Slides for Chapter 15: Coordination and Agreement

Figure 15.1
A network partition

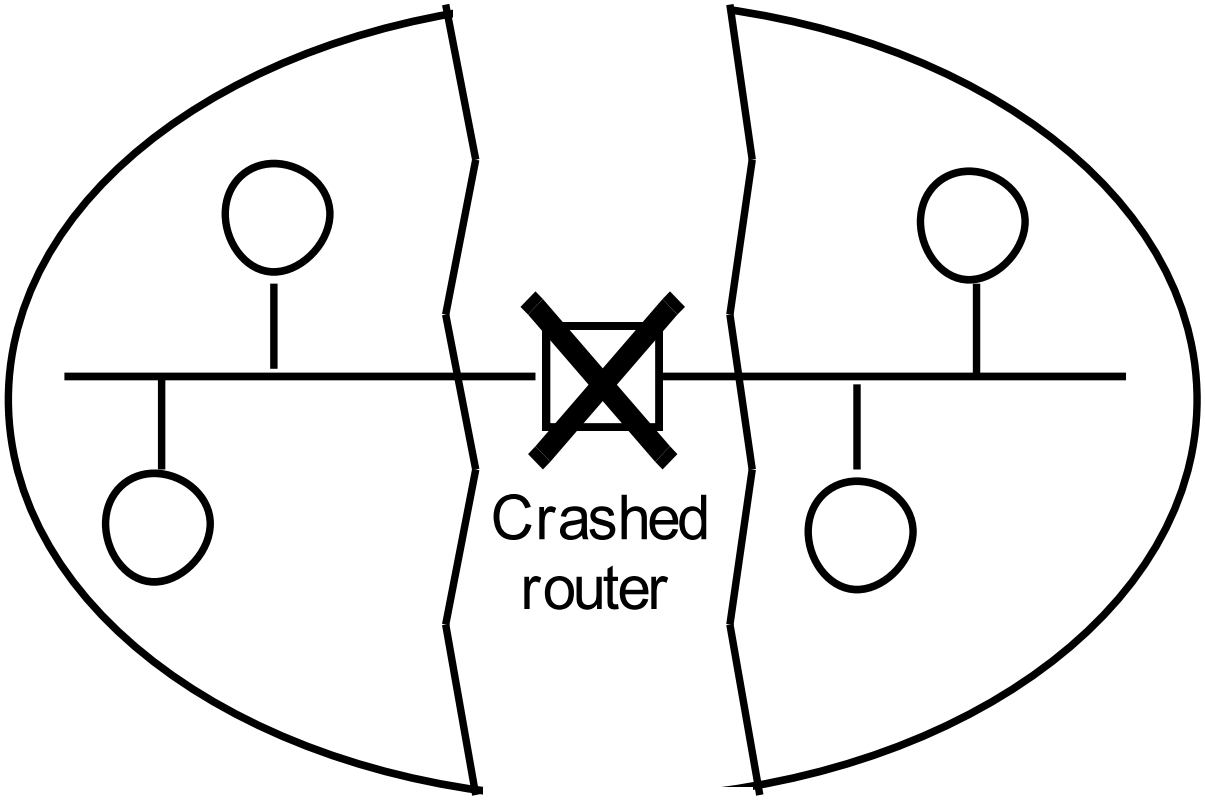


Figure 15.2
Server managing a mutual exclusion token for a set of processes

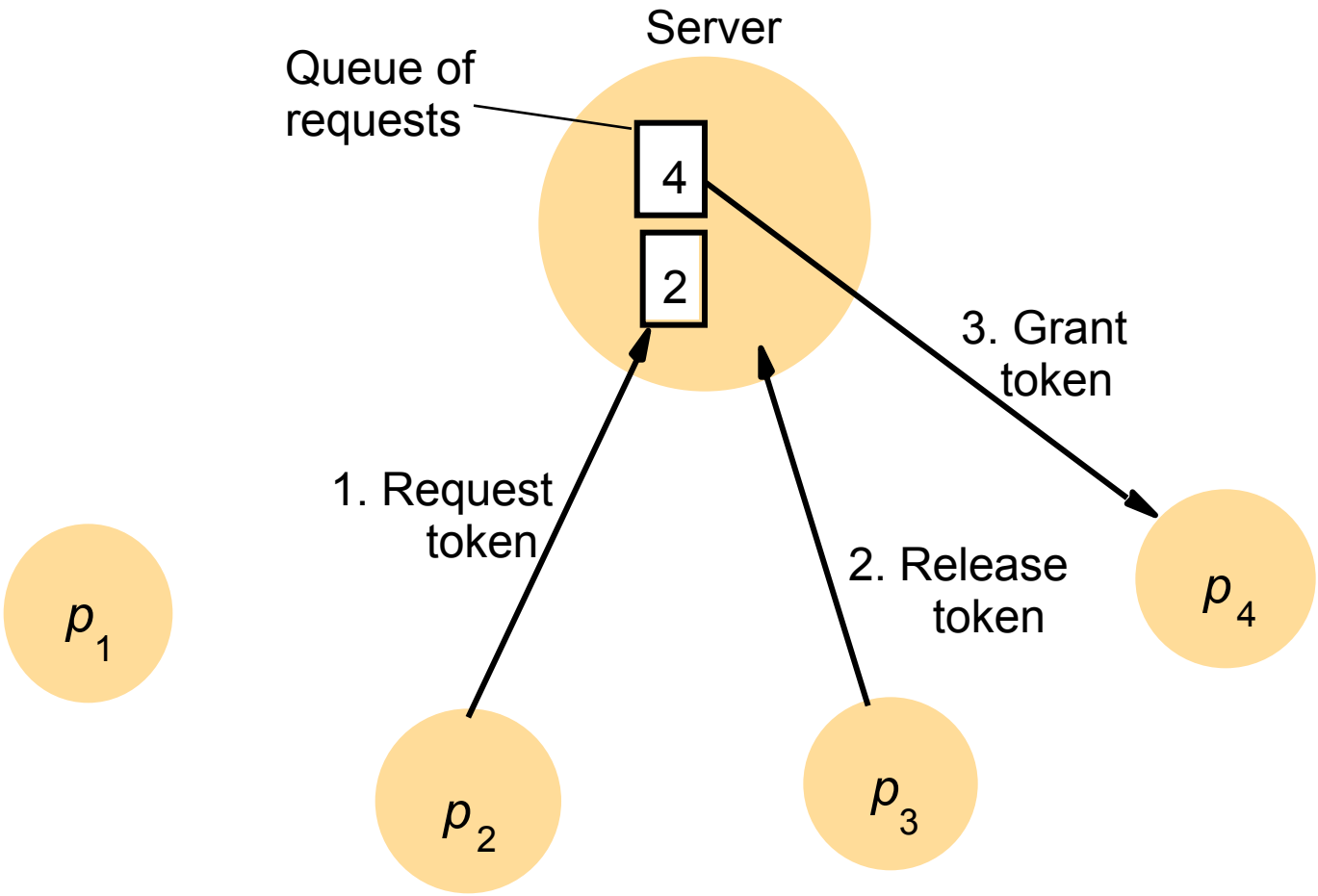


