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

Leslie Lamport

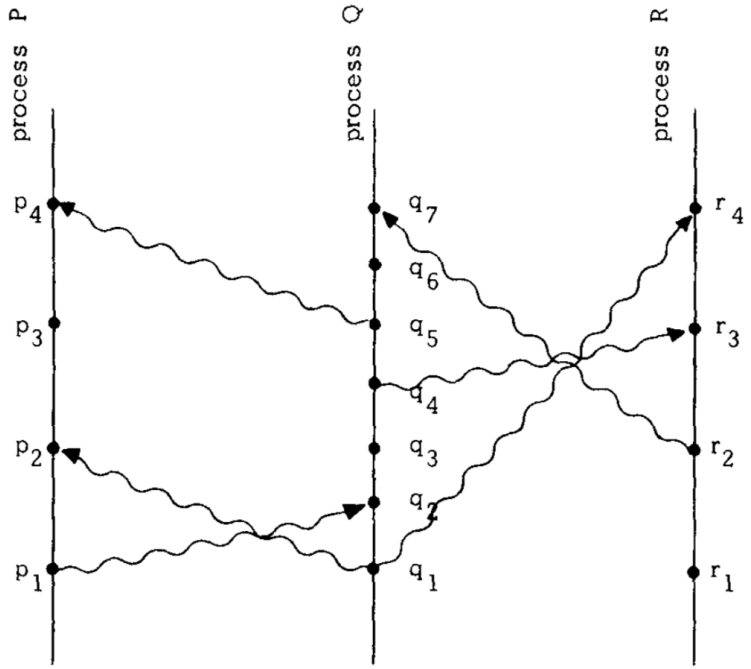
Yuncong Zhang

April 23, 2020

Outline

Events in Distributed System

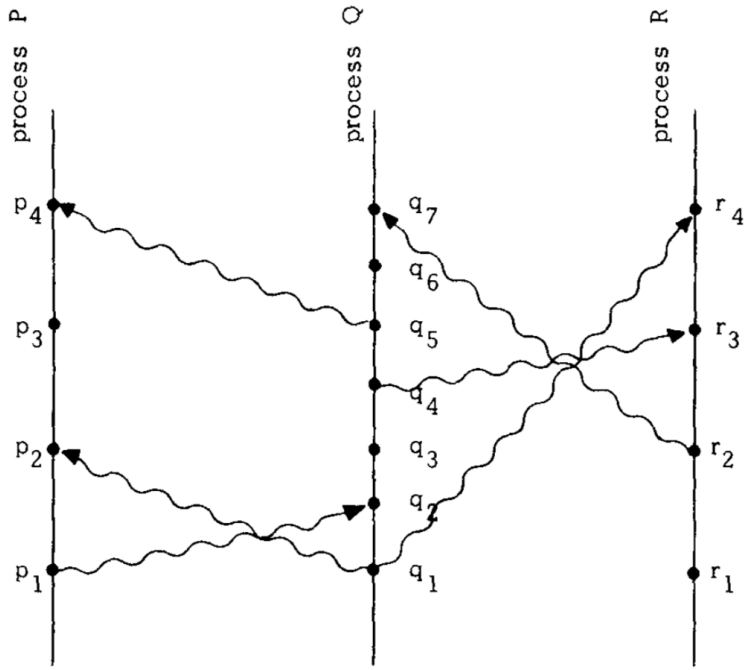
What does it mean by "*a* happens before *b*" in distributed system?



Events in Distributed System

What does it mean by "*a* happens before *b*" in distributed system?

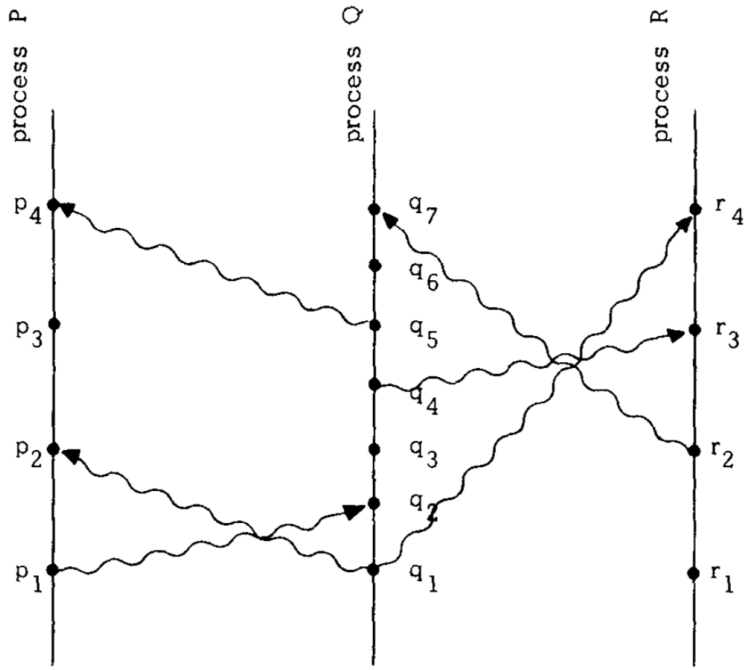
- Hard to define across asynchornized processes



Events in Distributed System

What does it mean by "*a* happens before *b*" in distributed system?

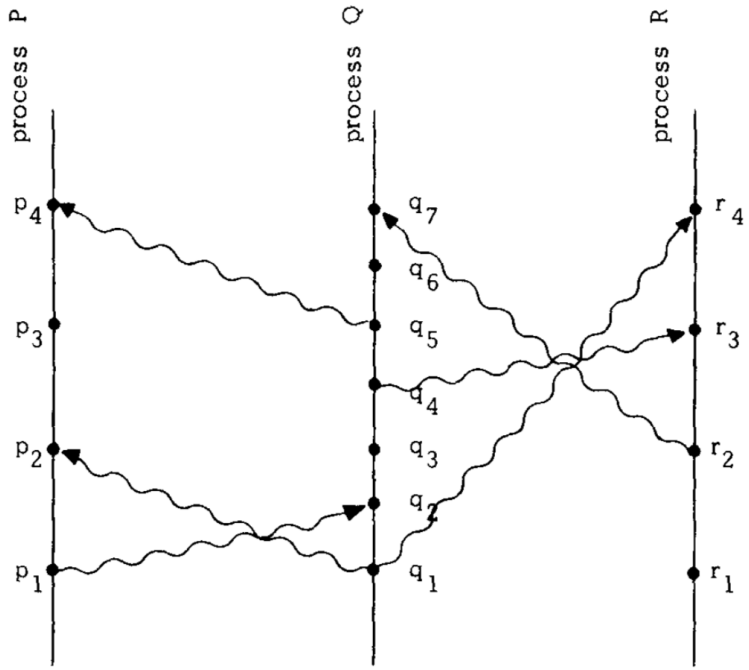
- Hard to define across asynchornized processes
- In one process, **earlier** events happens before **later** events



Events in Distributed System

What does it mean by "*a* happens before *b*" in distributed system?

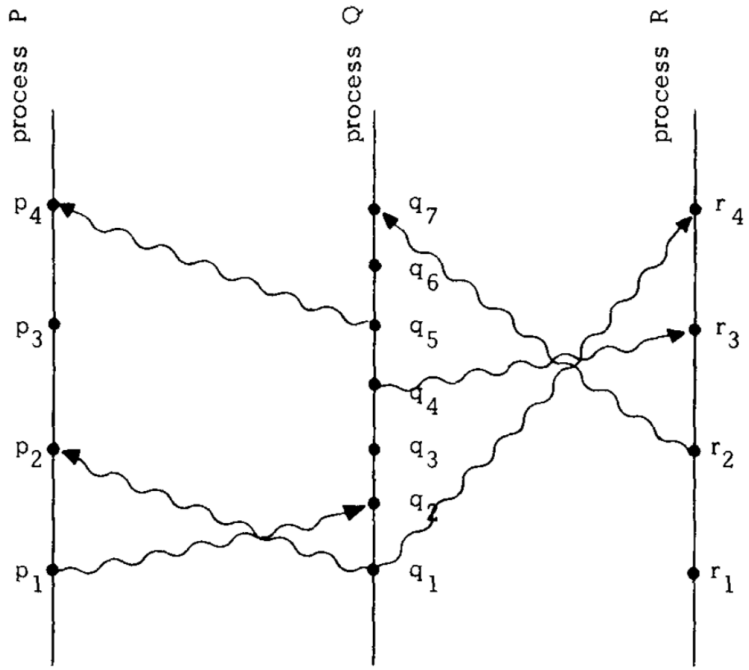
- Hard to define across asynchornized processes
- In one process, **earlier** events happens before **later** events
- **Sending message** happens before **receiving message**



Events in Distributed System

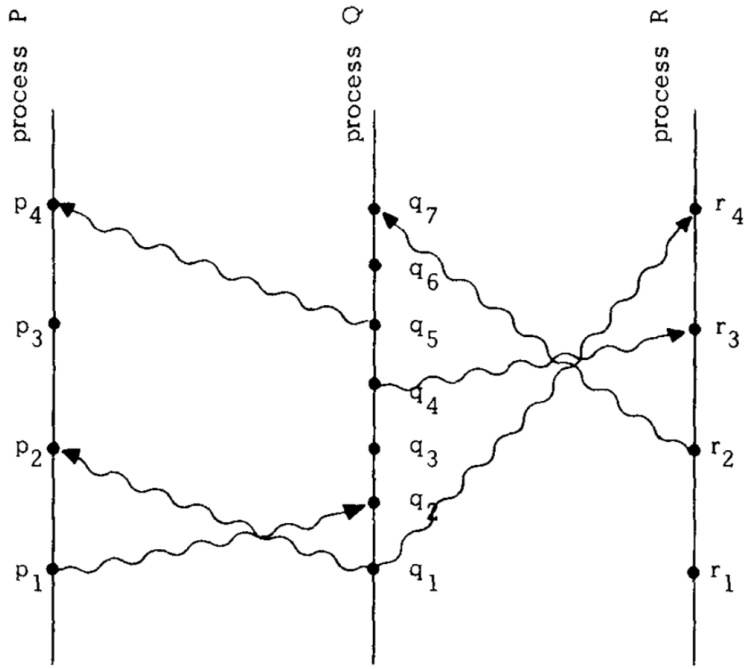
What does it mean by "*a* happens before *b*" in distributed system?

- Hard to define across asynchornized processes
- In one process, **earlier** events happens before **later** events
- **Sending message** happens before **receiving message**
- If *a* happens before *b*, and *b* happens before *c*, then *a* happens before *c*



Partial Ordering of Events

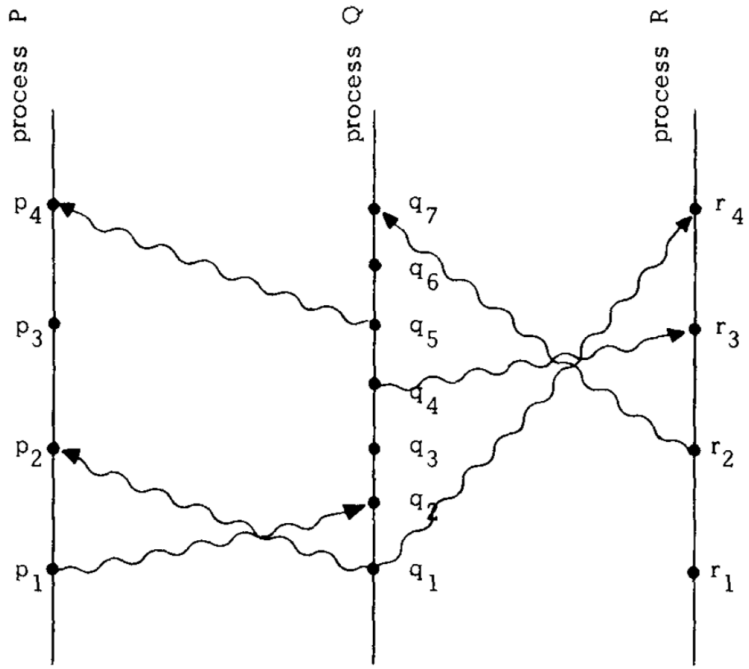
Denote by $a \rightarrow b$ if a happens before b.



Partial Ordering of Events

Denote by $a \rightarrow b$ if a happens before b.

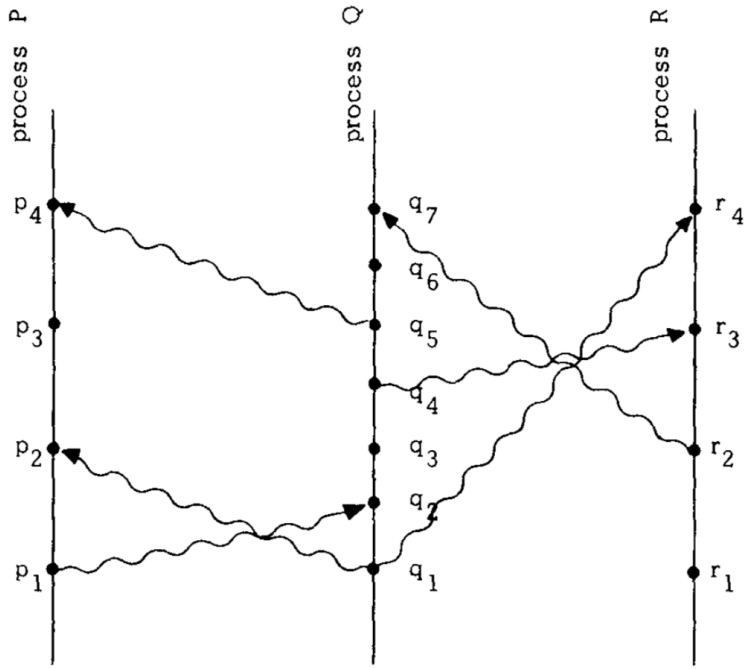
- "$\rightarrow$" defines a partial order: not all pairs of events are ordered



Partial Ordering of Events

Denote by $a \rightarrow b$ if a happens before b.

