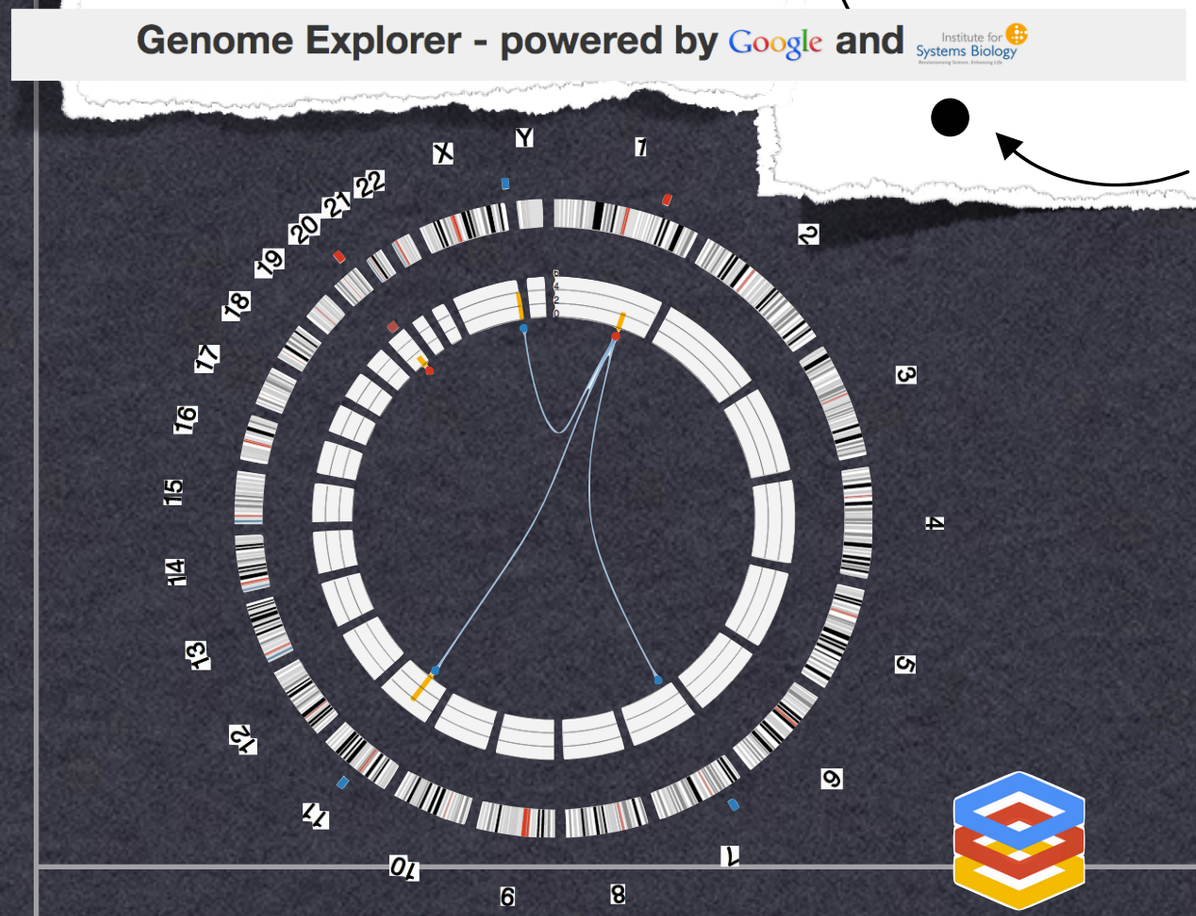
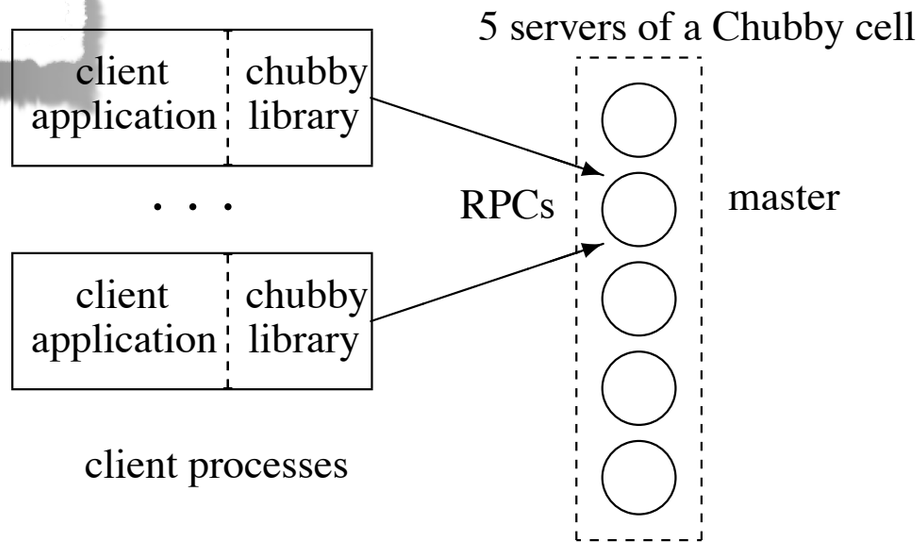










Google Compute Engine

DISTRIBUTED SYSTEMS

LOCAL AND DISTRIBUTED SYNCHRONIZATION

Last week ...

- https://www.mentimeter.com/app/presentation/alzk6k6n49pu1ve456xnxgqvfxzbfg6n/i2xrhx2pes8v

- Processes vs threads

- Threads and local synchronization

- The three heads of Cerberus

- https://www.mentimeter.com/app/presentation/alzk6k6n49pu1ve456xnxgqvfxzbfg6n/6m6jss2jjx5b

This lecture ...

- Local synchronization
  - semaphores
  - conditional variables
- Distributed synchronization
  - distributed time - an example
  - synchronizing real time
  - logical time

