Distributed Systems

5. Fault Tolerant Systems

Fault tolerance

- A system or a component fails due to a *fault*
- *Fault tolerance* means that the system continues to provide its services in presence of faults
- A distributed system may experience and should recover also from *partial failures*
- Fault categories in time
  - Transient
    - Occurs once and disappear
  - Intermittent
    - Occurs many times in an irregular way
  - Permanent

Different Types of Failures

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition f.*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary (Byzantine) failure	A server may produce arbitrary responses at arbitrary times

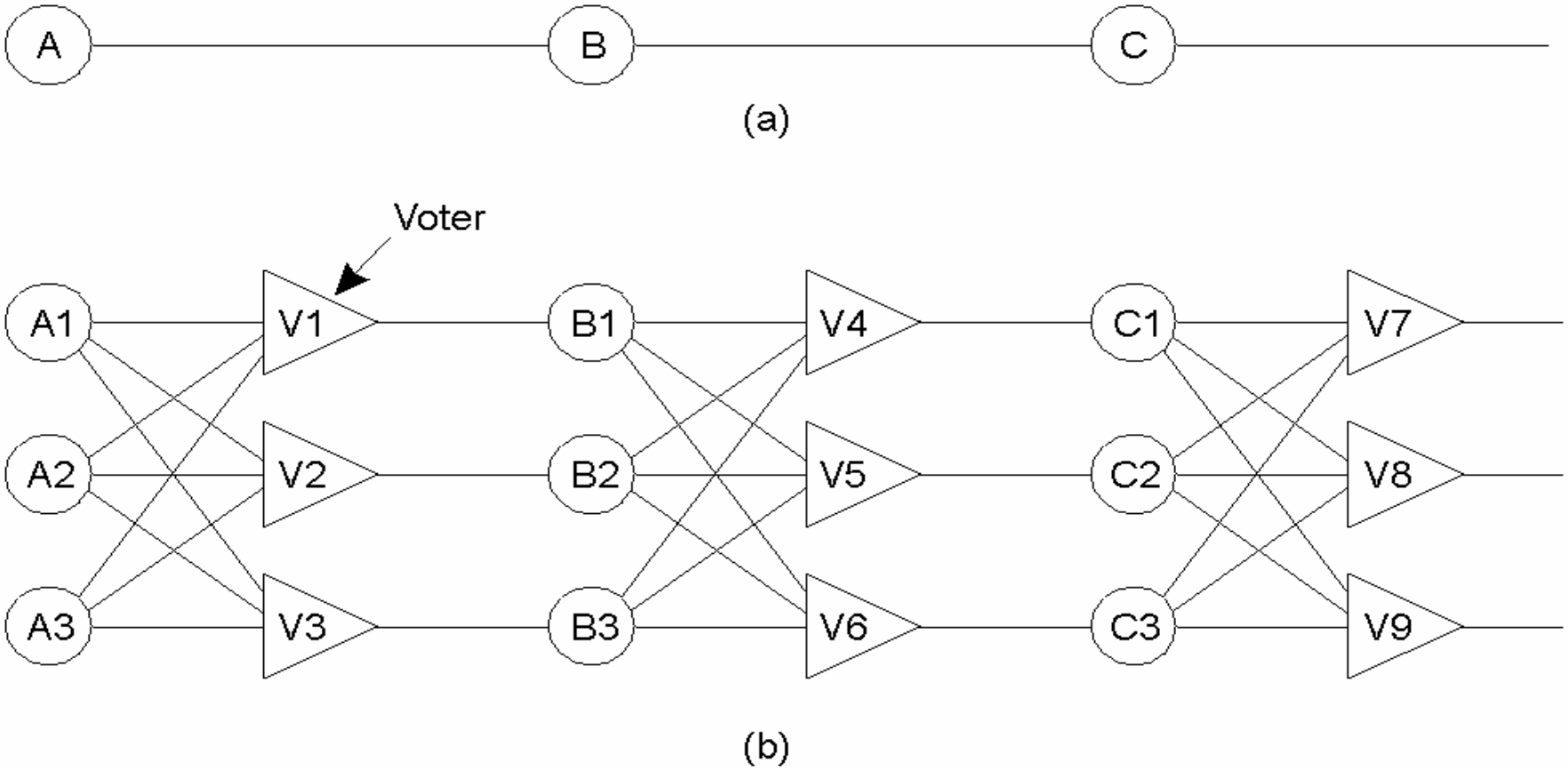
Dependable Systems

- Availability
  - The system is usable immediately at any time
- Reliability
  - A system works over a long period without error
    - A system crashing for a millisecond every hour has good availability but very poor reliability
- Safety
  - Temporal failures have no catastrophic consequences
- Maintainability
  - Failures can be repaired quickly and easily
- Security
  - System can resist attacks against its integrity