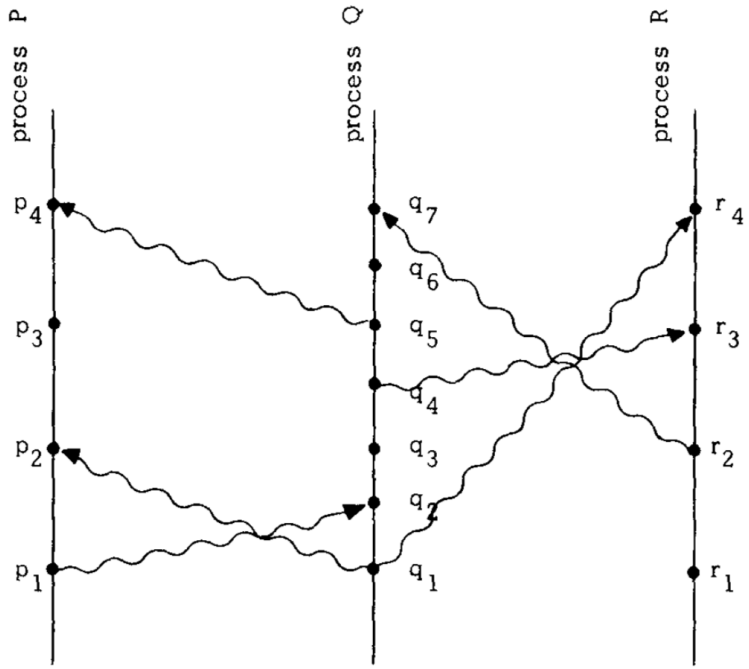
- "$\rightarrow$" defines a partial order: not all pairs of events are ordered
- If $a \nrightarrow b$ and $b \nrightarrow a$ then a and b are *concurrent*



Partial Ordering of Events

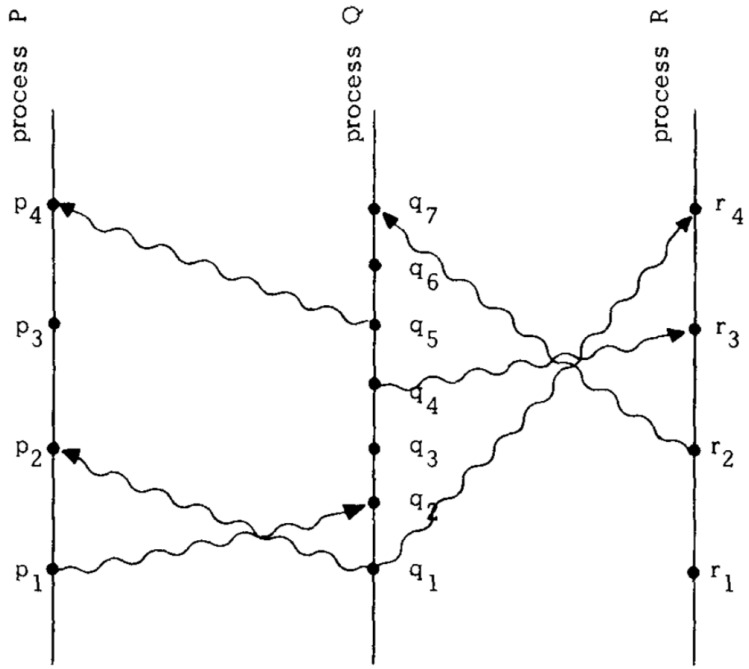
Denote by $a \rightarrow b$ if a happens before b.

- "$\rightarrow$" defines a partial order: not all pairs of events are ordered
- If $a \nrightarrow b$ and $b \nrightarrow a$ then a and b are *concurrent*
- $a \rightarrow b$ is equivalent to saying one can go from a to b in the diagram by moving forward in time along process and message lines.



Partial Ordering of Events

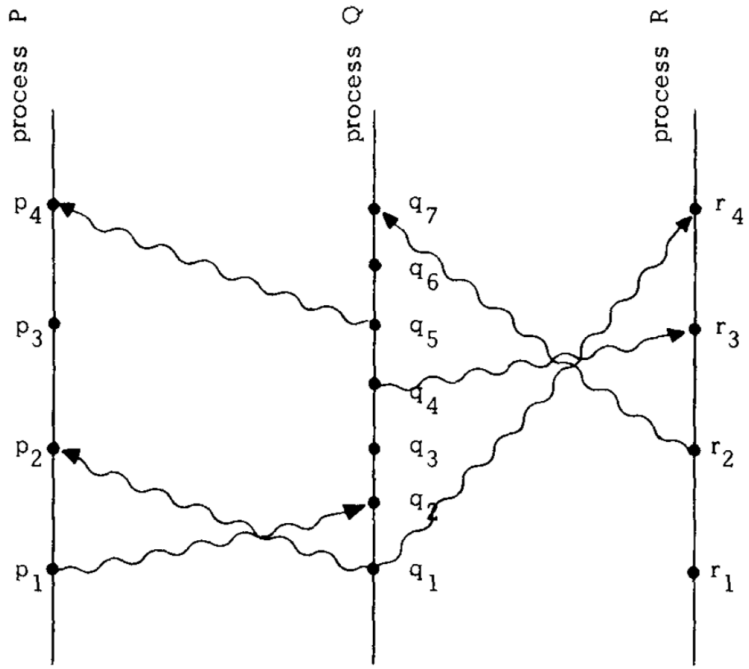
Example: $p_1 \rightarrow r_4$



Partial Ordering of Events

Example: $p_1 \rightarrow r_4$

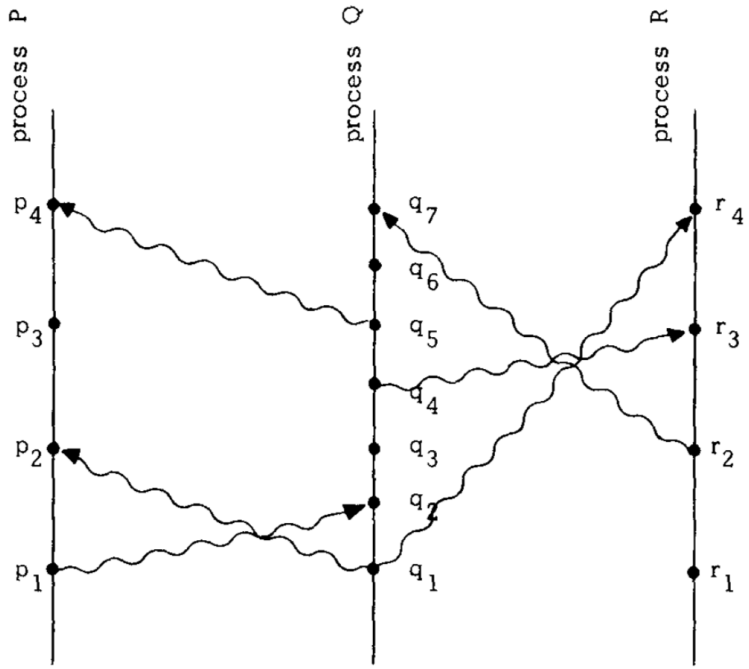
- $p_1 \rightarrow q_2$



Partial Ordering of Events

Example: $p_1 \rightarrow r_4$

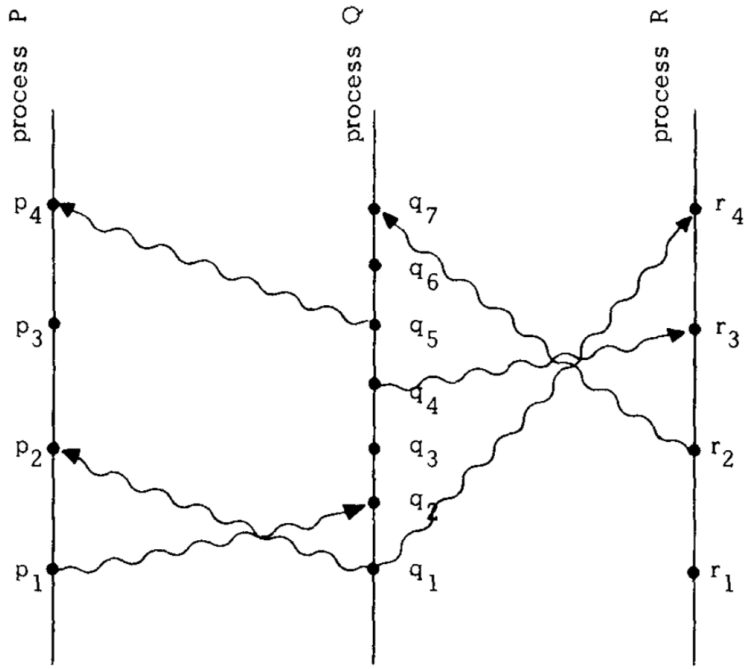
- $p_1 \rightarrow q_2$
- $q_2 \rightarrow q_4$



Partial Ordering of Events

Example: $p_1 \rightarrow r_4$

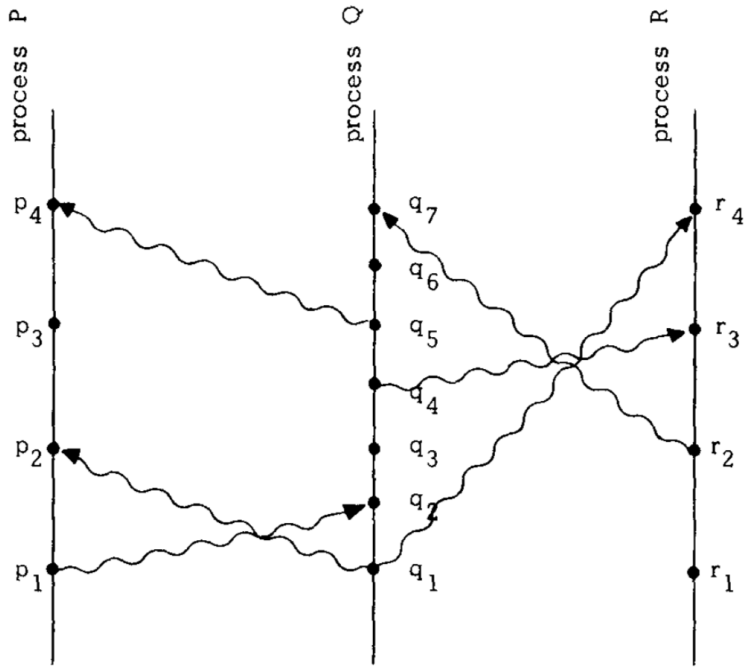
- $p_1 \rightarrow q_2$
- $q_2 \rightarrow q_4$
- $q_4 \rightarrow r_3$



Partial Ordering of Events

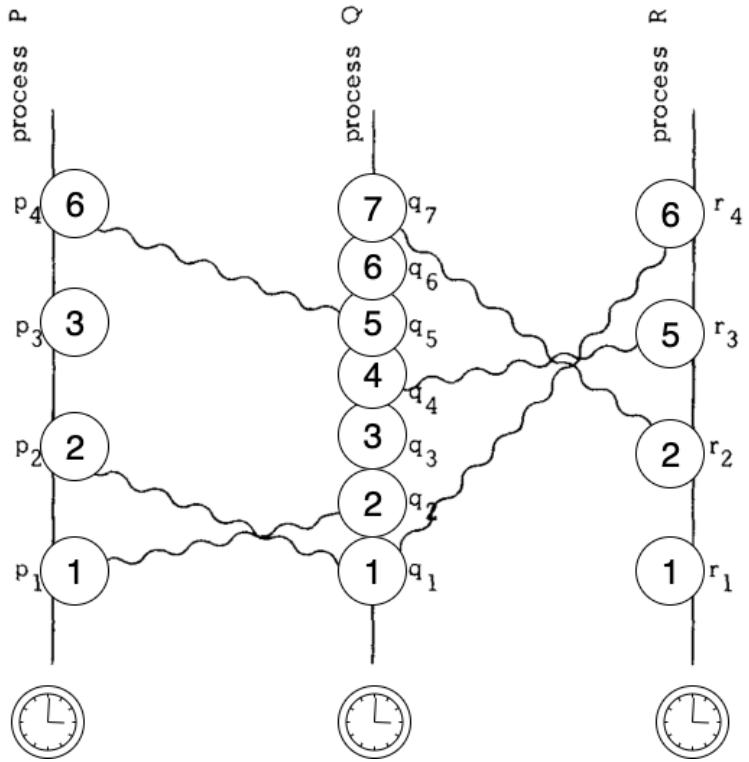
Example: $p_1 \rightarrow r_4$

- $p_1 \rightarrow q_2$
- $q_2 \rightarrow q_4$
- $q_4 \rightarrow r_3$
- $r_3 \rightarrow r_4$



Logical Clocks

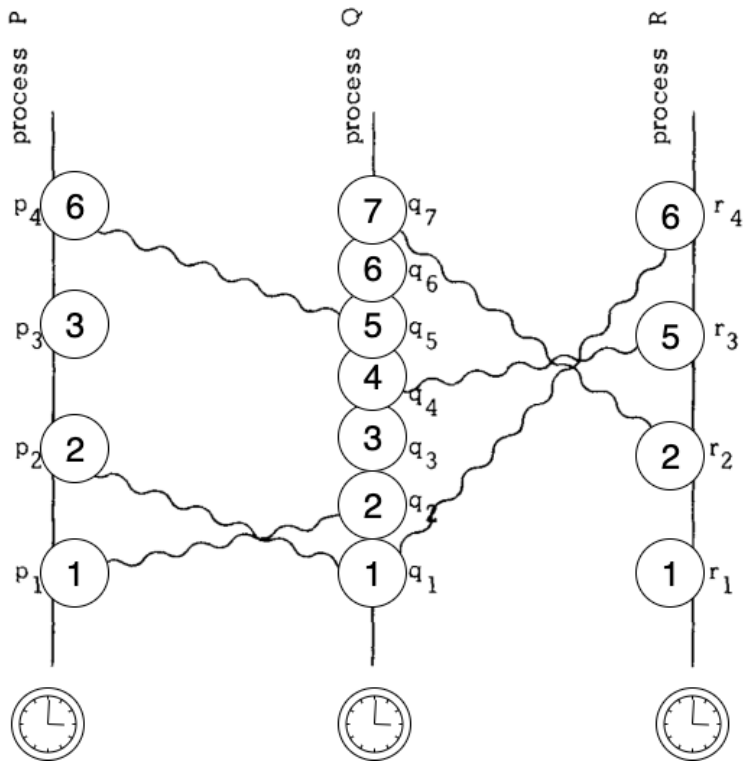
Logical clock is an assignment of numbers on events



