Time, Clocks, and the Ordering of Events in a Distributed System

By Leslie Lamport

Presented by Ben Landrum 10/7/25

Agenda

Leslie Lamport

Why we care about ordering events

Logical Clocks

Resource Exclusion

Physical Clocks

Clock Synchronization

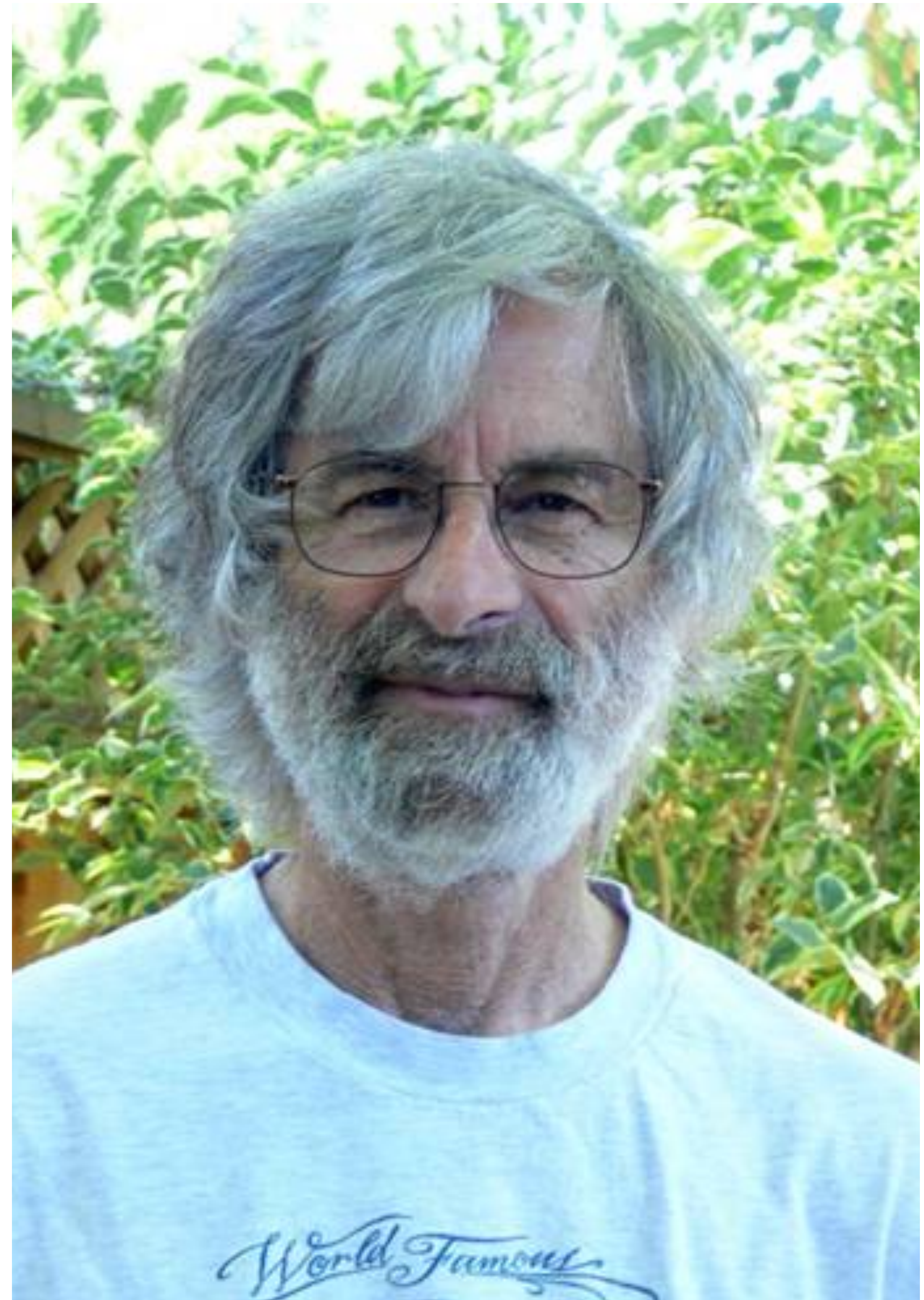
Network Time Protocol

Leslie Lamport

b.1941, PhD at Brandeis

Net worth: ?

- Developed many foundational ideas in distributed systems:
  - Logical clocks (this paper)
  - Paxos (won the Turing award primarily for this)
  - Byzantine generals problem
  - Sequential consistency
- Started LaTeX
- Worked in a handful of industry labs until retiring from MSR earlier this year



Ordering Events in Distributed Systems

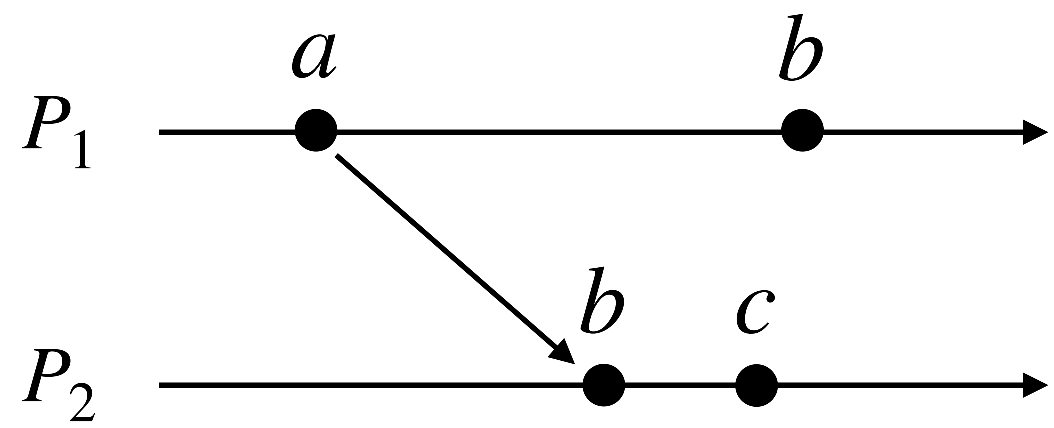
Why do we care?

