

Figure 11.1 A network partition

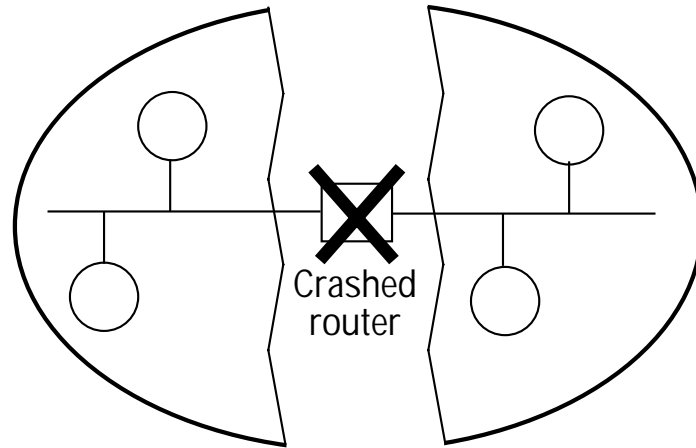


Figure 11.2 Server managing a mutual exclusion token for a set of processes

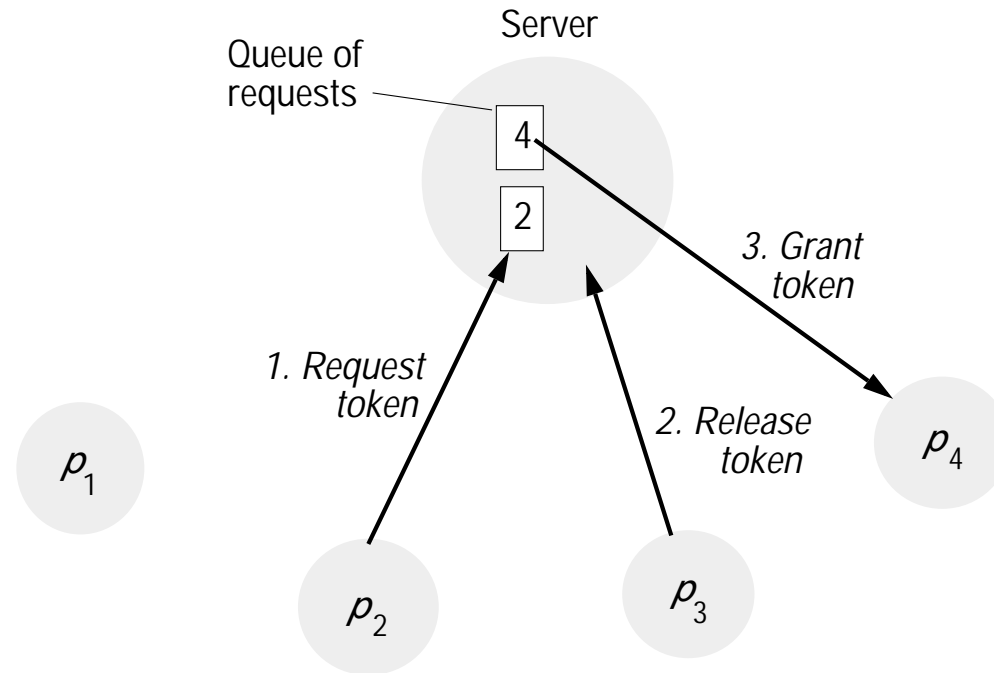


Figure 11.3 A ring of processes transferring a mutual exclusion token

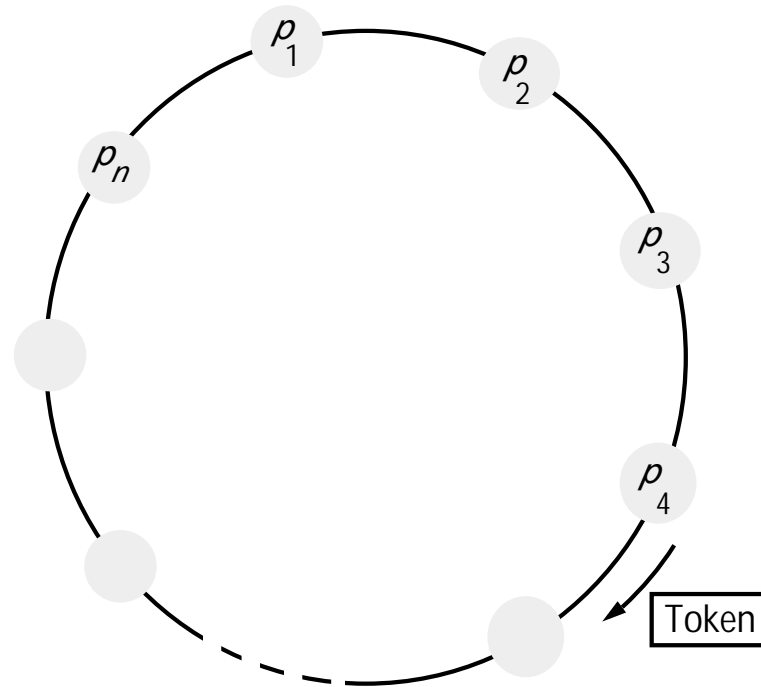


Figure 11.4 Ricart and Agrawala's algorithm

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 $\langle T_i, p_i \rangle$ *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 11.5 Multicast synchronization

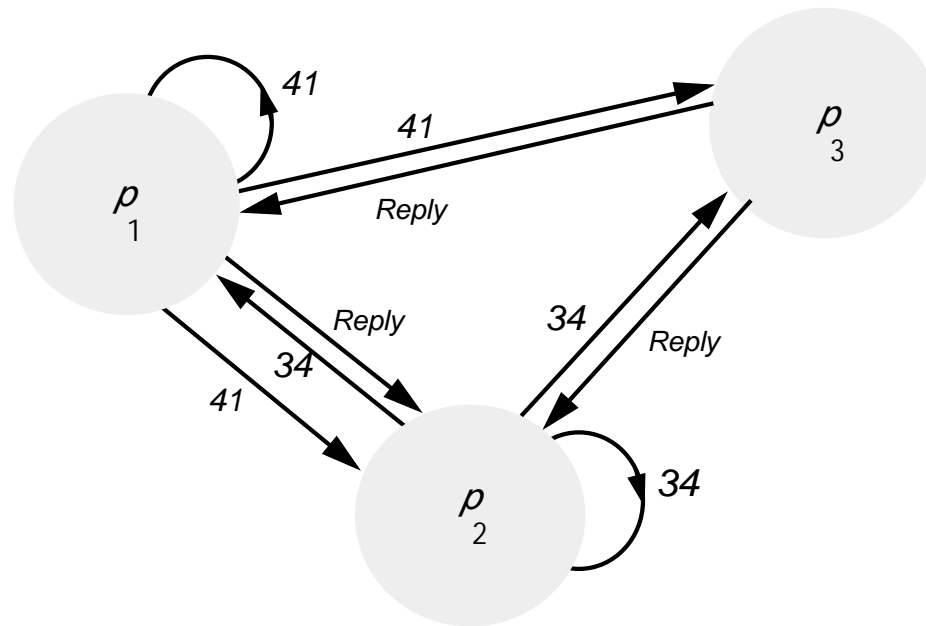


Figure 11.6 Maekawa's algorithm (cont'd on next slide)

On initialization

state := RELEASED;

voted := FALSE;

For p_i *to enter the critical section*

state := WANTED;

Multicast *request* to all processes in $V_i - \{p_i\}$;

Wait until (number of replies received = $(K - 1)$);

state := HELD;

On receipt of a request from p_i *at* p_j ($i \neq j$)

if (*state* = HELD or *voted* = TRUE)

then

 queue *request* from p_i without replying;

else

 send *reply* to p_i ;

voted := TRUE;

end if

Maekawa's algorithm (cont'd)