Failure Masking by Redundancy

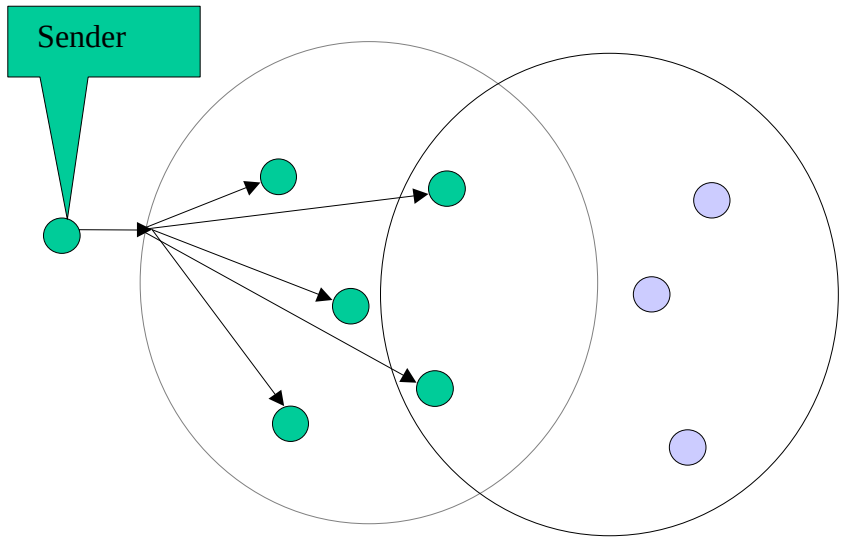
- Information redundancy
  - Extra bits are added (e.g. CRC)
- Time redundancy
  - Actions may be redone (e.g. transactions after abort)
- Physical redundancy
  - Hardware and software components may be multiplied (e.g. extra disk, extra engine in an airplane)
  - Triple modular redundancy (TMR)
    - Uses the principle of building a *majority* opinion
    - Each device is replicated 3 times, signals pass all 3 devices
    - If one device fails, a *voter* can reproduce the correct value based on 2 correct signals
    - At every stage 1 device and 1 voter may fail

Triple modular redundancy



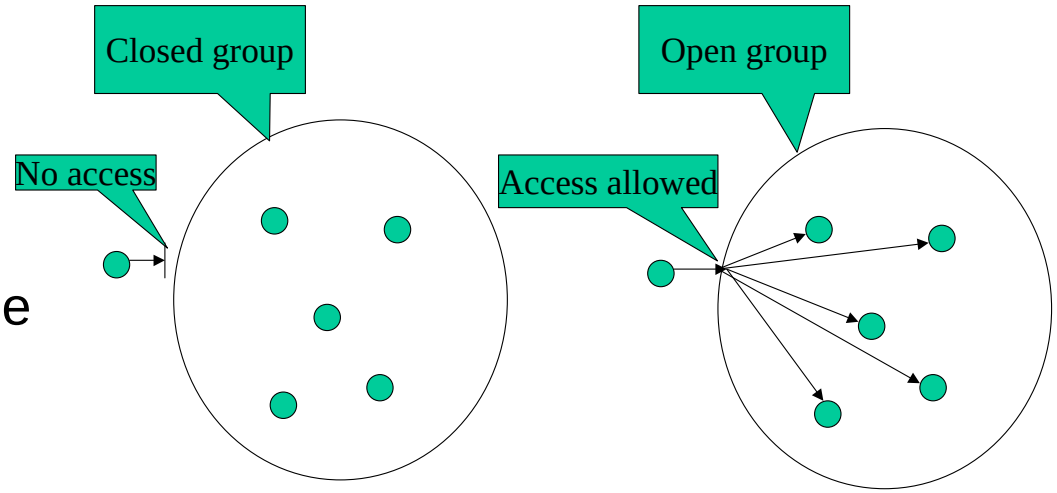
Group Communication

- A group of processes forms a logical unit
  - This creates redundancy, the basis for fault-tolerance
- One-to-many communication
  - As opposed to one-to-one communication
- Groups are dynamic
  - New groups can be created and destroyed
  - Processes can join and leave groups
  - Membership management is necessary
  - The same process maybe member of many groups
  - Groups may be overlapped



