Lecture 23

- Administration

Chapter 4: Global State and snapshot recording algorithms

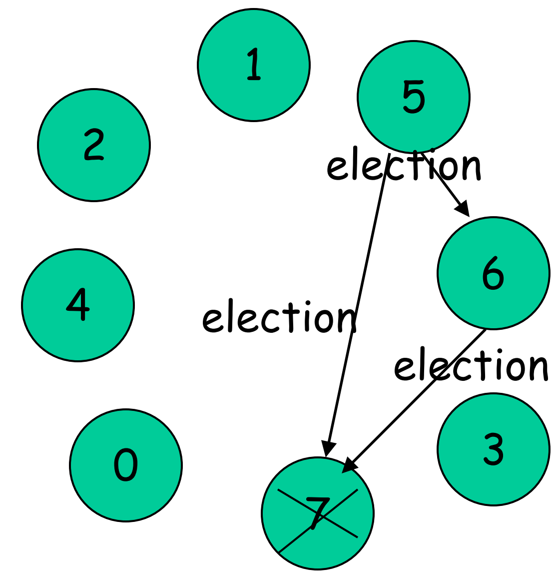
Bully Algorithm

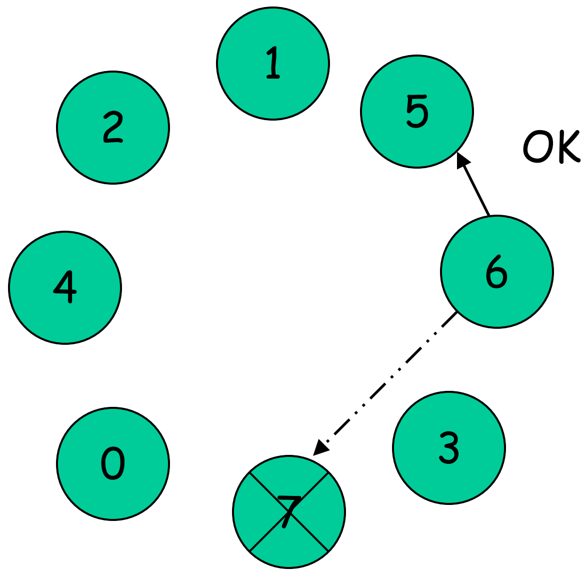
1. Send *election* message *(I want to be the leader)* to processes with *larger id*
2. Give up your bid if a process with *larger id* sends a *reply* message *(means no, you cannot be the leader).* In that case, wait for the *leader* message (*I am the leader*). Otherwise elect yourself the leader and send a *leader* message
3. If *no reply is received*, then elect yourself the leader, and broadcast a *leader* message.
4. If you receive a reply, but later don't receive a *leader* message from a process of larger id (i.e the leader-elect has crashed), then re-initiate election by sending *election* message.

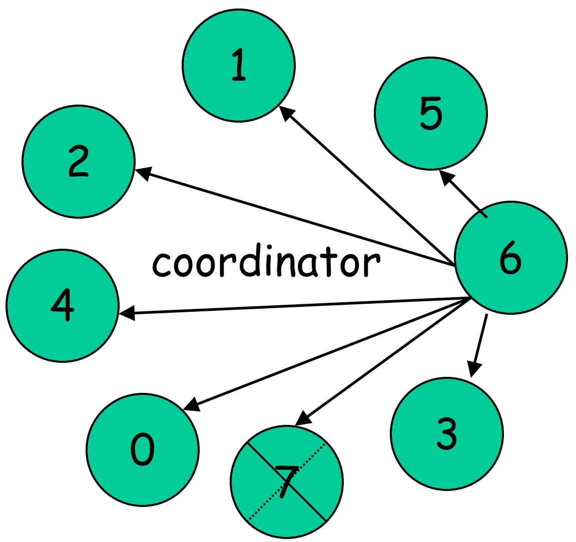
The process *q* now calls an election (if it has not already done so).

Repeat until no higher-level process responds. The last process to call an election “wins” the election.

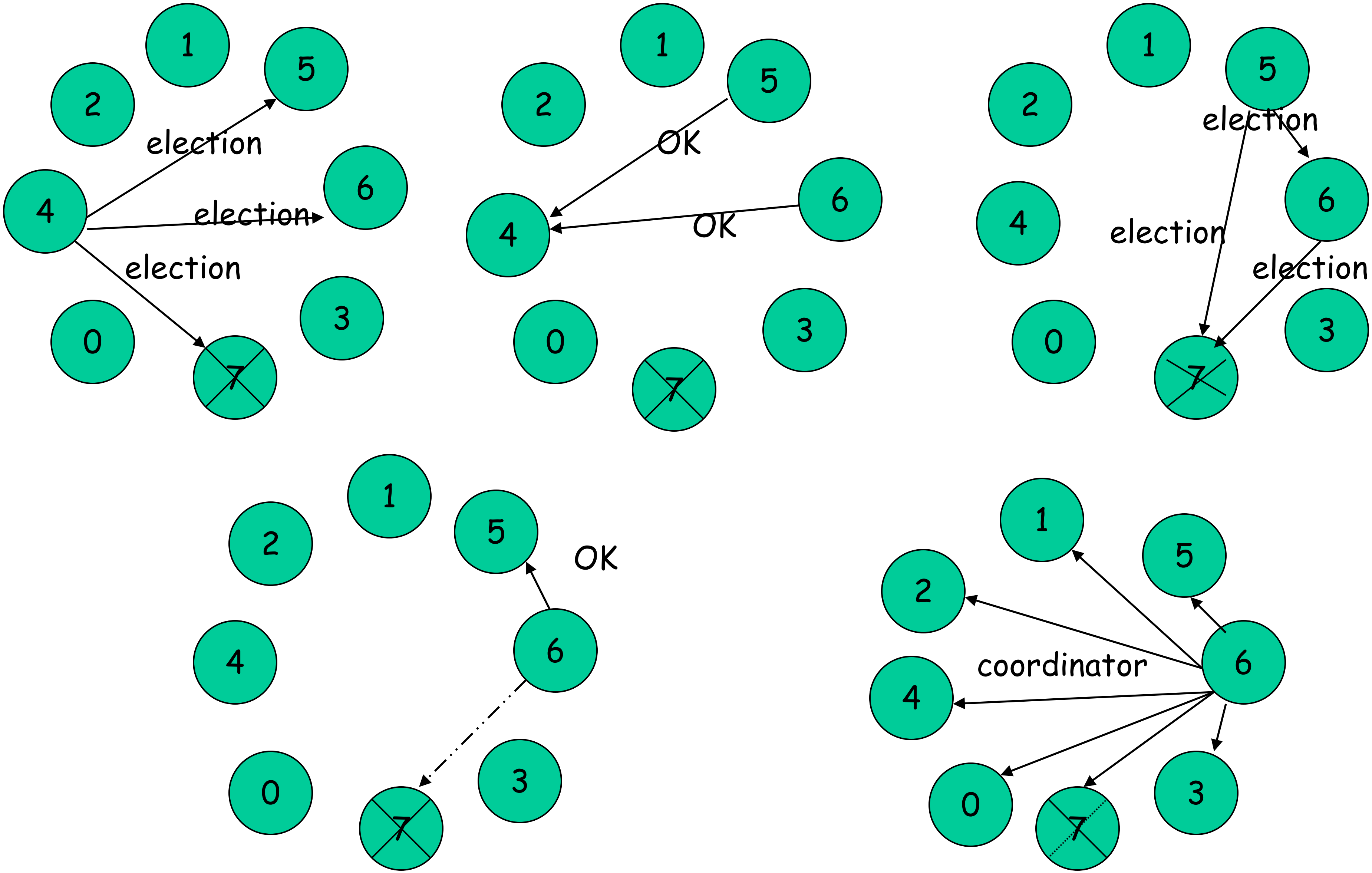
The winner sends a message to other processes announcing itself as the new coordinator.