Recall …

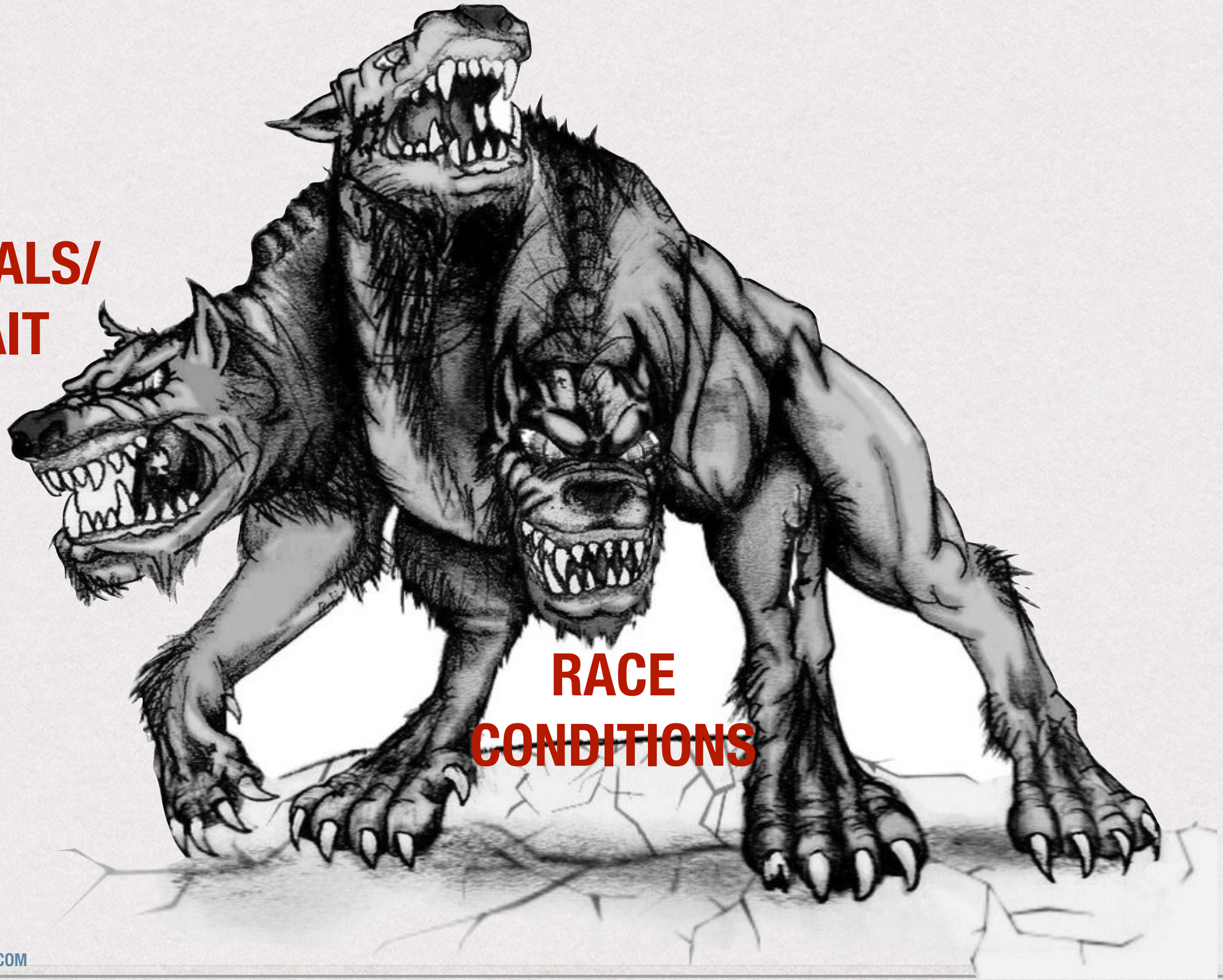
- What is a thread?
- What were the issues in multi-threaded programming?
- What was a race condition?
- How did we deal with race conditions?

Local Synchronization Mechanisms

- Locks
- Semaphores
- Barriers
- Mutexes
- ...

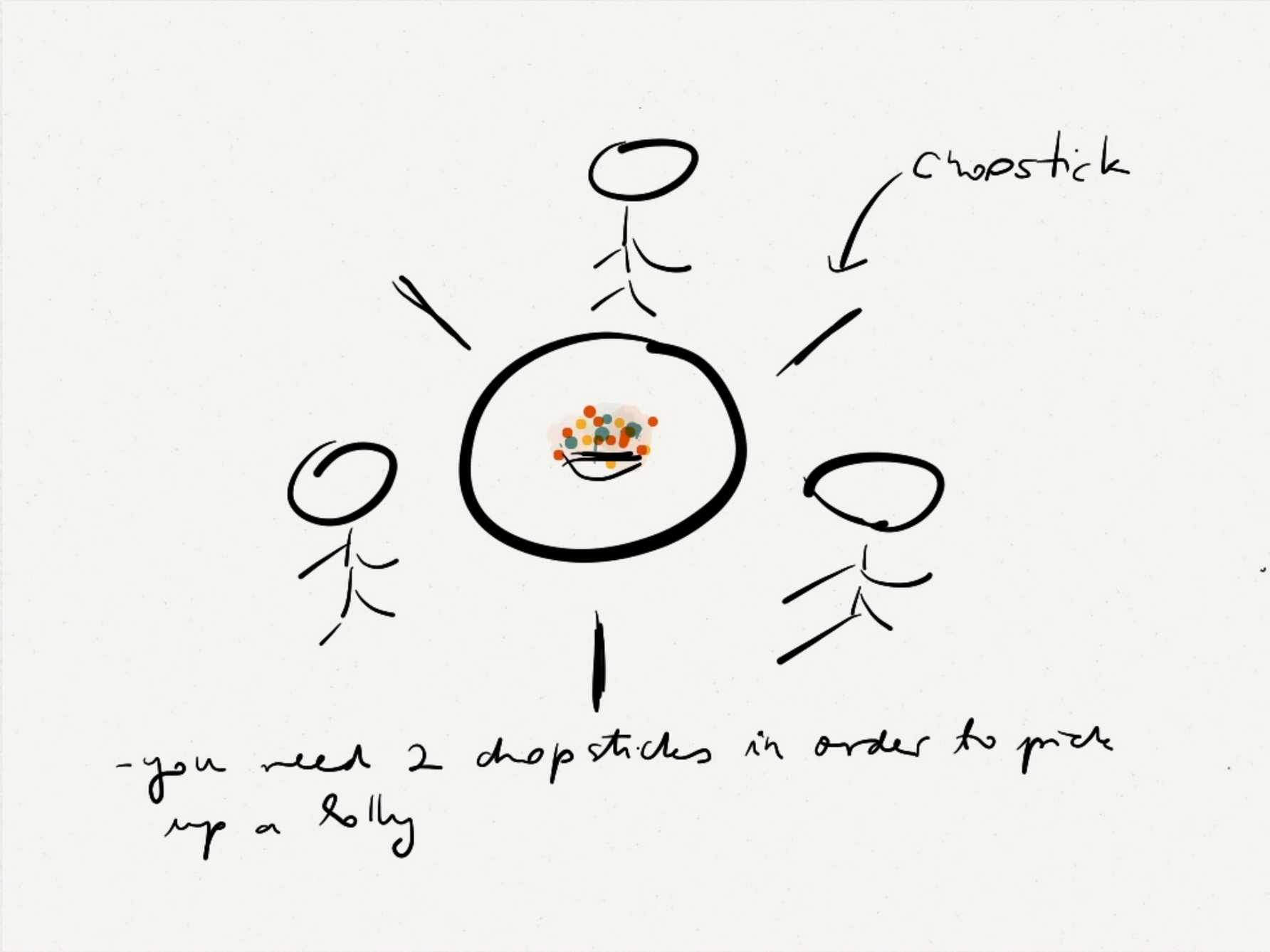
DEADLOCK

SIGNALS/ WAIT



RACE CONDITIONS

Group Activity



DINING PHILOSOPHERS

Group Activity

- Imagine that you and your other three group mates are sitting around a table on which there is a bowl of lollies and a number of chopsticks equal to the number of members in your team.
- You need two chopsticks to pick up one lolly. You can only pick up the chopsticks that are on your side, and picking up and putting down are actions that can be done one at a time. ***You and your team mates are silent.***
- Now further imagine that the lollies are your only source of sustenance - you need lollies to think, and you need to think to live. Design ***a protocol*** that ensures that everybody around the table gets a fair share of the lollies.

Edsger W. Dijkstra

- Fundamental contributions to the development of programming languages, graph theory, distributed systems
  - Shortest path algorithm
  - Reverse polish notation
  - Banker's algorithm and semaphores, self stabilization
  - Formal verification and CSP …
- Turing Award

Semaphores

- Simple abstraction to control access to a common resource
  - signal and wait
- Variable x that allows interaction via two operations
  - `wait(x):` `while (x == 0) wait; --x;`
  - `signal(x):` `++x;`
- Both operations are done atomically

Example - Producer Consumer

- One process (the producer) generates data items and another process (the consumer) receives and uses it

- They communicate using a queue of maximum size N with the conditions

  - The consumer must wait for the producer to produce something if the queue is empty.

  - The producer must wait for the consumer to consume something if the queue is full.

Group Activity

- In your groups, list examples of distributed systems where producer-consumer is used
  - team with maximum number of examples wins
  - time: 3 minutes!