- There are many applications where you care about the ordering of events
    - Financial system
    - Laser tag
    - Resource exclusion

Where was the internet in 1978?

- ARPANET started in 1969
- TCP/IP proposed in 1974, wouldn’t be the official packet switching protocol on ARPANET until 1983
- This is the start of distributed systems

The Partial Ordering



$\boldsymbol{a \rightarrow b}$

- If a happens before b on the same process, $a \rightarrow b$
- If a is the sending of a message and b is the receipt of that message, $a \rightarrow b$
- Transitive, such that $a \rightarrow b$ and $b \rightarrow c$ implies $a \rightarrow c$

- Not reflexive, such that $a \nrightarrow a$
- We say that two events a and b such that $a \nrightarrow b$ and $b \nrightarrow a$ are *concurrent*

Logical Clocks

How can we observe the partial ordering?

Clock: a function C from events to timestamps such that

$a \rightarrow b$ implies $C\langle a \rangle < C\langle b \rangle$

Logical Clocks

The *Clock Condition*

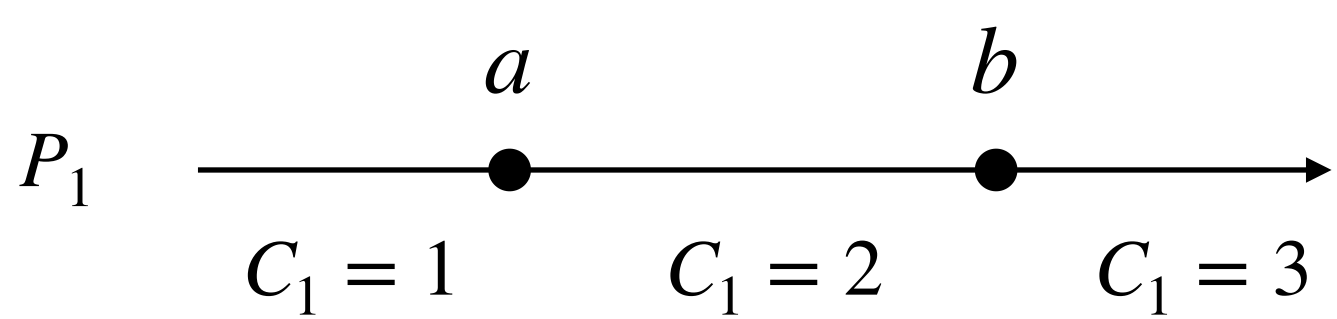
C1:

two events $a \rightarrow b$ in the same process P_i satisfy

$$C_i\langle a\rangle < C_i\langle b\rangle$$

IR1:

Every time an event (including communication) happens on process P_i, increment C_i



$$a \rightarrow b$$

Logical Clocks

The *Clock Condition*

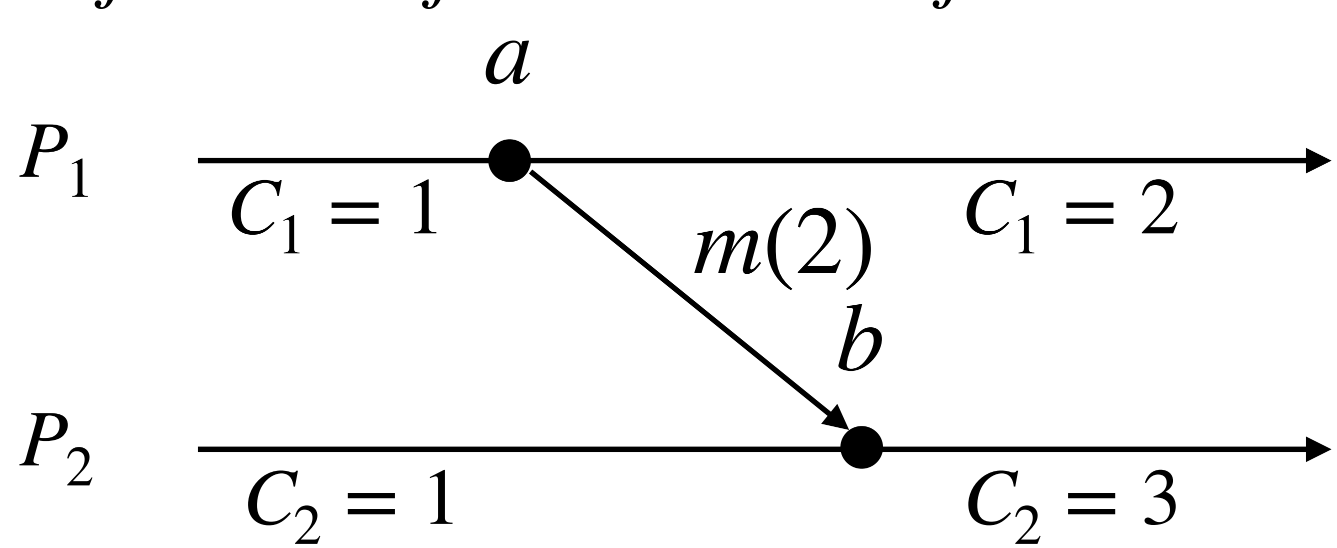
C2:

if a is the sending of a message from P_i, and b is the receipt of that message by P_j, $C_i\langle a\rangle < C_j\langle b\rangle$

IR2:

(a) every message contains a timestamp from the sender

(b) P_j sets $C_j := \max(C_j\langle b\rangle, C_i\langle a\rangle + \epsilon)$



$a \rightarrow b$

Discussion

- Are logical clocks robust to Byzantine faults? If not, what do we lose when there are adversarial timestamps being sent?
- Does it matter whether you increment the clock before vs after an event?