If 7 comes back on line, it will call an election



1
2
5
4
6
0
3
7
election
election
election
OK
OK
election
election
election
OK
coordinator

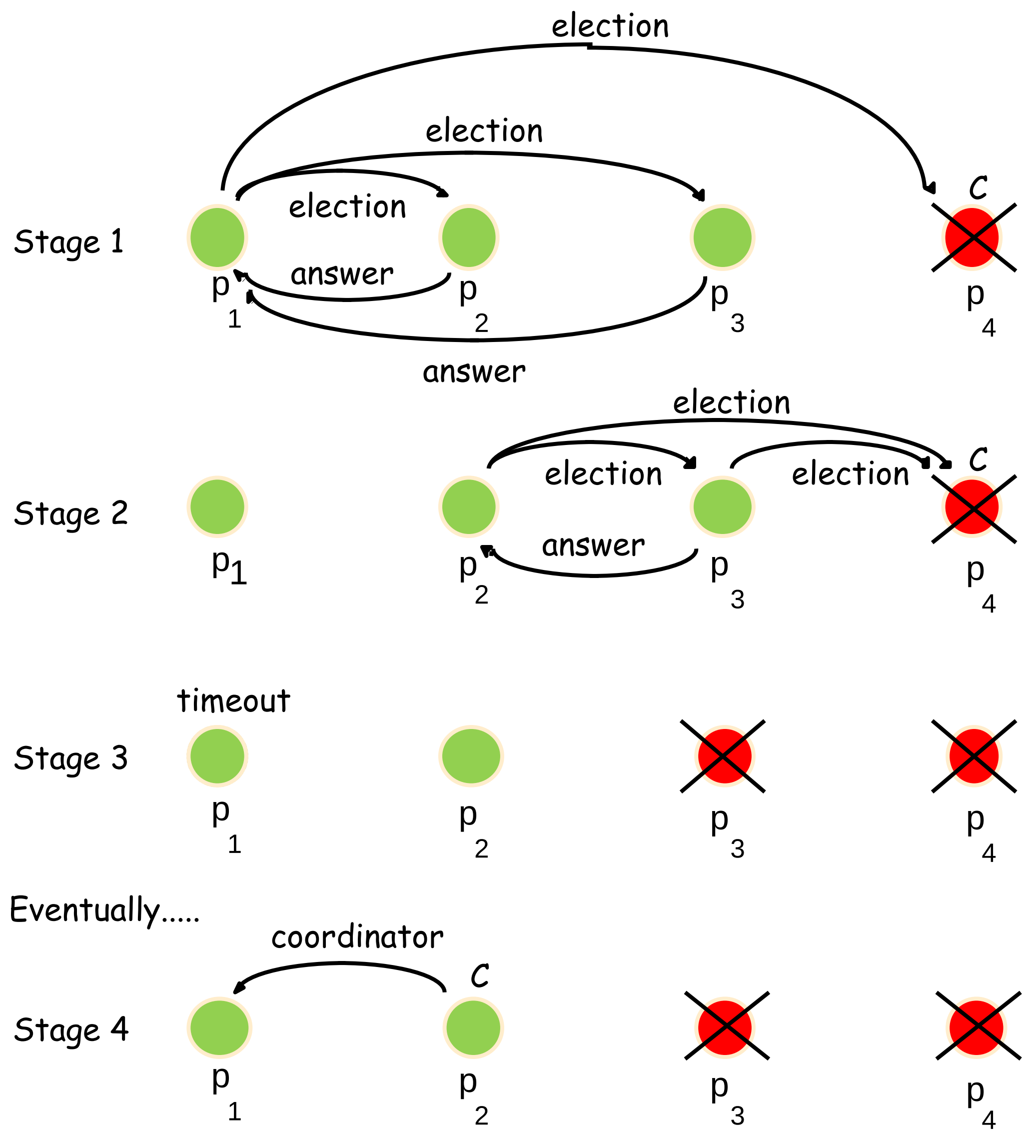
The Bully Algorithm

The coordinator p_4 fails and p_1 detects this starts election

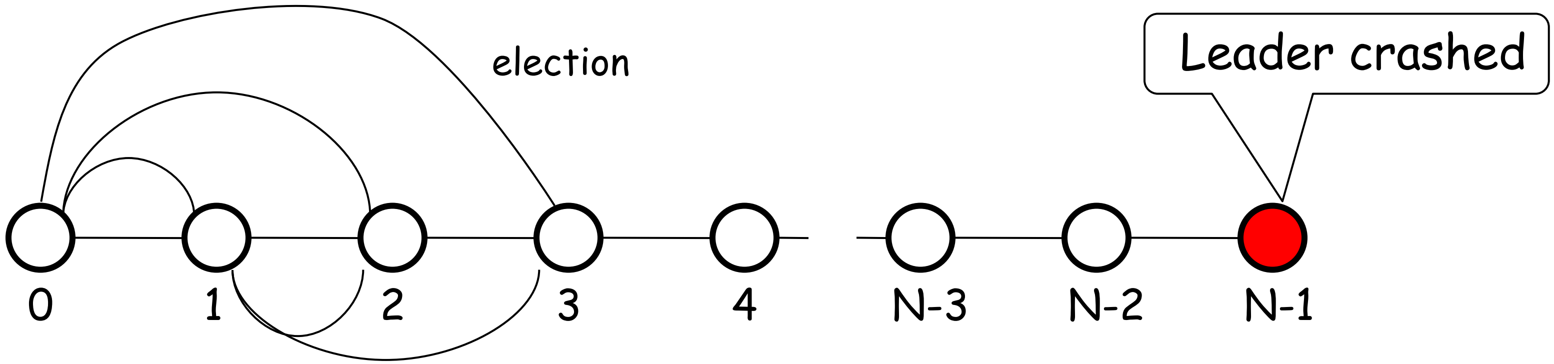
On the message from p_1, p_2 and p_3 start their own election

Before p_3 can announce victory it fails, assuming p_1 timeouts first.

Eventually p_2 can announce victory .



Worst Case



Node 0 sends N-1 election messages   So, 0 starts all over again
Node 1 sends N-2 election messages
Node N-2 sends 1 election messages etc

Finally, node N-2 will be elected leader, but before it sent the leader message, it crashed.

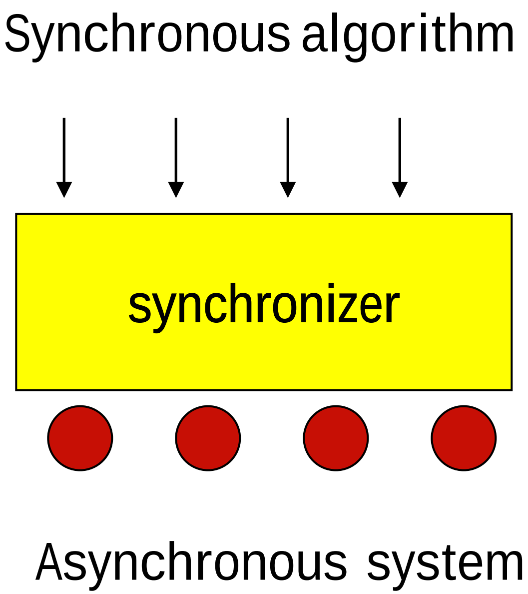
The worst-case message complexity = **$O(n^3)$ (This is bad)**

Synchronizers

Synchronous algorithms (round-based, where processes execute actions in lock-step synchrony) are easer to deal with than ***asynchronous algorithms***. In each round, a process

(1) receives messages from neighbors,
(2) performs local computation
(3) sends messages to ≥ 0 neighbors