Example

Q: WHAT HAPPENS IF WE HAVE MORE THAN ONE PRODUCER?

```
produce:

wait(emptyCount)

putItemIntoQueue(item)

signal(fullCount)
```

```
consume:

 wait(fullCount)

getItemFromQueue()

signal(emptyCount)
```

Example

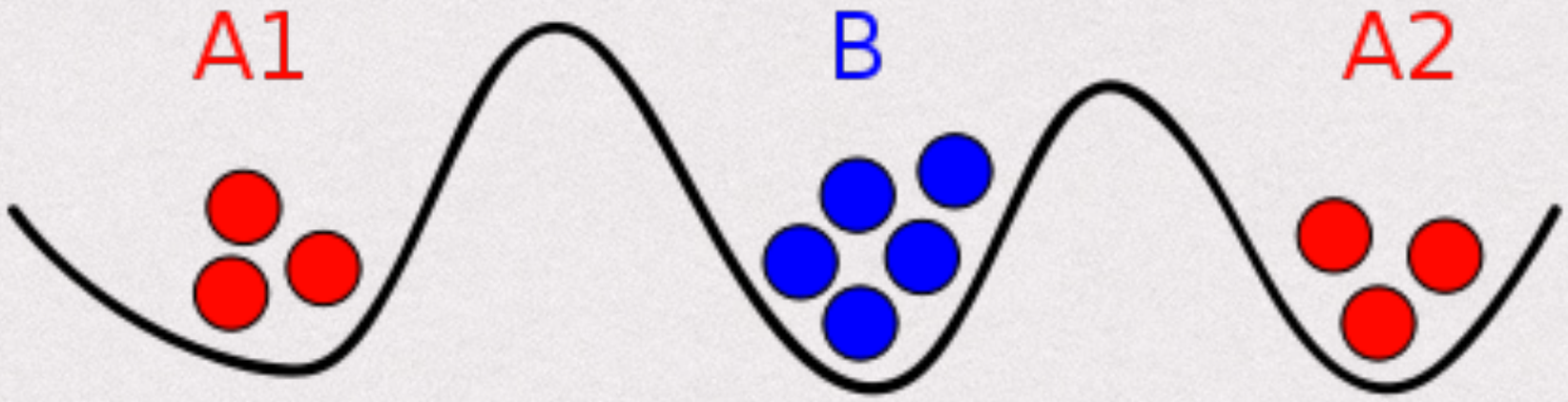
```
produce:

    wait(emptyCount)

    wait(useQueue)


putItemIntoQueue(item)

    signal(useQueue)

    signal(fullCount)
```

```
consume:

    wait(fullCount)

    wait(useQueue)

getItemFromQueue()

 signal(useQueue)

signal(emptyCount)
```

Distributed Synchronization

- Multiple processes/programs on different machines share the same resource: printer, file etc
- Things are worse when you consider failures

General's problem

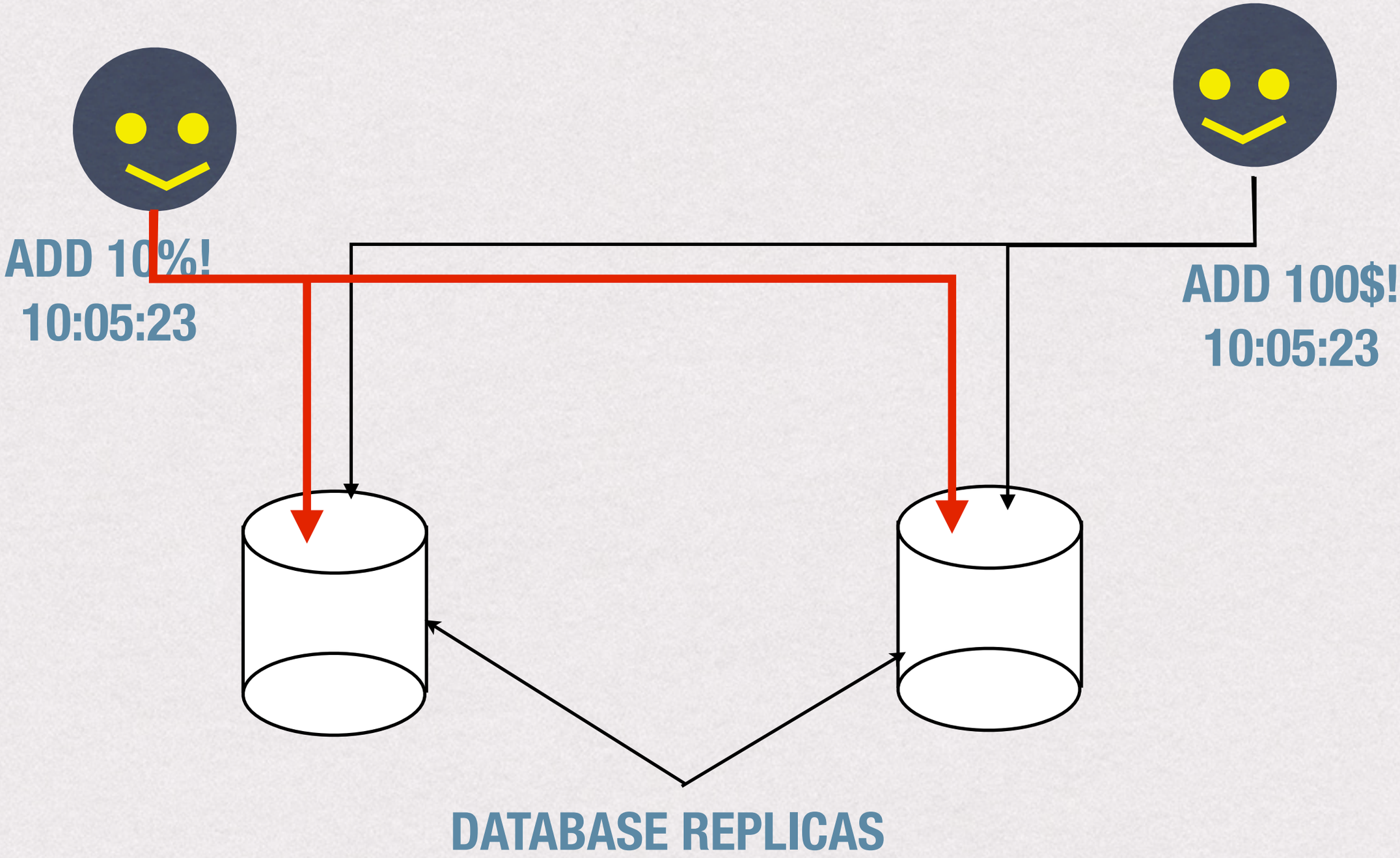


- Armies A1 and A2 need to communicate but their messengers may be captured by army B
- Clients/servers may crash, right in the middle of an RPC call
- Sent and received messages may be lost

