CC552 Lecture 4

The following slides are based on:

- An edited version of slides by professor Wyatt Lloyd taught in Princeton's COS418 course (CC license).
- An edited version of slides provided on the textbook website (M. van Steen and A.S. Tanenbaum, Distributed Systems, 3rd ed., distributed-systems.net, 2017)

Outline

Synchronization and Coordination (cont.)

- Lamport Clocks (Review)
- Totally-ordered Multicast
- Vector Clocks
- Mutual Exclusion in Distributed Systems
- Distributed Snapshots

Vector Clocks

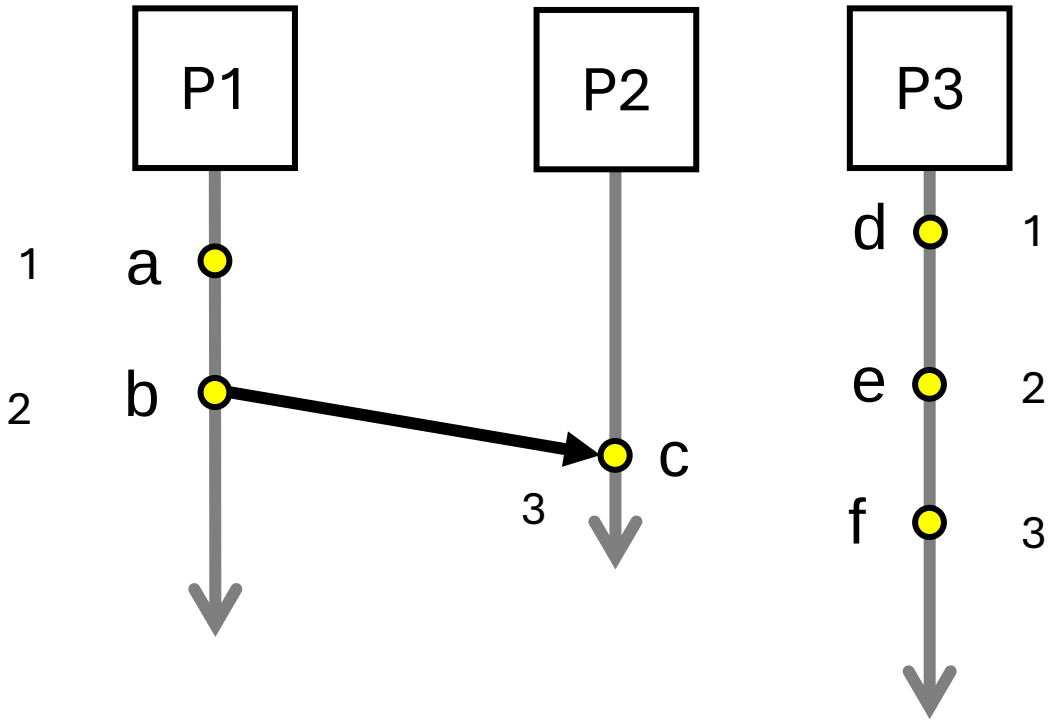
Lamport Clocks and Causality

- Lamport clock timestamps do not capture causality

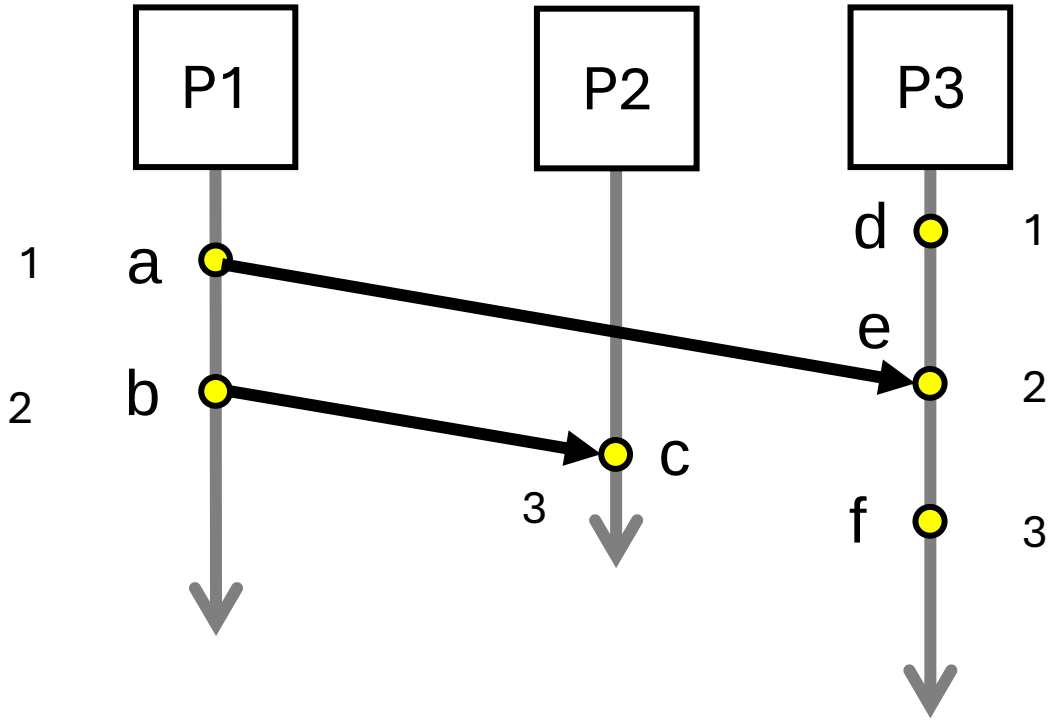
- Given two timestamps C(a) and C(z), want to know whether there's a chain of events linking them:

$$a \rightarrow b \rightarrow \ldots \rightarrow y \rightarrow z$$

Lamport Clocks and Causality



Does **a** causally precede **f** in the above figure?



Does **a** causally precede **f** in the above figure?

Note: The timestamps in both cases are the same, although the answers are different.

Causal Precedence: Event 1 causally precedes Event 2, when Event 2 **may** causally depend on Event 1, i.e., there may be information from Event 1 that is propagated into Event 2.

Vector clock: Introduction

- One integer can't order events in more than one process
- So, a Vector Clock (VC) is a vector of integers, one entry for each process in the entire distributed system
  - Label event e with VC(e) = [c_1, c_2 …, c_n]
    - Each entry c_k is a count of events in process k that causally precede e

Vector clock: Update rules

- Initially, all vectors are [0, 0, …, 0]

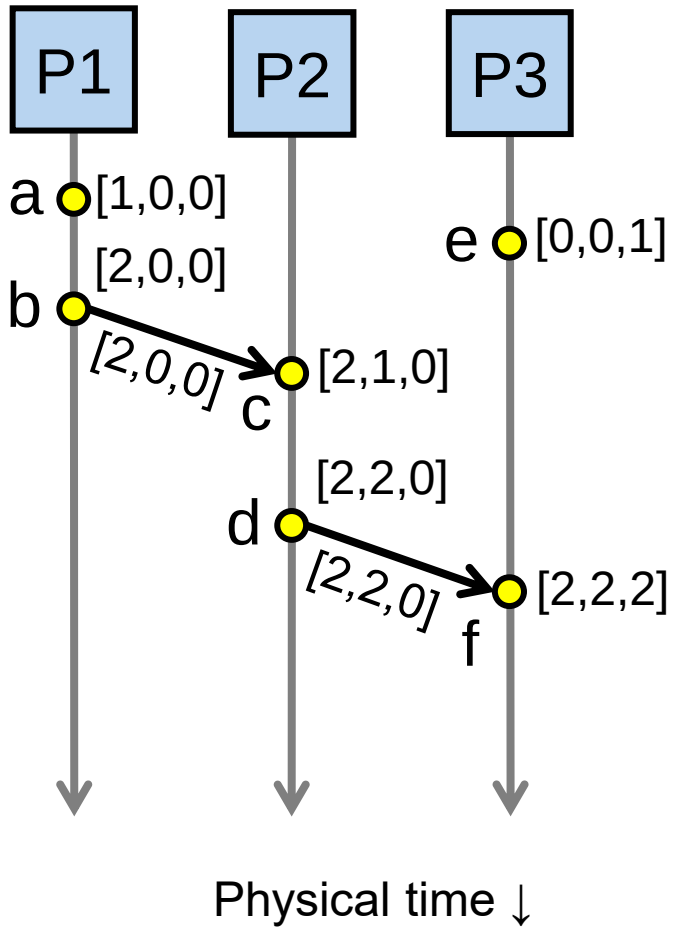
- Two update rules:

1. For each local event on process i, increment local entry c_i

2. If process j receives message with vector $[d_1, d_2, \ldots, d_n]$:
   - Set each local entry $c_k = \max\{c_k, d_k\}$
   - Increment local entry c_j

Vector clock: Example

- All processes' VCs start at [0, 0, 0]
- Applying local update rule
- Applying message rule
  - Local vector clock piggybacks on inter-process messages



Comparing vector timestamps

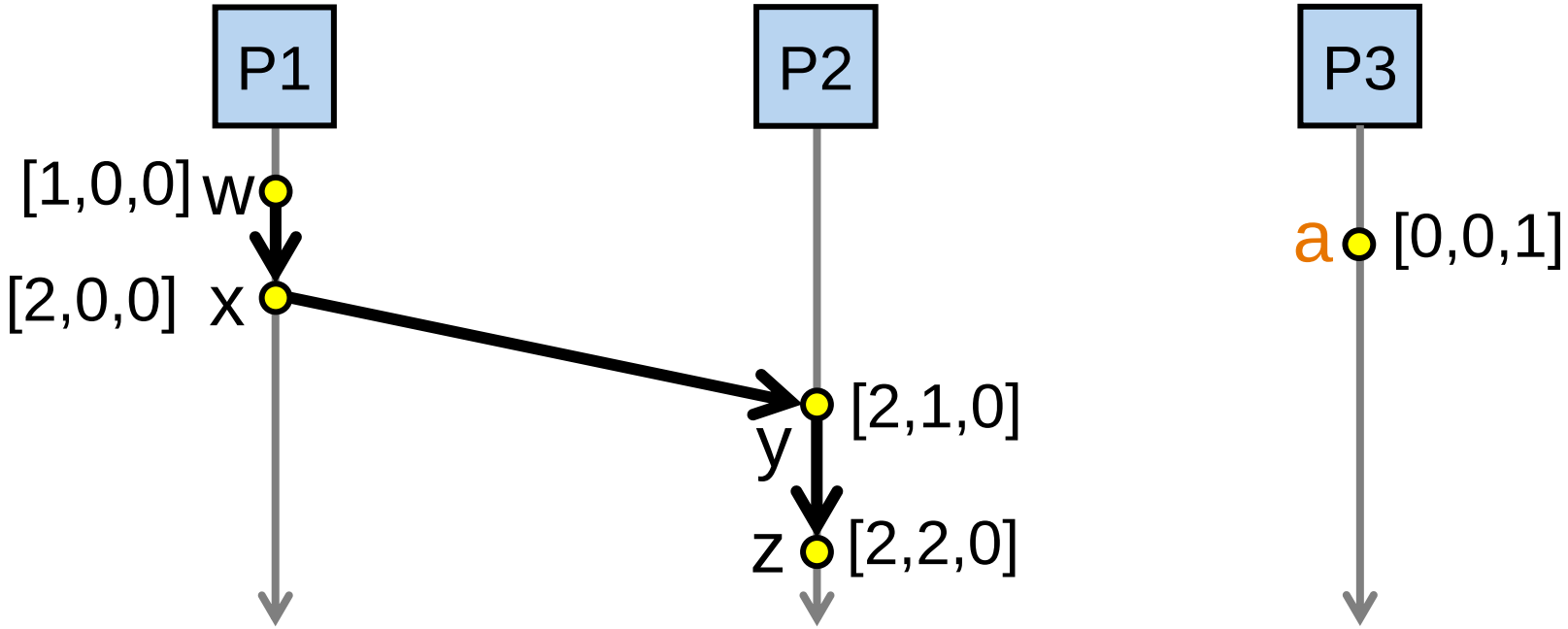
- Rule for comparing vector timestamps:
  - V(a) = V(b) when $a_k = b_k$ for all k
  - V(a) < V(b) when $a_k \leq b_k$ for all k and V(a) ≠ V(b)

- Concurrency:
  - V(a) || V(b) if $a_i < b_i$ and $a_j > b_j$, some i, j

Vector clocks capture causality

- V(w) < V(z) then there is a chain of events linked by Happens-Before (→) between a and z

- V(a) || V(w) then there is no such chain of events between a and w



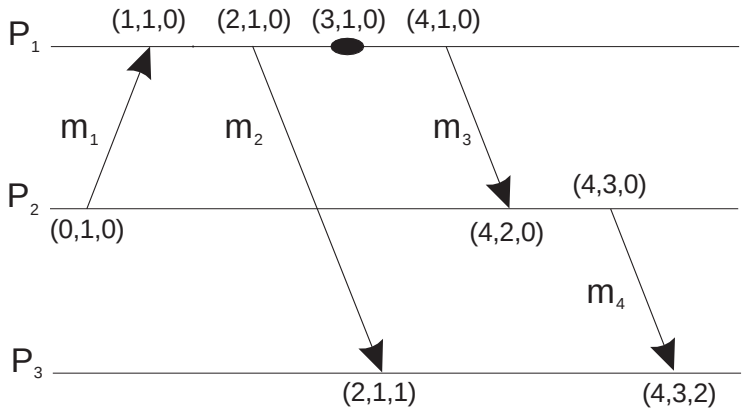
Comparing vector timestamps

- Rule for comparing vector timestamps:
  - V(a) = V(b) when $a_k = b_k$ for all k
    - They are the same event
  - V(a) < V(b) when $a_k \leq b_k$ for all k and V(a) ≠ V(b)
    - a → b

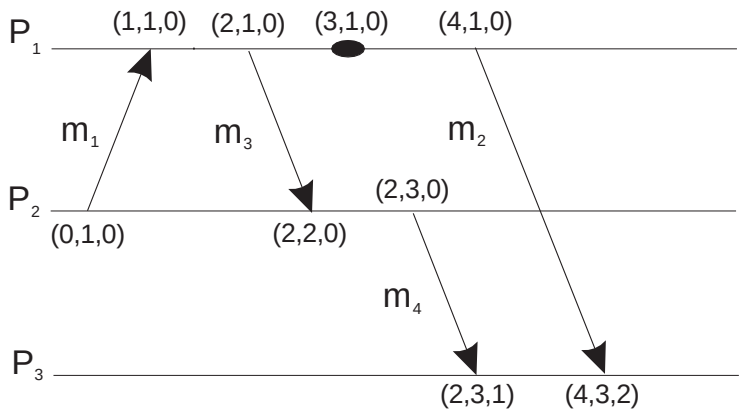
- Concurrency:
  - V(a) || V(b) if $a_i < b_i$ and $a_j > b_j$, some i, j
    - a || b

Vector clocks: Example

Capturing potential causality when exchanging messages



(a)