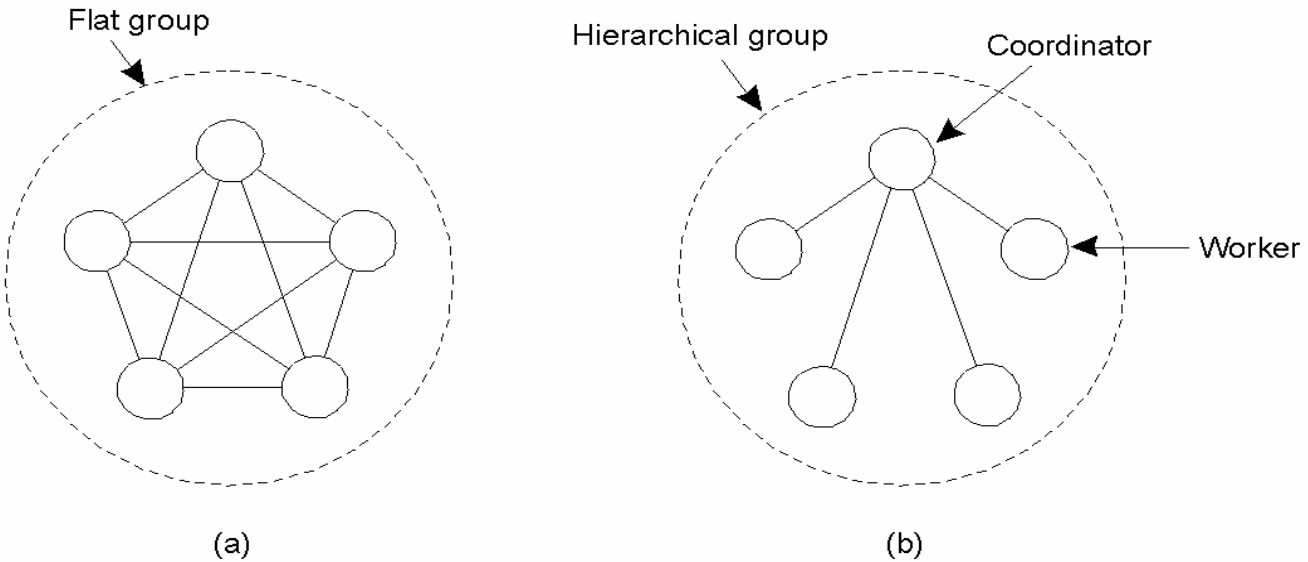
Open and closed groups

- Closed Groups
  - A process must first join the group, otherwise cannot access the members of the group
  - Main use in parallel proces
- Open Groups
  - Non-members can also access group-members
  - E.g. in a replicated server the server instances are the members and clients can send messages to the entire group



Flat and hierarchical groups

- Peer (or flat) groups
  - All processes are equal, fully symmetric, no single point of failure
  - Decisions are complicated → voting algorithms
- Hierarchical groups (one “master”)
  - Simple decisions can be made by the coordinator
  - Loss of the coordinator brings the entire group halt → needs election



Group Membership

- Controls joining and leaving of groups
- Entering and leaving must be atomic
  - All members must agree on the actual members atomically
  - Even in the case of implicit leaving – i.e. by crash of a member
- A group may get inoperable, because most members crash
  - Group must be recreated in this case
- Central group server
  - Easy to implement
  - Single point of failure
  - Central server easily becomes bottleneck
- Distributed group server
  - Difficult to implement
  - No single point of failure
  - No bottleneck due to central server