Making this a total ordering

- We want to define a tiebreaker for a total ordering $\Rightarrow$
  - This allows all processes to agree on the ordering
  - e.g. if $i < j$ then $C_i\langle a \rangle = C_j\langle b \rangle$ implies $b \Rightarrow a$

Resource Exclusion

Premise

A set of processes share a resource, but only one process can use it at a time

We want an algorithm that ensures:

- The resource has to be released by the holder for it to change hands
- Requests are granted in the order they're made
- If every lease is eventually released, every request is eventually granted

Resource Exclusion

Assumptions

Messages are received in the order they're sent

All messages are eventually delivered

Resource Exclusion

Setup

A

$C_A = 1$

Req(0, **A**)

queue_A

B

$C_B = 2$

Req(0, **A**)

queue_B

C

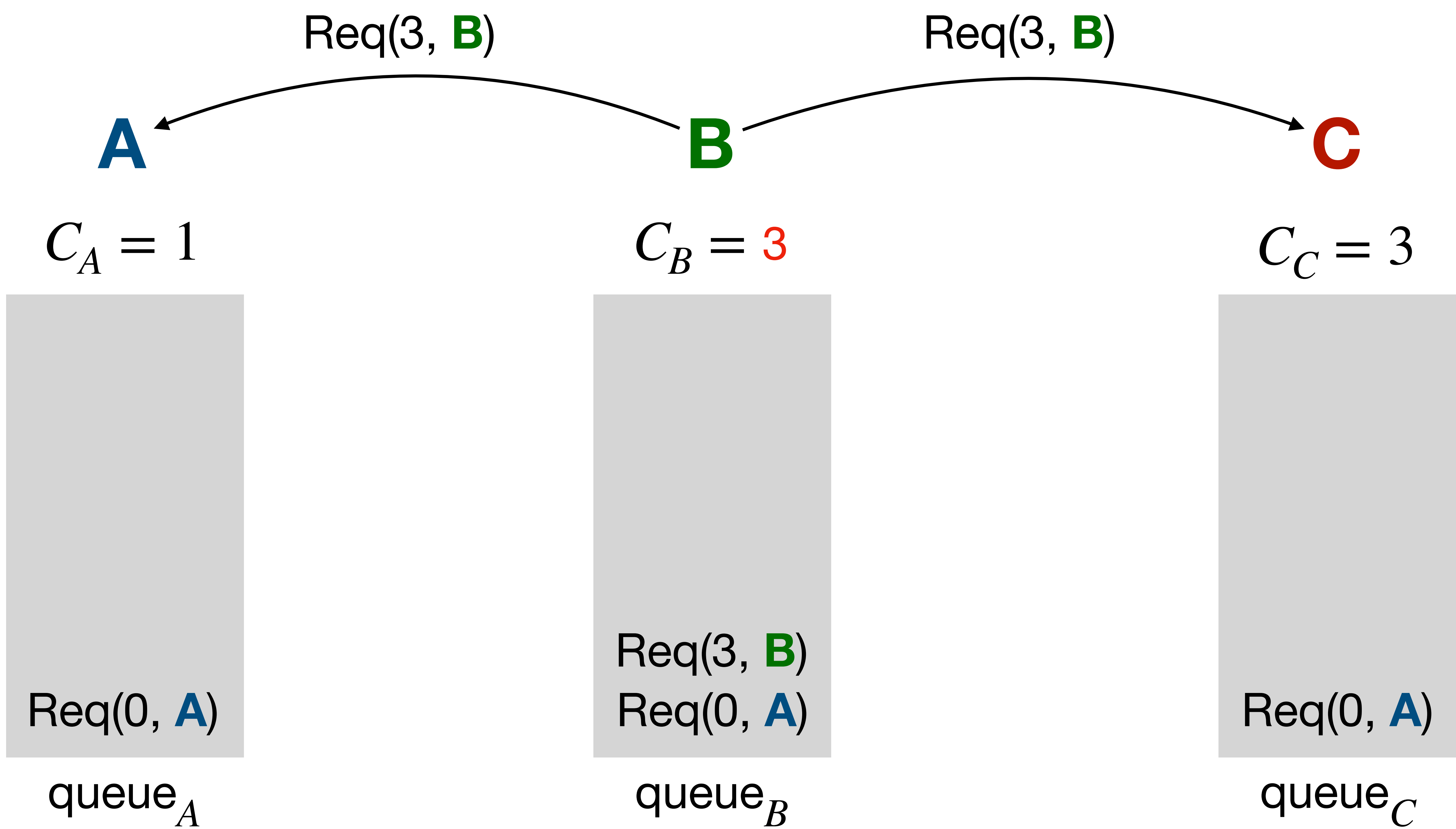
$C_C = 3$

Req(0, **A**)