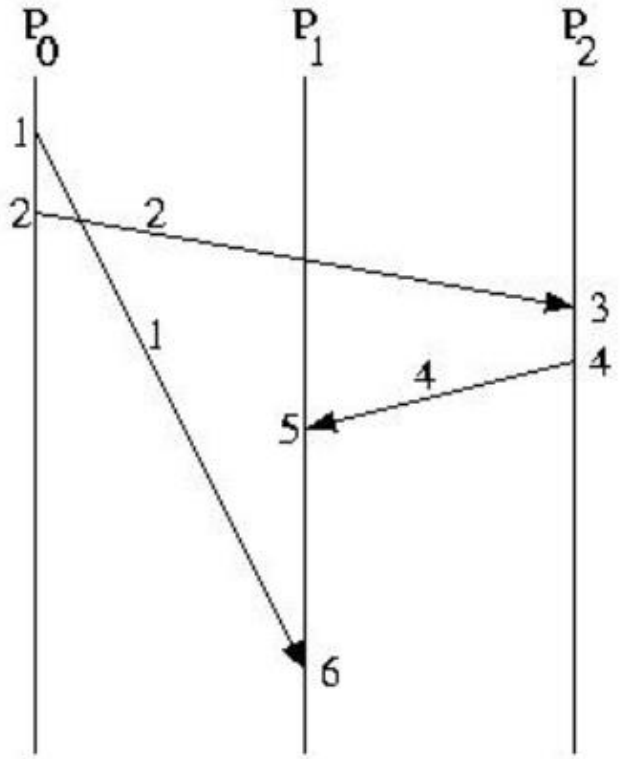
(b)

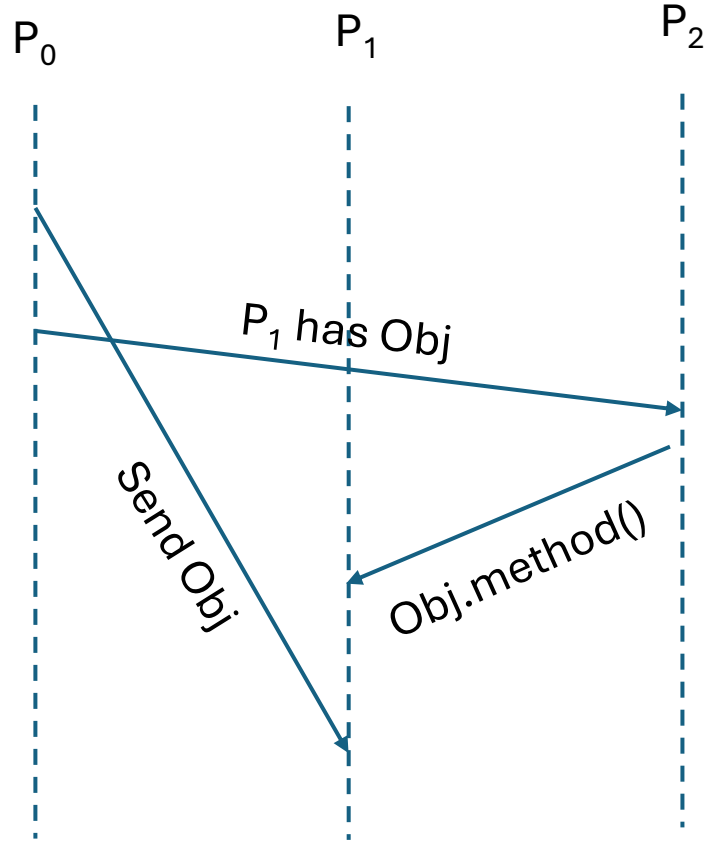
Analysis

Situation	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	**Conclusion**
(a)	(2, 1, 0)	(4, 3, 0)	Yes	No	m_2 may causally precede m_4
(b)	(4, 1, 0)	(2, 3, 0)	No	No	*Concurrent Events (may conflict)*

Causality Violations

- Consider this example (Source)



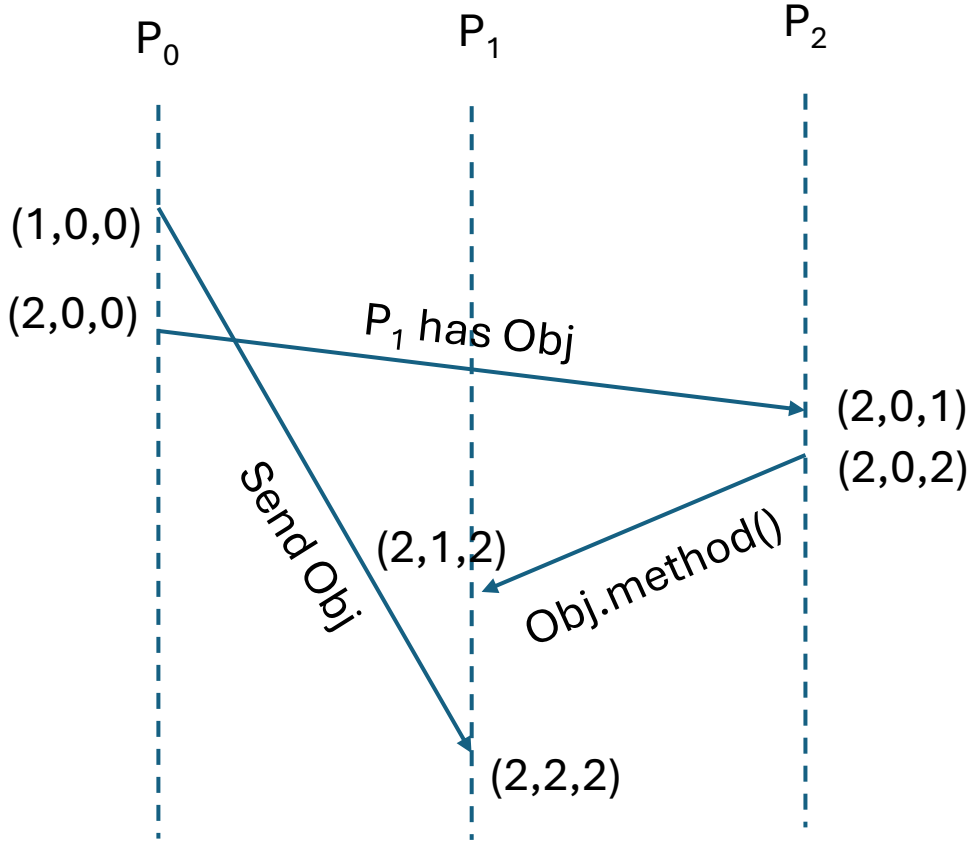


How can vector timestamps help here?

Causality Violations

Example (Source)

- To detect potential causality violations using vector timestamps:
  - Compare the timestamp of a newly received message to the local time.
  - If the message's timestamp is **less than** the local time vector, a (potential) causality violation has occurred.



What could have been done to avoid this scenario?

Mutual Exclusion in Distributed Systems

Mutual exclusion

Problem

A number of processes in a distributed system want exclusive access to some resource.

Basic solutions

Permission-based: A process wanting to enter its critical section, or access a resource, needs permission from other processes.

Token-based: A token is passed between processes. The one who has the token may proceed in its critical section, or pass it on when not interested.

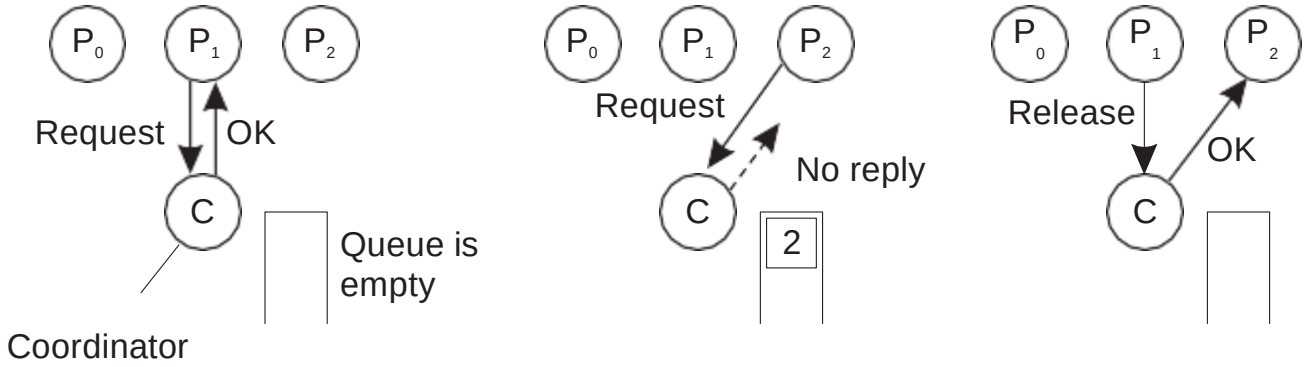
Properties of Token-based Solutions

- They provide guarantees for **safety** by avoiding what is known as **starvation**, as they can easily ensure that every process will get a chance at accessing the resource. They can also help with avoiding deadlocks.