Group Addressing

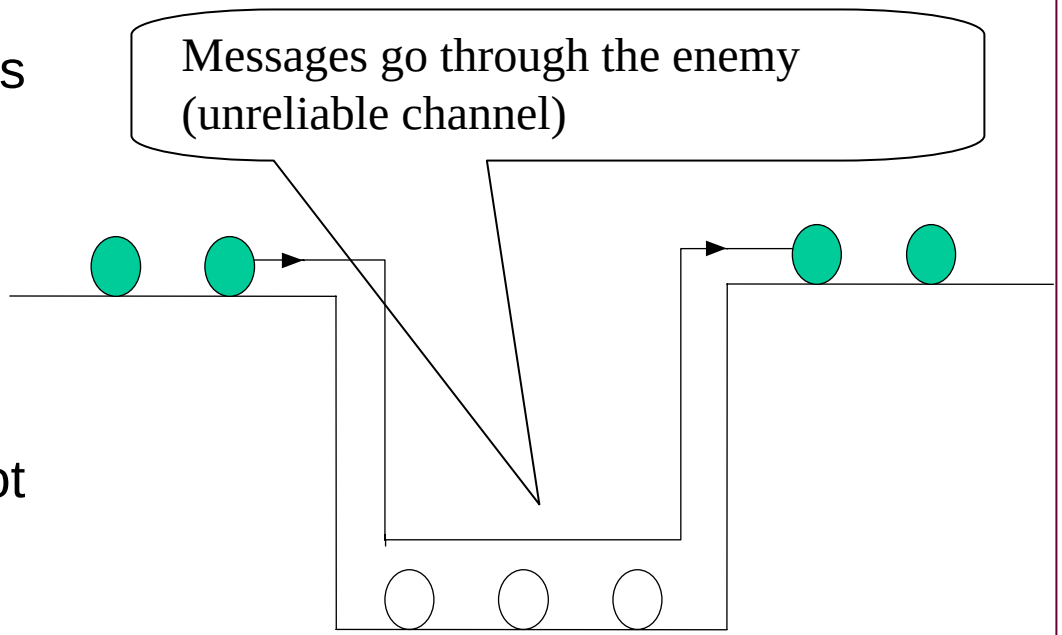
- Unicasting (single network receiver)
  - The system has to maintain a list of members
  - For *N* members *N* messages are necessary
- Broadcasting (all nodes of a nw. segment get the message)
  - The kernel may discard those that go to group-members not available on the given machine
- Multicasting (a selected group of nodes gets the message)
  - Group addresses can be mapped to multicast address
- Predicate Addressing
  - The receiver gets a Boolean expression. If this evaluates to true, the address is valid, otherwise not
  - The predicate may simply check group membership
  - It may contain other checks as well
    - E.g. the message should be accepted by all machines having some resources available (e.g. big main memory, magnetic tape etc.)

Failure Masking and Replication

- Groups may help in fault-tolerance
  - We replicate identical processes
  - Some of them may fail, the rest still works
- K fault tolerance
  - A system is *k fault tolerant*, if it "survives" the failure of k components
  - If k components simply stop
    - At least k+1 components are needed
  - If k components may produce wrong answers
    - At least 2k+1 components are needed to form a majority
    - In realistic cases we may need *more* – see later
  - We usually do not know, how many components will fail

Distributed agreement with faulty channels

- On an unreliable channel, in an asynchronous system, *no agreement is possible*, even with non-faulty processes
- The two-army problem
  - The divided dark army needs an agreement
  - Endless sequence of acknowledgments were necessary
  - If there was a *last* message, the sender of it still would not know, whether his message has arrived



Distributed Agreement with faulty processors

- Given is a set of processors $P = \{p_1, \ldots p_N\}$
- A subset $F \subset P$ is faulty, $P - F$ is not
- $\forall\ p_i \in P$ stores a value V_i
- During the agreement protocol, the processors calculate an agreement value A_i
- After the protocol ends the following two conditions hold:
  - $\forall\ (p_i, p_j) \in (P - F)$: $A_i = A_j$ (the agreement value)
  - The agreement value is a function of $\{V_i\} \in (P - F)$

Model of failure for distributed agreement

- An *"adversary"* (an "enemy") tries to make the protocol fail
- Most executions maybe correct but a few, unlikely executions are not
- The adversary may
  - Examine the global state
  - Schedule the execution protocol
  - Destroy or modify messages
  - Change the protocol at some of the processors
- For *synchronous systems*
  - There *are* some protocols to achieve a consensus
- For *asynchronous systems* a consensus is impossible
  - There is *no* algorithm that can guarantee that all non-failed processors agree on a value within finite time

Byzantine Agreement (1)