Figure 15.3
A ring of processes transferring a mutual exclusion token

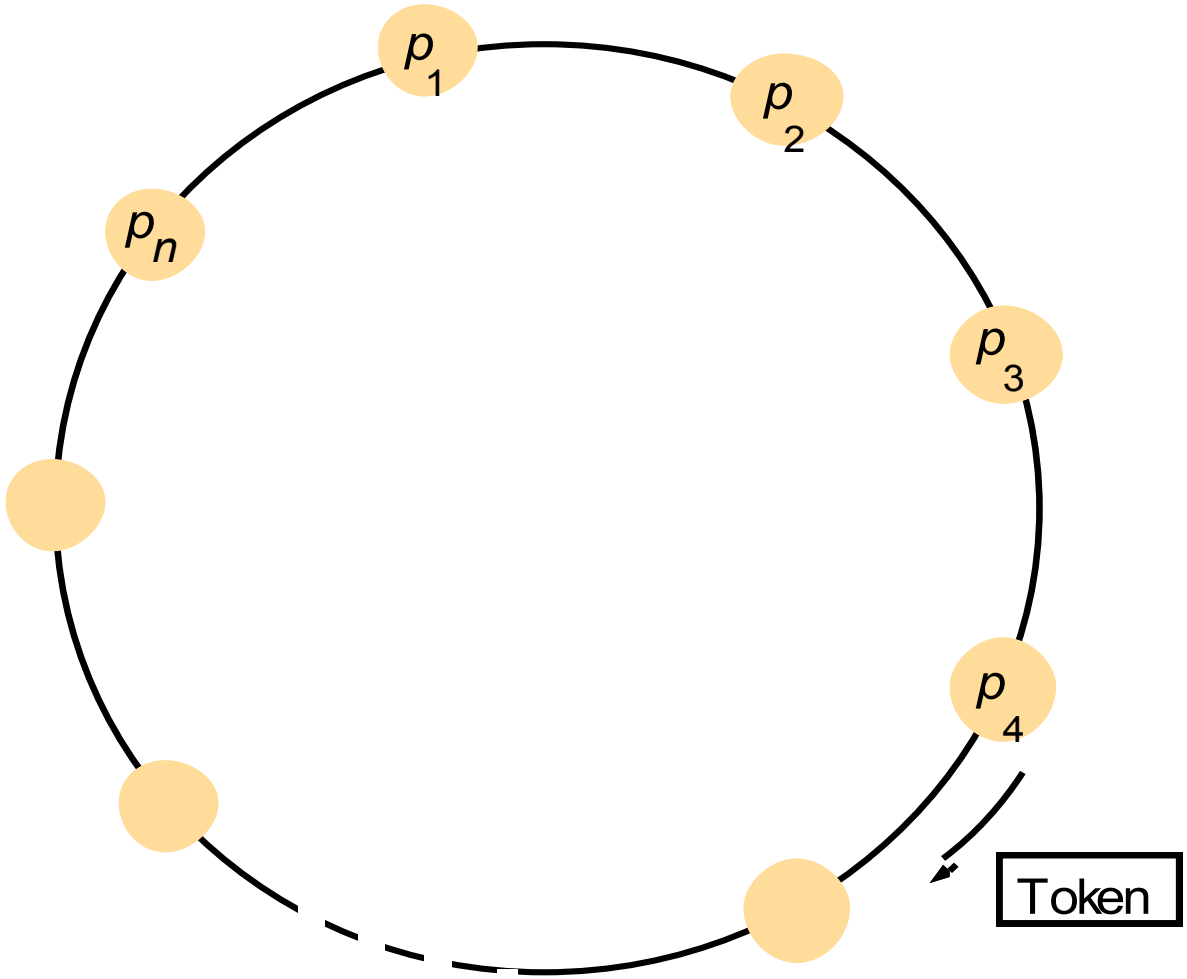


Figure 15.4
Ricart and Agrawala's algorithm

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = (*N* – 1));
    state := HELD;