- The main drawback of token-based solutions is when the token is lost (e.g. because the process holding it crashed), a careful distributed procedure needs to be started to ensure that a new token is created, but above all, **that it is also the only token.**

Permission-based, centralized

Simply use a coordinator



(a) Process P_1 asks the coordinator for permission to access a shared resource. Permission is granted.

(b) Process P_2 then asks permission to access the same resource. The coordinator does not reply.

(c) When P_1 releases the resource, it tells the coordinator, which then replies to P_2.

Drawbacks?

Mutual exclusion Ricart & Agrawala

This algorithm requires a total ordering of all events in the system as discussed earlier. That is, for any pair of events, such as messages, it must be unambiguous which one actually happened first.

When a process wants to access a shared resource, it builds a message containing the name of the resource, its process number, and the current (logical) time. It then sends the message to all other processes, conceptually including itself.

Return a response to a request only when:

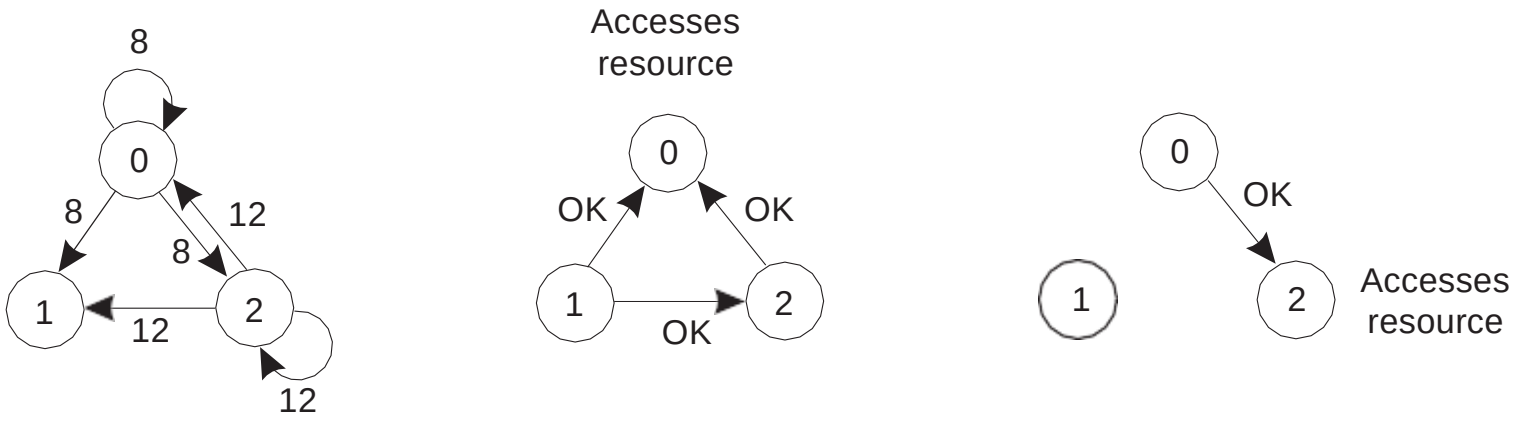
- The receiving process has no interest in the shared resource; or
- The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).

In all other cases, reply is deferred.

The above algorithm assumes that communication is reliable.

Mutual exclusion Ricart & Agrawala

Example with three processes



(a) (b) (c)

(a) Two processes want to access a shared resource at the same moment.

(b) P_0 has the lowest timestamp, so it wins.

(c) When process P_0 is done, it sends an *OK* also, so P_2 can now go ahead.

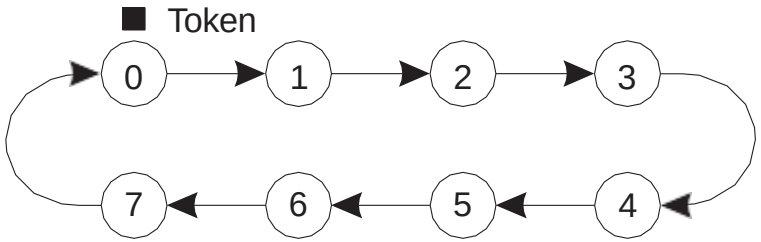
Drawbacks?

Mutual exclusion: Token ring algorithm

Essence

Organize processes in a logical ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to).

An overlay network constructed as a logical ring with a circulating token



Drawbacks?

Decentralized mutual exclusion

Principle

Assume every resource is replicated N times, with each replica having its own coordinator $\Rightarrow$ access requires a majority vote from $m > N/2$ coordinators. A coordinator always responds immediately to a request.

Assumption

When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted.

Decentralized mutual exclusion

How robust is this system?

- Let *p* be the probability that a coordinator resets.
- The probability P[*k*] that *k* out of *m* coordinators reset during the same interval is

$$\mathbb{P}[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

Decentralized mutual exclusion

How robust is this system?

- Let *p* be the probability that a coordinator resets.
- The probability P[*k*] that *k* out of *m* coordinators reset during the same interval is

$$\mathbb{P}[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

- *f* coordinators reset ⇒ correctness is violated when there is only a minority of nonfaulty coordinators: when $N-(m - f) >= m$, or, $f \geq 2m-N$.

Decentralized mutual exclusion

How robust is this system?

- Let *p* be the probability that a coordinator resets.
- The probability P[*k*] that *k* out of *m* coordinators reset during the same interval is

$$\mathbb{P}[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

- *f* coordinators reset ⇒ correctness is violated when there is only a minority of nonfaulty coordinators: when $N-(m - f) >= m$, or, $f \geq 2m-N$.
- The probability of a violation is $\sum_{k=2m - N}^{m}$ P[*k*].

What will happen if many nodes try to get access to the same resource at the same time?

Decentralized mutual exclusion

Violation probabilities for various parameter values

N	m	p	Violation
8	5	3 sec/hour	$< 10^{-5}$
8	6	3 sec/hour	$< 10^{-11}$
16	9	3 sec/hour	$< 10^{-4}$
16	12	3 sec/hour	$< 10^{-21}$
32	17	3 sec/hour	$< 10^{-4}$
32	24	3 sec/hour	$< 10^{-43}$

N	m	p	Violation
8	5	30 sec/hour	$< 10^{-3}$
8	6	30 sec/hour	$< 10^{-7}$
16	9	30 sec/hour	$< 10^{-2}$
16	12	30 sec/hour	$< 10^{-13}$
32	17	30 sec/hour	$< 10^{-2}$
32	24	30 sec/hour	$< 10^{-27}$

Table from 4th edition

Mutual exclusion: comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)
Centralized	3	2
Distributed	$2(N-1)$	$2(N-1)$
Token ring	$1, \ldots, \infty$	$0, \ldots, N-1$
Decentralized	$2kN + (k-1)N/2 + N, k = 1, 2, \ldots$	$2kN + (k-1)N/2$

Table from 4th edition

This algorithm requires sending N messages to coordinators, and receiving up to N responses, i.e., 2N.
If it does not get a majority, it will have to release (at most) N/2 votes.
A process may need to go through k ≥ 1 attempts, of which (k-1) will fail.