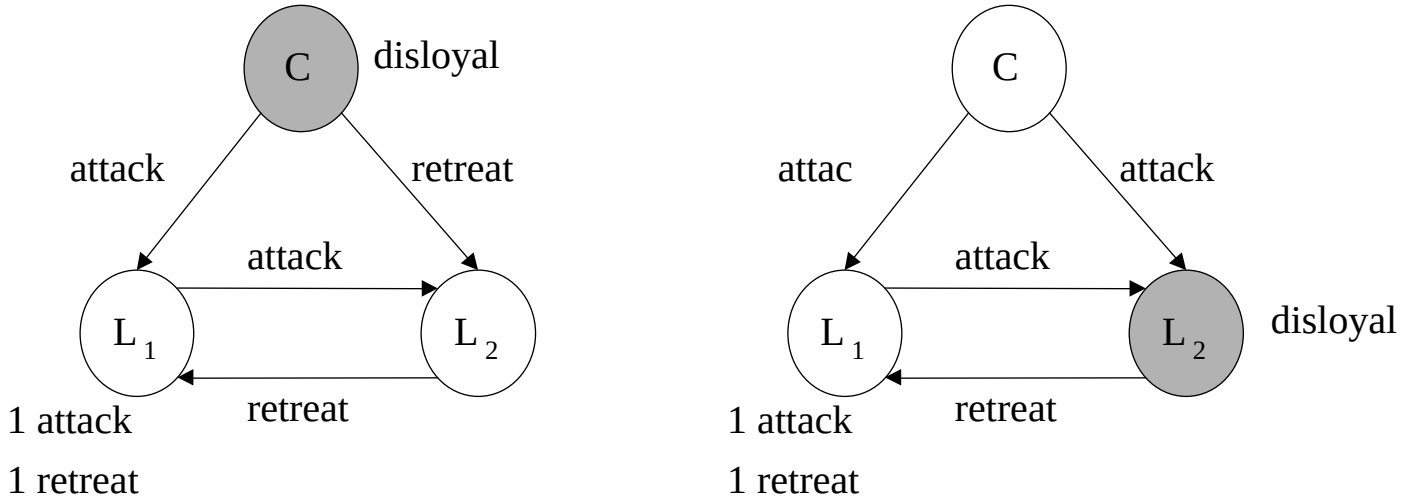
- Byzantine generals must coordinate their attacks against the army of the Turkish sultan
- *K* of them maybe *treacherous* (paid by the sultan)
- *1 commanding* and *N lieutenant* generals
- If the loyal generals agree, they win, otherwise they loose
- Failed processors may send arbitrary messages or none
- The system is *synchronous*
  - Non-faulty procs respond within *T*, non-answering procs are faulty
- The sender of a message can be identified by the receiver
- If each loyal general can agree on the opinion of the others (loyal or disloyal), loyal generals reach the same decision
- This needs a protocol for a *reliable broadcast*
  - Messages are seen in the same order by all procs – see later

Byzantine Agreement (2)

- Interactive consistency
  - If a loyal p_s sends V_s, all loyal generals agree on V_s
  - If the sender is treacherous, all loyal generals agree on the same value
- Suppose we know that only 1 general is treacherous
  - No consensus for 3 participants
    - There are not enough participants to form a majority
  - Either the commandant or one of the lieutenant is lying, the other two cannot figure out a consensus
  - Consensus for at least 4 participants
- If there are *t* traitors among *N* generals
  - An agreement cannot be reached if $N \leq 3t$
    - *2t+1* were only sufficient, if we knew, which one is the traitor!
  - An agreement can be reached if $N > 3t$, and if
    - The system is synchronous
    - Senders can be identified