request processing deferred here

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

Figure 15.5
Multicast synchronization

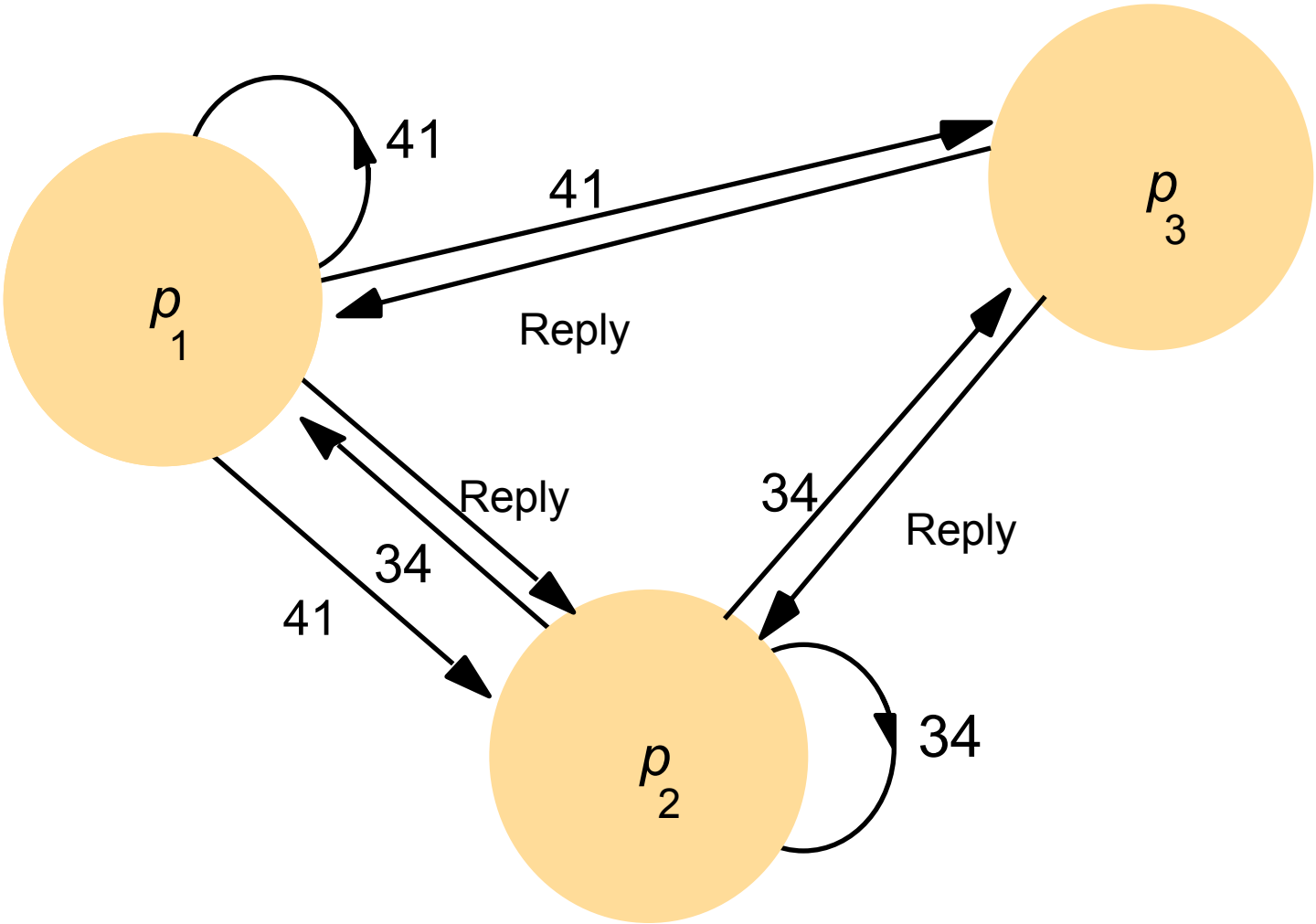
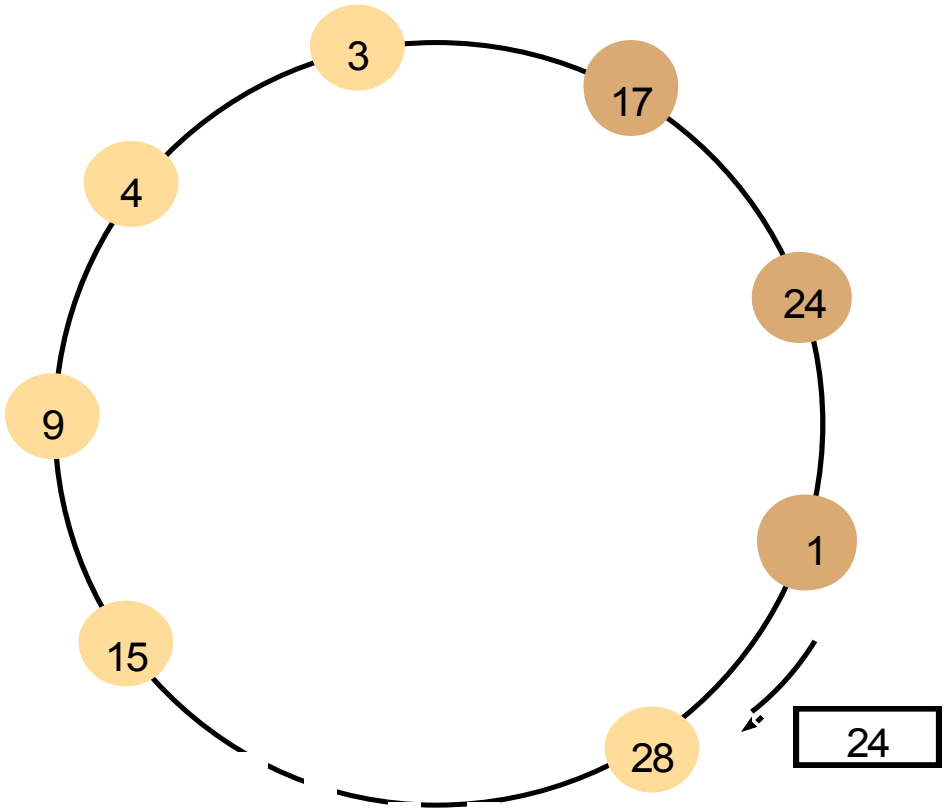