queue_C

Resource Exclusion

B requests the resource



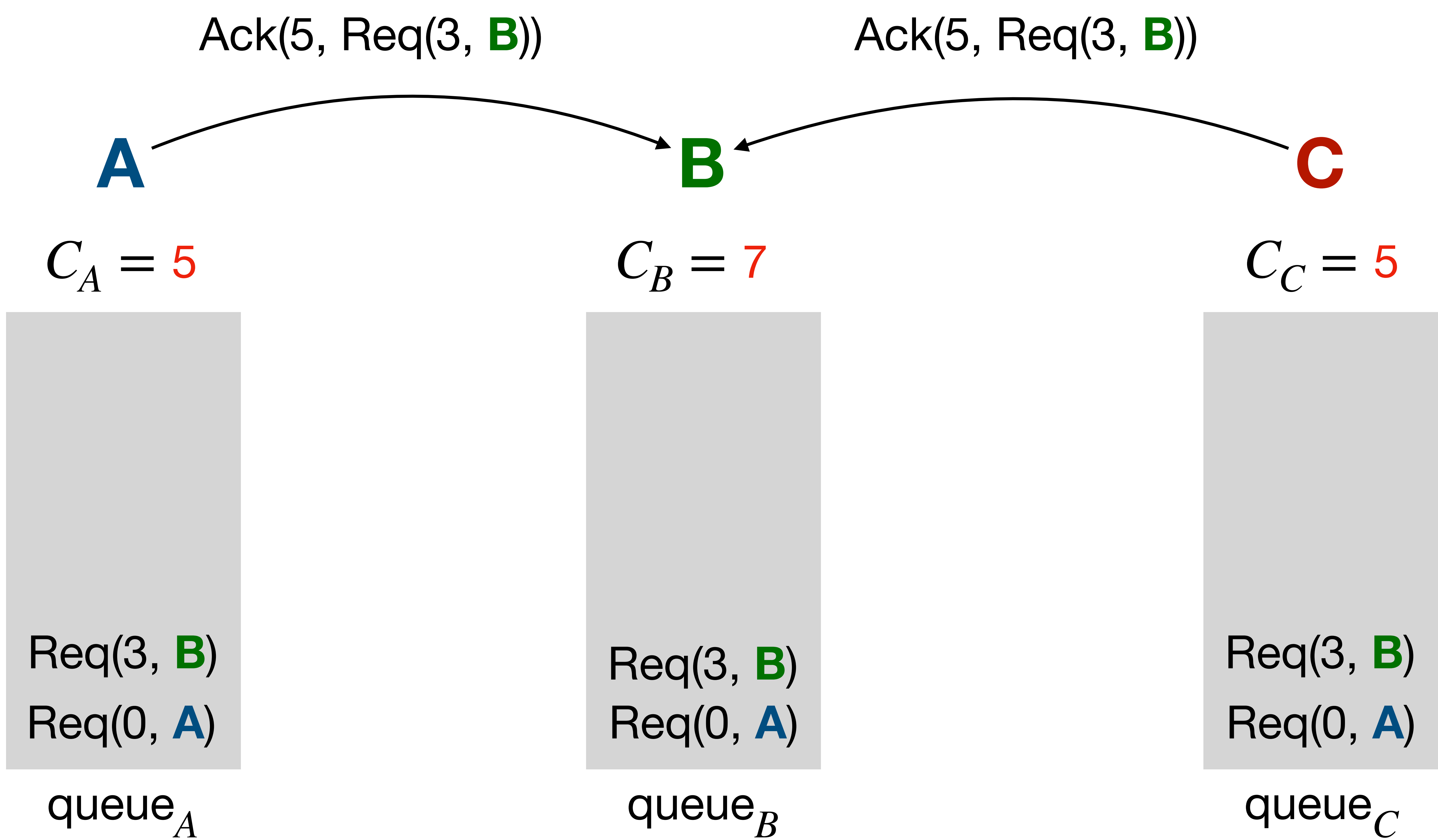
Resource Exclusion

A and C receive the request

A	B	C
$C_A = 4$	$C_B = 3$	$C_C = 4$
Req(3, **B**)	Req(3, **B**)	Req(3, **B**)
Req(0, **A**)	Req(0, **A**)	Req(0, **A**)
queue_A	queue_B	queue_C