For p_i *to exit the critical section*

$state := \text{RELEASED};$

Multicast *release* to all processes in $V_i - \{p_i\};$

On receipt of a release from p_i *at* p_j ($i \neq j$)

if (queue of requests is non-empty)

then

remove head of queue – from p_k , say;

send *reply* to p_k ;

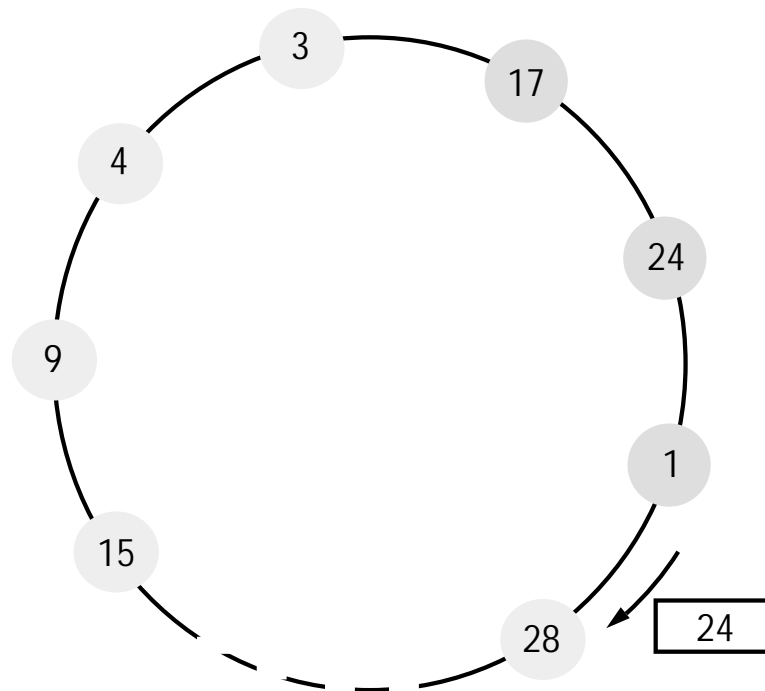
$voted := \text{TRUE};$

else