Figure 15.6
Maekawa's algorithm – part 1

On initialization
  state := RELEASED;
  voted := FALSE;
For p_i to enter the critical section
  state := WANTED;
  Multicast *request* to all processes in V_i;
  Wait until (number of replies received = K);
  state := HELD;
On receipt of a request from p_i at p_j
  if (*state* = HELD *or voted* = TRUE)
  then
    queue *request* from p_i without replying;
  else
    send *reply* to p_i;
    voted := TRUE;
  end if

For p_i to exit the critical section
  state := RELEASED;
  Multicast *release* to all processes in V_i;
On receipt of a release from p_i at p_j
  if (queue of requests is non-empty)
  then
    remove head of queue – from p_k, say;
    send *reply* to p_k;
    voted := TRUE;
  else
    voted := FALSE;
  end if

Figure 15.7
A ring-based election in progress



Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown in a darker colour

Figure 15.8
The bully algorithm

The election of coordinator p_2, after the failure of p_4 and then p_3

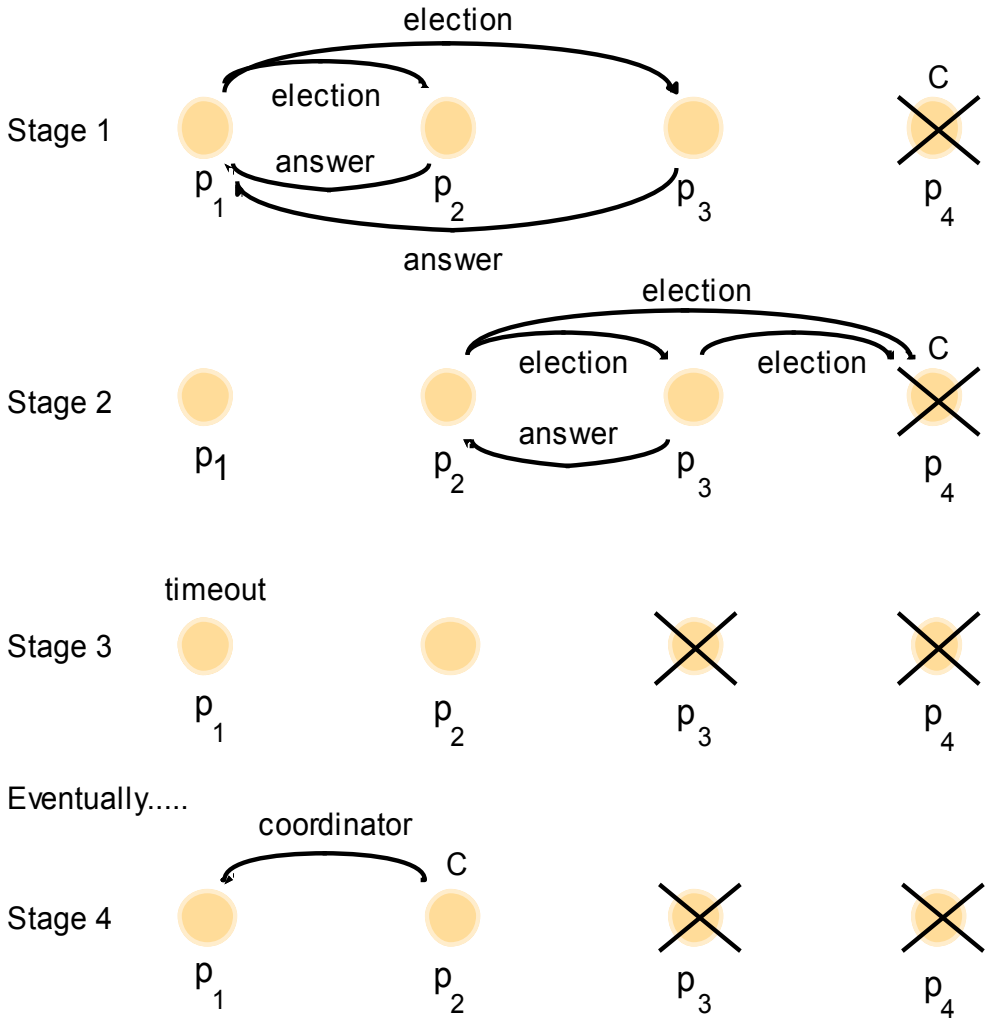


Figure 15.9
Reliable multicast algorithm

```
On initialization
    Received := {};

For process p to R-multicast message m to group g
    B-multicast(g, m);          // p ∈ g is included as a destination

On B-deliver(m) at process q with g = group(m)
    if (m ∉ Received)
    then
                Received := Received ∪ {m};
                if (q ≠ p) then B-multicast(g, m); end if
                R-deliver m;
    end if
```