A synchronizer is a protocol that enables synchronous algorithms to run on asynchronous platforms



Synchronizers

Simulate a synchronous network over an asynchronous underlying network

Possible in the absence of failures

Enables us to use simple synchronous algorithms even when the underlying network is asynchronous

Synchronous network abstraction: A message sent in *pulse* i is received at pulse i+1

Synchronizer indicates when a process can generate a pulse

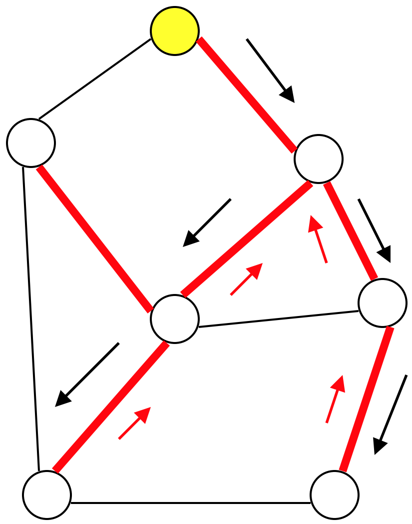
A process can go from pulse i to i+1 only when it has received and acted on all messages sent during pulse i-1

Synchronizers

In each pulse:

- A process receives messages sent during the previous pulse
- It then performs internal computation and sends out messages if required
- It can execute the next pulse only when the synchronizer permits it

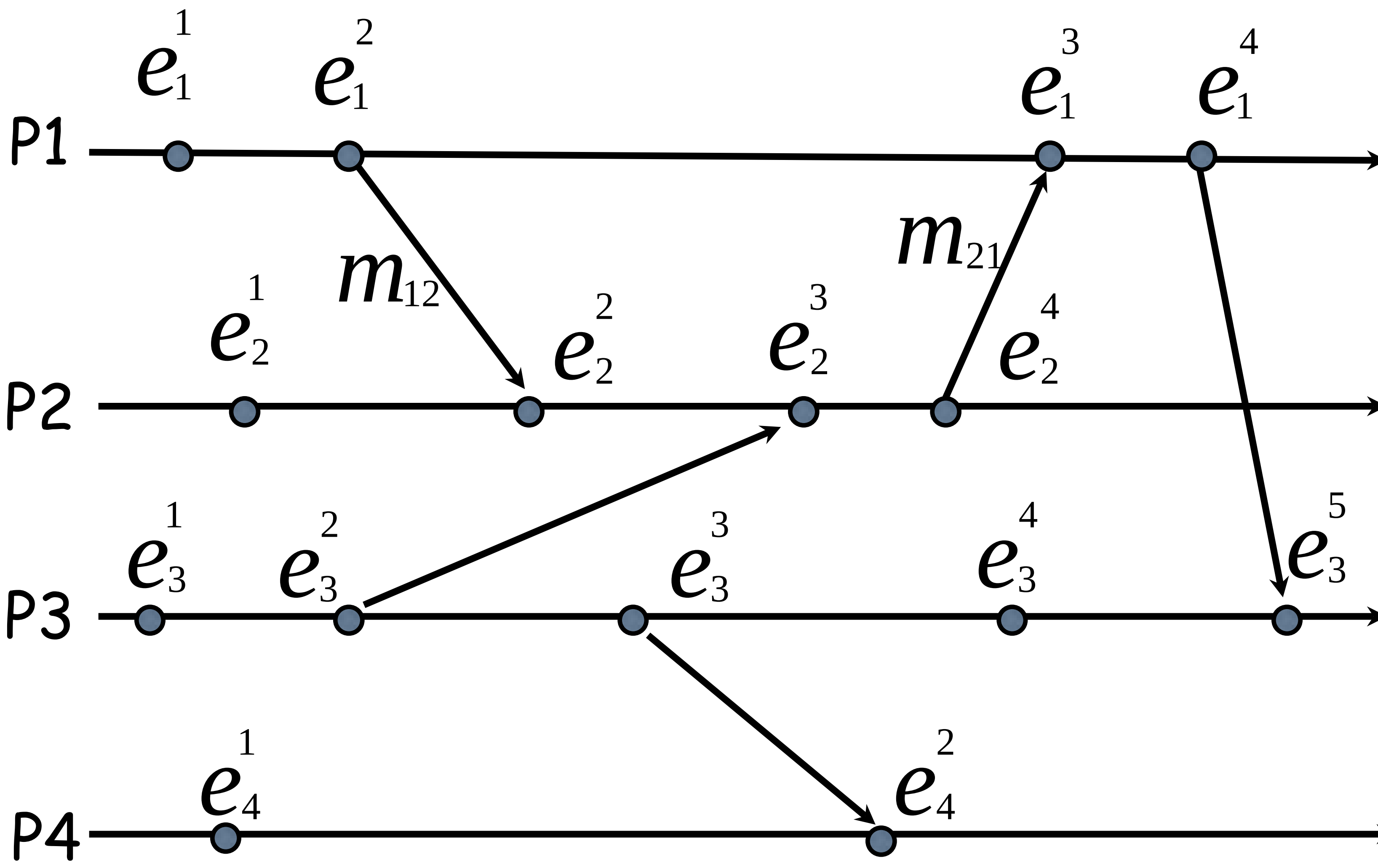
Tree synchronizer



Form a spanning tree with any node as the root. The root initiates the simulation of each tick by sending message m(j) for each clock tick j down the tree. Each process responds with ack(j) and then with a safe(j) message (that represents the fact that the entire subtree under it is safe). When the root receives safe(j) from every child, it initiates the simulation of clock tick (j+1)

Message complexity $M(\beta)$ = 3 (N-1)
since three messages (m, ack, safe) flow along each edge of the tree.

Time complexity $T(\beta)$ = depth of the tree.
For a balanced tree, this is O(log N)


P1
P2
P3
P4
e_1^1
e_1^2
e_1^3
e_1^4
m_{12}
m_{21}
e_2^1
e_2^2
e_2^3
e_2^4
e_3^1
e_3^2
e_3^3
e_3^4
e_3^5
e_4^1
e_4^2

State of Channel

All messages that have been sent but not yet received.

$$S_{ij}^{x,y} = \left\{ m_{ij} : \mathrm{send}(m_{ij}) \le \mathrm{recv}(m_{i,j}) > LS_j^y \right\}$$

LS_j^y The state of process j after the occurrence of event e_j^y

Global State

$$GS = \left\{ \bigcup_i LS_i^{x_i}, \bigcup_{j,k} S_{jk}^{y_j, z_k} \right\}$$

Consistent or Inconsistent

Terms

- Concurrent
- Cut some Global State
- Consistent, transitless (no outstanding messages), strongly consistent (consistent and transitless)

Chandy and Lamport Snapshot

Marker-Sending Rule for a Process p:

p records its state;

for (each channel *C* directed away from p, with a marker not sent)