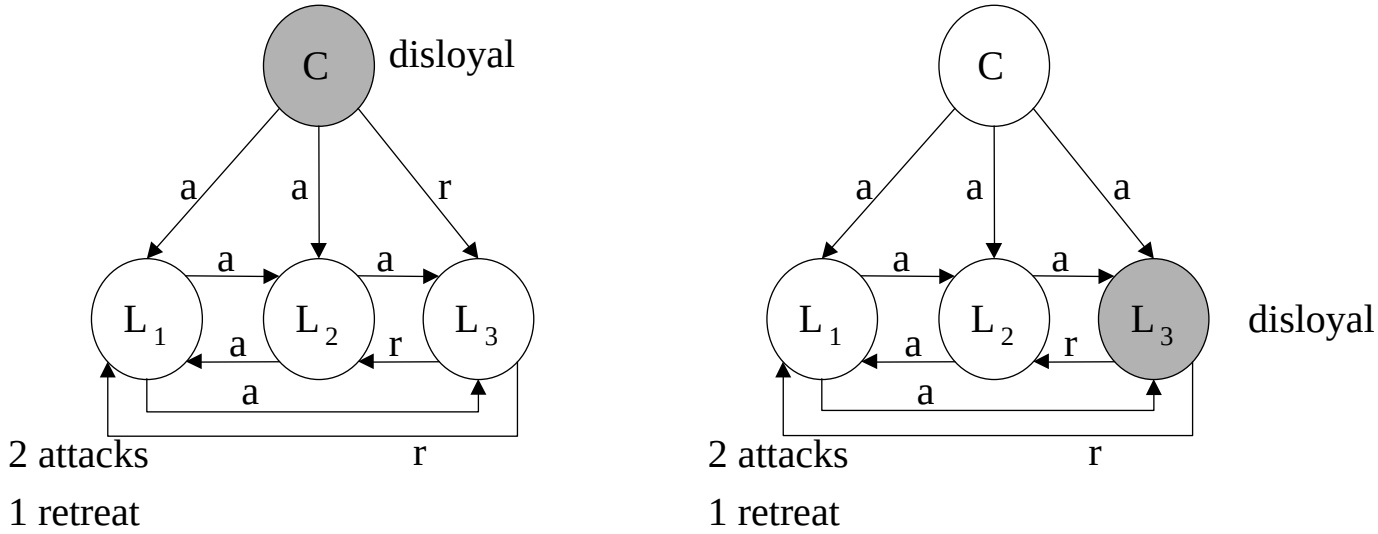
Byzantine Agreement (3)

- Assume we have 3 generals, at most 1 of them is a traitor
- In one case the commander is disloyal in the other case L_2
- L_1 receives in both cases 1 attack and 1 retreat message – no agreement is possible
- Further communication does not help – no new information



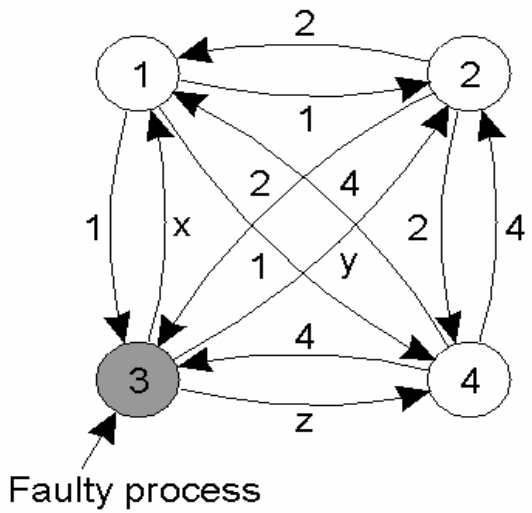
Byzantine Agreement (4)

- Assume we have 4 generals, at most 1 of them is a traitor
- In one case the commander is disloyal, in the other case L_3
- The loyal generals can agree in both cases on attack
  - In the first case L_1 – L_3 will attack
    - The loyal generals win, even if the commander wanted to “fool” them
  - In the second case C and L_1 and L_2 will agree
- If a general does not answer, a default is assumed – retreat



Byzantine Agreement (5)

- If not just a Boolean value is to agree (e.g. the strength of the troops): *Value vector*
  a) The generals announce their troop strengths (in battalions)
  b) The vectors that each general assembles based on (a)
  c) The vectors the loyal generals receive



(a)

1	Got(1, 2, x, 4)
2	Got(1, 2, y, 4)
3	Got(1, 2, 3, 4)
4	Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

Byzantine Broadcast Algorithm (1)

- The algorithm BG(k) works for *k* (or less) traitor
- Performing a broadcast that can tolerate *k* traitors requires that the lieutenants perform a broadcast that can tolerate *k-1* traitors (recursive algorithm):
  - If the commander is a traitor the loyal lieutenants have to agree – having max. *k-1* disloyal lieutenants
- *Voting vectors* contain the votes of all
- Correctness of the algorithm can be proved by induction
- Complexity: $O(N^k)$ for BG(k)
  - Unpractical, but can be improved

Byzantine Broadcast Algorithm (2)

Base Case

BG_Send(0, v, li)

The commander broadcasts v to every lieutenant on li,
with k = 0 faulty processors – everybody gets the message

BG_Receive(0)

Return the value sent to you or *retreat* if no message is received

Recursive Case

BG_Send(k, v, li)

Send *v* to every lieutenant on *li*

BG_Receive(k)