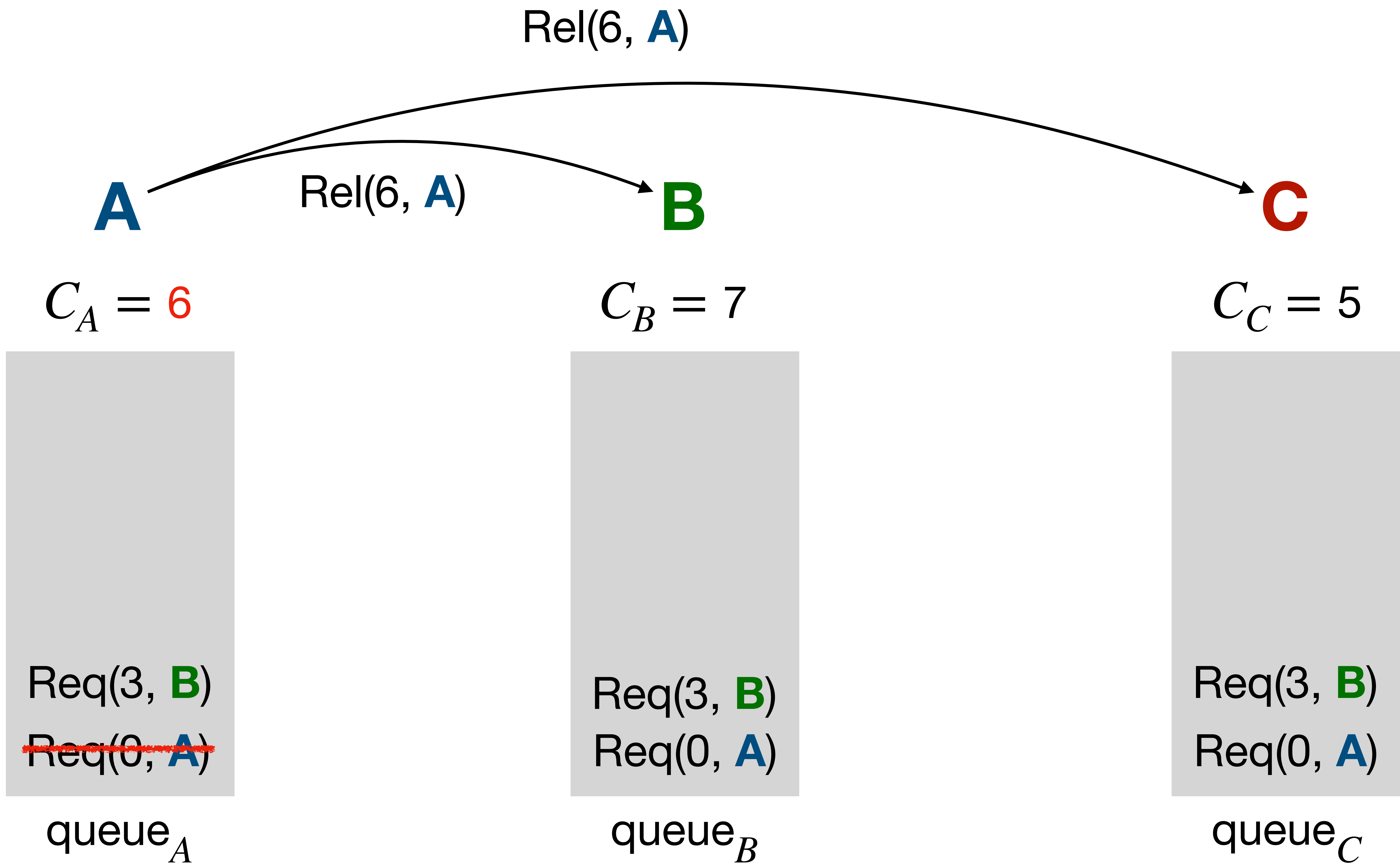
Resource Exclusion

A and C acknowledge the request



Resource Exclusion

A releases the resource



Resource Exclusion

B and C receive the release & B starts using the resource

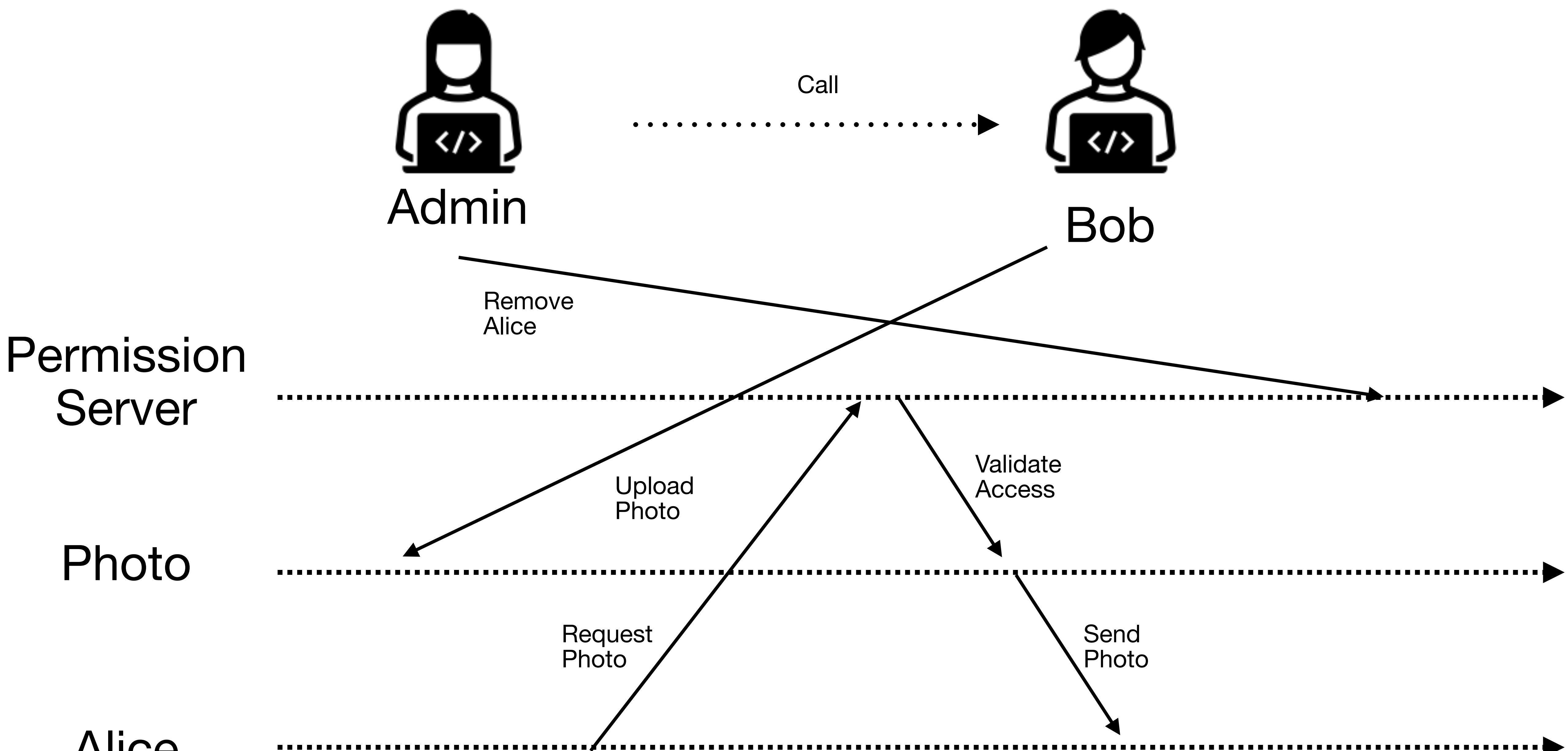
A	**B**	**C**
$C_A = 6$	$C_B = 8$	$C_C = 7$
Req(3, **B**)	Req(3, **B**)	Req(3, **B**)
~~Req(0, **A**)~~	~~Req(0, **A**)~~	~~Req(0, **A**)~~
queue_A	queue_B	queue_C