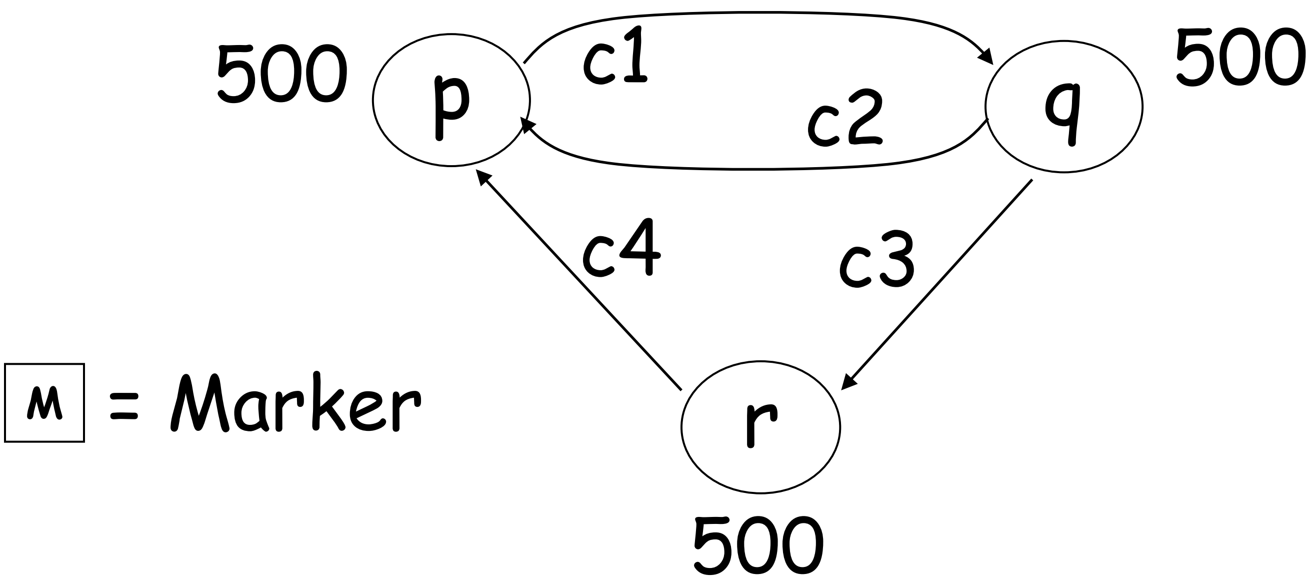
{ p sends one marker along *C* before p sends any further messages along *C*; }

Marker-Receiving Rule for a Process q:

if (q has not recorded its state) then
{ q records the state of *C* as the empty sequence;
execute marker-sending rule.
}
else { q records the state of *C* as the sequence of message received along *C* after q's state was recorded and before q received the marker along *C*.
}

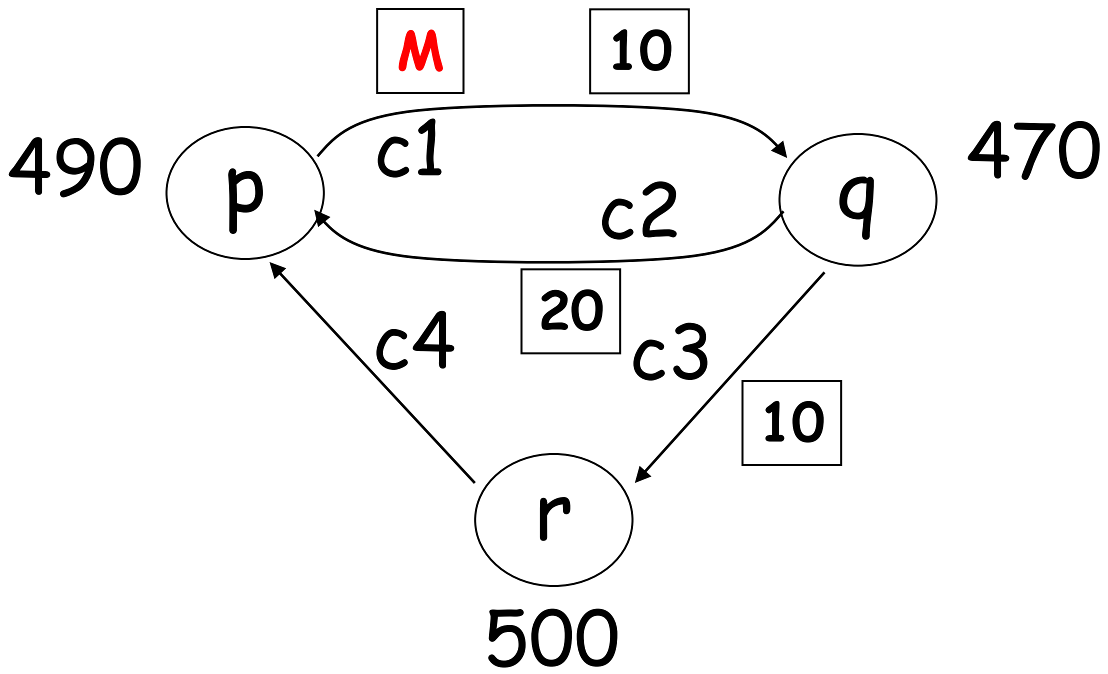
- When a process receives a mark, it knows that a snapshot is in process.
- An individual node knows that it is done when it records its own state and all the states in my incoming channels.

Example -- initial



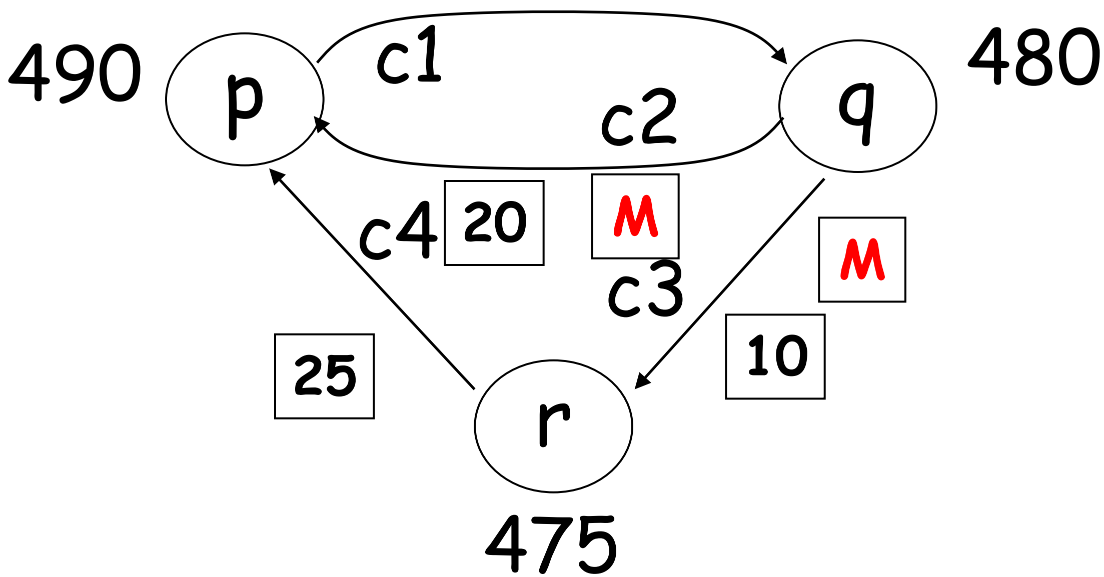
Node	Recorded state				
		c1	c2	c3	c4
p			{}		{}
q		{}			
r				{}	