Let *v* be the value sent to you, or *retreat* if no value is sent

Let *li* be the set of lieutenants who have never broadcast *v* (i.e. the delivery list of this message)

BG-Send(k – 1, v, li – self)

Use BG_Receive(k-1) to receive $v_i \ \forall \ i \in li - self$

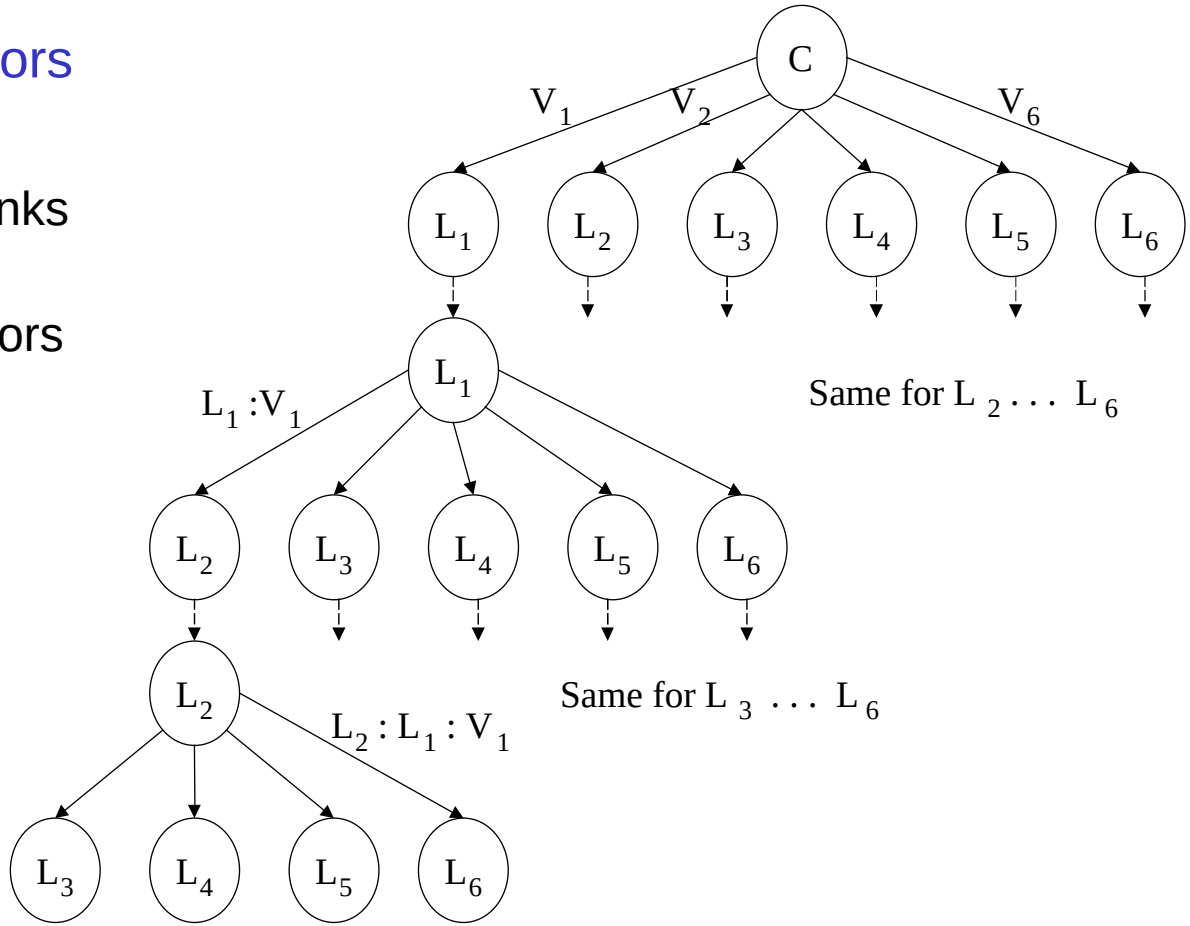
return majority(v, v_1, ... $v_{|li|-1}$)

or *retreat*, if no majority exists (half is attack, half is retreat and *n* is even)

Byzantine Broadcast Algorithm (3)

- Example:

  7 generals, 2 traitors
- Virtual tree
  - Shows, who thinks what of whom
  - The voting vectors can be seen as well