Figure 15.10
The hold-back queue for arriving multicast messages

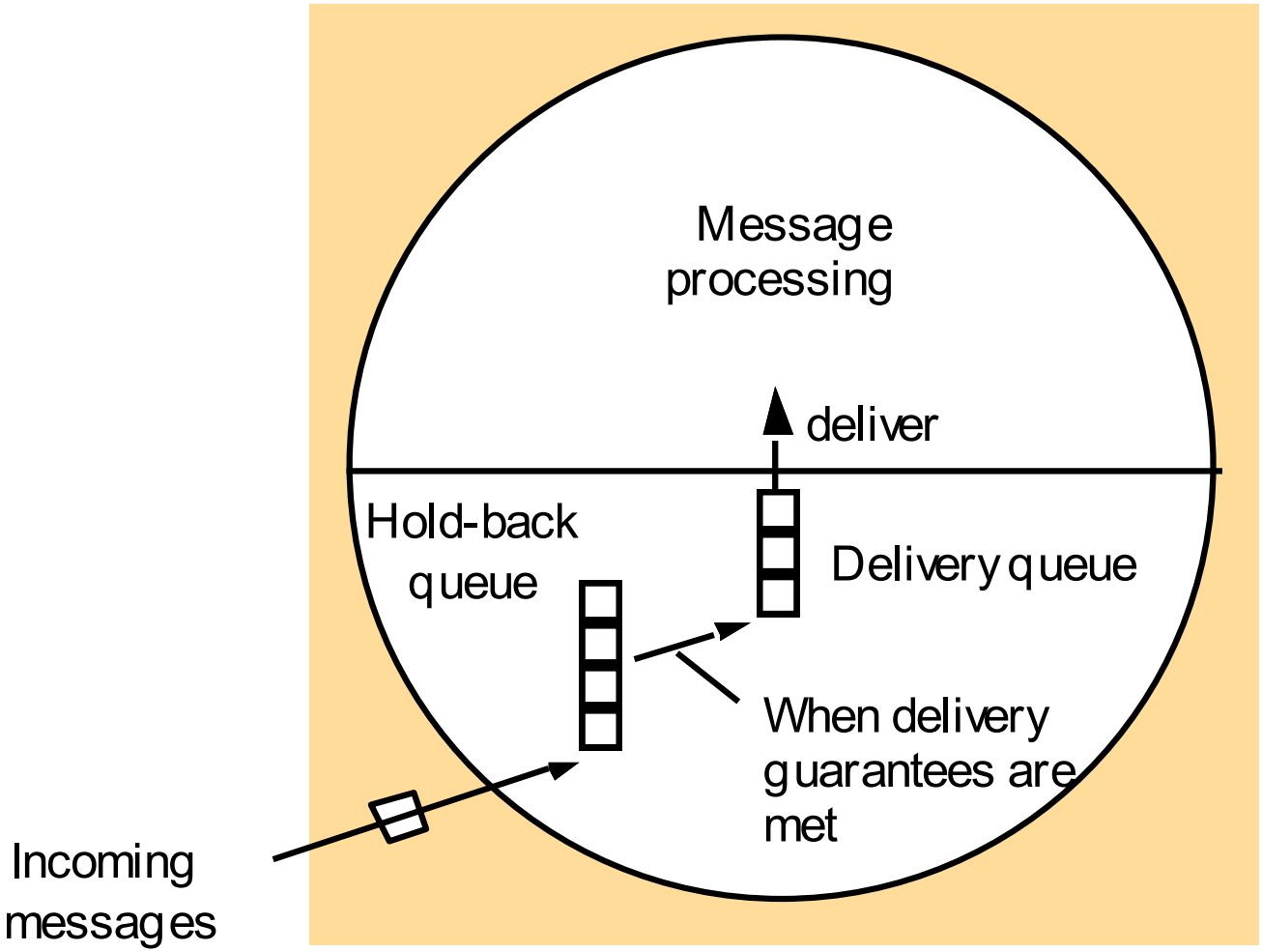


Figure 15.11
Total, FIFO and causal ordering of multicast messages

Notice the consistent ordering of totally ordered messages T_1 and T_2, the FIFO-related messages F_1 and F_2 and the causally related messages C_1 and C_3 – and the otherwise arbitrary delivery ordering of messages.