Finally, if it did get enough votes, it will have to send an additional N release messages later while releasing the lock.

Election algorithms

Principle

An algorithm requires that some process acts as a coordinator. The question is how to select this special process dynamically.

Note

In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions ⇒ single point of failure.

Basic Assumptions in the Next Algorithms

- All processes have unique id's
- All processes know id's of all processes in the system (but not if they are up or down)
- Election means identifying the process with the highest id that is up

Election by bullying

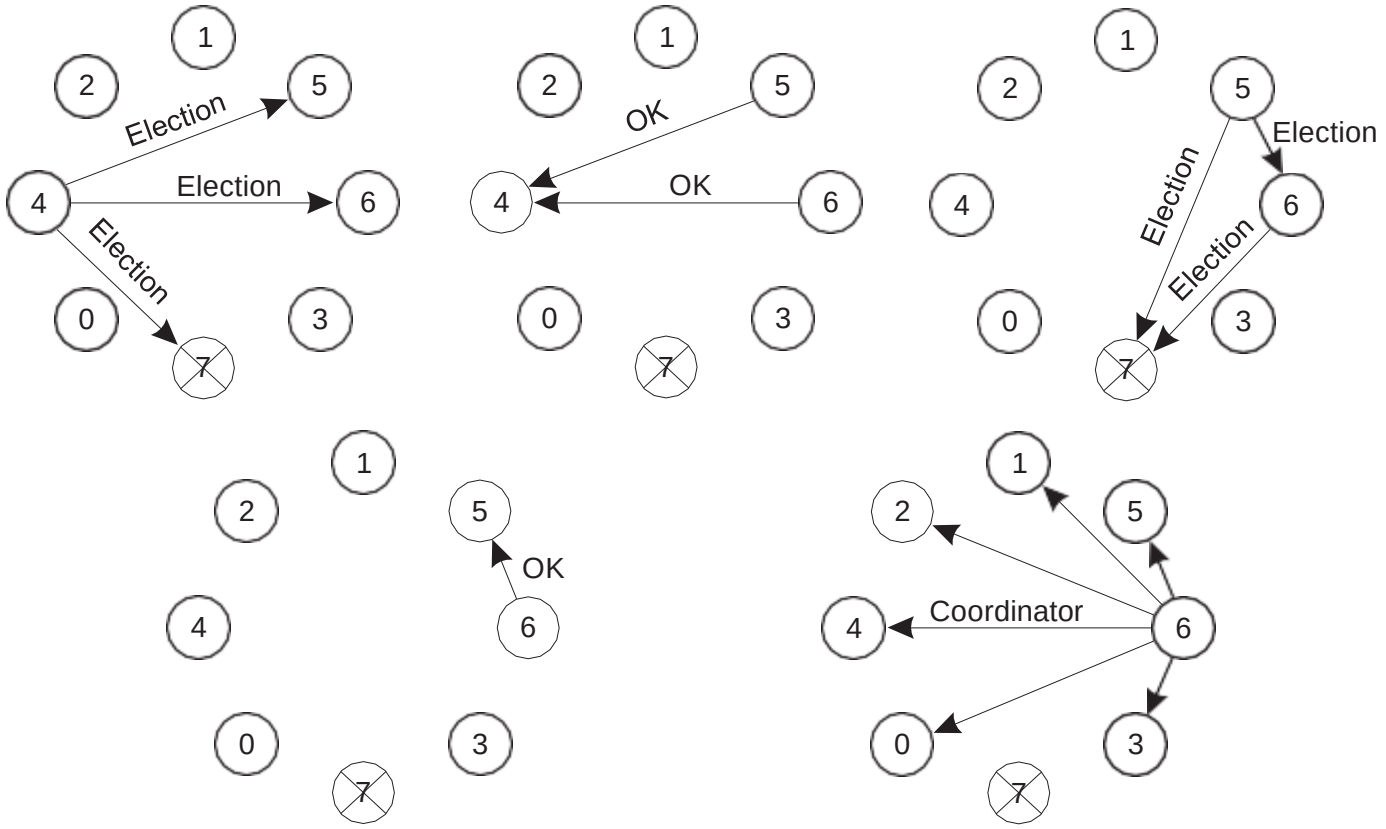
Principle

Consider N processes $\{P_0, \ldots, P_{N-1}\}$ and let $id(P_k) = k$. When a process P_k notices that the coordinator is no longer responding to requests, it initiates an election:

1. P_k sends an *ELECTION* message to all processes with higher identifiers: $P_{k+1}, P_{k+2}, \ldots, P_{N-1}$.
2. If no one responds, P_k wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over and P_k's job is done.

Election by bullying

The bully election algorithm



Note: If a process recovers from failure, it can hold an election in the same way.
If P7 is ever restarted (the one with the highest ID), it will send all the others a COORDINATOR message

Election in a ring

Principle

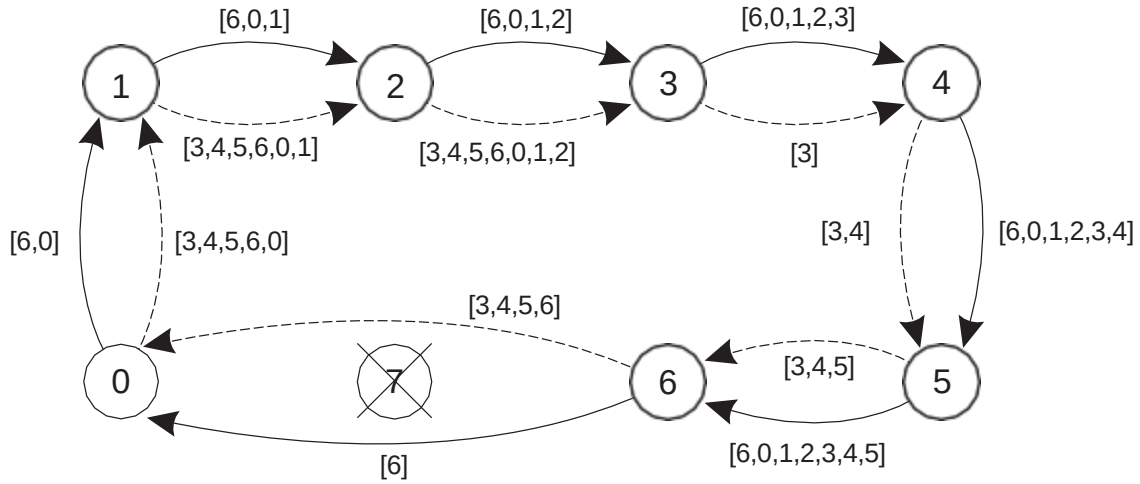
Process priority is obtained by organizing processes into a (logical) ring. Process with the highest priority should be elected as coordinator.

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.
- If a message is passed on, the sender adds itself to the list. When it gets back to the initiator, everyone had a chance to make its presence known.
- The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.

Note: This coordinator message will also help the nodes know who the members of the new ring are.