Logical clocks now

- Vector clocks (1988)
  - Each process tracks a logical clock for every other process, and sends a ‘vector’ of clocks as its timestamp
  - Gives a complete view of which events are causally dependent

A weakness of ⇒

We often care about the *actual* ordering of events



The Strong Clock Condition

Ensuring total ordering

- Let $\twoheadrightarrow$ be the partial ordering defined by the *actual* order that events in the system happened in
- Events outside the system (such as the phone call) could be placed in the order
- Clocks which track *physical* time could provide such an ordering

Important definitions

Offset: In absolute terms, the difference between $C_i(t)$ and $C_j(t)$

"How far off are our clocks?"

Skew: The difference between $\frac{dC_i(t)}{dt}$ and $\frac{dC_j(t)}{dt}$

"How much faster is your clock than mine?"

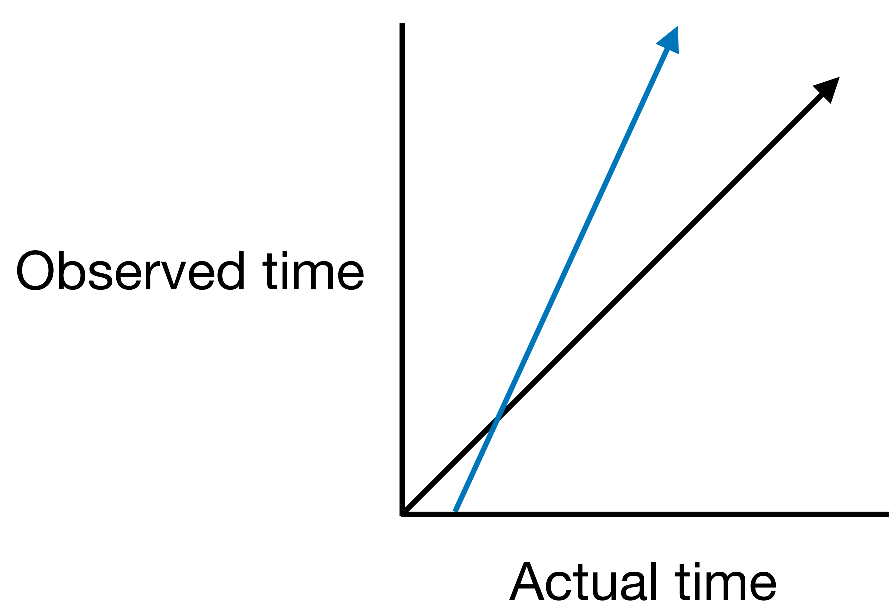
Physical Clocks

A reasonable set of properties

Let C_i be the continuous, differentiable clock kept by process P_i

C_i should progress forward at ~the same rate as actual time (have small **skew**)

PC1: $\exists \kappa \ll 1$ such that $\forall i, \left| \frac{dC_i(t)}{dt} - 1 \right| < \kappa$