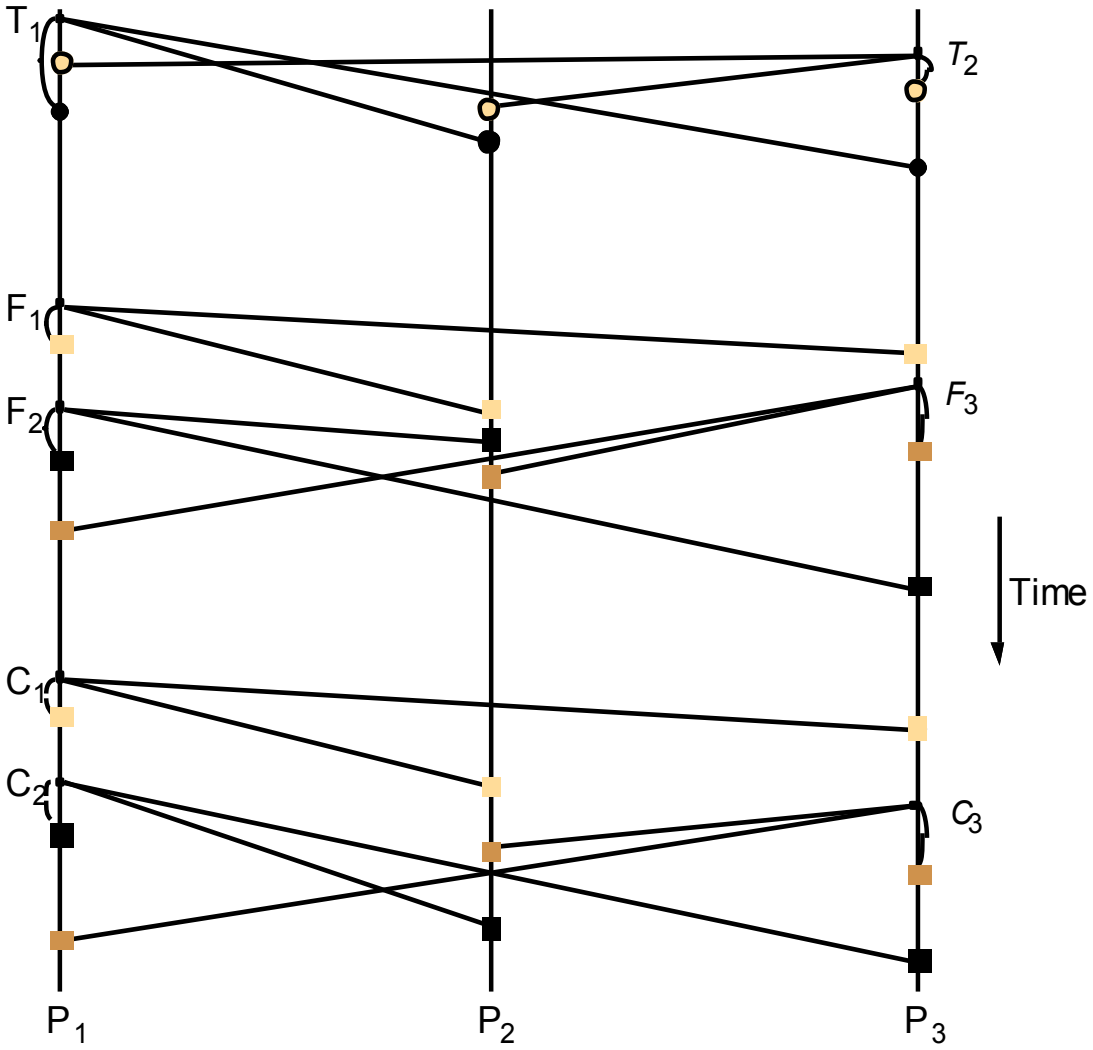


Figure 15.12
Display from bulletin board program

Bulletin board: os.interesting		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Figure 15.13
Total ordering using a sequencer

1. Algorithm for group member p

On initialization: $r_g := 0;$

To TO-multicast message m to group g
    B-multicast$(g \cup \{sequencer(g)\}, <m, i>);$

On B-deliver$(<m, i>)$ *with* $g = group(m)$
    Place $<m, i>$ in hold-back queue;

On B-deliver$(m_{order} = <$"order", $i, S>)$ *with* $g = group(m_{order})$
    wait until $<m, i>$ in hold-back queue and $S = r_g$;
    TO-deliver m; // (after deleting it from the hold-back queue)
    $r_g = S + 1;$

2. Algorithm for sequencer of g

On initialization: $s_g := 0;$

On B-deliver$(<m, i>)$ *with* $g = group(m)$
    B-multicast$(g,$ <"order", $i, s_g>);$
    $s_g := s_g + 1;$