Election in a ring

Election algorithm using a ring



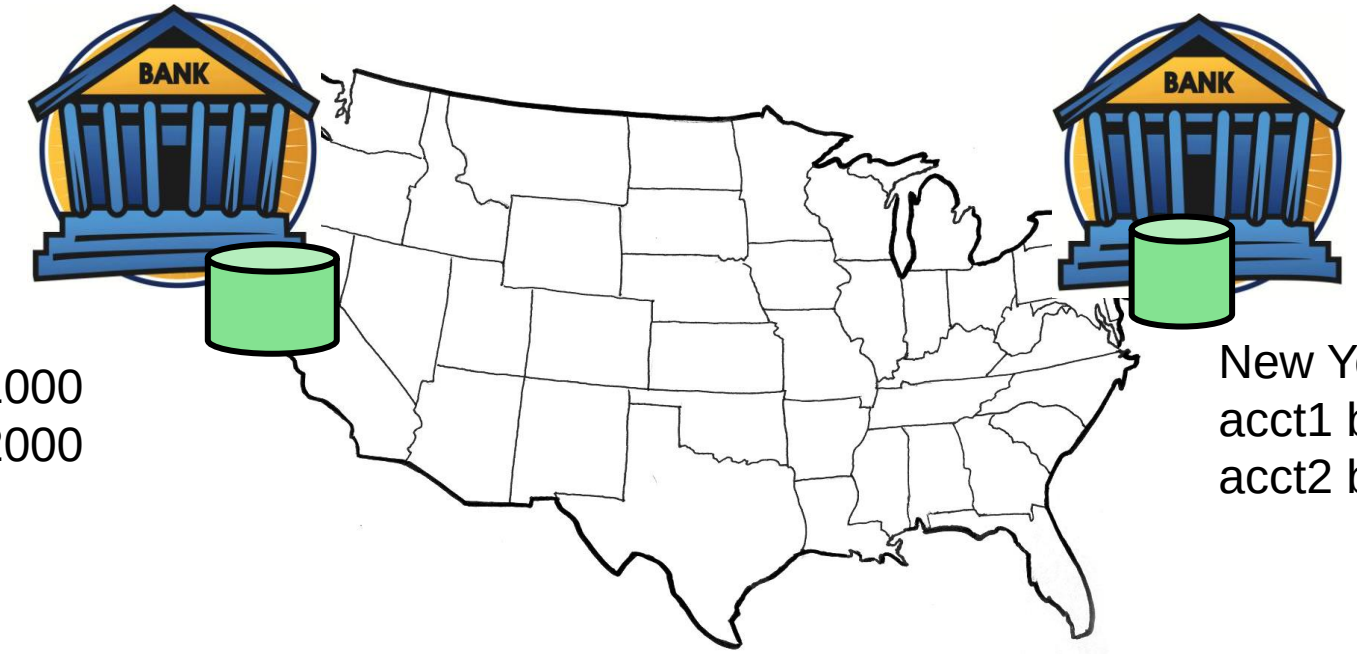
- The solid line shows the election messages initiated by P_6
- The dashed one the messages by P_3

Notes: This figure just shows points 1 and 2 in the previous slide.
Later, P3 and P6 will convert the messages they receive eventually into type COORDINATOR and send them again across the ring. This will help the network identify who the coordinator is, and the members of the new ring.

Distributed Snapshots

Distributed Snapshots

- What is the state of a distributed system?



San Francisco
acct1 balance = $1000
acct2 balance = $2000

New York
acct1 balance = $1000
acct2 balance = $2000

System model

- N processes in the system with no process failures
  - Each process has some state it keeps track of

- There are two first-in, first-out, unidirectional channels between every process pair P and Q
  - Call them channel(P, Q) and channel(Q, P)
  - The channel has state, too: the set of messages inside
  - All messages sent on channels arrive intact, unduplicated, in order

Aside: FIFO communication channel

- “All messages sent on channels arrive intact, unduplicated, in order”

- Q: Arrive?
- Q: Intact?
- Q: Unduplicated?
- Q: In order?

- At-least-once retransmission
- Network layer checksums
- At-most-once deduplication
- Sender includes sequence numbers, receiver only delivers in sequence order

- TCP provides all of these when processes don’t fail

Global snapshot is global state

- Each distributed application has a number of processes running on a number of physical servers

- These processes communicate with each other via channels

- A global snapshot captures
  1. The local states of each process (e.g., program variables), and
  2. The state of each communication channel

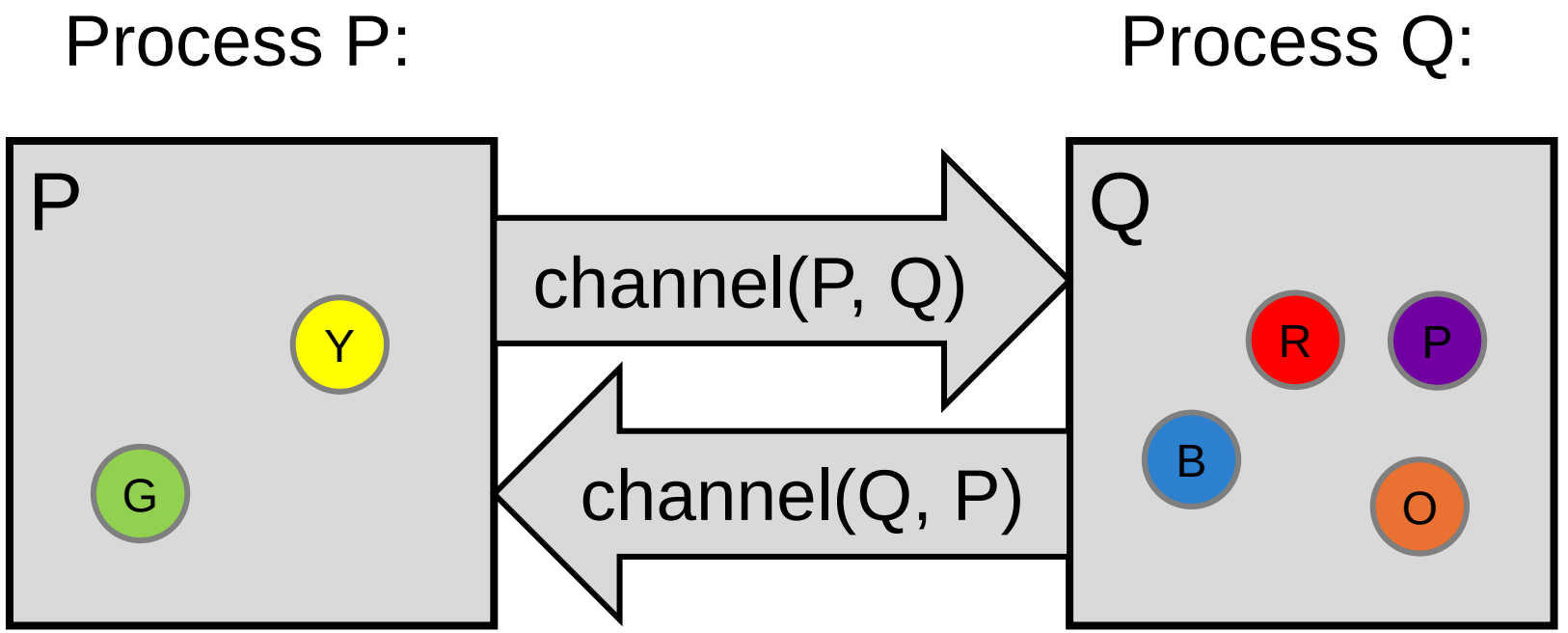
Why do we need snapshots?