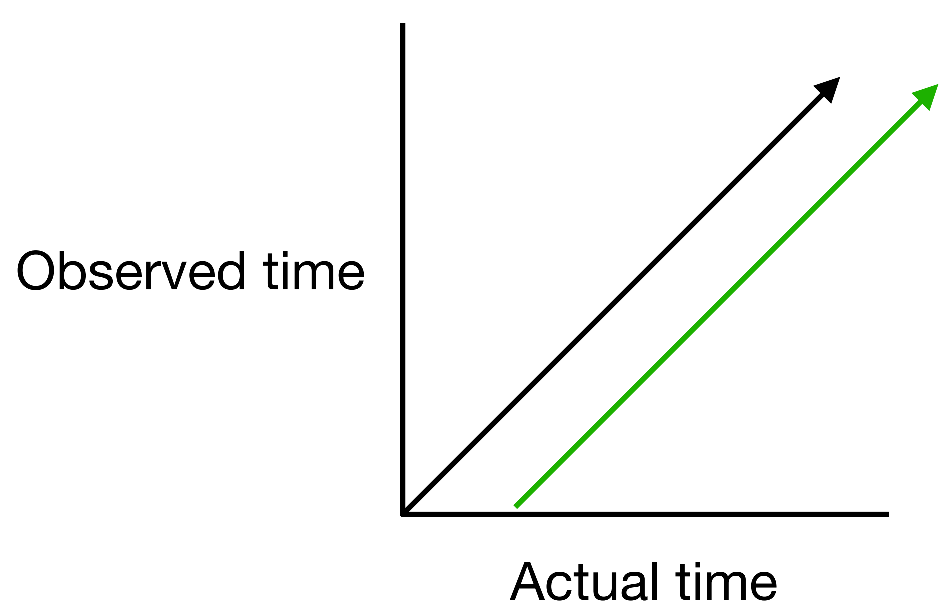


Physical Clocks

A reasonable set of properties

All the C_i should ~agree on what time it is (have small **offset**)

PC2: For some small constant ϵ, $\forall i, j, |C_i(t) - C_j(t)| < \epsilon$



Ensuring clocks respect $\twoheadrightarrow$

And thereby avoid 'anomalous behavior'

- Let μ be a time shorter than the minimum latency of a message
- $C_i(t+\mu) - C_j(t) > 0$ must hold for all i, j.
- This gives us the κ and ϵ we need for our physical clocks to respect the physical ordering: $C_i(t+\mu) - C_j(t) > (1-\kappa)\mu$ implies $\epsilon(1-\kappa) \leq \mu$

Update rules for physical clocks

Modified from logical clock rules

IR1:

Every time an event (including communication) happens on process P_i, increment C_i

becomes

IR1':

Any time P_i is not receiving a message, C_i is differentiable and $\frac{dC_i(t)}{dt} > 0$, such that its time is moving forward

Update rules for physical clocks

Modified from logical clock rules

IR2:

(a) every message contains a timestamp from the sender

(b) P_j sets $C_j := \max(C_j\langle b \rangle, C_i\langle a \rangle + \epsilon)$

becomes

IR2':

(a) every message contains a timestamp from the sender

(b) upon receiving a message with timestamp t_m at time t', P_j sets $C_j := \max(C_j(t'), t_m + \mu_m)$, where μ_m is a lower bound on the latency of the message.

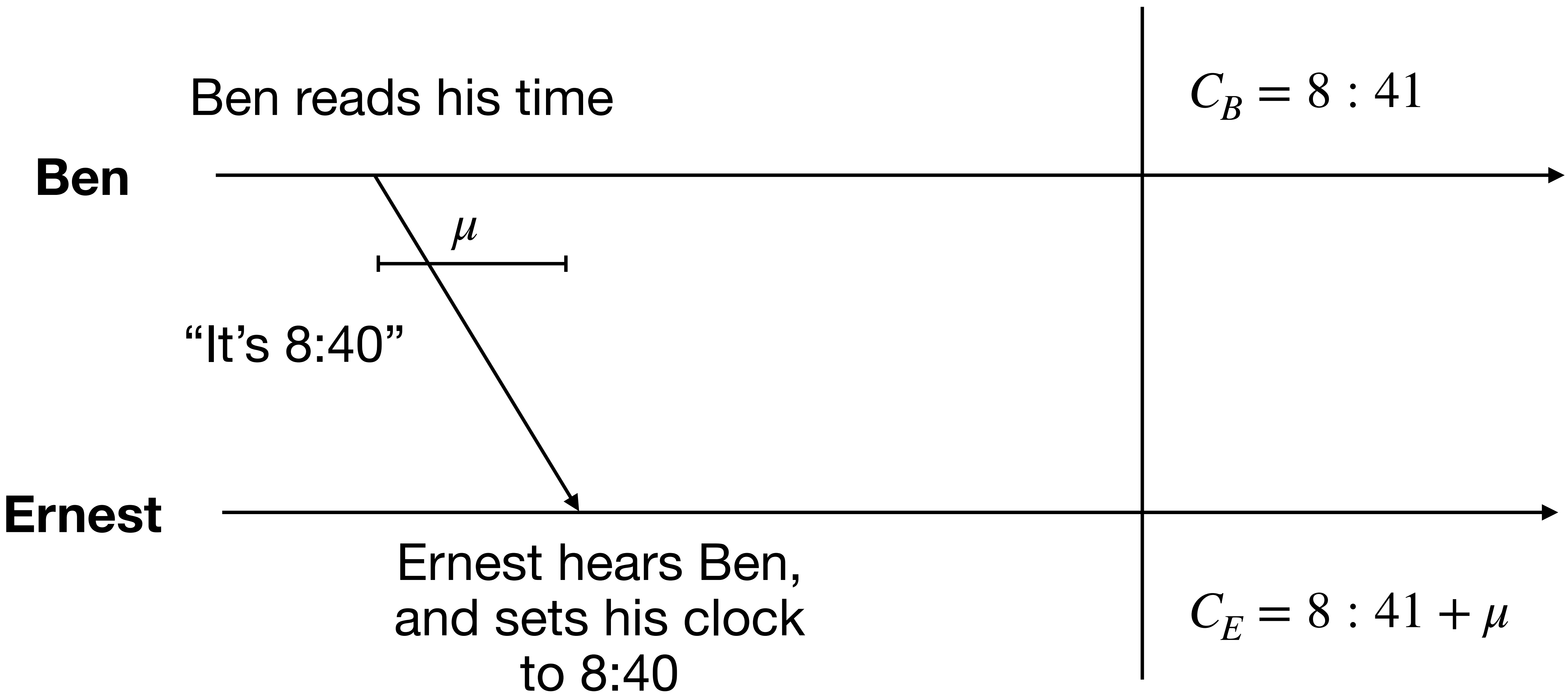
Discussion

- Why do clocks have to be monotonic?