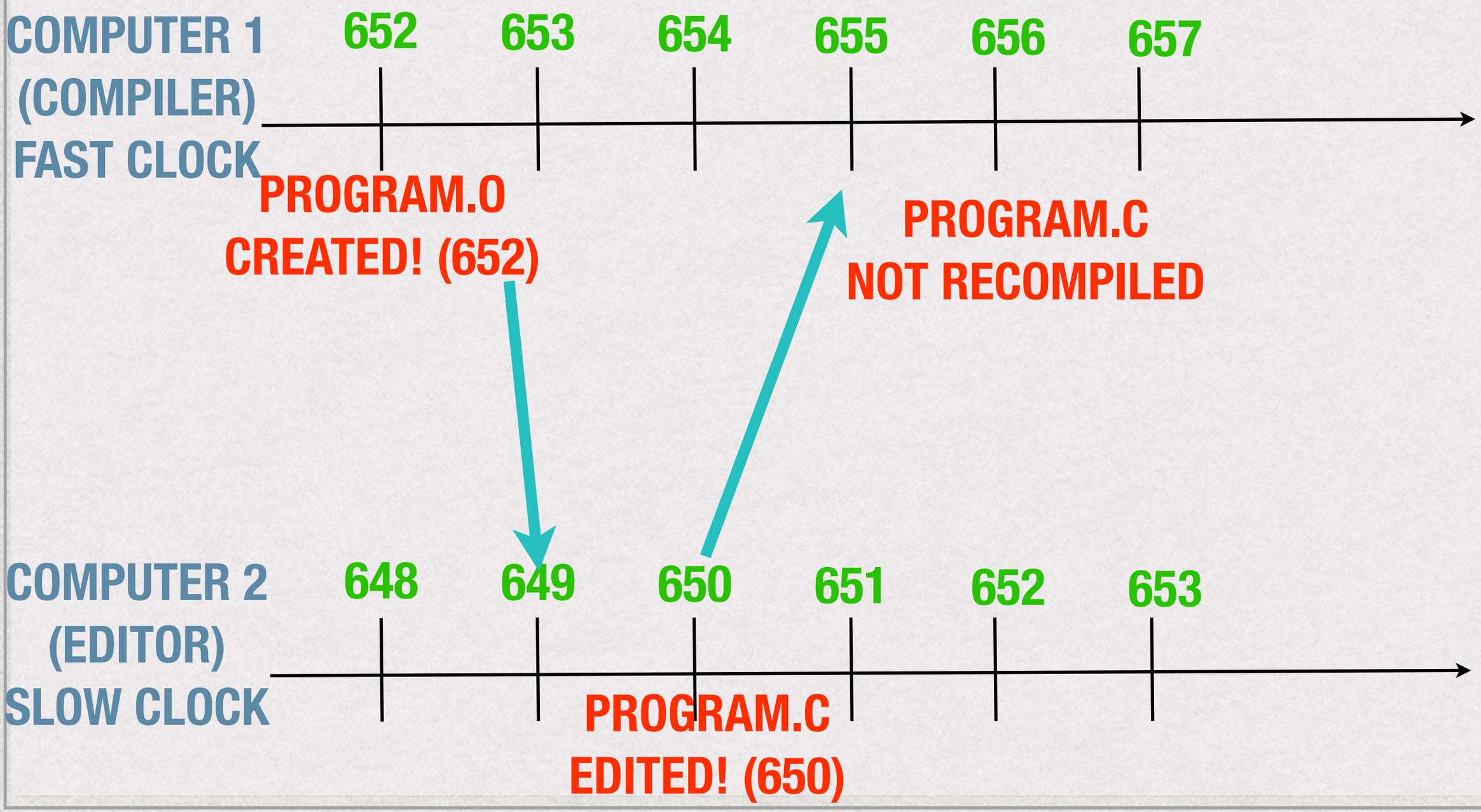
Distributed Synchronization Mechanisms

- **Logical clocks**: clock synchronization is a real issue in distributed systems, hence they often maintain logical clocks, which count operations on the shared resource
- **Consensus:** multiple machines reach majority agreement over the operations they should perform and their ordering
- **Data consistency protocols:** replicas evolve their states in pre- defined ways so as to reach a common state despite different views of the input
- **Distributed locking services:** machines grab locks from a centralized, but still distributed, locking service, so as to coordinate their accesses to shared resources (e.g., files)
- **Distributed transactions:** an operation that involves multiple services either succeeds or fails at all of them

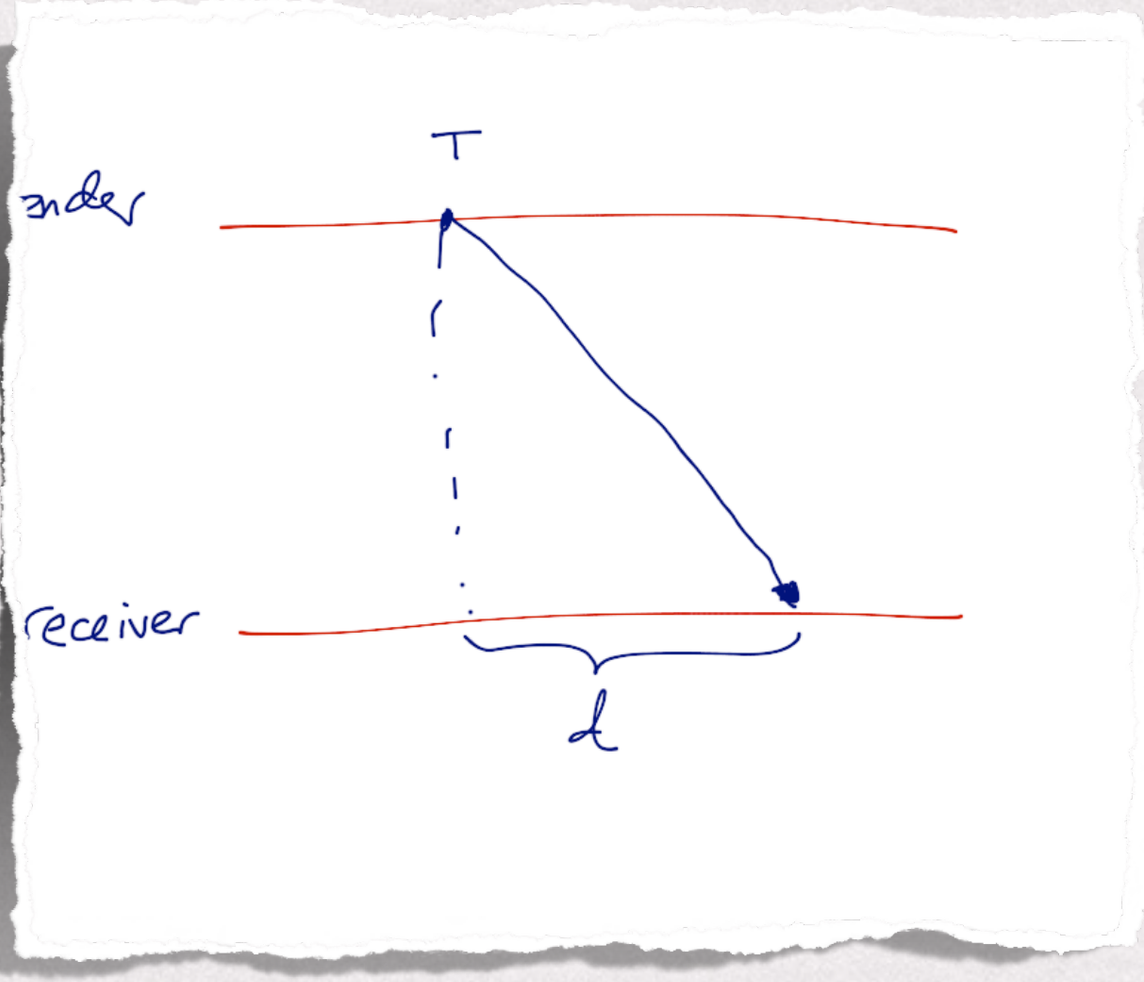
Distributed Synchronization Example I



Distributed Synchronization Example II - Distributed Make



In a perfect world ...