$voted := \text{FALSE};$

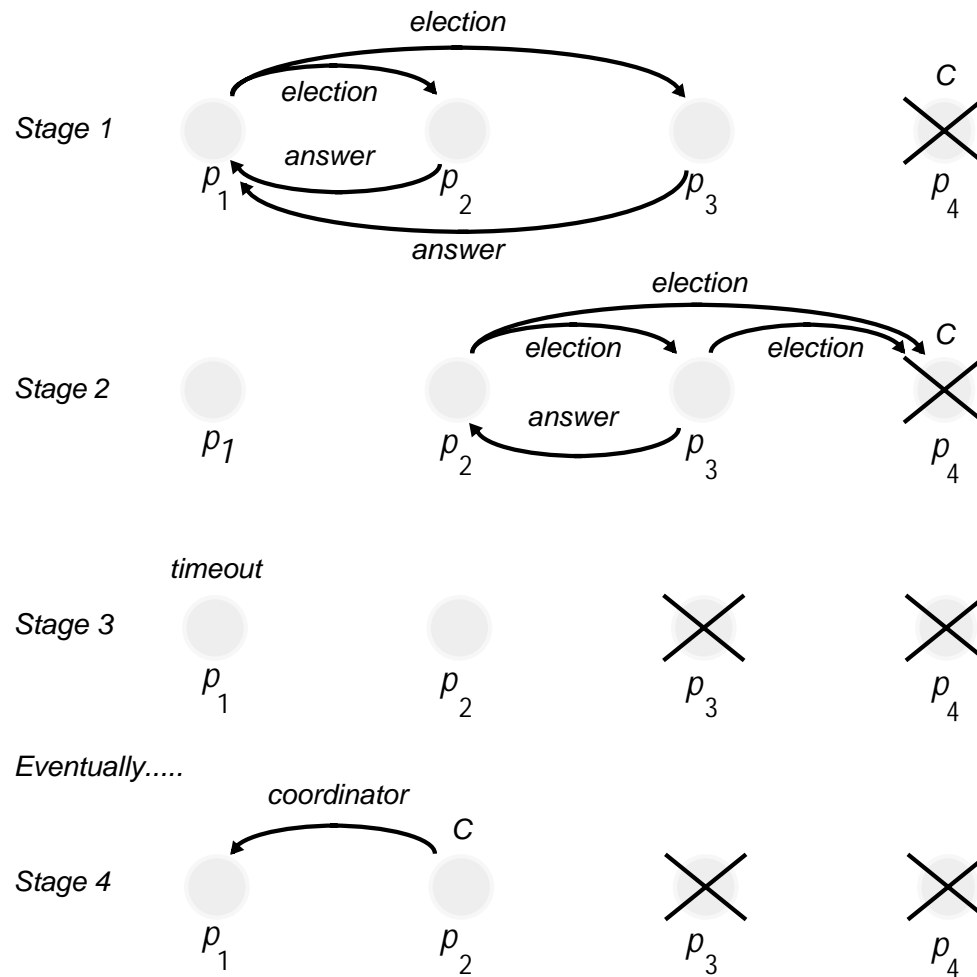
end if

Figure 11.7 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 darkened

Figure 11.8 The bully algorithm



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

Figure 11.9 Open and closed groups

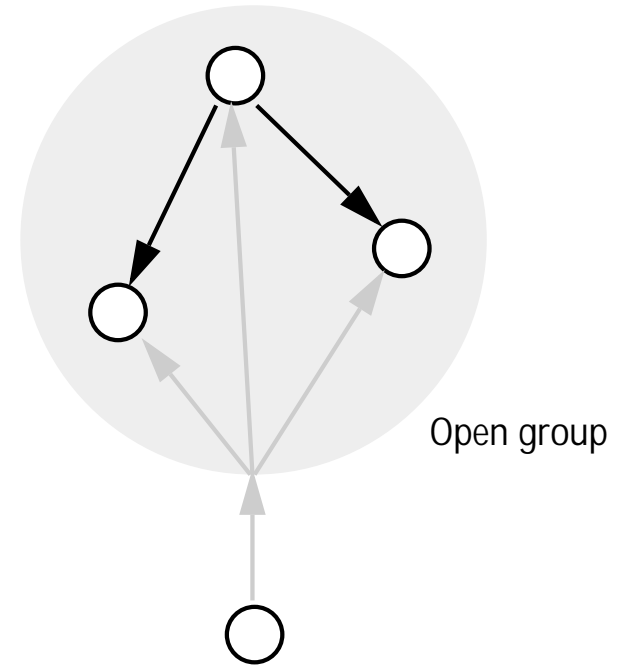
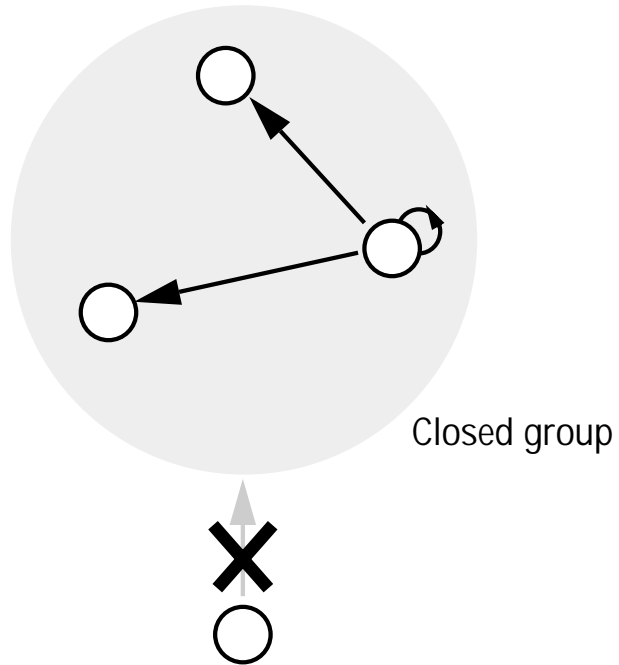