Network Time Protocol (NTP)

Why it's hard to synchronize clocks

A minimal 2 party example



Why it's hard to synchronize clocks

We can't measure μ!

- It's often impossible to measure 1 way latency
- We can only measure the **round trip latency**
- **This even extends to the speed of light**

Prior art

“What time is it?”

- Timestamp broadcasts by radio
- NIST Automated Computer Time Service (ACTS) (1988)
- Many standards for sending a timestamp over the internet
    - IP suite daytime protocol
    - IP suite time protocol
    - ICMP timestamp protocol
- Unix timed daemon keeps in sync with a master clock

Discussion

- What's wrong with the radio broadcast approach?
- What about GPS made it unsuitable for precise calibration of clocks until the Clinton administration? What about now?

NTP Overview

1. Send and receive NTP packets from peers in your NTP subnet

2. Collect several observations from each peer, and take the lowest offset as the most reliable measurement, recording things like jitter as indicators of peer quality

3. Filter out untrustworthy or unreliable peers

4. Use the ***offset*** between your clock and a weighted average of your trustworthy peers to adjust your ***skew***

$$T_{i-2} \approx T_{i-3} + \frac{\delta}{2} + \theta$$

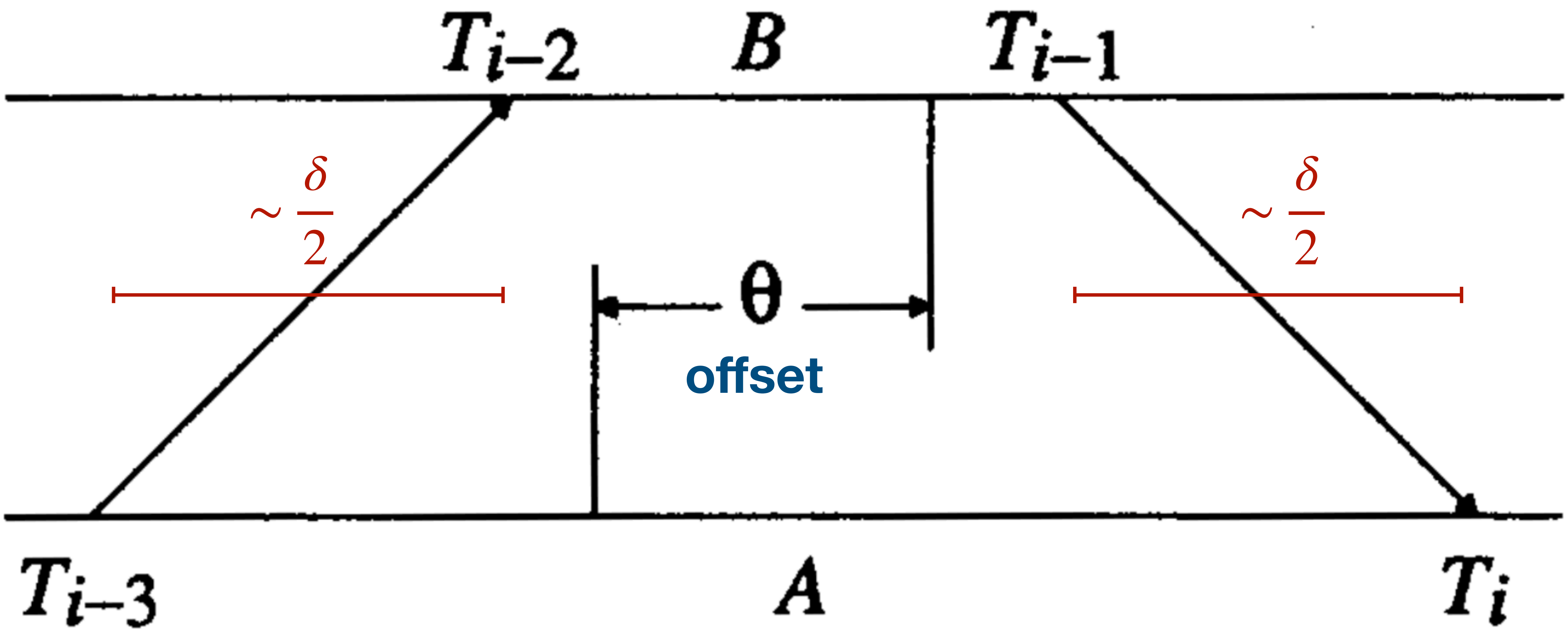


Fig. 3. Measuring delay and offset. $T_i \approx T_{i-1} + \frac{\delta}{2} - \theta$

$$\theta_{\text{measured}} = \frac{(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)}{2}$$

Adjusting your clock

- When adjusting, we can’t set our clock to a lower value
- Instead of changing the value outright, we tweak our **skew** up and down so our **offset** to the consensus of our peers will approach 0
- The bigger our **offset**, the more we change our **skew**
- Physically, the clock hardware is counting some physical phenomenon, and we adjust how many of that phenomenon are in a second

Discussion

- NTP packets contain T_{i-3}, T_{i-2}, and T_{i-1}. Why is T_{i-3} needed?
- What changes could you make to the networking hardware to make clock synchronization more precise?
- At what level of the stack (NIC/driver/kernel/user space) should the timestamp for a message be computed?