Logical Clocks

Logical clock is an assignment of numbers on events

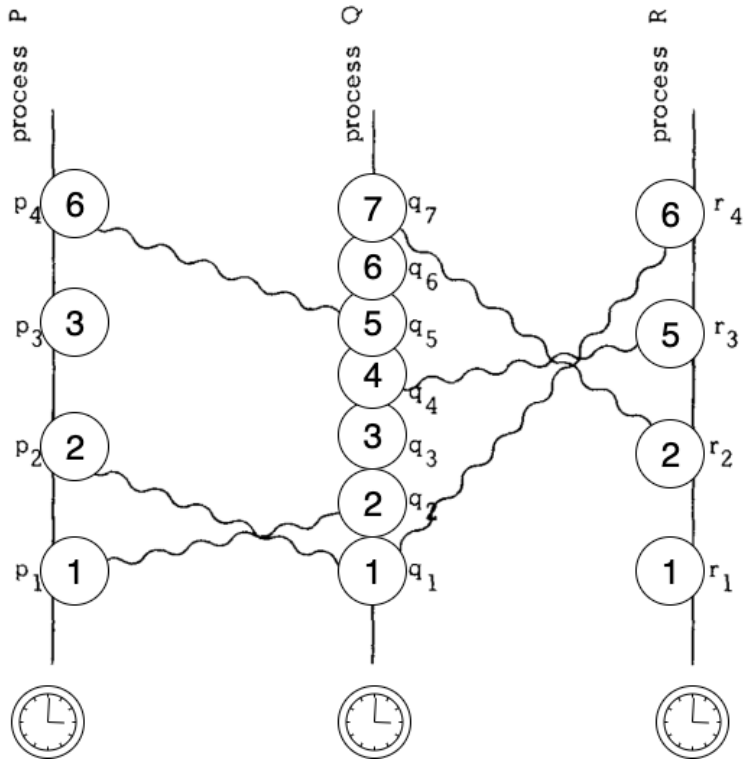
- Local clock C_i assigns number $C_i\langle a \rangle$ to event a in process P_i



Logical Clocks

Logical clock is an assignment of numbers on events

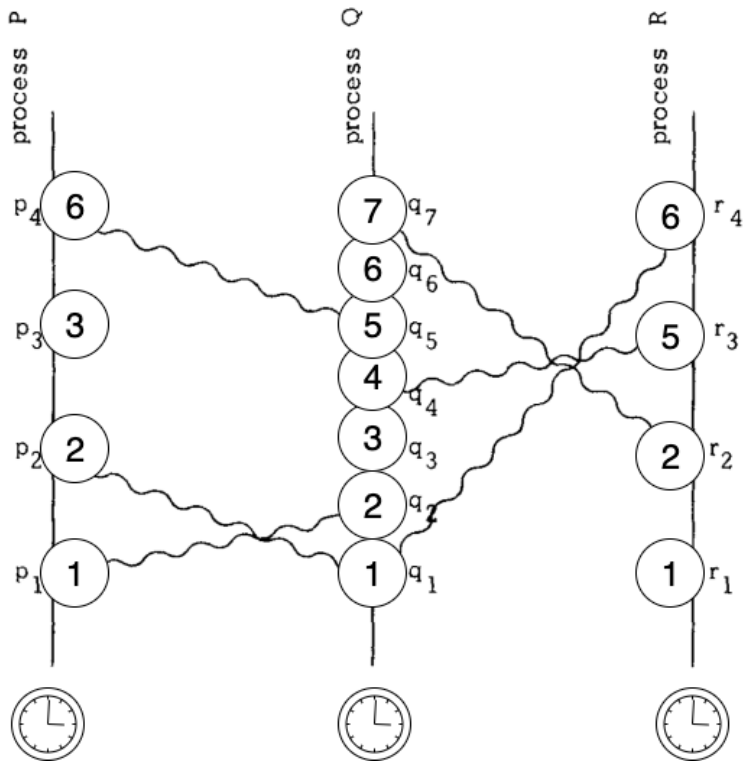
- Local clock C_i assigns number $C_i\langle a\rangle$ to event a in process P_i
- Global clock C defined by $C\langle a\rangle = C_i\langle a\rangle$ if a is in process P_i



Logical Clocks

Clock Condition

For any events a, b:
if $a \rightarrow b$ then $C\langle a \rangle < C\langle b \rangle$



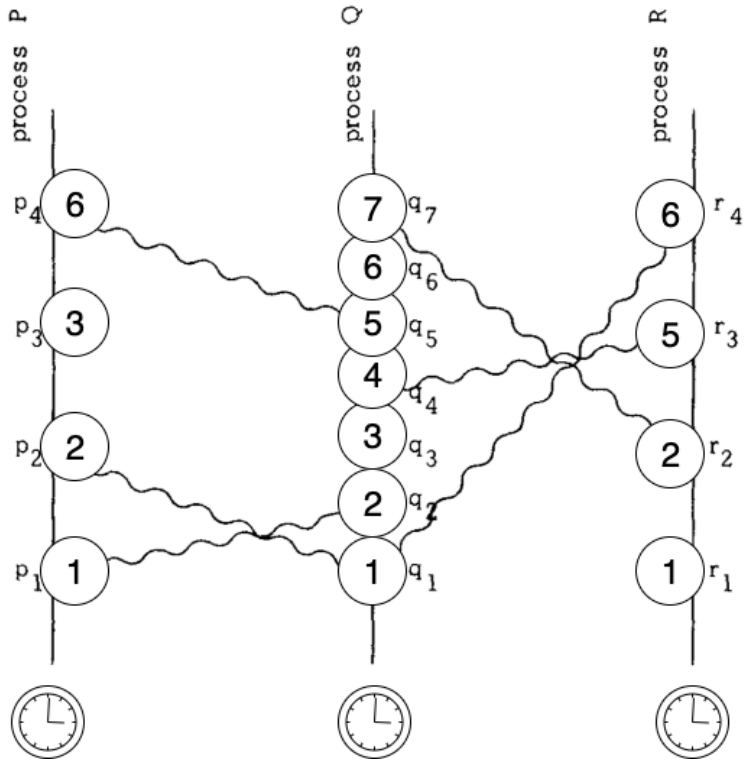
Logical Clocks

Clock Condition

For any events a, b:
if $a \rightarrow b$ then $C\langle a \rangle < C\langle b \rangle$

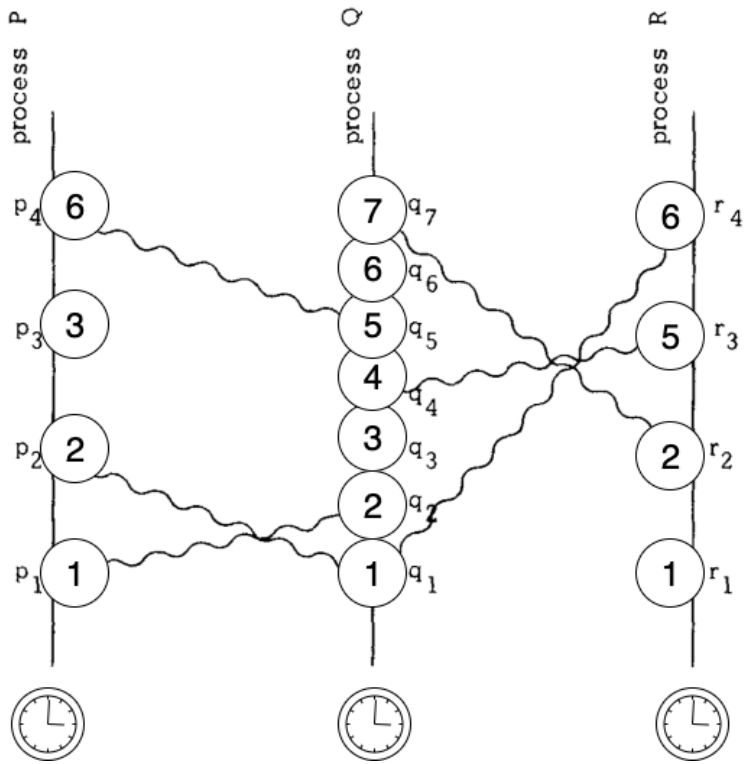
Remark

The converse is not required:
$C\langle a \rangle < C\langle b \rangle$ does not imply $a \rightarrow b$
Because that would require concurrent events to have equal clock values.



Logical Clocks

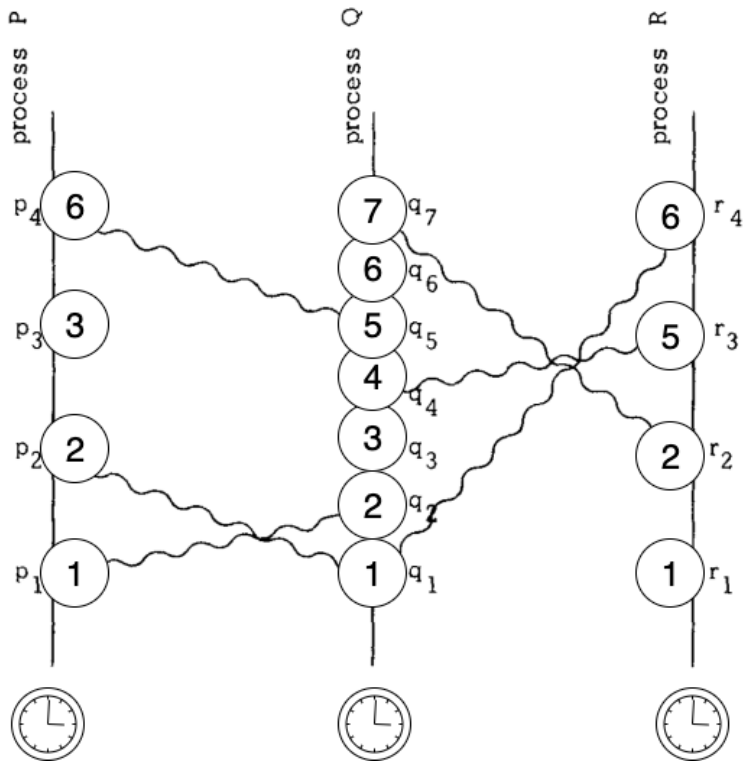
Implement the logical clock:



Logical Clocks

Implement the logical clock:

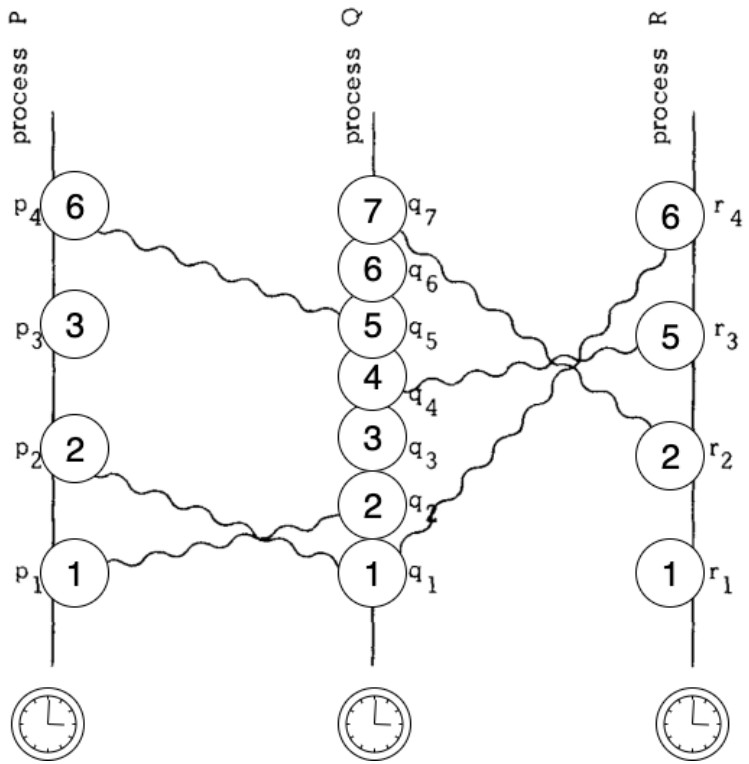
- Each process P_i maintains C_i



Logical Clocks

Implement the logical clock:

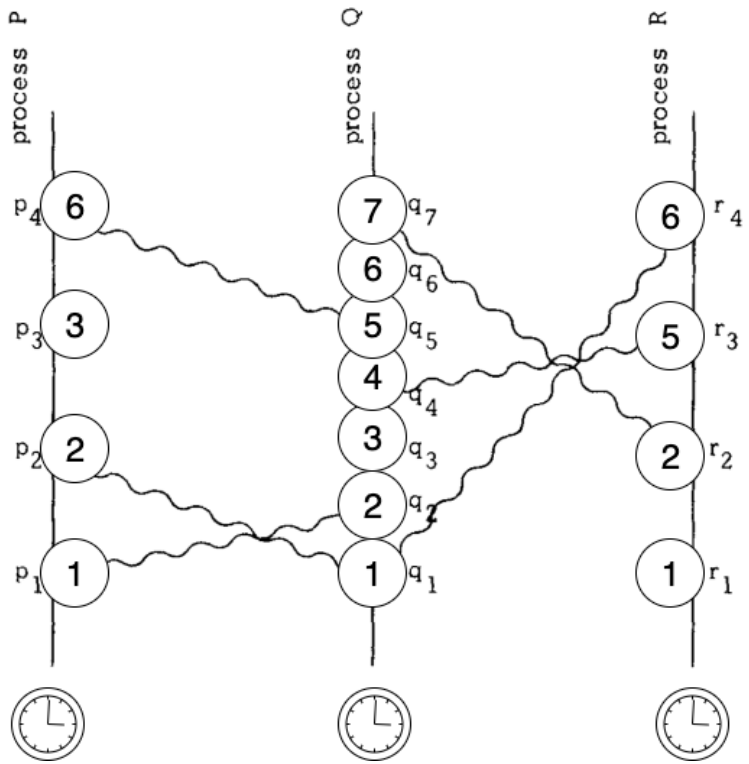
- Each process P_i maintains C_i
- P_i increments C_i for each new events



Logical Clocks

Implement the logical clock:

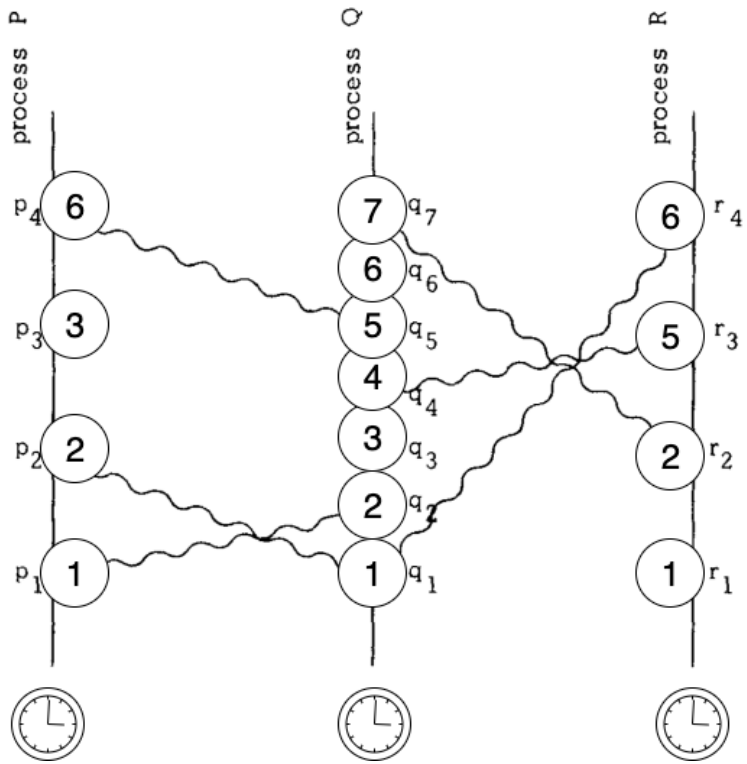
- Each process P_i maintains C_i
- P_i increments C_i for each new events
- Each message m is identified with the event a that sends it, and timestamped by $T_m = C_i\langle a \rangle$



Logical Clocks

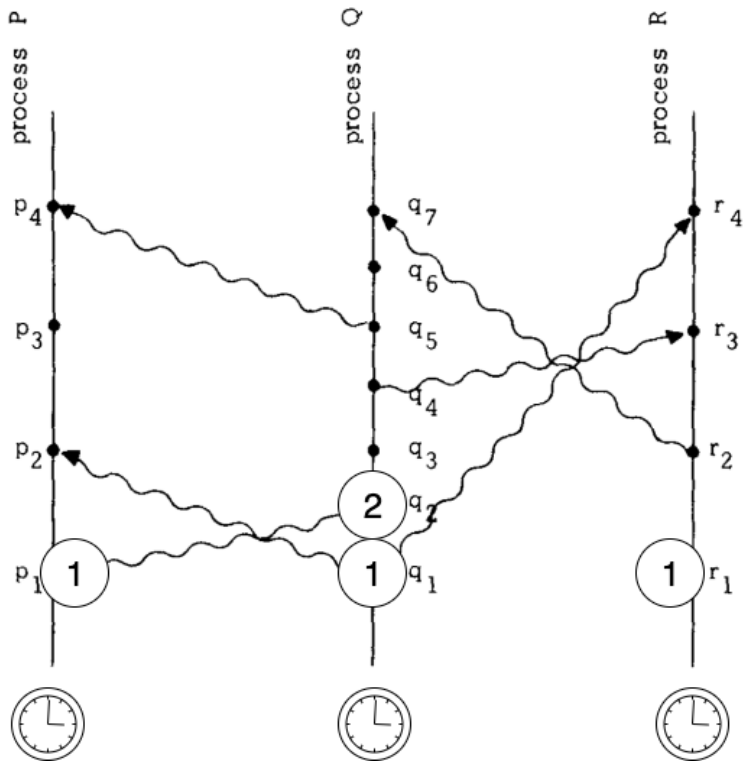
Implement the logical clock:

- Each process P_i maintains C_i
- P_i increments C_i for each new events
- Each message m is identified with the event a that sends it, and timestamped by $T_m = C_i\langle a \rangle$
- On receiving message m, P_j sets C_j to be greater than both T_m and current clock value