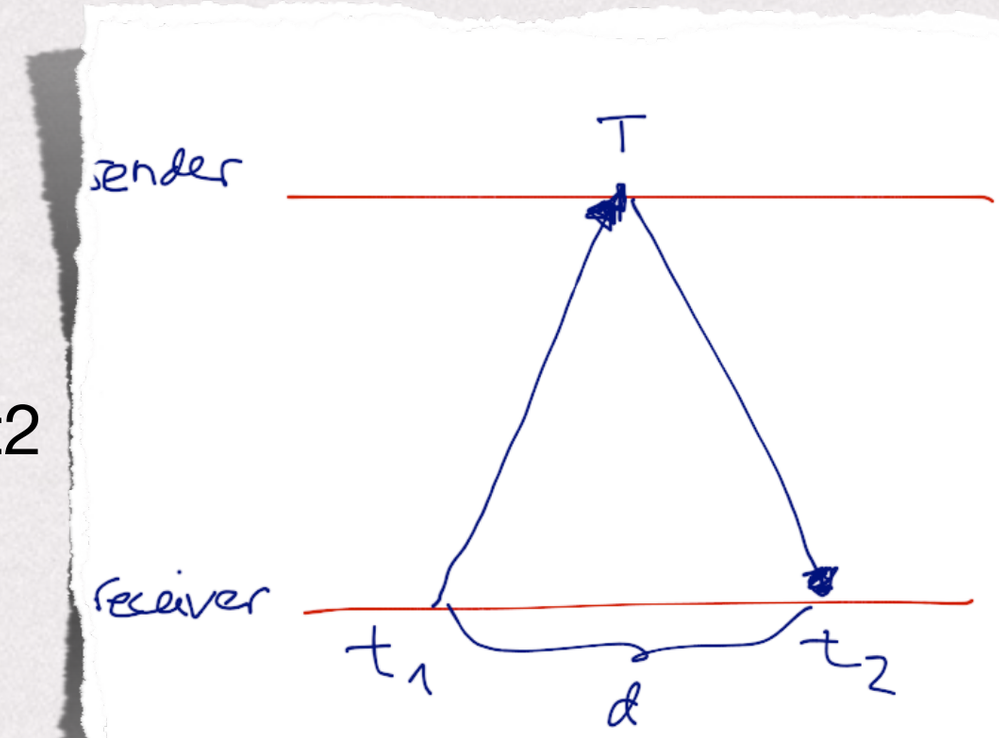
- Messages always arrive:
  - propagation delay ***exactly*** d
- Sender sends time T in a message
- Receiver sets clock to T+d
  - exact synchronization

Distributed Synchronization

- Synchronizing real clocks
  - Cristian's algorithm
  - The Berkeley Algorithm
  - Network Time Protocol (NTP)
- Logical time
  - Lamport logical clocks
  - Vector clocks

Cristian's algorithm

- Request time, get reply
  - Measure actual round-trip time d
- Sender's time was T between t1 and t2
- Receiver sets time to T+d/2
- Synchronization error is at most d/2
  - Can retry until we get a relatively small d



The Berkley Algorithm

- Master uses Cristian's algorithm to get time from many clients
  - Computes average time
  - Can discard outliers
- Sends time adjustments back to all clients

The Network Time Protocol

- Uses a hierarchy of time servers
  - Class 1 servers have highly-accurate clocks connected directly to atomic clocks, etc.
- Class 2 servers get time from only Class 1 and Class 2 servers
- Class 3 servers get time from any server
- Synchronization similar to Cristian's algorithm
  - Modified to use multiple one-way messages instead of immediate round-trip
- Accuracy: Local~1ms, Global~10ms

Real Synchronization not needed

- Usually distributed systems do not need exact real time, but an agreement on some time and some order
- E.g.: suppose file servers S1 and S2 receive two update requests, W1 and W2, for file F
- They need to apply W1 and W2 in the same order, but they don’t really care precisely which order...

Logical Time

- Capture just the order between events without caring about the actual time when the events happen

- Time at each process is well-defined

- Definition (→i): We say e →i e' if e happens before e' at process i

Global Logical Time

- Definition (→): We define $e \rightarrow e'$ using the following rules:
  - Local ordering: $e \rightarrow e'$ if $e \rightarrow_i e'$ for any process *i*
  - Messages: send(m) → receive(m) for any message m
  - Transitivity: $e \rightarrow e''$ if $e \rightarrow e'$ and $e' \rightarrow e''$
- We say e “happens before” e’ if $e \rightarrow e'$

Concurrency

- -> is a partial order
- Definition (concurrency): We say e is concurrent with e' (written e $\|$ e') if neither $e \rightarrow e'$ nor $e' \rightarrow e$

Lamport Logical Clocks

- Assigns logical timestamps to events consistent with "happens before" ordering; we want a clock such that:

$$\text{If } e \rightarrow e', \text{ then } L(e) \leq L(e')$$

- But not the converse
  – L(e) < L(e') does not imply e → e'

- Similar rules for concurrency
  – L(e) = L(e') implies e ‖ e' (for distinct e,e') – e ‖ e' does not imply L(e) = L(e')

- => Lamport clocks arbitrarily order some concurrent events

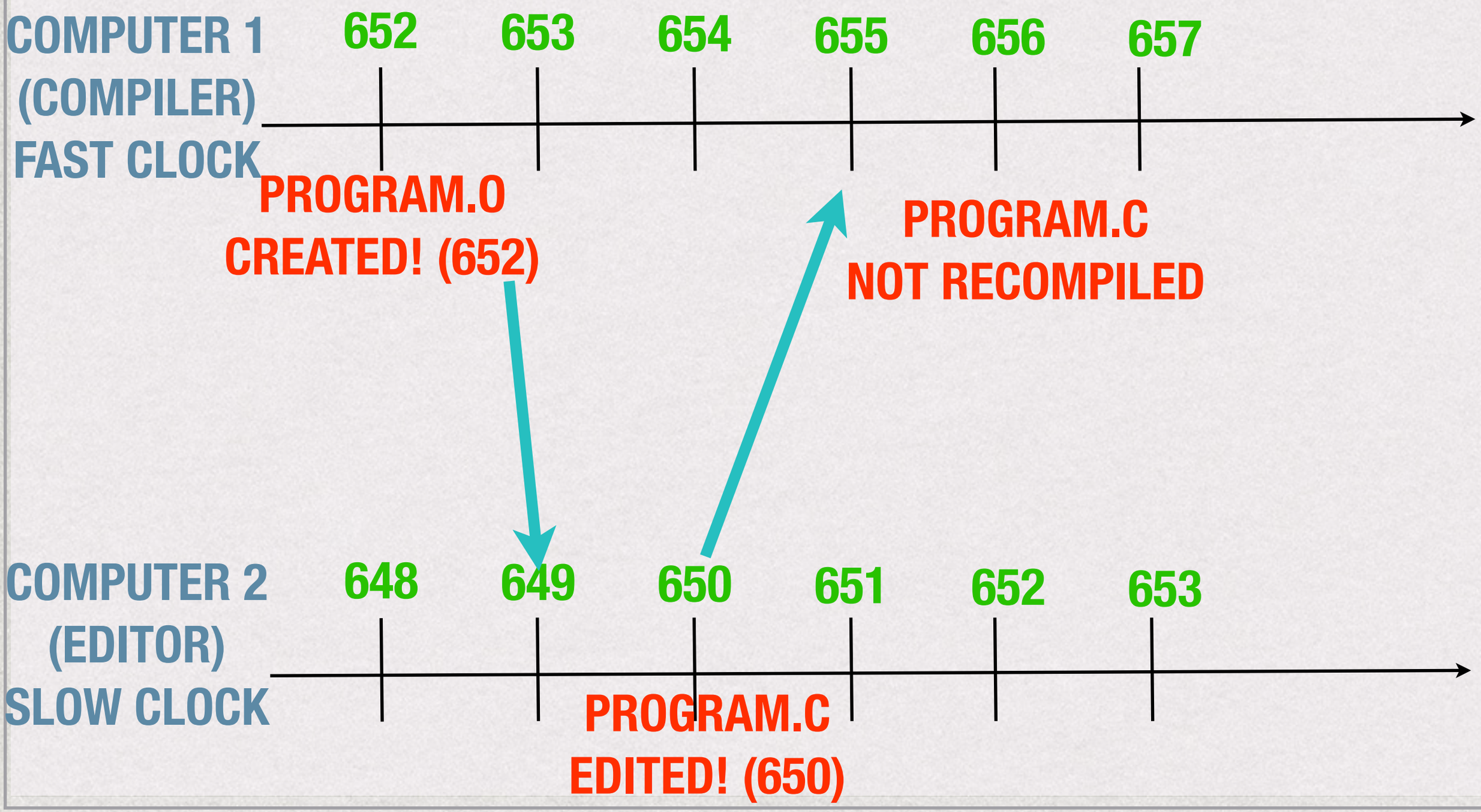
Lamport's Algorithm

- Each process i keeps a local clock, Li
- Three rules:

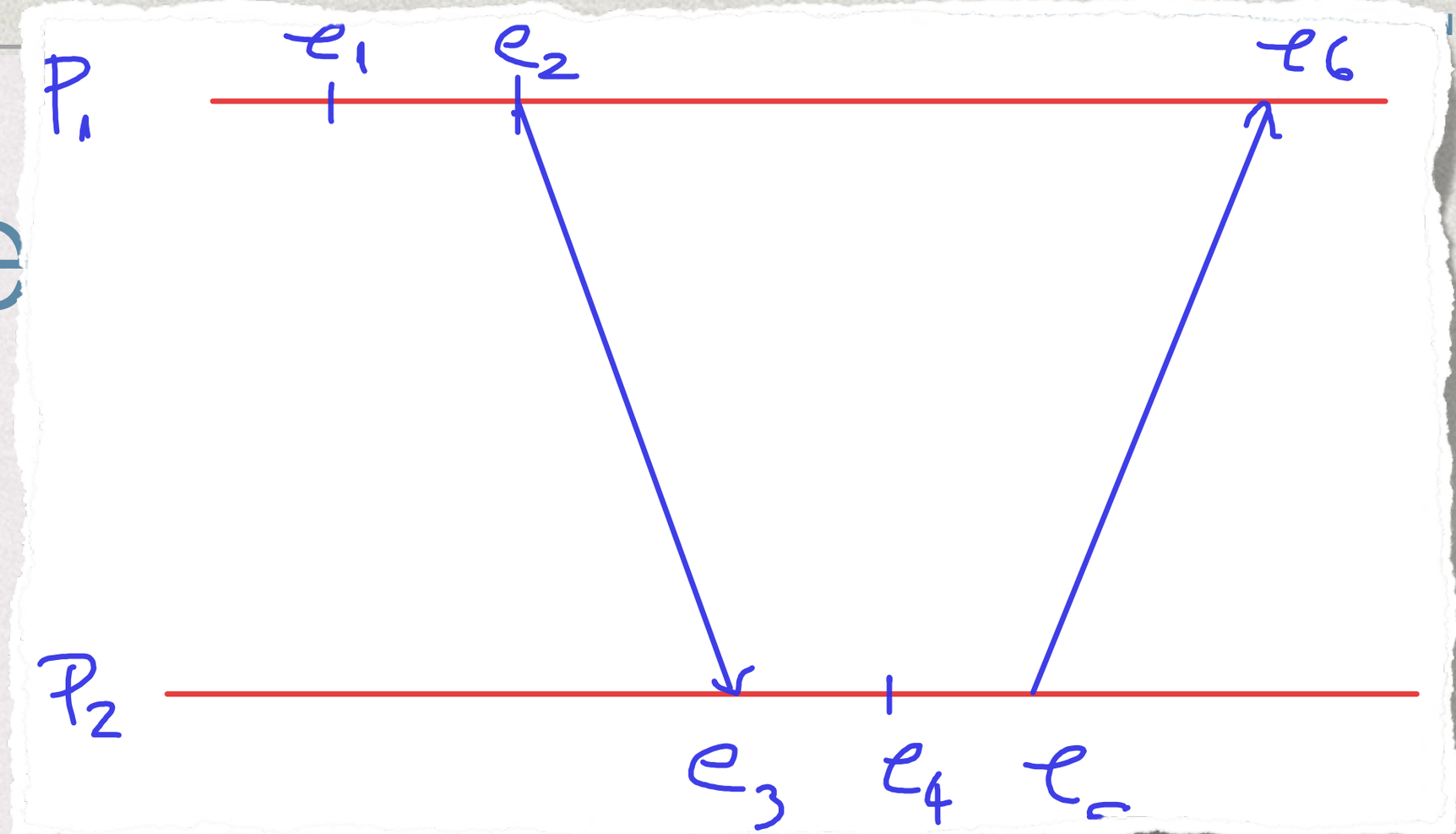
1. At process i, increment Li before each event
2. To send a message m at process i, apply rule 1 and then include the current local time in the message: i.e., send(m,Li)
3. When receiving a message (m,t) at process j, set Lj = max(Lj,t) and then apply rule 1 before time-stamping the receive event

- The global time L(e) of an event e is just its local time
- For an event e at process i, L(e) = Li(e)

Example


COMPUTER 1
(COMPILER)
FAST CLOCK
652
653
654
655
656
657
PROGRAM.O
CREATED! (652)
PROGRAM.C
NOT RECOMPILED
COMPUTER 2
(EDITOR)
SLOW CLOCK
648
649
650
651
652
653
PROGRAM.C
EDITED! (650)