- Checkpointing: Restart if the application fails
- Detecting deadlocks: The snapshot can examine the current application state
  - Process A grabs Lock 1, B grabs 2, A waits for 2, B waits for 1... ... ...
- Other debugging: A little easier to work with than printf…

System model: Graphical example

- Let's represent process state as a set of colored tokens

- Suppose there are two processes, P and Q:



Correct global snapshot = Exactly one of each token

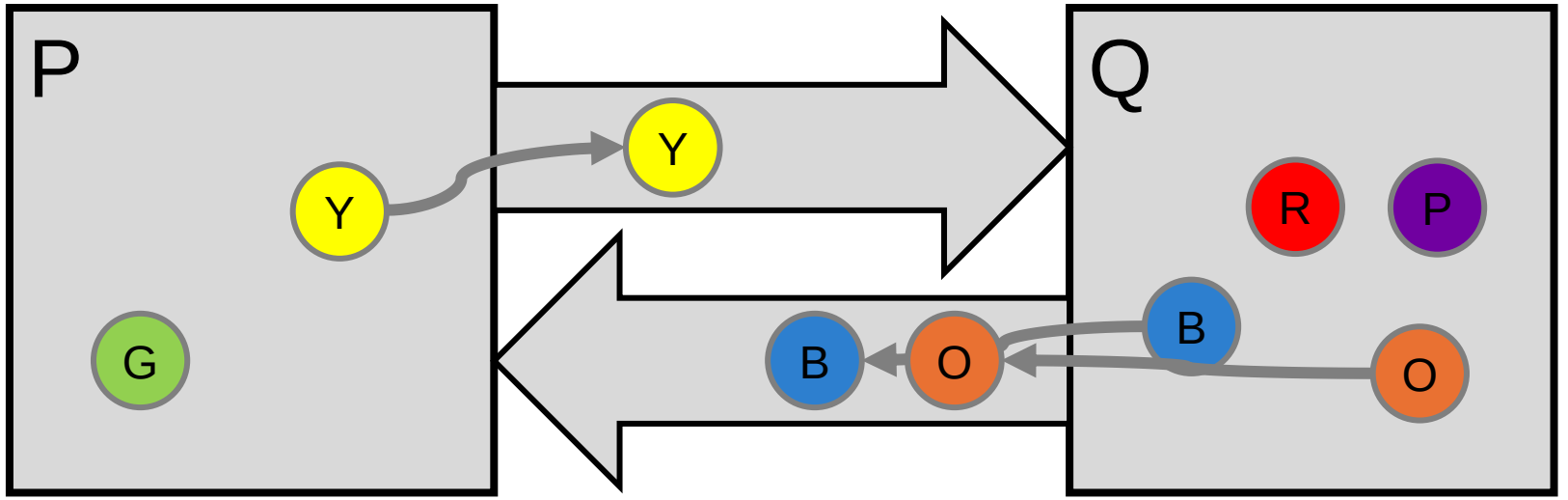
When is inconsistency possible?

- Suppose we take snapshots only from a process perspective
- Suppose snapshots happen independently at each process
- Let’s look at the implications...

Problem: Disappearing tokens

- P, Q put tokens into channels, then snapshot

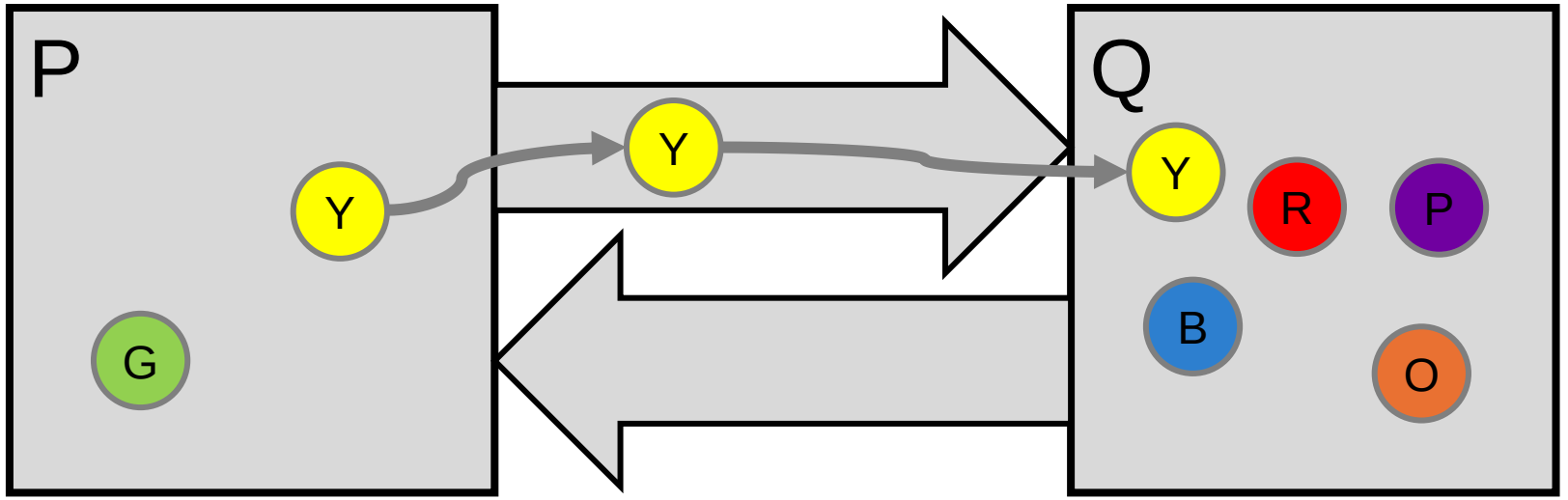
This snapshot misses Y, B, and O tokens



Problem: Duplicated tokens

- P snapshots, then sends Y
- Q receives Y, then snapshots

This snapshot duplicates the Y token



P = { G, Y }

Q = { Y, R, P, B, O }

Idea: “Marker” messages

- What went wrong? We should have captured the state of the channels as well

- Let’s send a marker message ▲ to track this state
  - Distinct from other messages
  - Channels deliver marker and other messages FIFO

Chandy-Lamport Algorithm: Overview

- We'll designate one node (say P) to start the snapshot
  - Without any steps in between, P:
    1. Records its local state ("snapshots")
    2. Sends a marker on each outbound channel

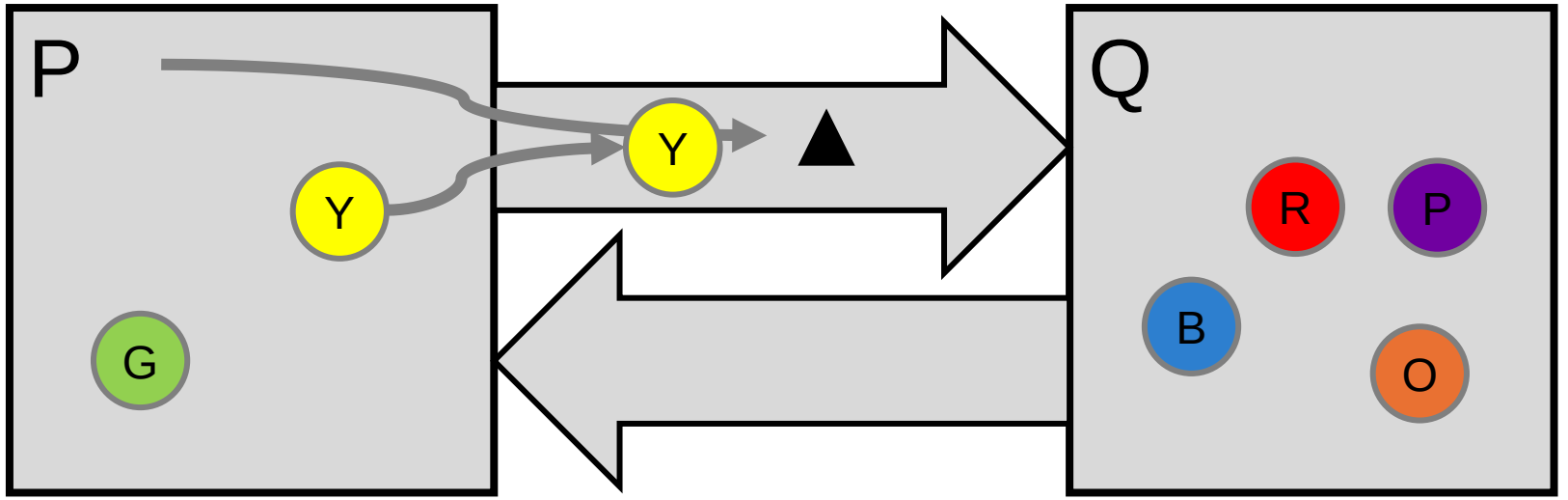
- Nodes remember whether they have snapshotted