Example - step 1



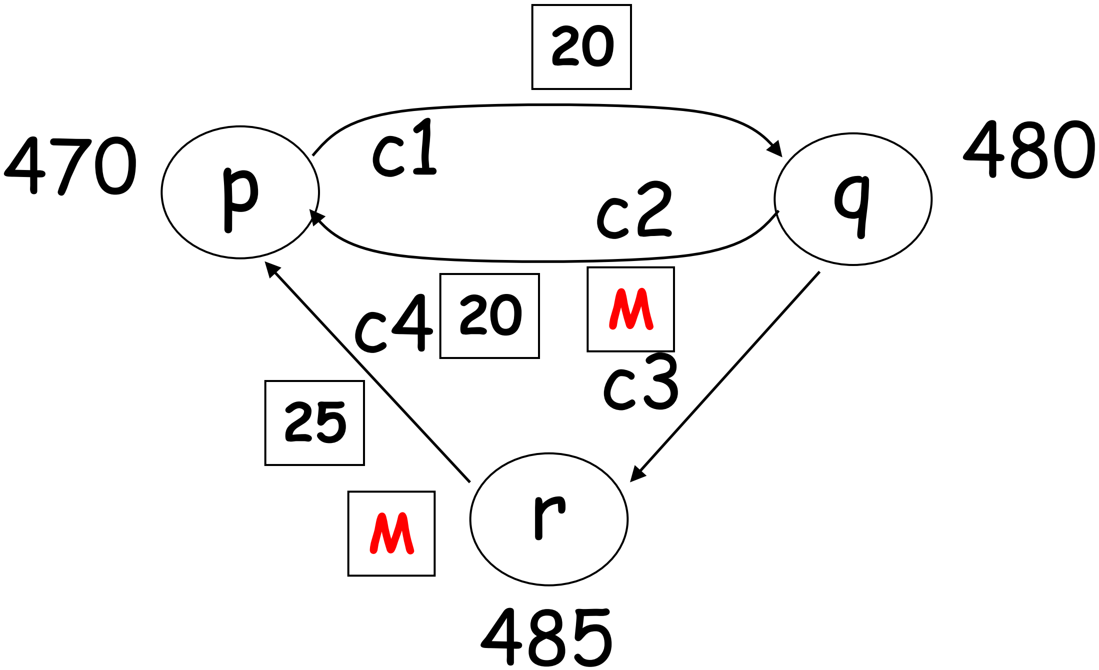
Node	Recorded state				
	state	c1	c2	c3	c4
p	490		{}		{}
q		{}			
r				{}	

Example - step 2



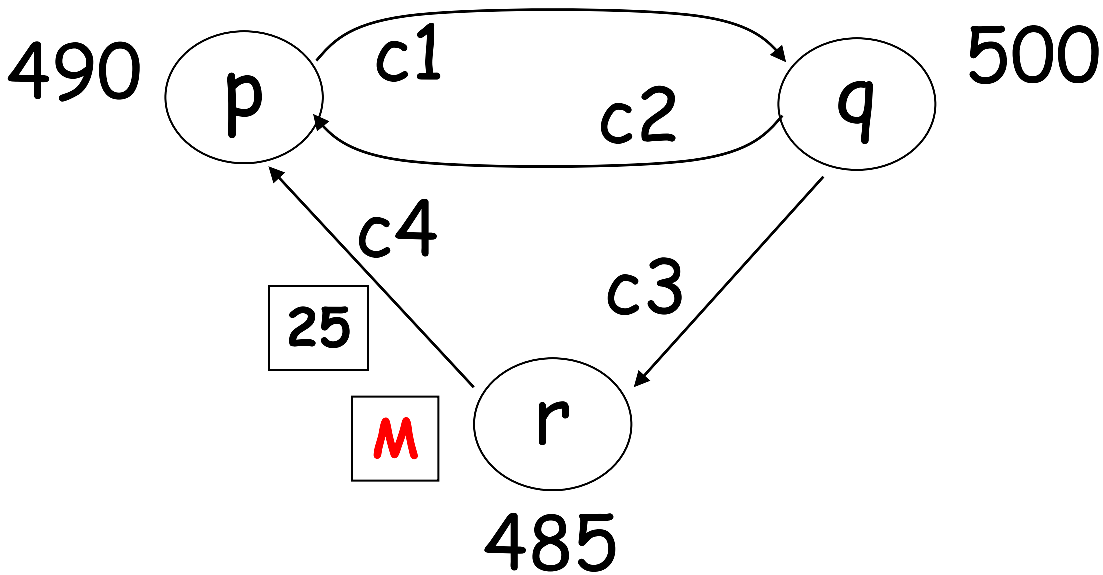
Node	Recorded state				
	state	c1	c2	c3	c4
p	490		{}		{}
q	480	{empty}			
r				{}	

Example - step 3



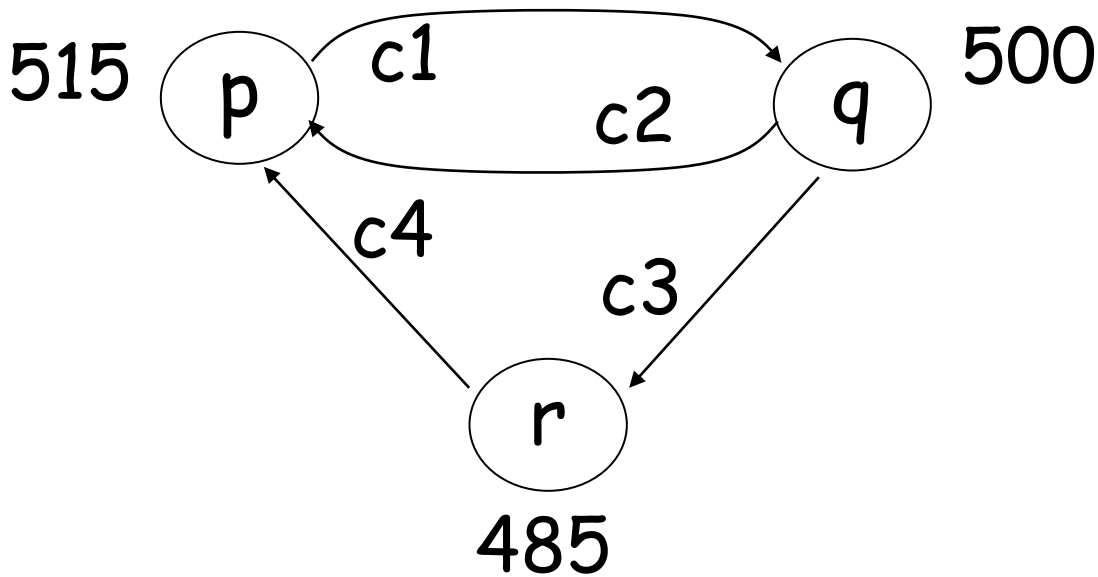
Node	Recorded state				
	state	c1	c2	c3	c4
p	490		{}		{}
q	480	{empty}			
r	485			{empty}	

Example - step 4



Node	Recorded state				
	state	c1	c2	c3	c4
p	490		{20}		{}
q	480	{empty}			
r	485			{empty}	

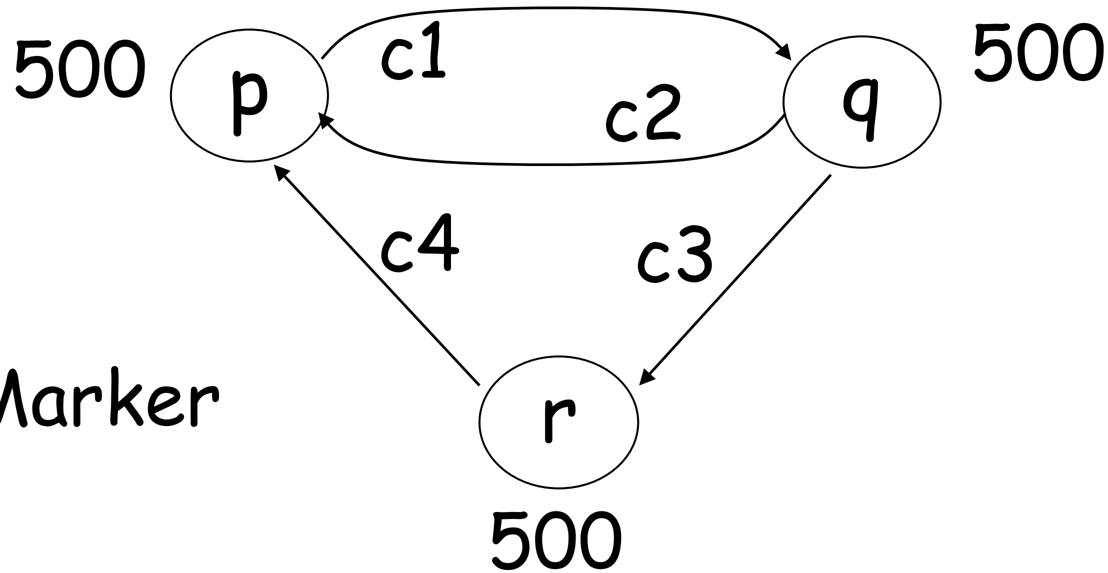
Example - step 5



Node	Recorded state				
	state	c1	c2	c3	c4
p	490		{20}		{25}
q	480	{empty}			
r	485			{empty}	

What if more than one initiate?