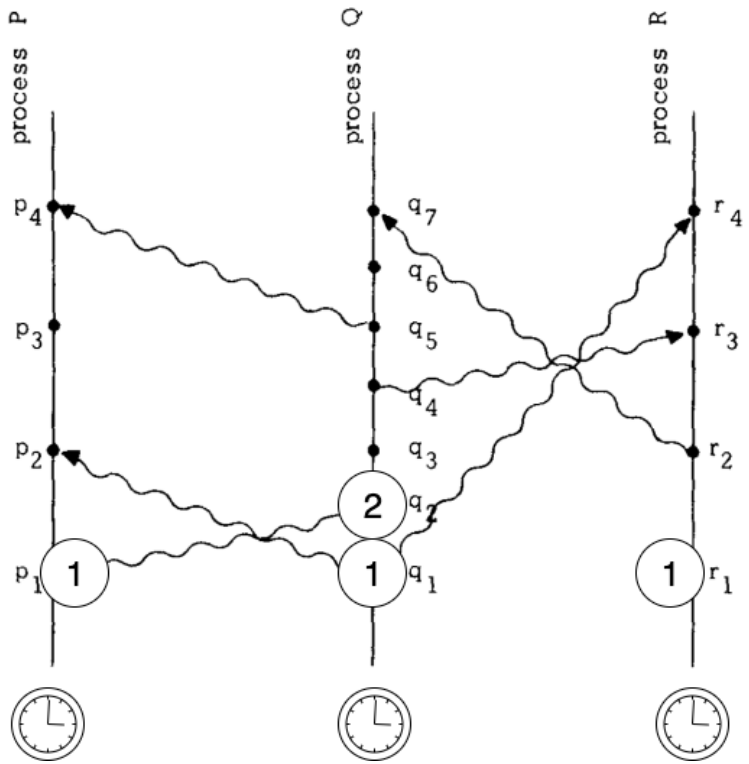
Logical Clocks

Example



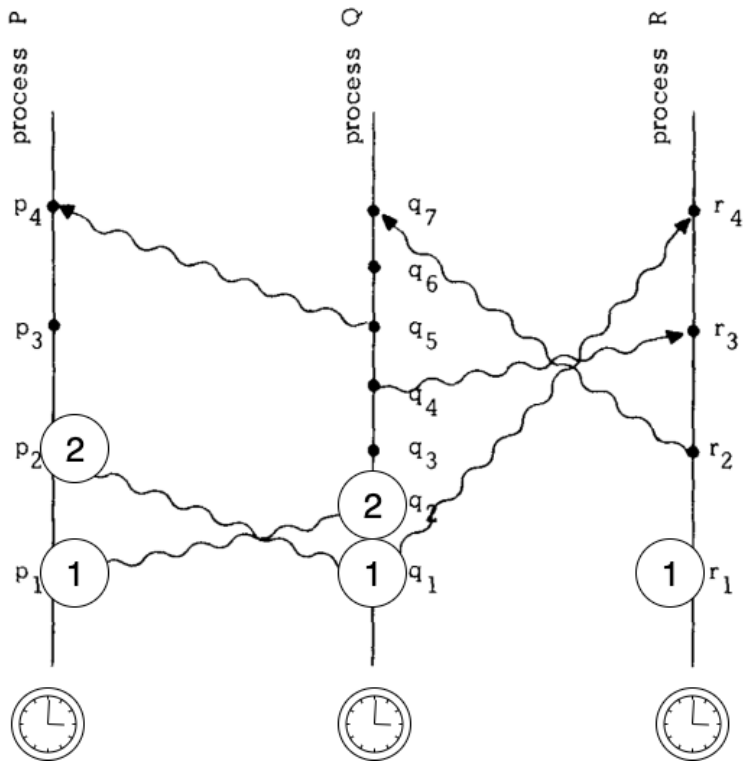
Logical Clocks

Process Q receives message p_1, updates clock to 2



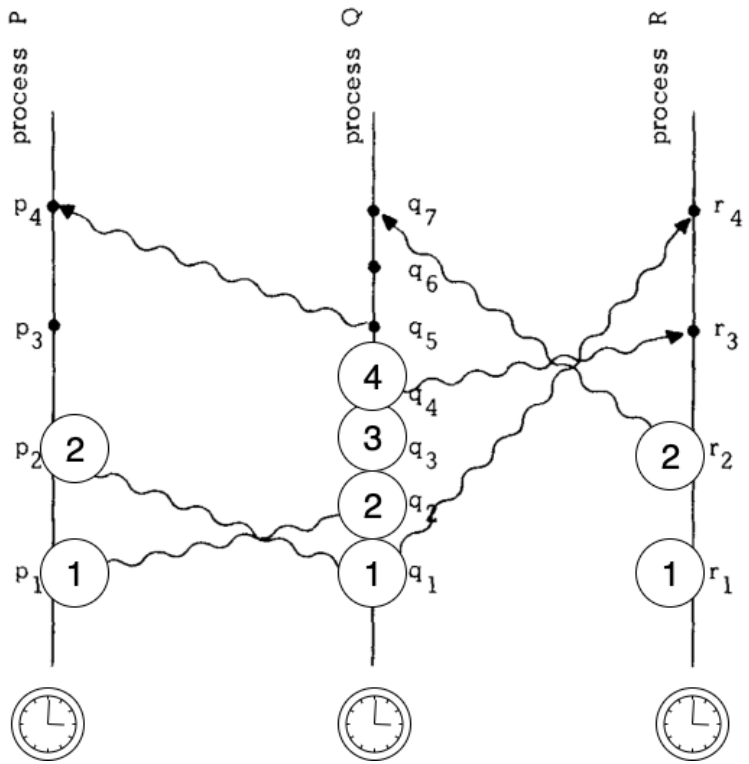
Logical Clocks

Process P receives message q_1, updates clock to 2



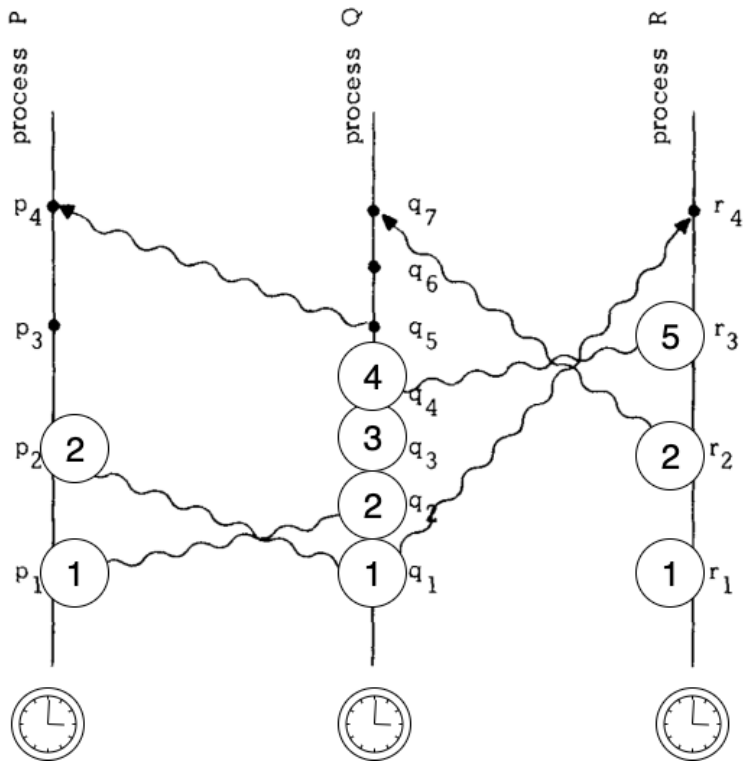
Logical Clocks

Proceeds until Q sends a message to R at event q_4 with timestamp 4



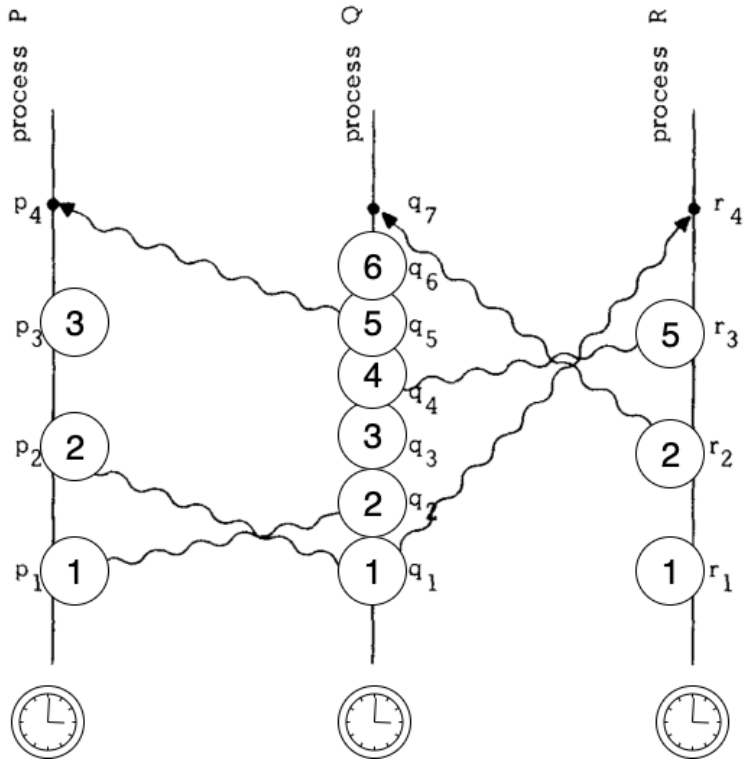
Logical Clocks

Process R receives the message with timestamp 4, and updates clock to 5



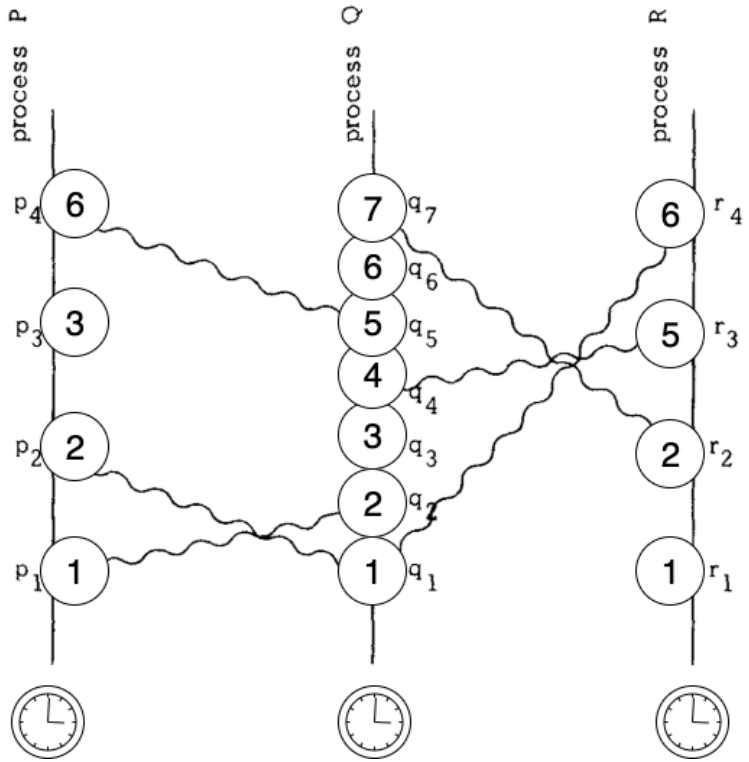
Logical Clocks

Process Q sends message to P with timestamp 5



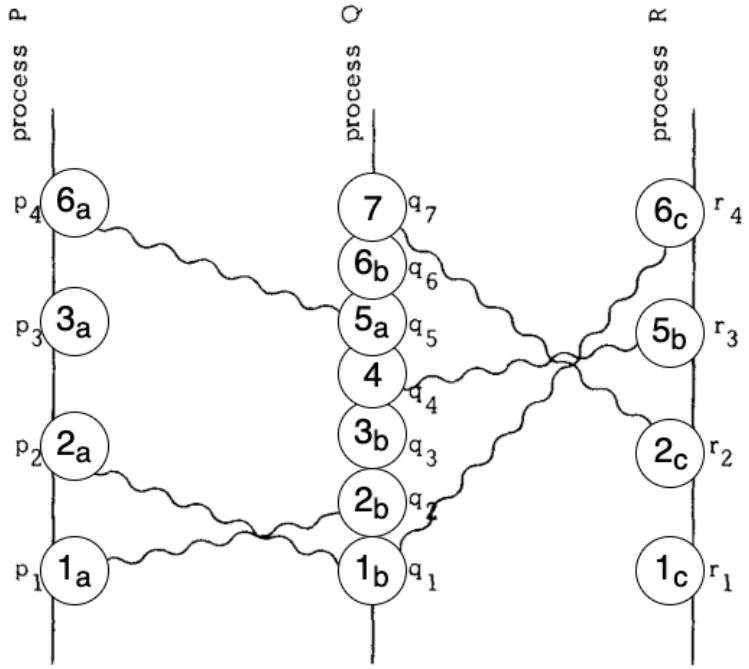
Logical Clocks

Process P updates clock on receiving message with timestamp 5. Clocks of processes Q and R are not affected by messages.



Total Ordering

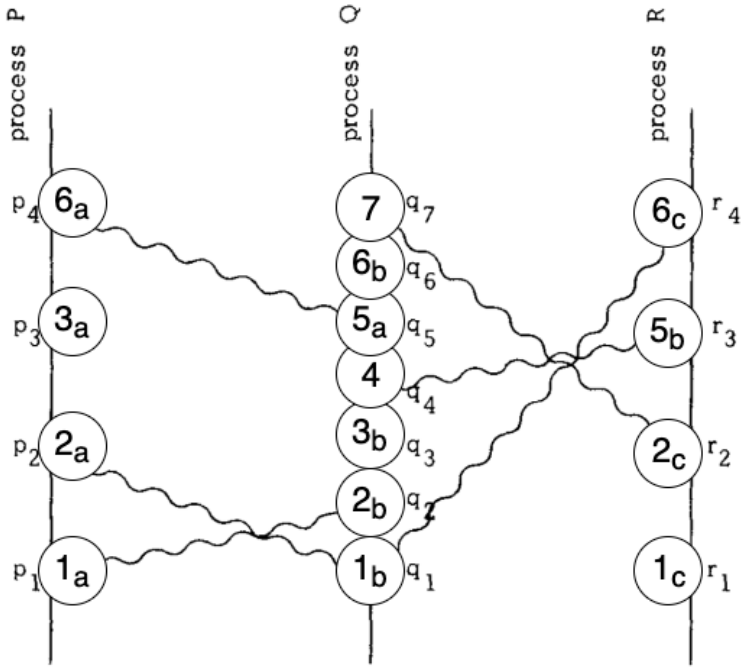
With the logical clock, we are ready to define a total order "$\Rightarrow$" for all events.



Total Ordering

With the logical clock, we are ready to define a total order "$\Rightarrow$" for all events.

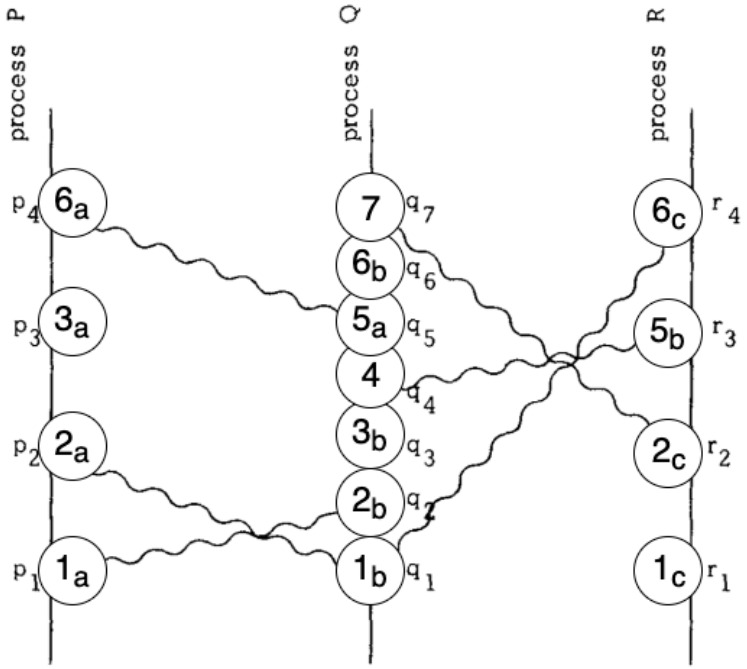
- Use the logical clock as the primary index



Total Ordering

With the logical clock, we are ready to define a total order "$\Rightarrow$" for all events.

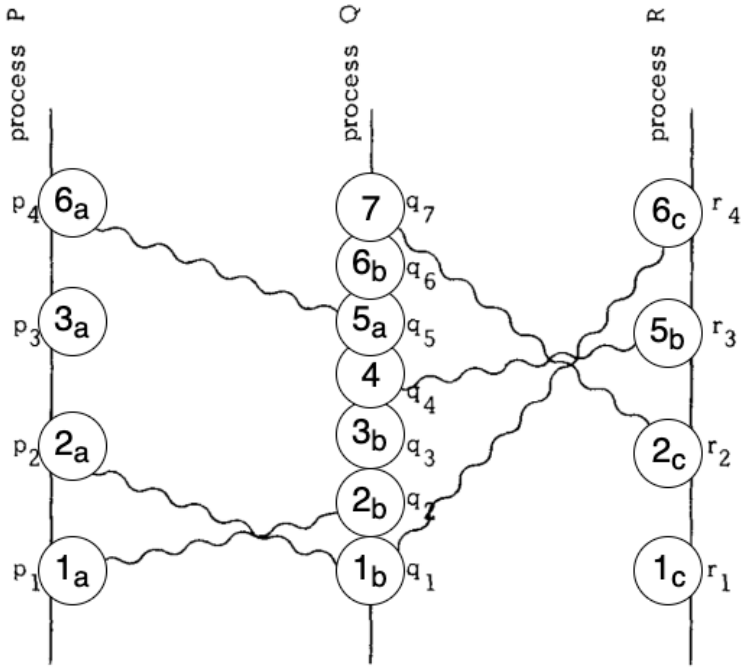
- Use the logical clock as the primary index
- Define a total order $\prec$ over the processes as secondary index



Total Ordering

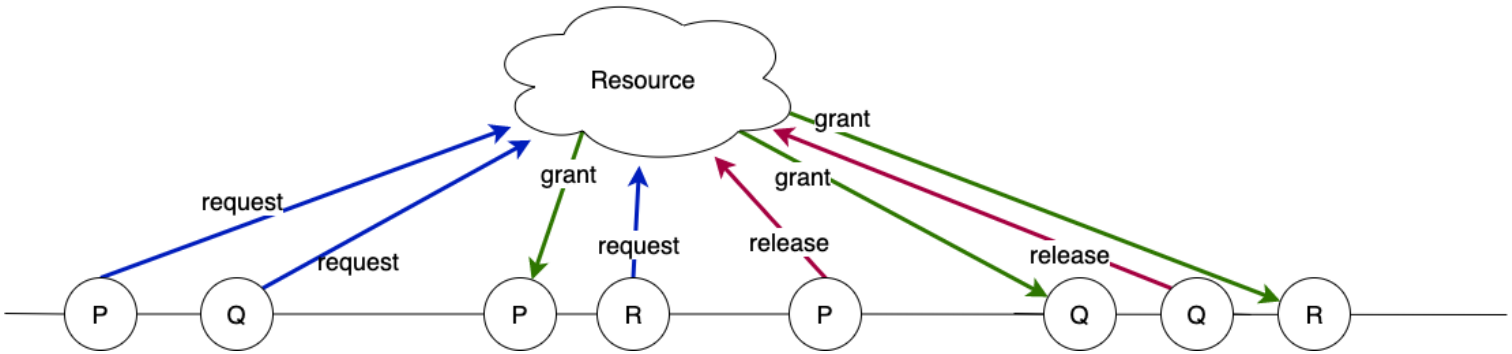
With the logical clock, we are ready to define a total order "$\Rightarrow$" for all events.

- Use the logical clock as the primary index
- Define a total order $\prec$ over the processes as secondary index
- Formally, for events a in P_i and b in P_j, $a \Rightarrow b$ if and only if either
  - $C_i\langle a\rangle < C_j\langle b\rangle$ or;
  - $C_i\langle a\rangle = C_j\langle b\rangle$ and $P_i \prec P_j$



Mutual Exclusion

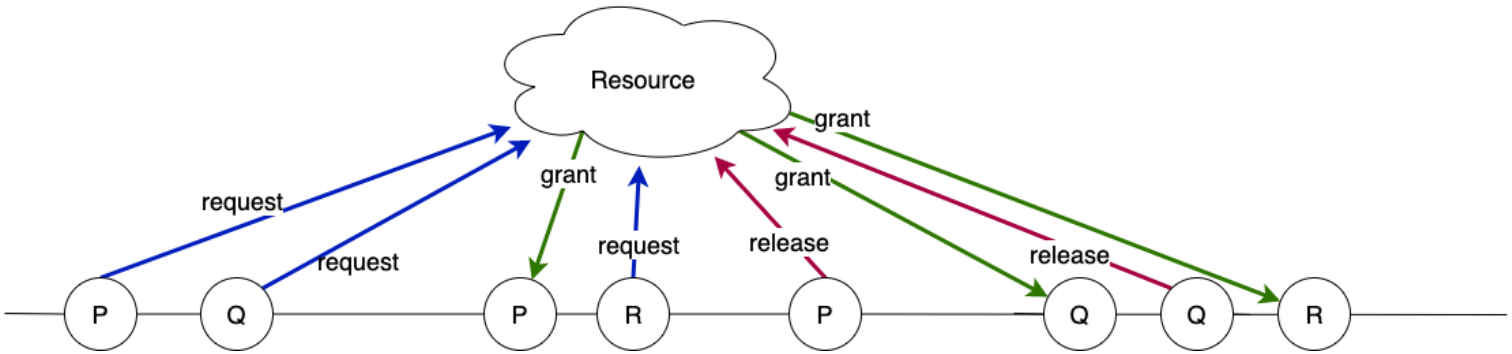
Mutual exclusion is common in implementing resource sharing in distributed systems. In our protocol, we assume:



Mutual Exclusion

Mutual exclusion is common in implementing resource sharing in distributed systems. In our protocol, we assume:

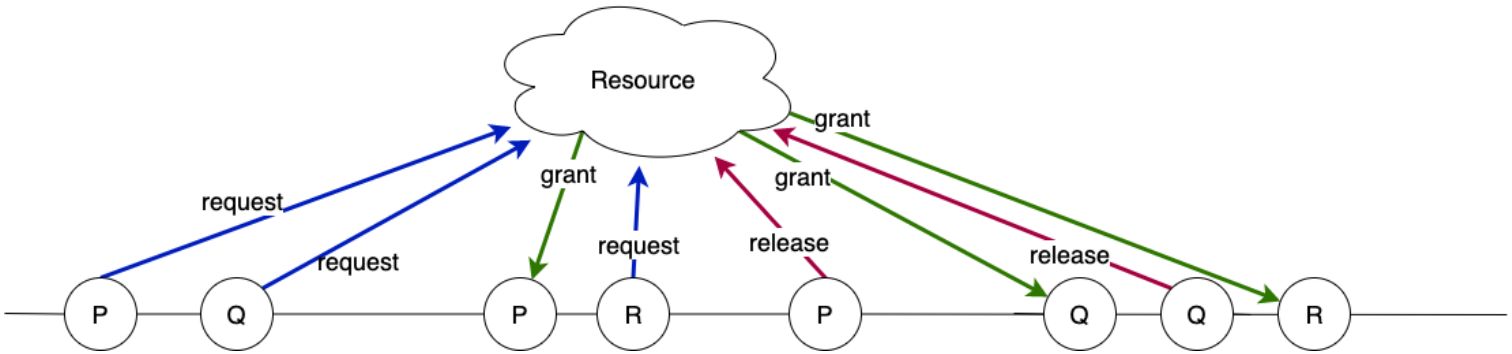
- Fixed collection of processes



Mutual Exclusion

Mutual exclusion is common in implementing resource sharing in distributed systems. In our protocol, we assume:

- Fixed collection of processes
- Single resource

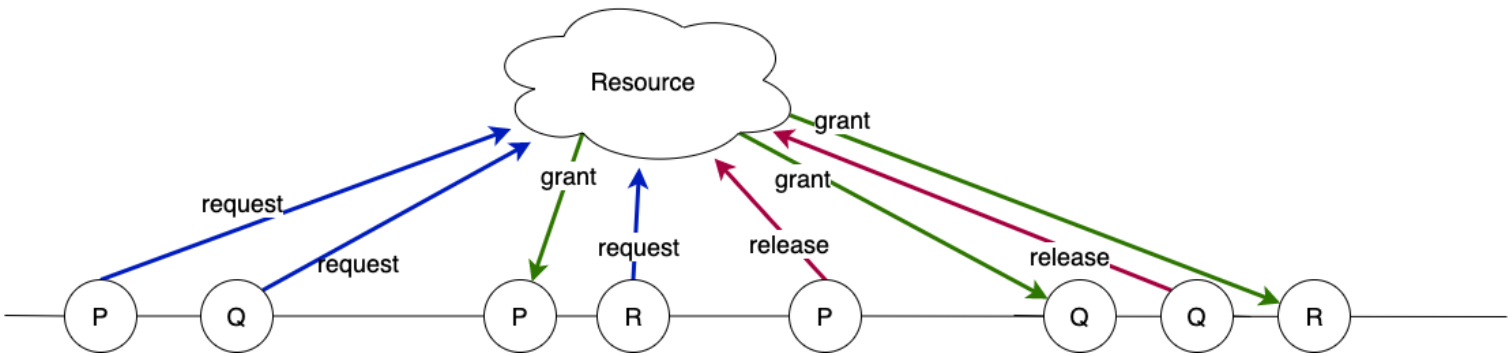


Mutual Exclusion

Mutual exclusion is common in implementing resource sharing in distributed systems. In our protocol, we assume:

- Fixed collection of processes
- Single resource

We want a protocol satisfying following conditions:



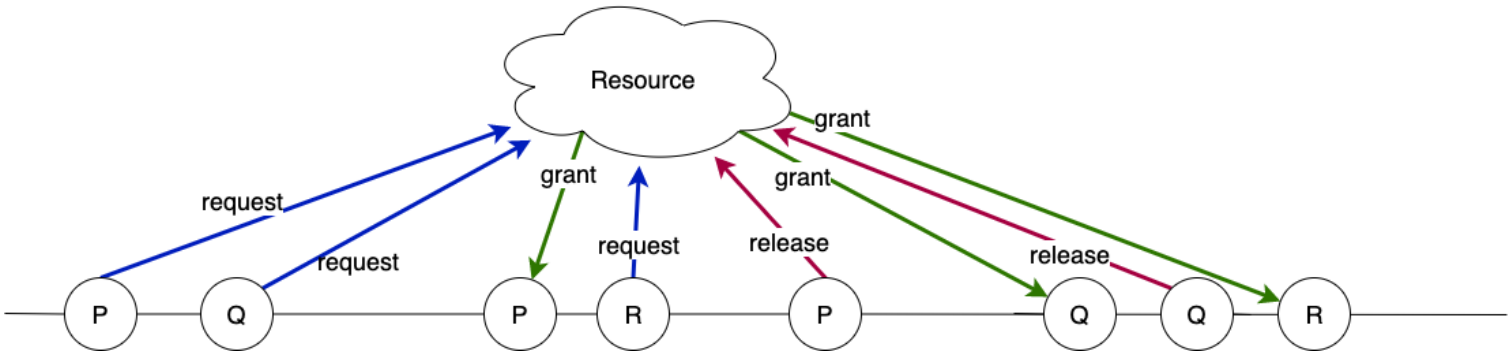
Mutual Exclusion

Mutual exclusion is common in implementing resource sharing in distributed systems. In our protocol, we assume:

- Fixed collection of processes
- Single resource

We want a protocol satisfying following conditions:

- **(I) Mutual exclusion**: once granted, must be released before granted again.



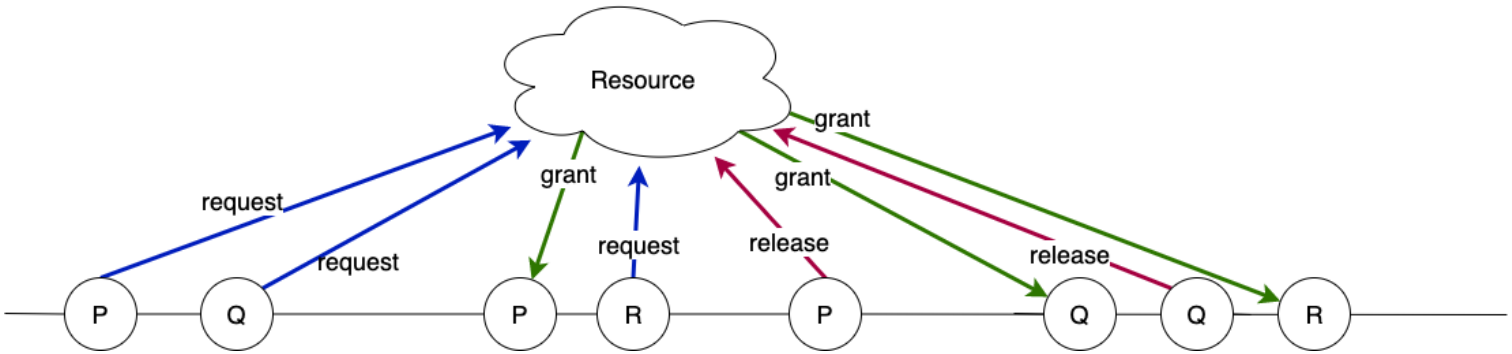
Mutual Exclusion

Mutual exclusion is common in implementing resource sharing in distributed systems. In our protocol, we assume:

- Fixed collection of processes
- Single resource

We want a protocol satisfying following conditions:

- **(I) Mutual exclusion**: once granted, must be released before granted again.
- **(II) In Order**: Earlier requests granted ealier



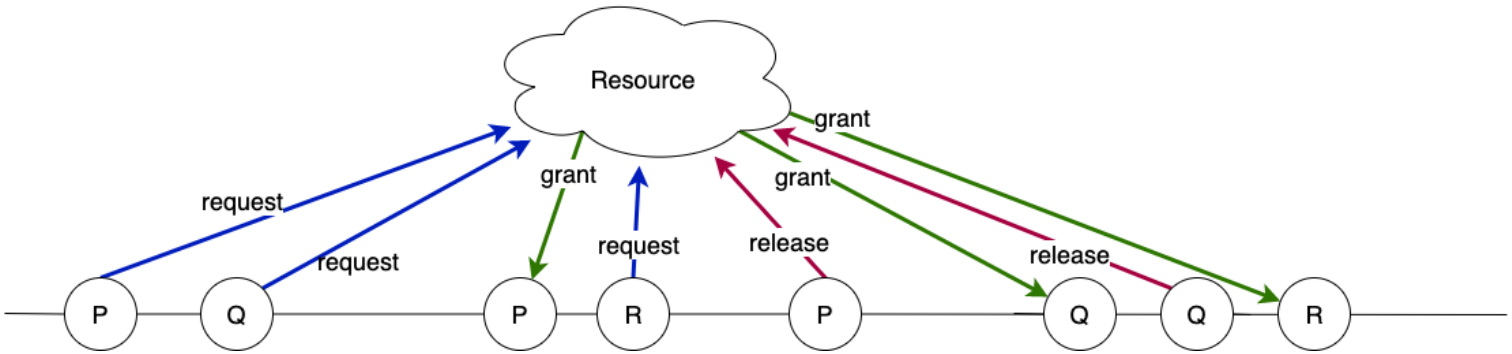
Mutual Exclusion

Mutual exclusion is common in implementing resource sharing in distributed systems. In our protocol, we assume:

- Fixed collection of processes
- Single resource

We want a protocol satisfying following conditions:

- **(I) Mutual exclusion**: once granted, must be released before granted again.
- **(II) In Order**: Earlier requests granted ealier
- **(III) Accessibility**: If every granted request eventually releases, then every request is eventually granted.



Mutual Exclusion

To simplify the implementation, some assumptions are needed to avoid extra details.

Mutual Exclusion

To simplify the implementation, some assumptions are needed to avoid extra details.

- For any P_i and P_j, the messages sent from P_i to P_j are received in order

Mutual Exclusion

To simplify the implementation, some assumptions are needed to avoid extra details.

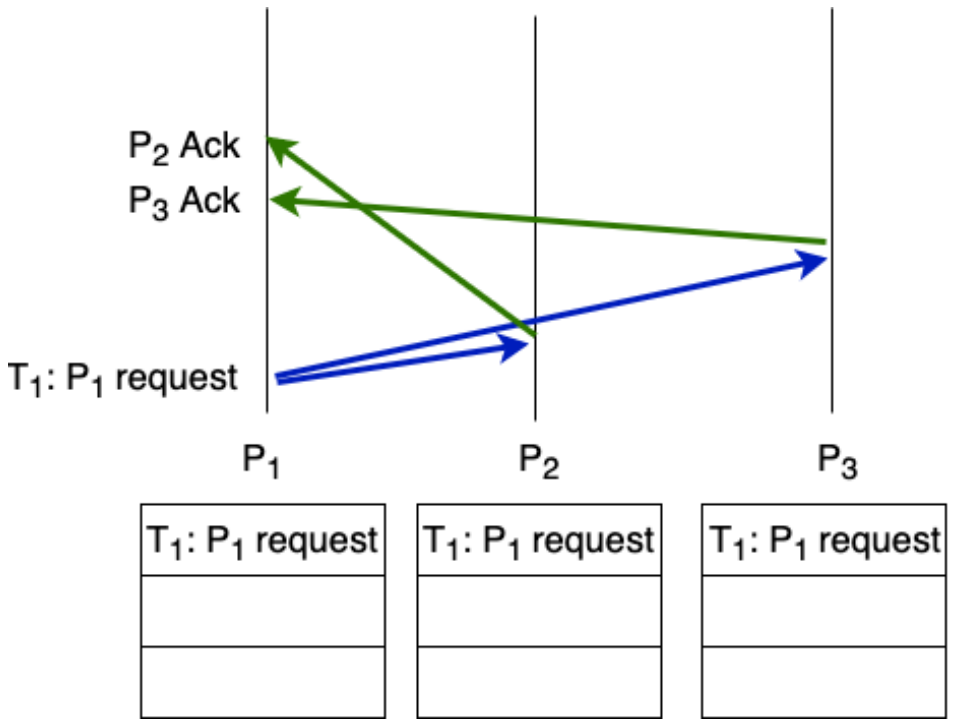
- For any P_i and P_j, the messages sent from P_i to P_j are received in order
- Every message is eventually received

Mutual Exclusion

To simplify the implementation, some assumptions are needed to avoid extra details.