Figure 11.10 Reliable multicast algorithm

On initialization

Received := {};

For process p to R-multicast message m to group g

B-multicast(g, m); // $p \in g$ is included as a destination

On B-deliver(m) at process q with g = group(m)

if (m \notin Received)

then

Received := Received \cup {m};

if (q \neq p) then B-multicast(g, m); end if

R-deliver m;

end if

Figure 11.11 The hold-back queue for arriving multicast messages

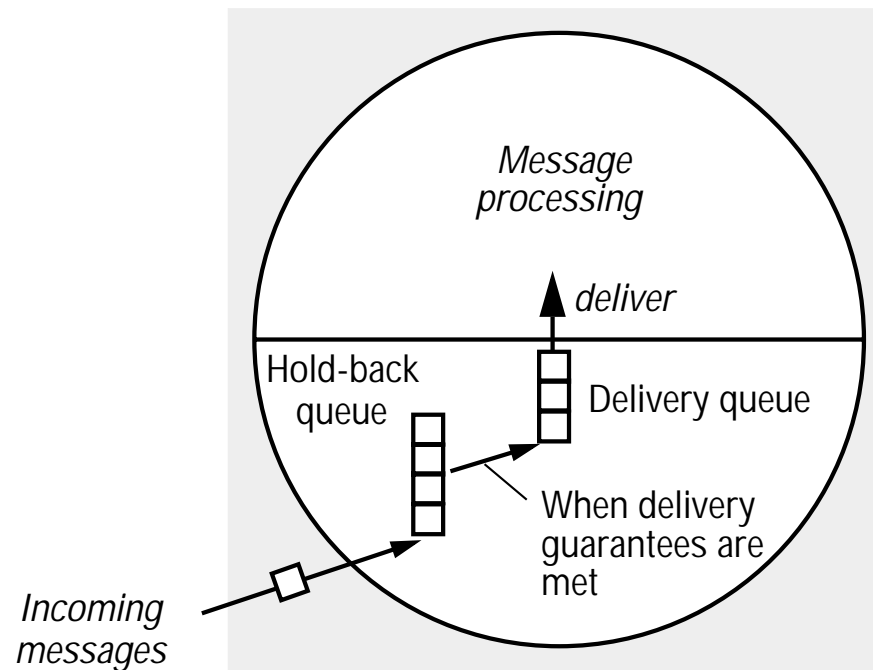
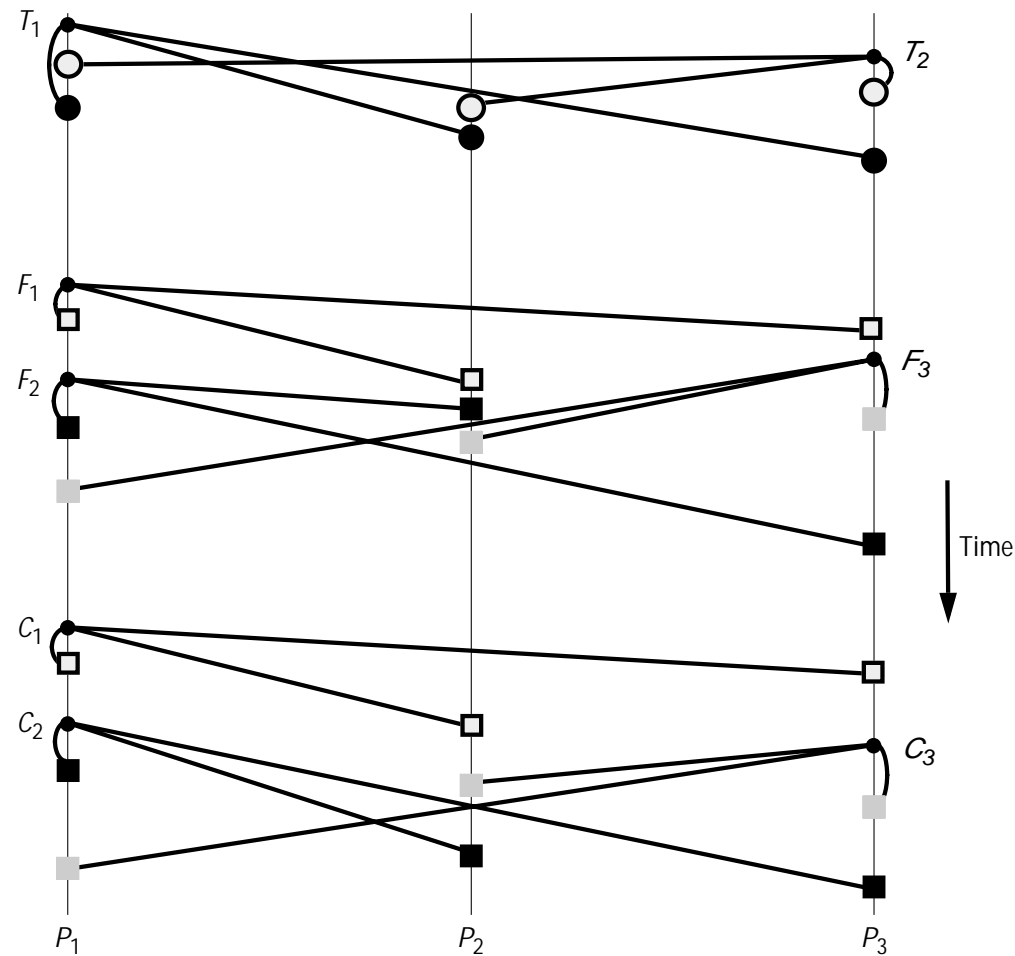