Example



e_1 – program.o created on P_1

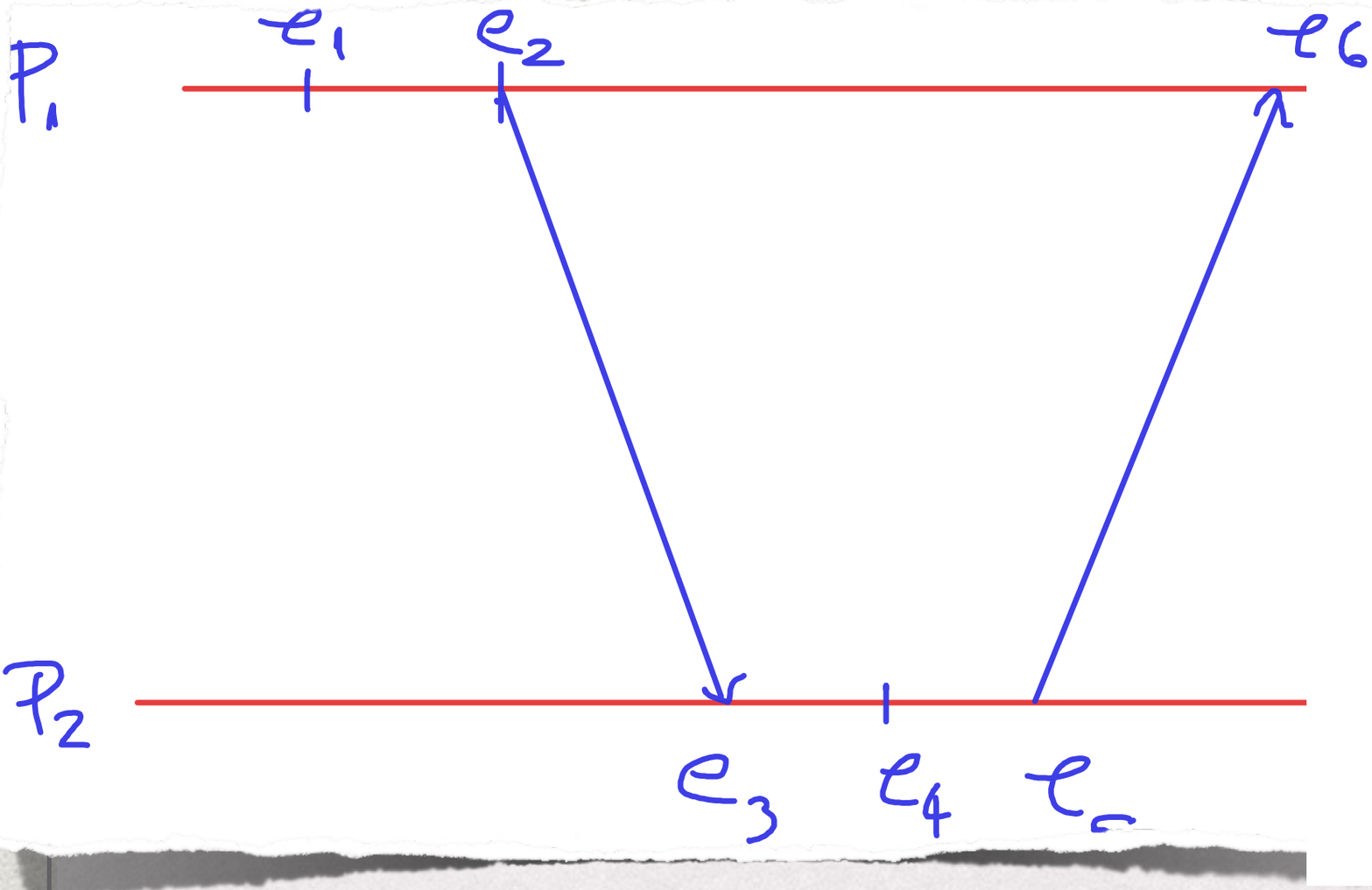
e_2 – message sent to P_2

e_3 – message arrives at P_2

e_4 – program.c edited on P_2

e_5 – message sent to P_1

e_6 – message arrives at P_1

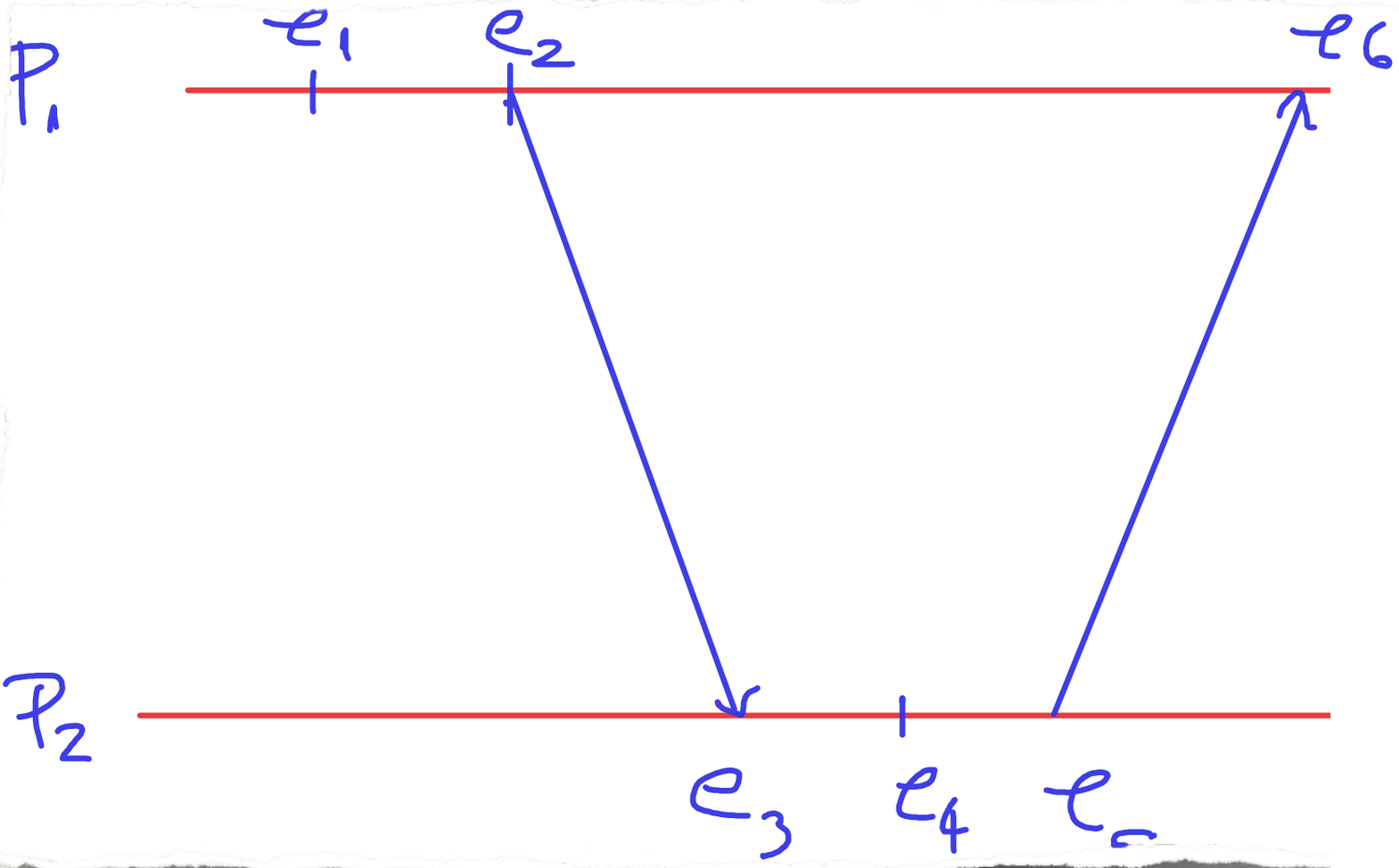


1. At process i, increment Li before each event

2. To send a message m at process i, apply rule 1 and then include the current local time in the message: i.e., send(m,Li)

3. When receiving a message (m,t) at process j, set Lj = max(Lj,t) and then apply rule 1 before time-stamping the receive event

At process 1: $L(P_1) = 1$ for e_1; $L(P_1) = 2$ for e_2; $send(m, 2)$;
At process 2: $L(P_2) = 1$ for e_3; $L(P_2) = max(1, 2) + 1 = 2 + 1$ for e_3;
At process 2: $L(P_2) = 4$ for e_4; $L(P_2) = 5$ for e_5; $send(m, 5)$;
At process 1: $L(P_1) = 3$ for e_6; $L(P_1) = max(3, 5) + 1 = 6$ for e_6
$\implies L(e_4) > L(e_1)$



1. At process i, increment Li before each event

2. To send a message m at process i, apply rule 1 and then include the current local time in the message: i.e., send(m,Li)

3. When receiving a message (m,t) at process j, set Lj = max(Lj,t) and then apply rule 1 before time-stamping the receive event

Or, even simpler:

$$e_1 \rightarrow e_2; e_2 \rightarrow e_3 \implies e_1 \rightarrow e_3;$$

$$e_3 \rightarrow e_4 \implies e_1 \rightarrow e_4 \implies L(e_4) \geq L(e_1)$$

Total order Lamport Clocks

- Many systems require a total ordering of events
- Use Lamport's algorithm, but break ties using process ID

$$L(e) = M * L_i(e) + i$$

  - M - maximum number of processes

Where are these used?!