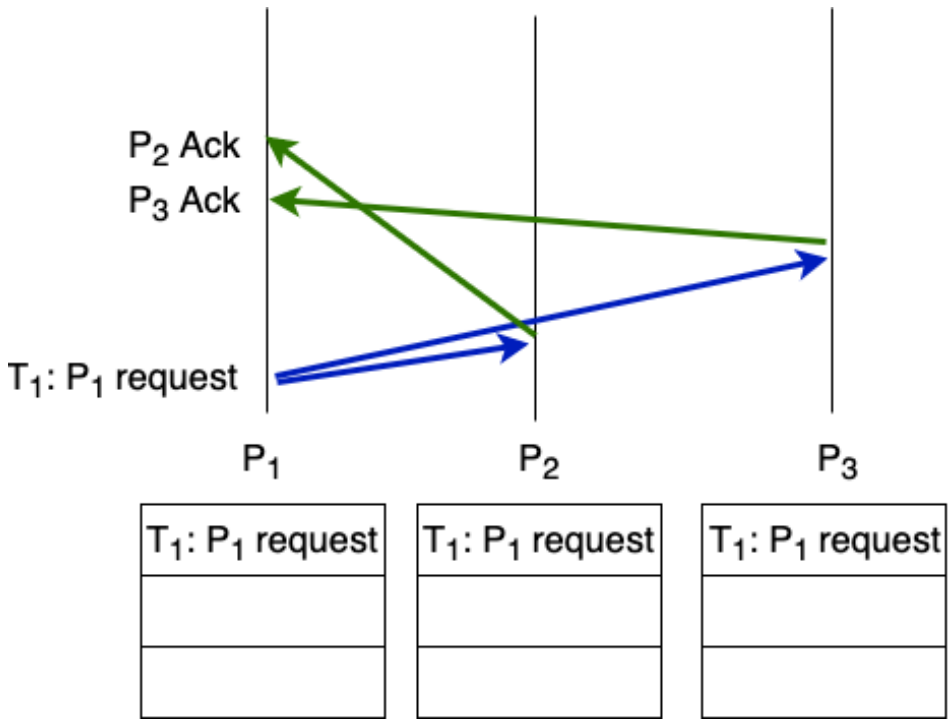
- For any P_i and P_j, the messages sent from P_i to P_j are received in order
- Every message is eventually received
- A process can send messages directly to every other process

Mutual Exclusion



Mutual Exclusion

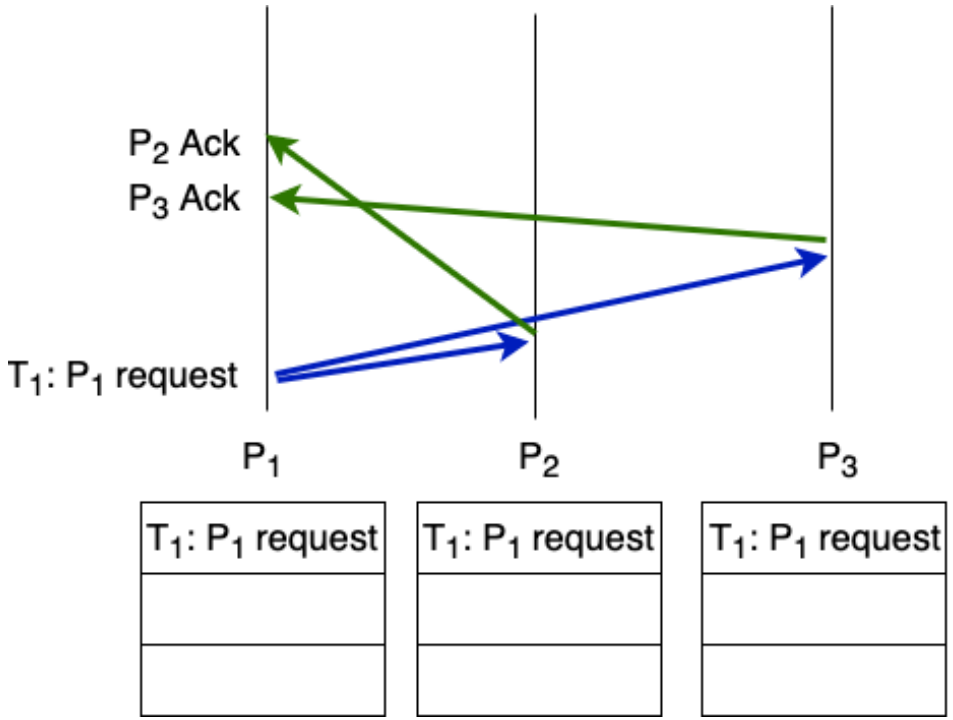
Each process maintains its own *request queue*.



Mutual Exclusion

Each process maintains its own *request queue*.

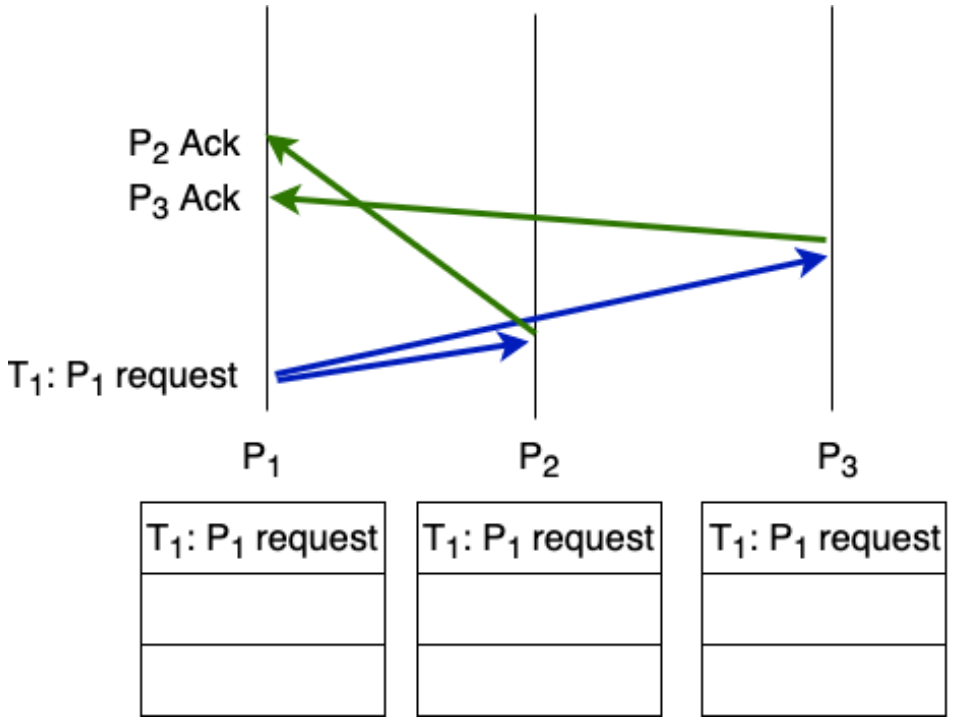
- If P_i wants the resource, it sends a message $m = \langle T_m : P_i \text{ requests resource} \rangle$ to every other process, and puts m on its request queue



Mutual Exclusion

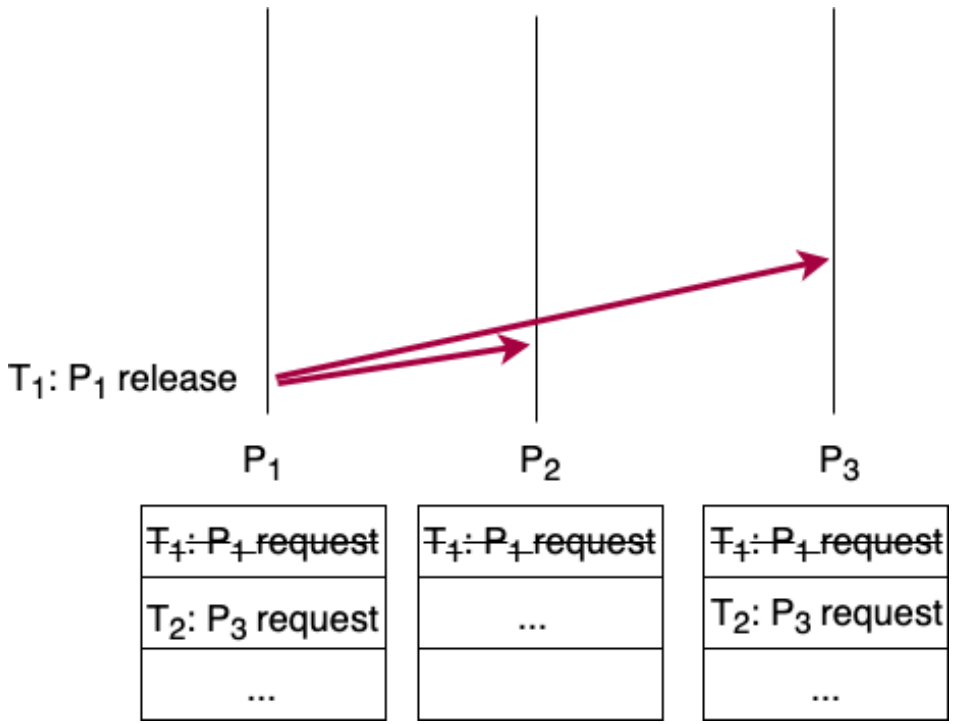
Each process maintains its own *request queue*.

- If P_i wants the resource, it sends a message $m = \langle T_m : P_i \text{ requests resource} \rangle$ to every other process, and puts m on its request queue
- When P_j receives m, it puts m on its request queue, and sends an acknowledgement message to P_i



Mutual Exclusion

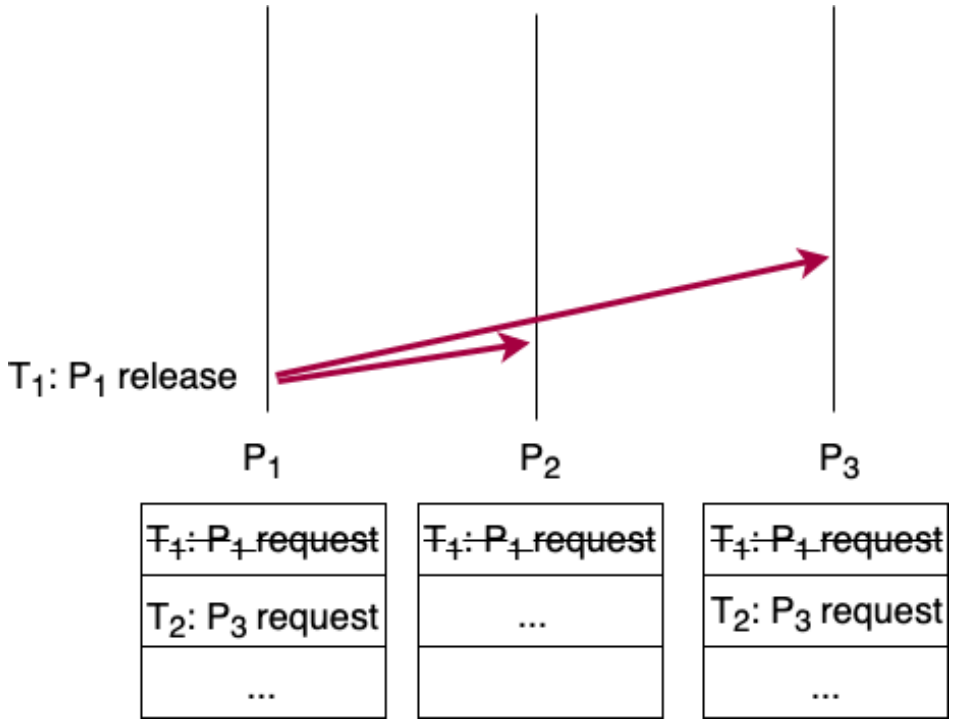
Releasing the resource works in similar way, without acknowledgement.



Mutual Exclusion

Releasing the resource works in similar way, without acknowledgement.

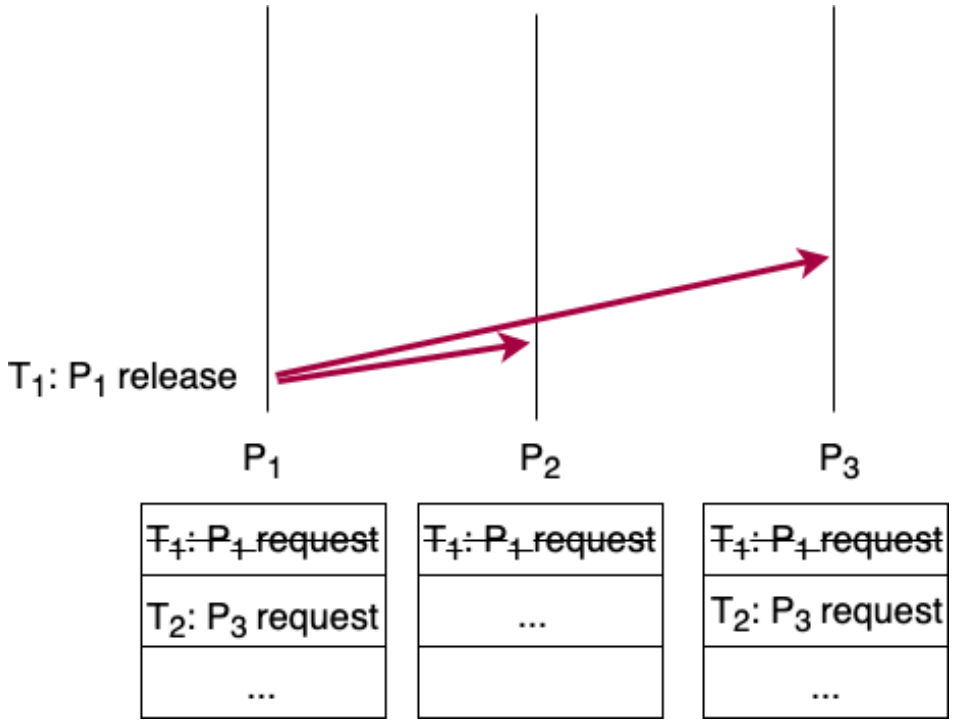
- If P_i wants to release the resource, it sends a message $\langle T'_m : P_i \textit{ releases resource} \rangle$ to other processes, and removes any $\langle T_m : P_i \textit{ requests resource} \rangle$ from its request queue



Mutual Exclusion

Releasing the resource works in similar way, without acknowledgement.

- If P_i wants to release the resource, it sends a message $\langle T'_m : P_i \textit{ releases resource} \rangle$ to other processes, and removes any $\langle T_m : P_i \textit{ requests resource} \rangle$ from its request queue
- When P_j receives $\langle T'_m : P_i \textit{ releases resource} \rangle$, it removes any $\langle T_m : P_i \textit{ requests resource} \rangle$ from its request queue



Mutual Exclusion

P_i is granted the resource if the following conditions hold.

Mutual Exclusion

P_i is granted the resource if the following conditions hold.