Figure 15.14
The ISIS algorithm for total ordering

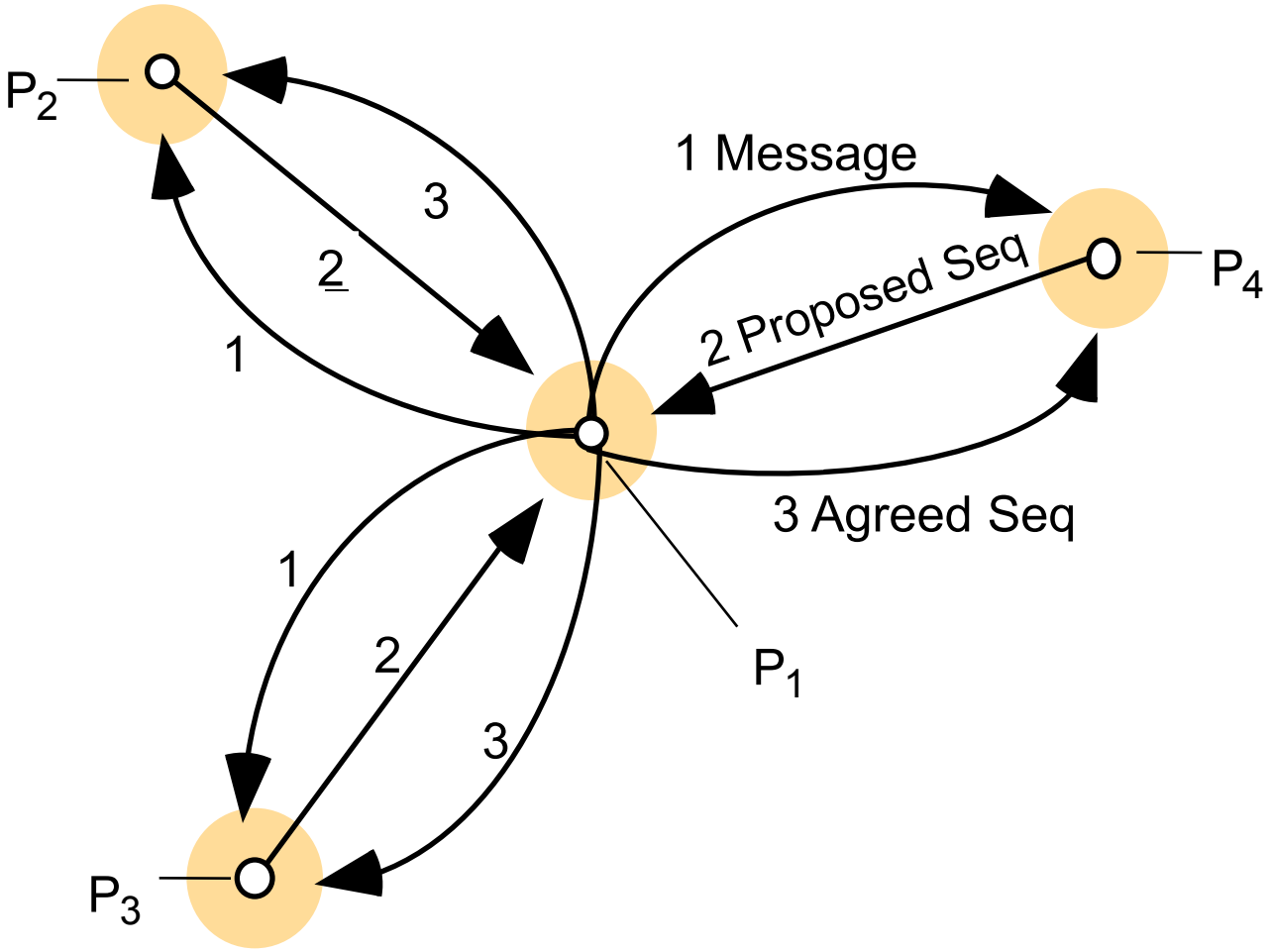


Figure 15.15
Causal ordering using vector timestamps

Algorithm for group member p_i $(i = 1, 2\ldots, N)$

On initialization

$V_i^g[j] := 0$ $(j = 1, 2\ldots, N)$;

To CO-multicast message m to group g

$V_i^g[i] := V_i^g[i] + 1$;

B-multicast$(g, <V_i^g, m>)$;

On B-deliver$(<V_j^g, m>)$ *from* p_j, *with* $g = group(m)$

place $<V_j^g, m>$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;

CO-deliver m; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1$;

Figure 15.16
Consensus for three processes

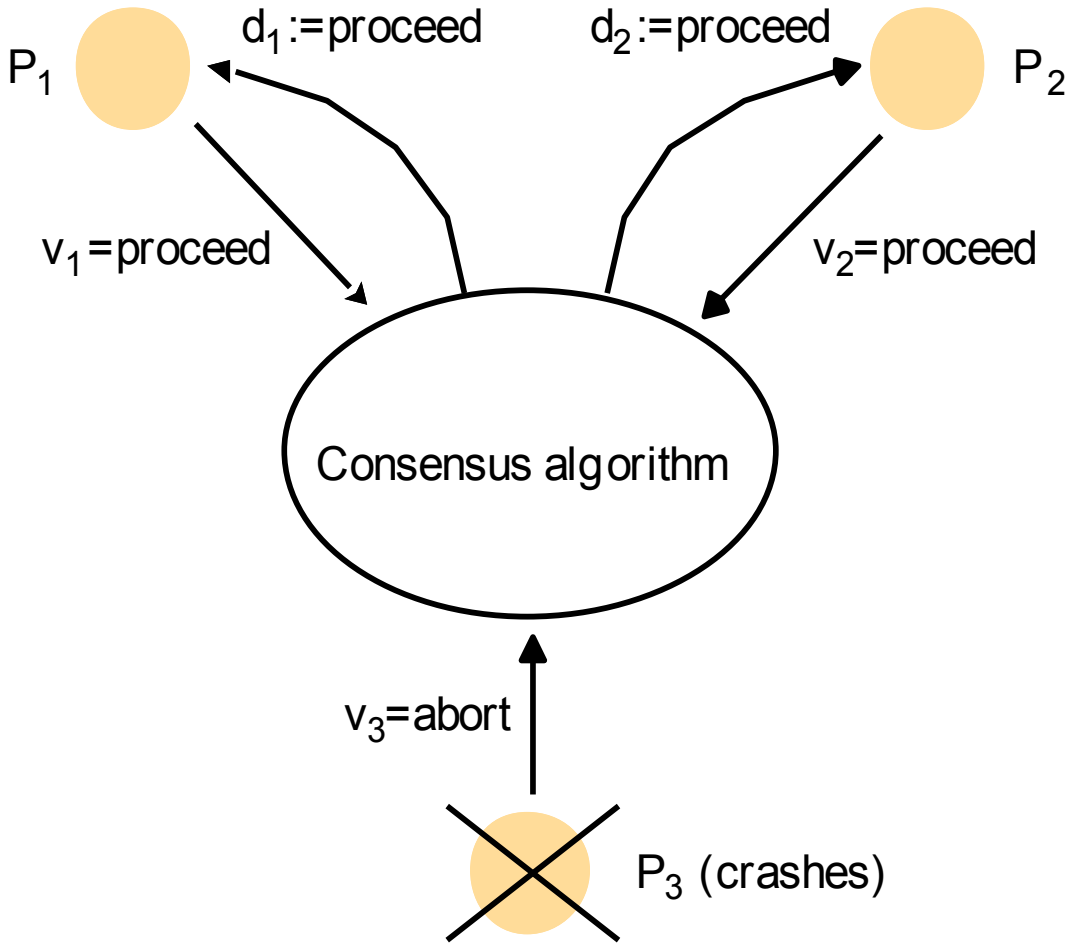


Figure 15.17
Consensus in a synchronous system

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \leq r \leq f + 1)$

B-multicast$(g, Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round r)