Figure 11.12 Total, FIFO and causal ordering of multicast messages



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 11.13 Display from bulletin board program

| Bulletin board: <i>os.interesting</i> | | |
|---------------------------------------|-------------|------------------|
| 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 11.14 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($g \cup \{\text{sequencer}(g)\}$, $\langle m, i \rangle$);

On B-deliver($\langle m, i \rangle$) with $g = \text{group}(m)$

Place $\langle m, i \rangle$ in hold-back queue;

On B-deliver($m_{\text{order}} = \langle \text{"order"}, i, S \rangle$) with $g = \text{group}(m_{\text{order}})$

wait until $\langle m, i \rangle$ 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($\langle m, i \rangle$) with $g = \text{group}(m)$

B-multicast(g , $\langle \text{"order"}, i, s_g \rangle$);

$s_g := s_g + 1$;

Figure 11.15 The ISIS algorithm for total ordering

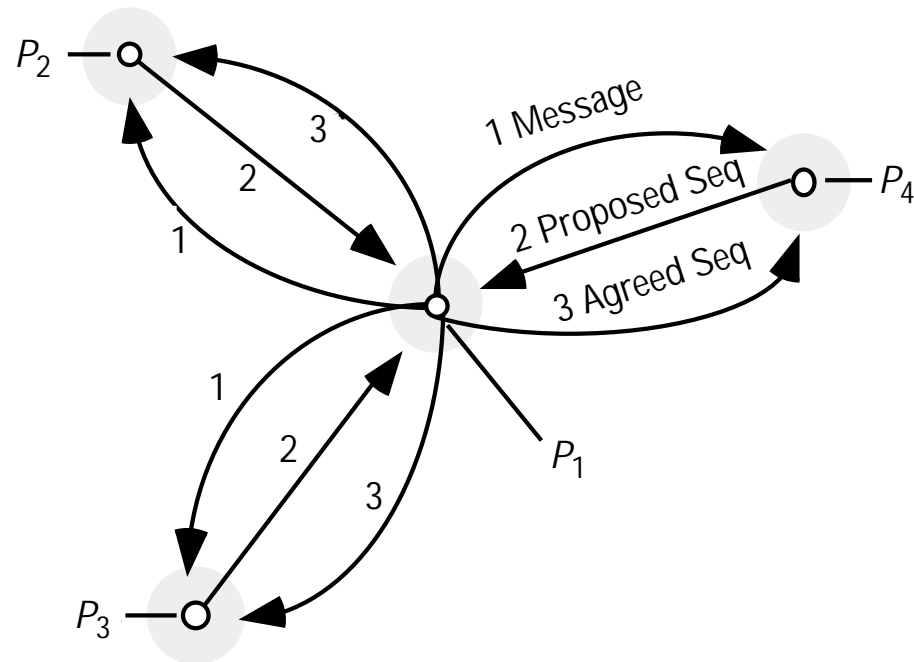


Figure 11.16 Causal ordering using vector timestamps

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

On initialization

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

To CO-multicast message m to group g

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

B-multicast($g, \langle V_i^g, m \rangle$);

On B-deliver($\langle V_j^g, m \rangle$) from p_j , with $g = \text{group}(m)$

place $\langle V_j^g, m \rangle$ 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 11.17 Consensus for three processes

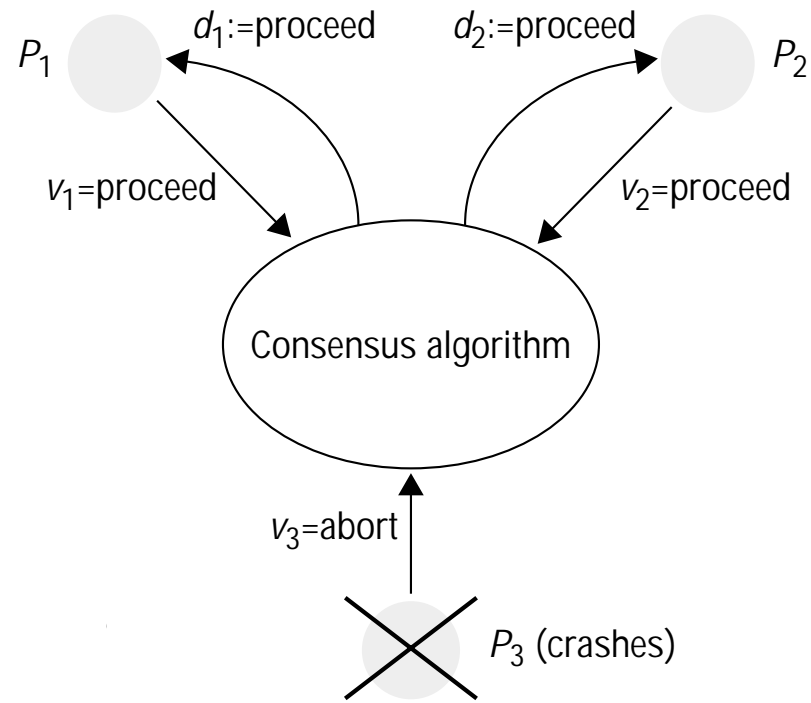


Figure 11.18 Consensus in a synchronous system

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

$B\text{-multicast}(g, Values_i^r - Values_i^{r-1});$ // 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

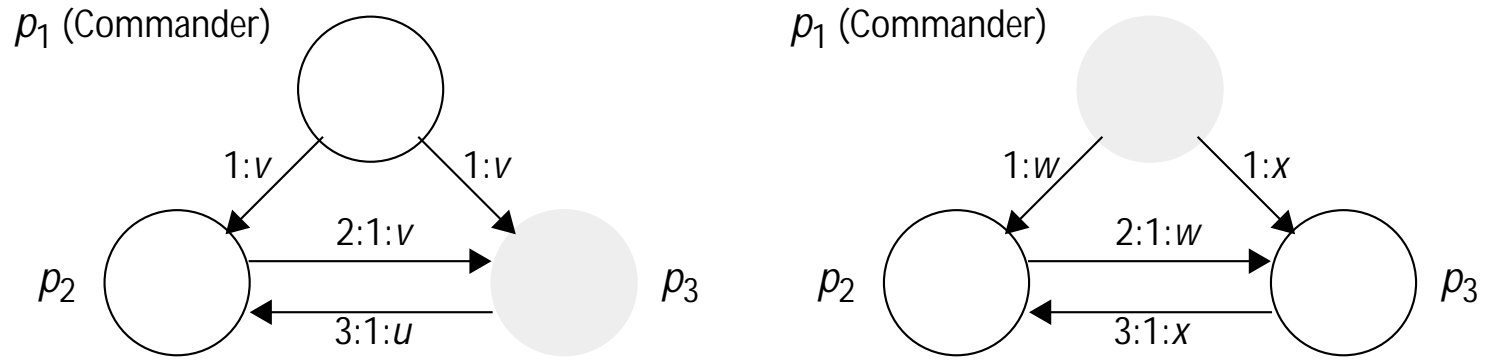
$Values_i^{r+1} := Values_i^{r+1} \cup V_j;$