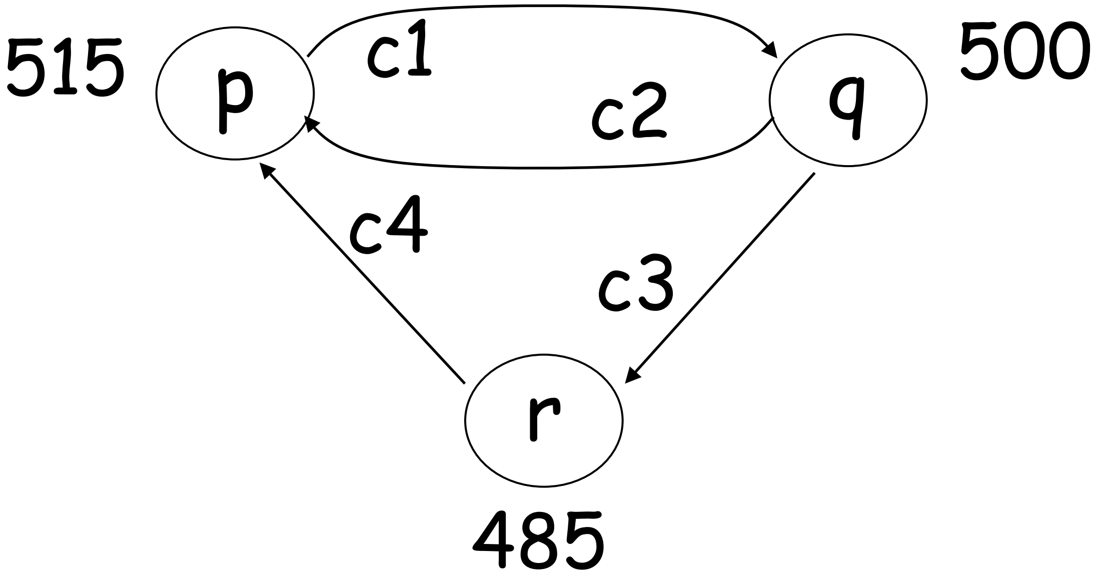
Example -- initial



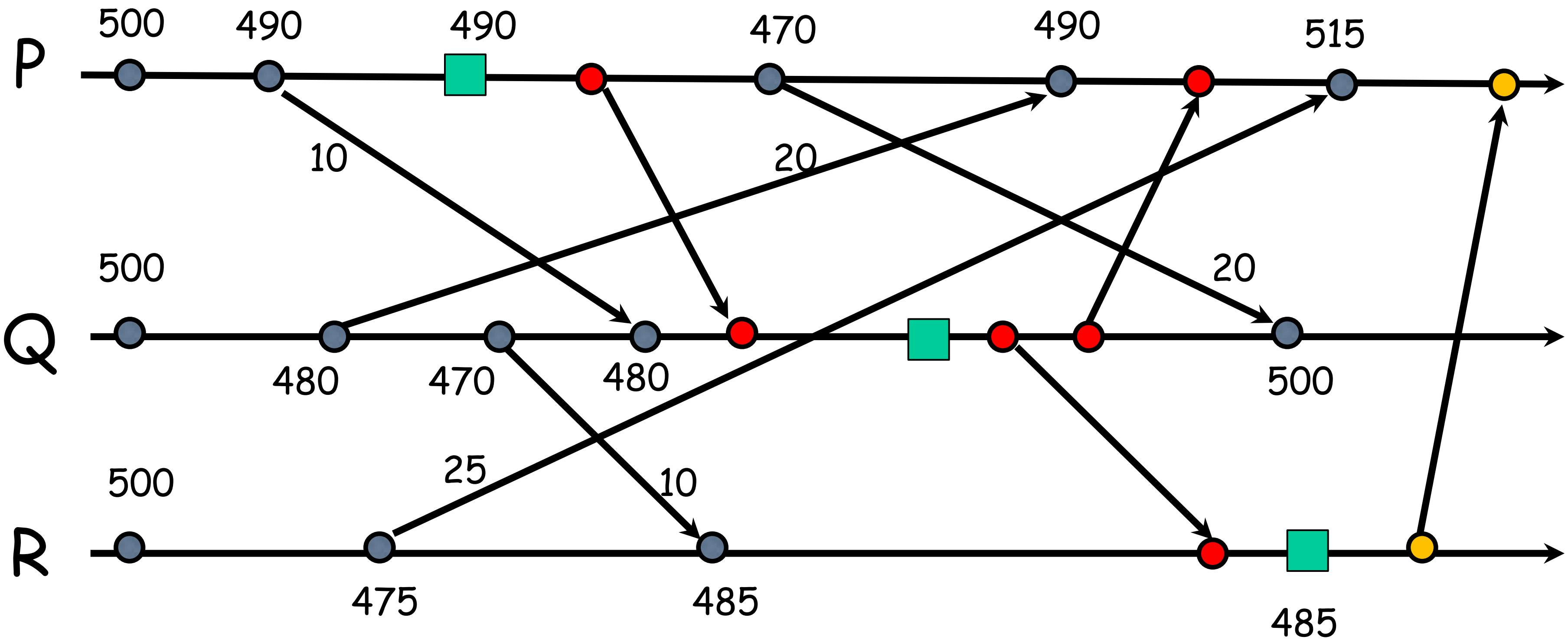
M = Marker

Node	Recorded state				
		c1	c2	c3	c4
p			{}		{}
q		{}			
r				{}	