- On receiving a marker, a non-snapshotted node performs steps (1) and (2) above

Chandy-Lamport: Sending process

- P snapshots and sends marker, then sends Y

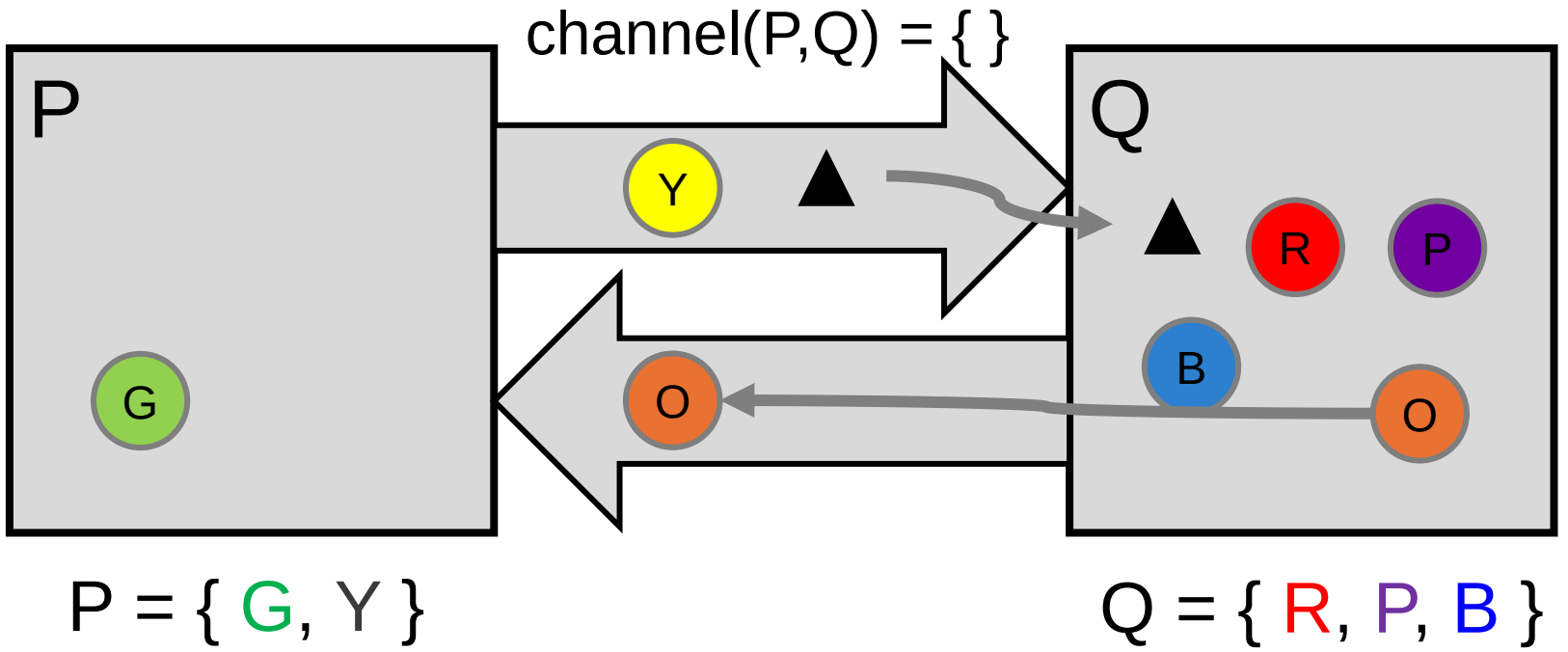
- Send Rule: Send marker on all outgoing channels
  - Immediately after snapshot
  - Before sending any further messages



snap: P = { G, Y }

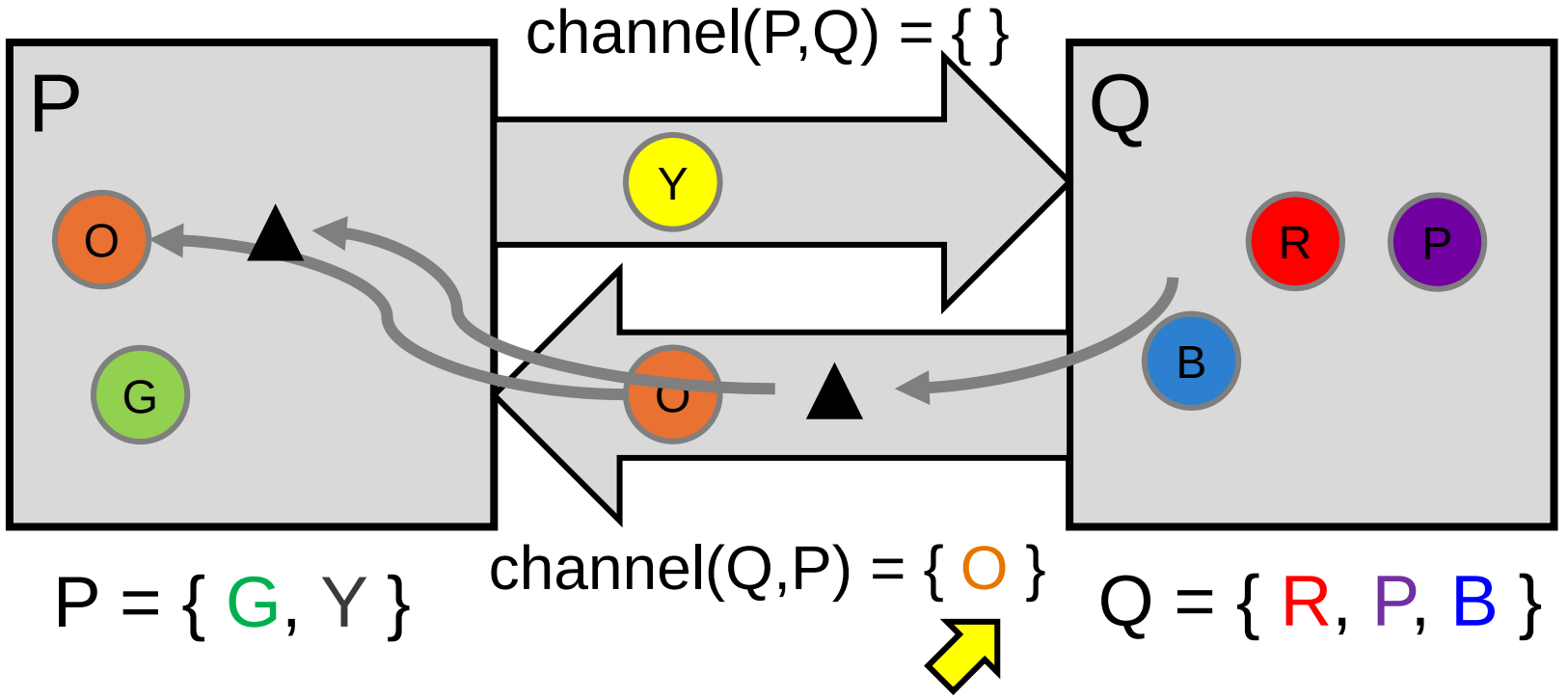
Chandy-Lamport: Receiving process (1/2)

- At the same time, Q sends orange token O
- Then, Q receives marker ▲
- Receive Rule (if not yet snapshotted)
  - On receiving marker on channel c record c's state as empty



Chandy-Lamport: Receiving process (2/2)

- Q sends marker to P
- P receives orange token O, then marker ▲
- Receive Rule (if already snapshotted):
  - On receiving marker on c record c's state: all msgs from c since snapshot



Terminating a Snapshot

- Distributed algorithm: No one process decides when it terminates
- Eventually, all processes have received a marker (and recorded their own state)
- All processes have received a marker on all the N–1 incoming channels (and recorded their states)
- Later, a central server can gather the local states to build a global snapshot