- There is a $\langle T_m : P_i \textit{ requests resource} \rangle$ in its request queue ordered before any other messages inside the queue by relation "$\Rightarrow$"

Mutual Exclusion

P_i is granted the resource if the following conditions hold.

- There is a $\langle T_m : P_i \textit{ requests resource} \rangle$ in its request queue ordered before any other messages inside the queue by relation "$\Rightarrow$"
- P_i received the acknowledgements from every other process

Mutual Exclusion

P_i is granted the resource if the following conditions hold.

- There is a $\langle T_m : P_i \textit{ requests resource} \rangle$ in its request queue ordered before any other messages inside the queue by relation "$\Rightarrow$"
- P_i received the acknowledgements from every other process

Now P_i can safely access the resource, assured that nobody else is granted the resource.

Mutual Exclusion

P_i is granted the resource if the following conditions hold.

- There is a $\langle T_m : P_i \textit{ requests resource} \rangle$ in its request queue ordered before any other messages inside the queue by relation "$\Rightarrow$"
- P_i received the acknowledgements from every other process

Now P_i can safely access the resource, assured that nobody else is granted the resource.
But why does this work?

Mutual Exclusion

The above protocol satisfies conditions (I)(II) and (III).

Mutual Exclusion

The above protocol satisfies conditions (I)(II) and (III).

- **Conditions I and II:** From the point of view of $m = \langle T_m : P_i \textit{ requests resource} \rangle$, any request m' issued later by P_j is in either of the following cases:
  - Acknowledged by P_i, then P_j must also have received m, and m' is superceded by m in P_j's queue
  - Not acknowledged by P_i

  In any case, m' is not granted.

Mutual Exclusion

The above protocol satisfies conditions (I)(II) and (III).

- **Conditions I and II:** From the point of view of $m = \langle T_m : P_i \textit{ requests resource} \rangle$, any request m' issued later by P_j is in either of the following cases:
  - Acknowledged by P_i, then P_j must also have received m, and m' is superceded by m in P_j's queue
  - Not acknowledged by P_i

  In any case, m' is not granted. In summary, for any request m, no request issued later than m will be granted before m is released.

Mutual Exclusion

The above protocol satisfies conditions (I)(II) and (III).

- **Conditions I and II:** From the point of view of $m = \langle T_m : P_i \textit{ requests resource} \rangle$, any request m' issued later by P_j is in either of the following cases:
  - Acknowledged by P_i, then P_j must also have received m, and m' is superceded by m in P_j's queue
  - Not acknowledged by P_i

  In any case, m' is not granted. In summary, for any request m, no request issued later than m will be granted before m is released.
- **Condition III:** For each message $m = \langle T_m : P_i \textit{ requests resource} \rangle$:
  - m will be the oldest in P_i's queue eventually
  - P_i will receive all acknowledgements eventually

Mutual Exclusion

Remark

Mutual Exclusion

Remark

- No centralized authority, everything happens automatically in sending and receiving messages

Mutual Exclusion

Remark

- No centralized authority, everything happens automatically in sending and receiving messages
- Requires all parties trusted. Malicious party can easily forge timestamps, refuse to send acknowledgement, etc.

Mutual Exclusion

Remark

- No centralized authority, everything happens automatically in sending and receiving messages
- Requires all parties trusted. Malicious party can easily forge timestamps, refuse to send acknowledgement, etc.
- The shared resource is an abstract concept, which can be anything

Mutual Exclusion

Remark

- No centralized authority, everything happens automatically in sending and receiving messages
- Requires all parties trusted. Malicious party can easily forge timestamps, refuse to send acknowledgement, etc.
- The shared resource is an abstract concept, which can be anything
- Can be easily extended to multiple resources

Mutual Exclusion

Remark

- No centralized authority, everything happens automatically in sending and receiving messages
- Requires all parties trusted. Malicious party can easily forge timestamps, refuse to send acknowledgement, etc.
- The shared resource is an abstract concept, which can be anything
- Can be easily extended to multiple resources

Applied to consensus protocol?

Mutual Exclusion

Remark

- No centralized authority, everything happens automatically in sending and receiving messages
- Requires all parties trusted. Malicious party can easily forge timestamps, refuse to send acknowledgement, etc.
- The shared resource is an abstract concept, which can be anything
- Can be easily extended to multiple resources

Applied to consensus protocol?

- Cryptographic mechanisms for trustlessness?

Mutual Exclusion

Remark

- No centralized authority, everything happens automatically in sending and receiving messages
- Requires all parties trusted. Malicious party can easily forge timestamps, refuse to send acknowledgement, etc.
- The shared resource is an abstract concept, which can be anything
- Can be easily extended to multiple resources

Applied to consensus protocol?

- Cryptographic mechanisms for trustlessness?
- Shared resource can be: leadership, write permission, ...

Physical Clock

Let t denote the real, ideal physical time, which is not available to any processes

Physical Clock

Let t denote the real, ideal physical time, which is not available to any processes

- P_i only has access to a function of t, called a clock, denoted by $C_i(t)$ which is a (almost) differentiable function

Physical Clock

Let t denote the real, ideal physical time, which is not available to any processes

- P_i only has access to a function of t, called a clock, denoted by $C_i(t)$ which is a (almost) differentiable function
- The clock rates $dC_i(t)/dt$ approximate 1

Physical Clock

Let t denote the real, ideal physical time, which is not available to any processes

- P_i only has access to a function of t, called a clock, denoted by $C_i(t)$ which is a (almost) differentiable function
- The clock rates $dC_i(t)/dt$ approximate 1
- The idea is processes sending to each other synchronization messages, to ensure that $|C_i(t) - C_j(t)| \leq \varepsilon$ for all t and all pairs of i, j, where ε is related to parameters including:

Physical Clock

Let t denote the real, ideal physical time, which is not available to any processes

- P_i only has access to a function of t, called a clock, denoted by $C_i(t)$ which is a (almost) differentiable function
- The clock rates $dC_i(t)/dt$ approximate 1
- The idea is processes sending to each other synchronization messages, to ensure that $|C_i(t) - C_j(t)| \leq \varepsilon$ for all t and all pairs of i, j, where ε is related to parameters including:
  - Global bound on message delay

Physical Clock

Let t denote the real, ideal physical time, which is not available to any processes

- P_i only has access to a function of t, called a clock, denoted by $C_i(t)$ which is a (almost) differentiable function
- The clock rates $dC_i(t)/dt$ approximate 1
- The idea is processes sending to each other synchronization messages, to ensure that $|C_i(t) - C_j(t)| \leq \varepsilon$ for all t and all pairs of i, j, where ε is related to parameters including:
  - Global bound on message delay
  - Frequencies of synchronization messages sent to each other

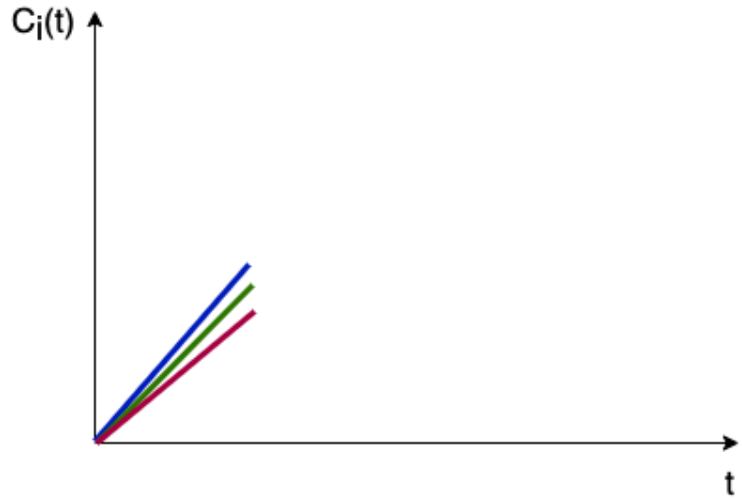
Physical Clock

Let t denote the real, ideal physical time, which is not available to any processes

- P_i only has access to a function of t, called a clock, denoted by $C_i(t)$ which is a (almost) differentiable function
- The clock rates $dC_i(t)/dt$ approximate 1
- The idea is processes sending to each other synchronization messages, to ensure that $|C_i(t) - C_j(t)| \leq \varepsilon$ for all t and all pairs of i, j, where ε is related to parameters including:
  - Global bound on message delay
  - Frequencies of synchronization messages sent to each other
  - Accuracy of the local clocks

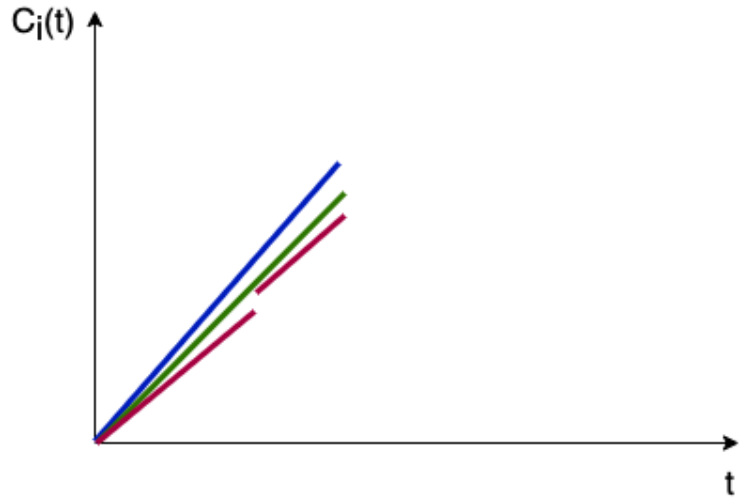
Physical Clock

The clocks run at different rates and start deviating from each other



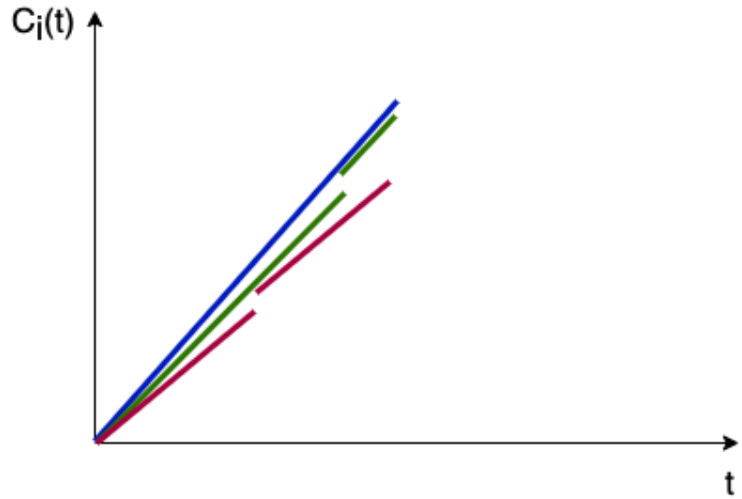
Physical Clock

The third process receives a message from second process, and resets clock.



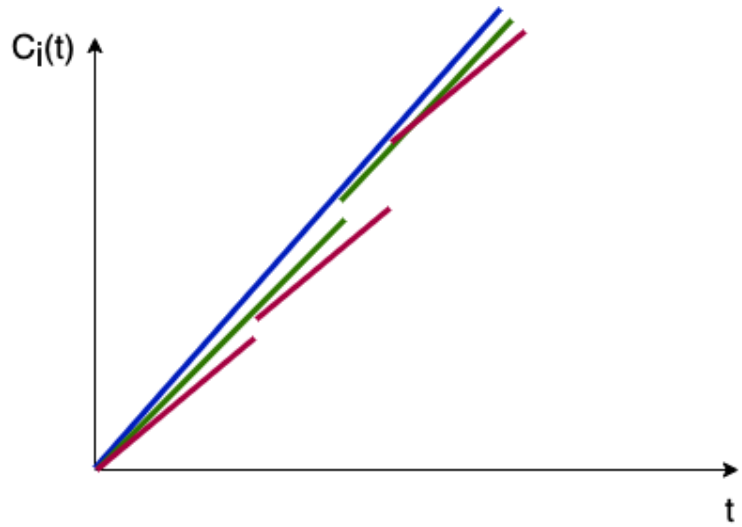
Physical Clock

The second process receives a message from first process, and resets clock.



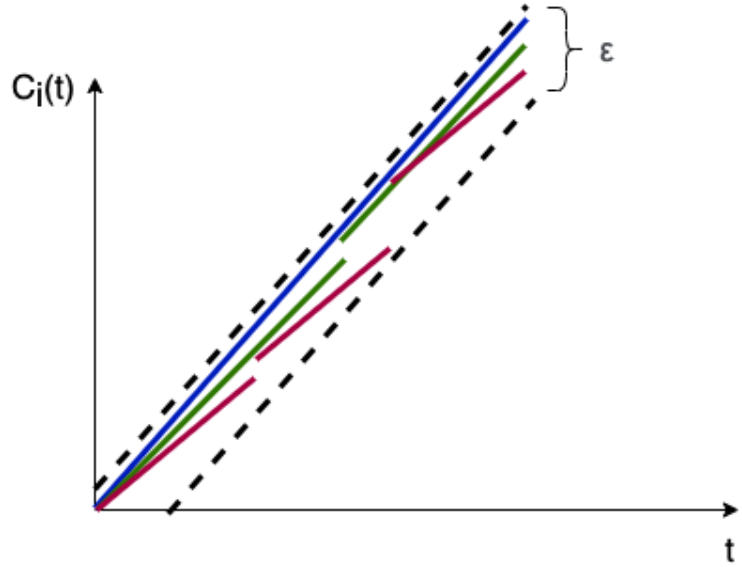
Physical Clock

The third process receives a message from first process, and resets clock.



Physical Clock

Given appropriate assumptions, the clocks are syncronized within bounded error.



Summary

- Total ordering
  - Partial order "happens before" on events: $\rightarrow$
  - Logical clock that respects "$\rightarrow$"
  - Logical clock + Total order on processes $\Rightarrow$ total order on events
- Protocol that achieves mutual exclusion, and something more
- Physical clock

Q&A