}

After ($f + 1$) *rounds*

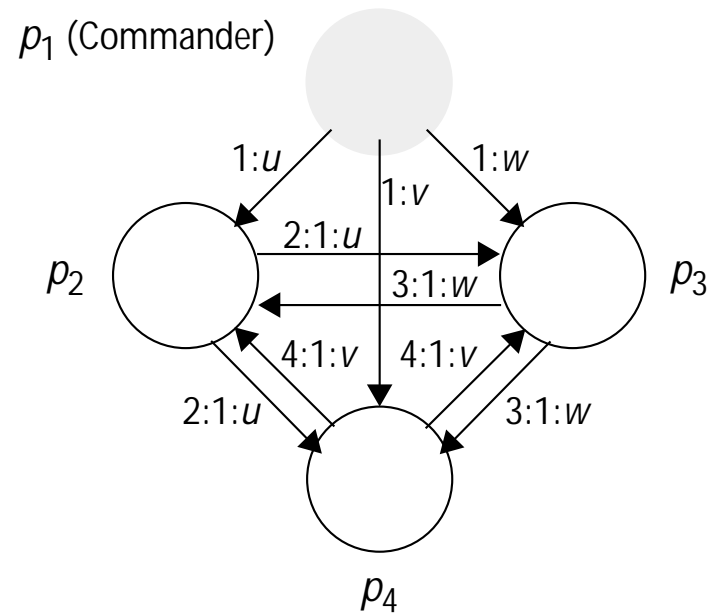
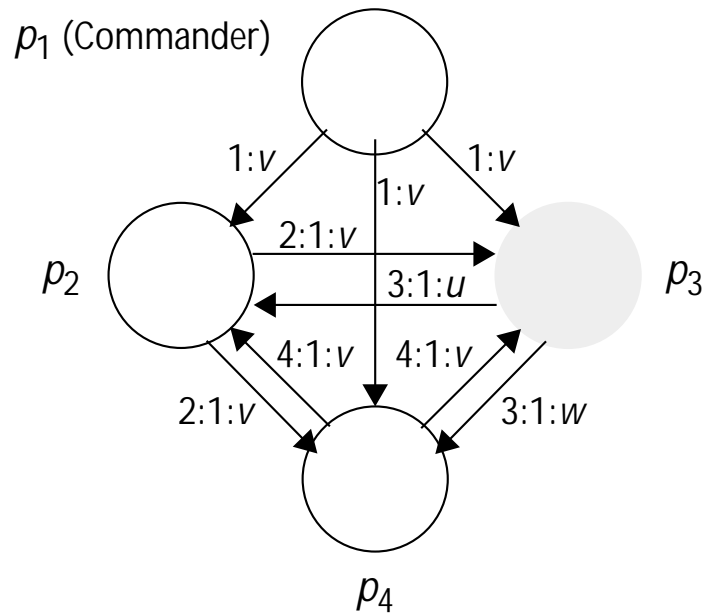
Assign $d_i = \text{minimum}(Values_i^{f+1});$

Figure 11.19 Three byzantine generals



Faulty processes are shown shaded

Figure 11.20 Four byzantine generals



Faulty processes are shown shaded