- Anywhere where you would need to ensure (some) order of events
- Distributed mutual exclusion

Distributed Mutual Exclusion

- Maintain mutual exclusion among distributed processes
- Each process executes a loop

```
perform local ops

request critical section

perform CS ops

leave critical section

perform local ops
```

- During critical section, processes interact with other remote processes or directly with the shared resource
  - *send message to a shared file server asking it to write something to a file*

Distributed Mutual Exclusion: Goals

- Similar to regular mutual exclusion
- Safety
  - at most one process holds the lock at any time
- Liveness: progress
  - if nobody holds the lock, a processor requesting it will acquire it
- Fairness: bounded wait and in-order (in logical time!)
  - processes will not wait indefinitely;
  - will acquire locks in order of request

Distributed Mutual Exclusion: Performance Goals

- Minimize message traffic
  - distributed mutual exclusion is solved by sending messages
- Minimize synchronization delay
  - at most one process holds the lock at any time
- Assumptions
  - network is reliable but asynchronous
  - processes may fail at any time!!

Plan

- Before entering critical section, process must obtain permission from others (***Safety***)
- When exiting critical section, process must let others know that it has finished (***Liveness***)
- Process should allow others that have asked for permission earlier to enter the critical section (***Fairness***)

Centralized Lock Server

- To enter critical section
  - send **REQUEST** to central server
  - wait for permission
- To leave critical section
  - send **RELEASE** to central server
- Server
  - logs all requests in a queue
  - sends **OK** to process at head of queue

ADVANTAGES?

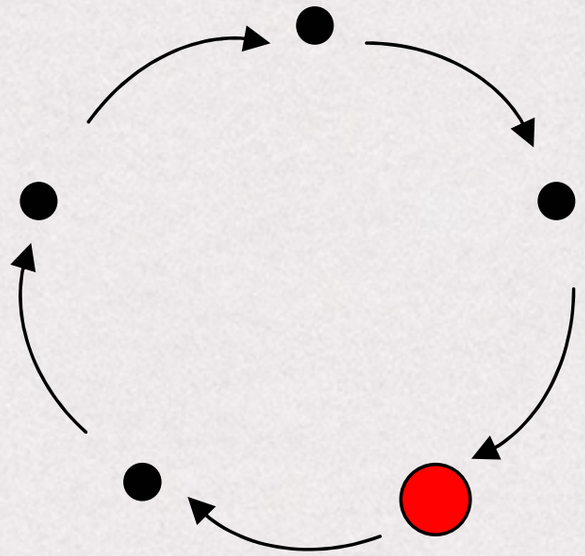
DISADVANTAGES?

Centralized Lock Server

- **Advantages**
  - Simple! YAY!
  - Only 3 messages required
- **Disadvantages**
  - Single point of failure; single performance bottleneck
  - Does not achieve fairness
  - Must elect central server

A ring-based algorithm

- Pass token around ring
  - can enter CS only if you hold token
- Problems
  - not in-order
  - long synchronization delay: need to wait N-1 messages for N processes
  - Unreliable: any process failure breaks the ring
- Can be improved by piggy-backing on the token with the time of the earliest known outstanding request



Lamport's algorithm

- Requesting process
  - Enters its **REQUEST** in its own queue (ordered by time stamps)
  - Sends a request to every node.
  - Wait for replies from all other nodes.
  - If own request is at the head of the queue and all replies have been received, enter critical section.
  - Upon exiting the critical section, send a **RELEASE** message to every process.

Lamport's algorithm

- Other process
  - If receiving **REQUEST** from process j:
    - enter it in own request queue ordered by time stamps;
    - if waiting for **REPLY** from j for an earlier request, wait until j replies
    - else **REPLY** with timestamp
- If receiving **RELEASE**, remove corresponding request from the queue
- If own request is at the head of the queue and all replies have been received, enter critical section.

Initial state

t	action
42	(start)

QUEUE P1:

t	action
11	(start)

QUEUE P2:

QUEUE P3:

t	action
14	(start)

P3 initiates request

t	action
42	(start)

QUEUE P1:

t	action
11	(start)

QUEUE P2:

QUEUE P3: <15.3>

t	action
14	(start)
15	request<15,3>

P1 P2 receive and reply

t	action
42	(start)
43	recv<15,3>
44	reply 1 to <15,3>

QUEUE P1: <15, 3>

t	action
11	(start)
16	recv<15,3>
17	reply 2 to <15,3>

QUEUE P2: <15,3>

QUEUE P3: <15,3>

t	action
14	(start)
15	request<15,3>

P3 receives replies

It's in front of its queue

Can enter CS

t	action
42	(start)
43	recv<15,3>
44	reply 1 to <15,3>

QUEUE P1: <15, 3>

t	action
11	(start)
16	recv<15,3>
17	reply 2 to <15,3>

QUEUE P2: <15,3>

QUEUE P3: <15,3>

t	action
14	(start)
15	request<15,3>
18	recv reply 2
45	recv reply 1
46	run CS

LAMPORT CLOCKS!!!

P1 and P2 initiate request

Concurrently!!

t	action
42	(start)
43	recv<15,3>
44	reply 1 to <15,3>
45	request <45,1>

QUEUE P1: <45,1>

t	action
11	(start)
16	recv<15,3>
17	reply 2 to <15,3>
18	request<18,2>

QUEUE P2: <18,2>

QUEUE P3:

t	action
14	(start)
15	request<15,3>
18	recv reply 2
45	recv reply 1
46	run CS

LAMPORT CLOCKS!!!

P3 gets requests and replies

t	action
42	(start)
43	recv<15,3>
44	reply 1 to <15,3>
45	request <45,1>
49	recv reply 3

QUEUE P1: <45,1>

t	action
11	(start)
16	recv<15,3>
17	reply 2 to <15,3>
18	request<18,2>
51	recv reply 3

QUEUE P2: <18,2>,

QUEUE P3: <18,2> <45,1>

t	action
14	(start)
15	request<15,3>
18	recv reply 2
45	recv reply 1
46	run CS
47	recv <45,1>
48	reply 3 to <45,1>
49	recv <18,2>
50	reply 3 to <18,2>

LAMPORT CLOCKS!!!

P2 gets request <45,1>

Delays reply because

<18,2> is an earlier request to which P1 has not replied

t	action
42	(start)
43	recv<15,3>
44	reply 1 to <15,3>
45	request <45,1>
49	recv reply 3

QUEUE P1: <45,1>

t	action
11	(start)
16	recv<15,3>
17	reply 2 to <15,3>
18	request<18,2>
51	recv reply 3
52	recv <45,1>

QUEUE P2: <18,2>, <45,1>

QUEUE P3: <18,2> <45,1>

t	action
14	(start)
15	request<15,3>
18	recv reply 2
45	recv reply 1
46	run CS
47	recv <45,1>
48	reply 3 to <45,1>
49	recv <18,2>
50	reply 3 to <18,2>

LAMPORT CLOCKS!!!

P1 gets <18,2>

Replies

t	action
42	(start)
43	recv<15,3>
44	reply 1 to <15,3>
45	request <45,1>
49	recv reply 3
50	recv<18,2>
51	reply 1 to <18,2>

QUEUE P1: <18,2>,<45,1>

t	action
11	(start)
16	recv<15,3>
17	reply 2 to <15,3>
18	request<18,2>
51	recv reply 3
52	recv <45,1>
53	recv reply 1
54	reply 2 to <45,1>
55	run CS

QUEUE P2: <18,2>, <45,1>

QUEUE P3: <18,2> <45,1>

t	action
14	(start)
15	request<15,3>
18	recv reply 2
45	recv reply 1
46	run CS
47	recv <45,1>
48	reply 3 to <45,1>
49	recv <18,2>
50	reply 3 to <18,2>

LAMPORT CLOCKS!!!

Lamport's Algorithm

- **Advantages**
  - Fair
  - short sychronization delay
- **Disadvantages**
  - any process failure halts progress
  - 3*(N-1) messages per entry/exit