{

On B-deliver(V_j) *from some* p_j

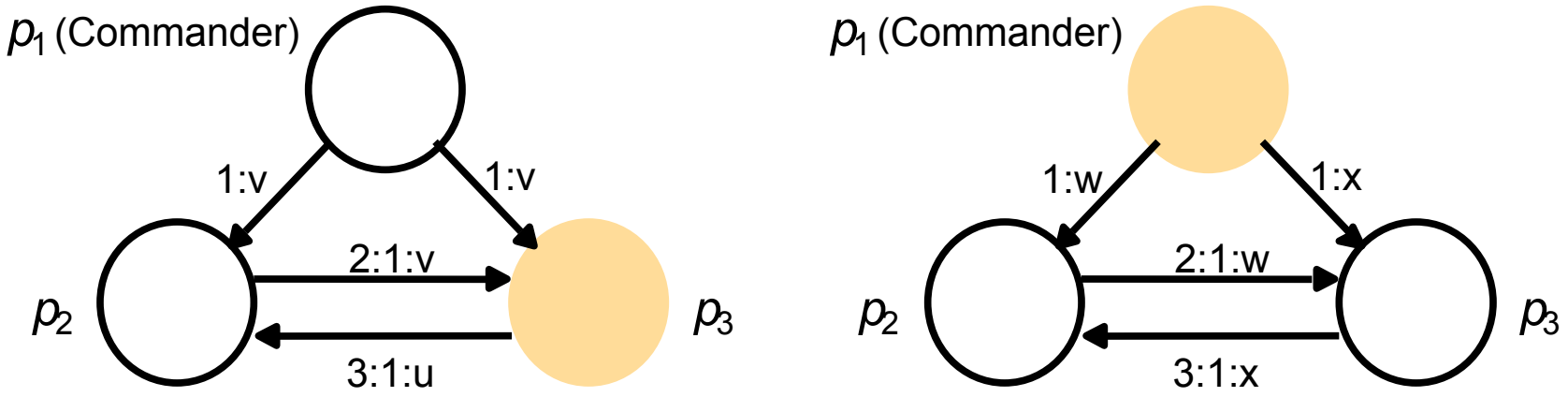
$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f + 1)$ *rounds*

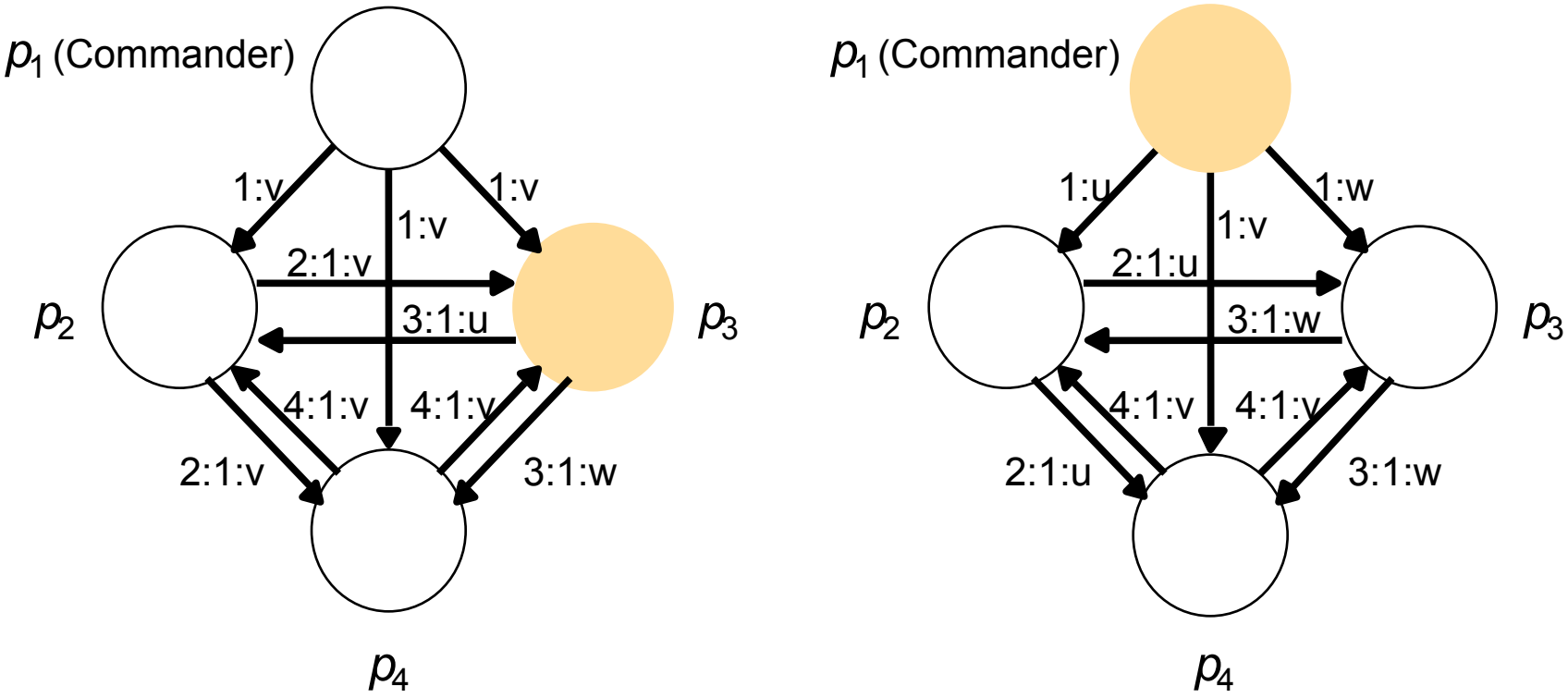
Assign $d_i = minimum(Values_i^{f+1})$;

Figure 15.18
Three Byzantine generals



Faulty processes are shown coloured

Figure 15.19
Four Byzantine generals



Faulty processes are shown coloured