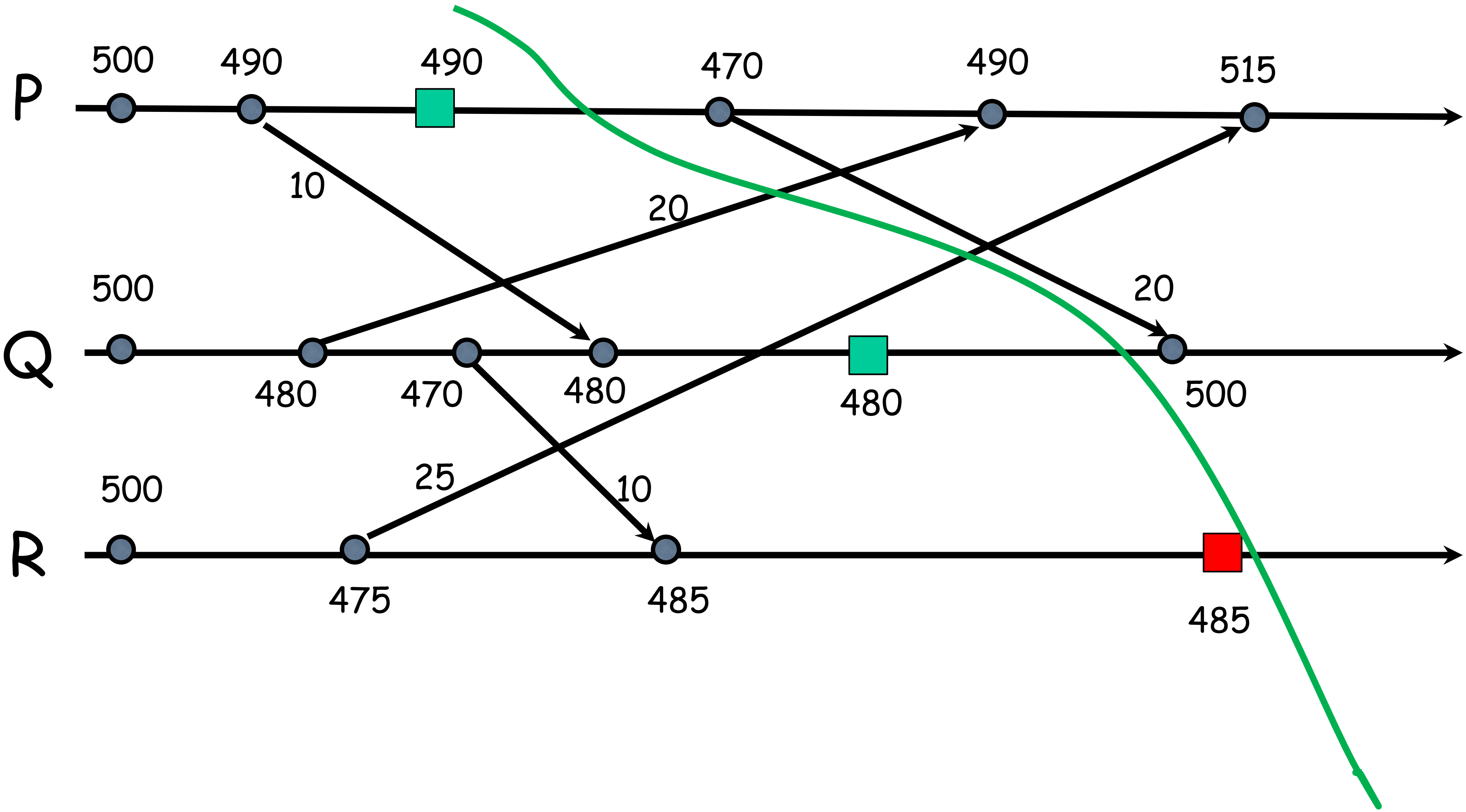
Example - step 5



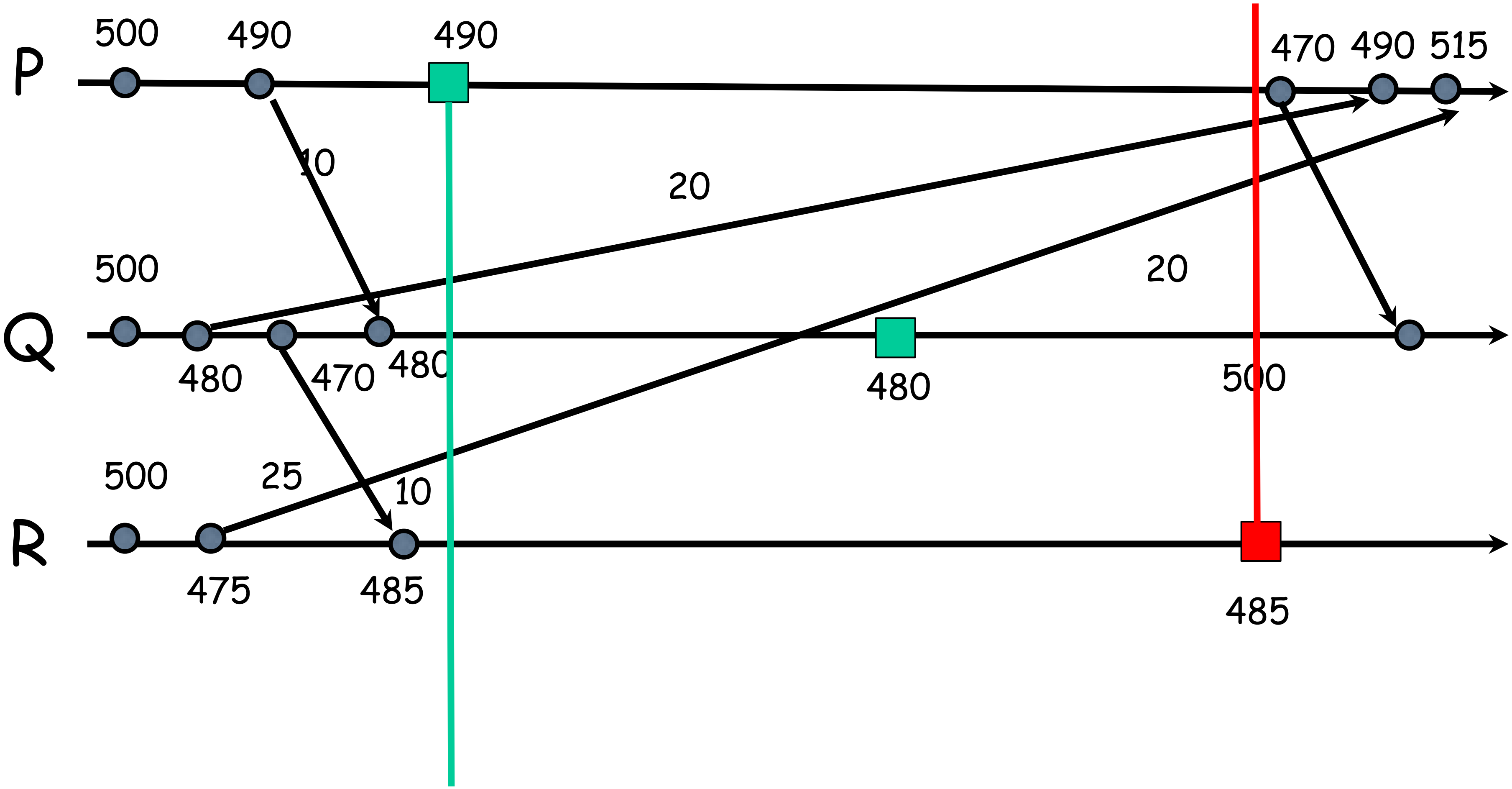
Node	Recorded state				
	state	c1	c2	c3	c4
p	490		{20}		{25}
q	480	{empty}			
r	485			{empty}	

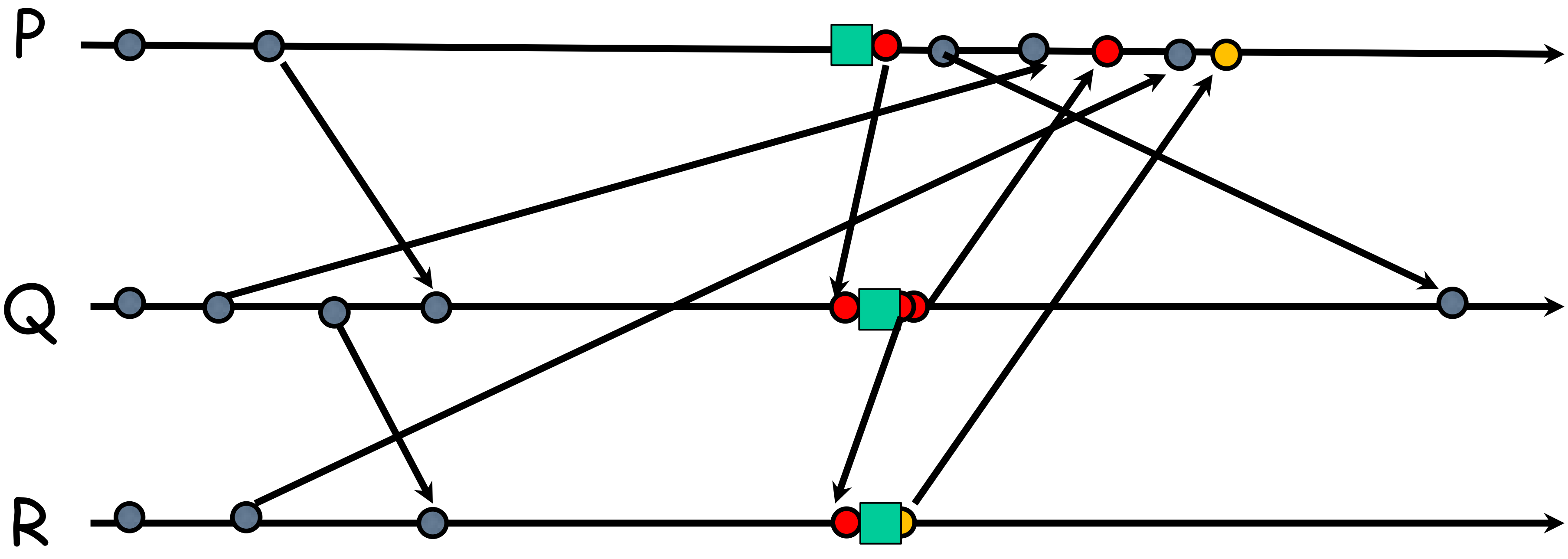


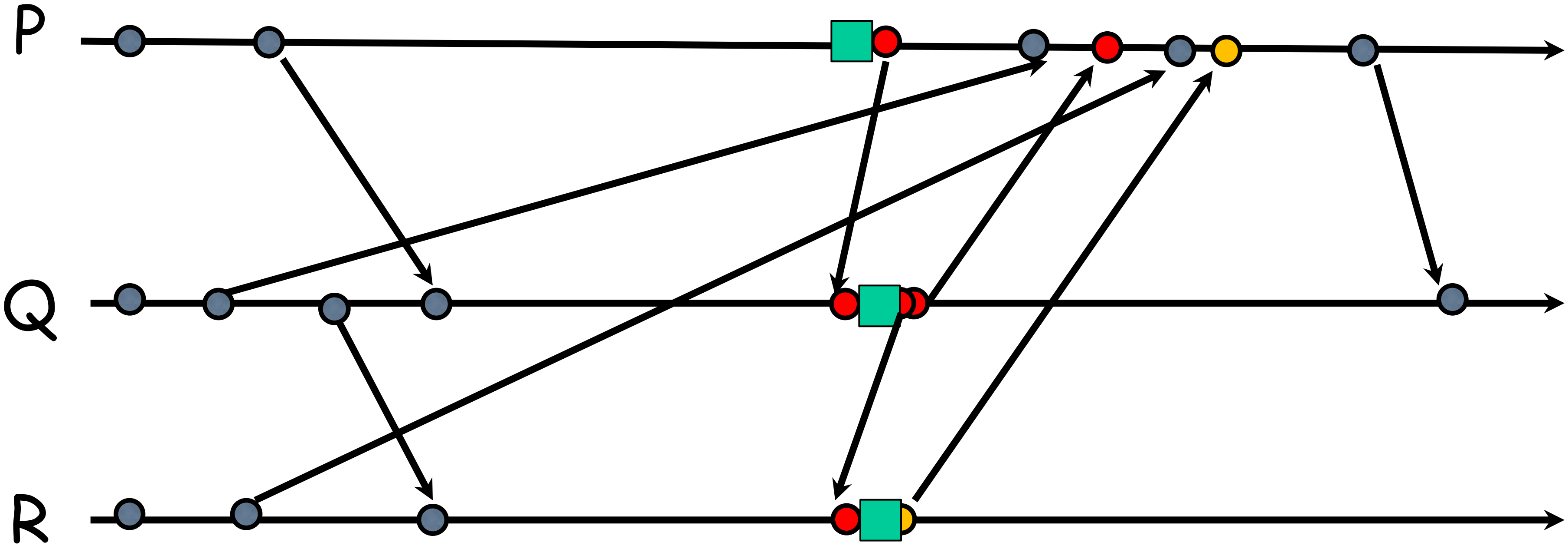
P
Q
R
500
490
490
470
490
515
10
20
500
20
480
470
480
500
500
25
10
475
485
485

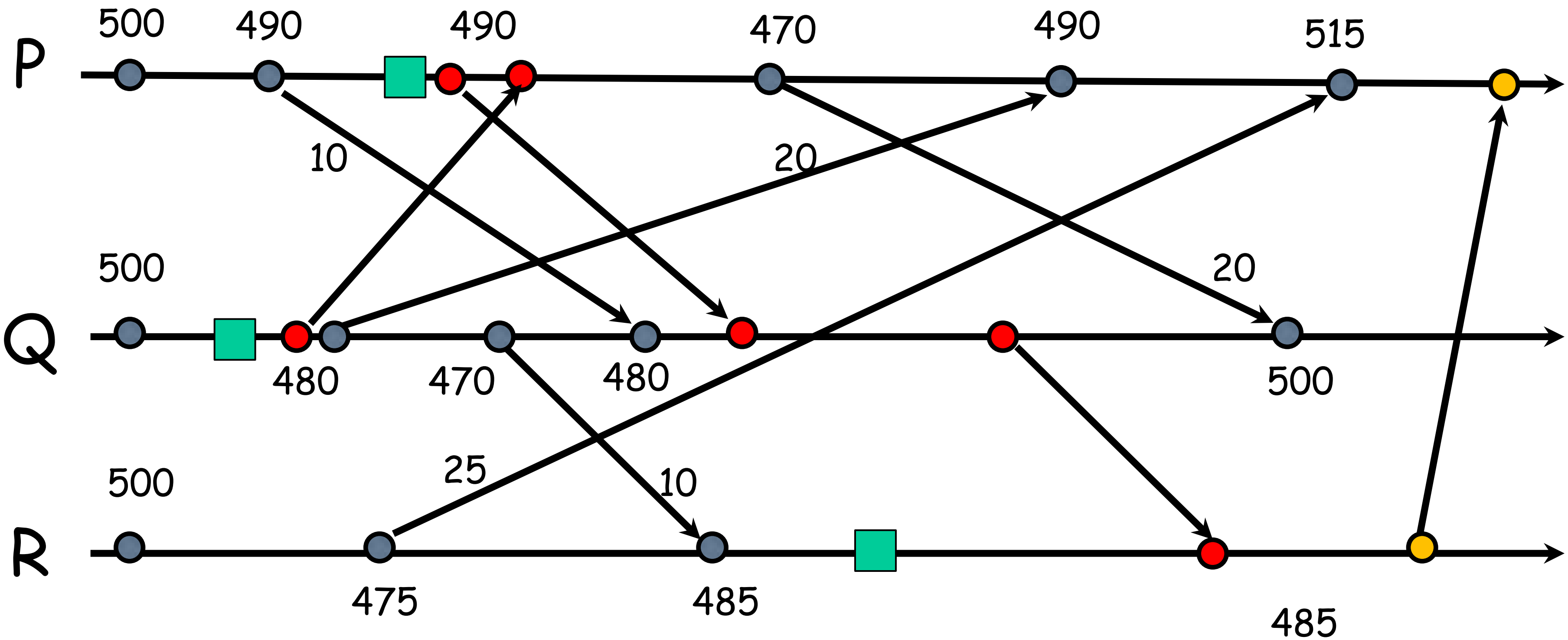


P
Q
R
500
490
490
470
490
515
10
20
500
20
480
470
480
480
500
500
25
10
475
485
485


P
Q
R
500
490
490
470
490
515
10
20
500
20
480
470
480
480
500
500
25
10
475
485
485


P
Q
R


P
Q
R


P
Q
R
500
490
490
470
490
515
10
20
500
480
470
480
20
500
25
10
500
475
485
485

What if more than one initiate?