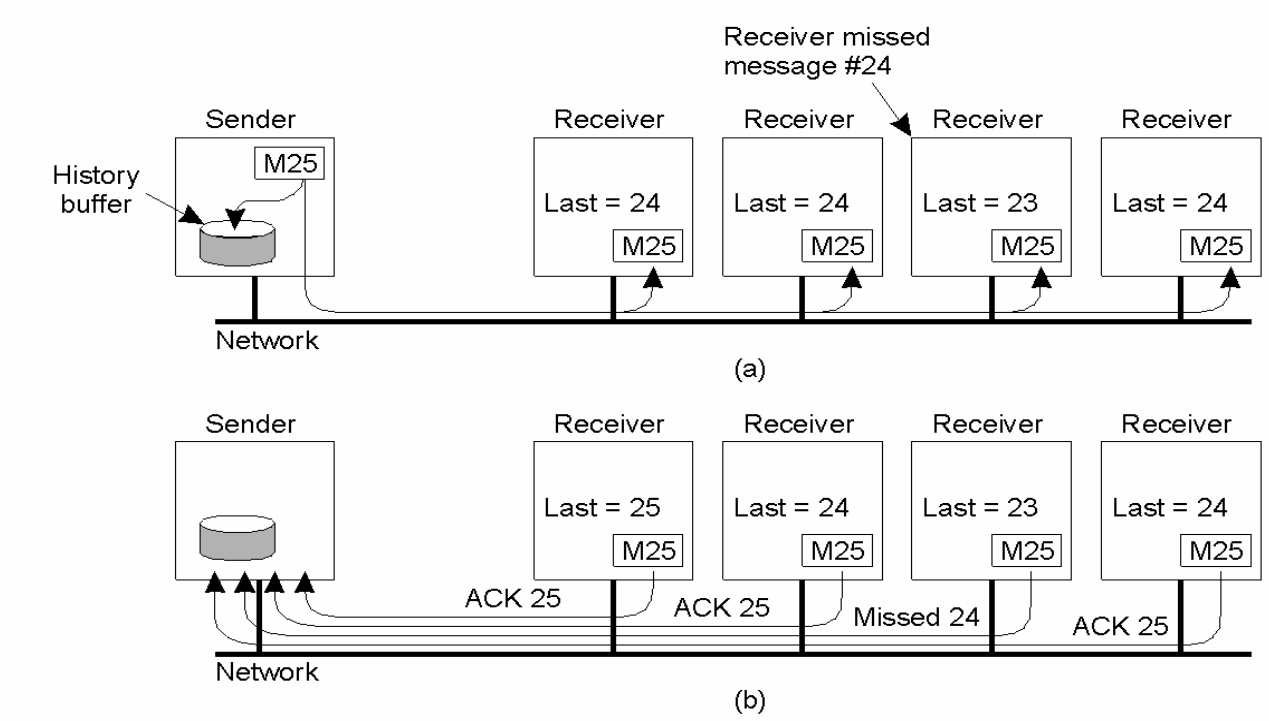
Byzantine Broadcast Algorithm (4)

- Commander broadcasts its order to 6 lieutenants
- Each lieutenant sends it to the 5 other lieutenants
- Each lieutenant broadcasts to the other 4 what he heard the other lieutenants say
- V_i represents the value sent to L_i, $L_i{:}V_i$ is L_i's rebroadcast, $L_j{:}L_i{:}V_i$ is L_j's rebroadcast of what L_i said
- After L_1 finishes its rebroadcast $L_1{:}V_1$
  - Each processor has a consensus of what the other processors think that L_1 broadcast
  - E.g. L_2 has seen: $(L_3{:}L_1{:}V_1, L_4{:}L_1{:}V_1, L_5{:}L_1{:}V_1, L_6{:}L_1{:}V_1)$
  - L_2 can compute the majority function for L_1's value
- After BG(1) finishes, each processor has a consensus of what the other processors received for their commands
  - E.g. L_1 has seen: $(L_2{:}V_2, L_3{:}V_3, L_4{:}V_4, L_5{:}V_5, L_6{:}V_6)$
  - It may decide on the commander's order by taking the majority opinion of the majority opinions

Reliable Multicast

- Reliable multicast
  - Each member of the group should get the message
  - Reliable point-to-point (TCP) channels don't suffice
  - What, if the sender crashes, or a new process joins during message delivery?
- Weak reliable multicast
  - We assume that the groups remains unchanged during the given message delivery
  - We assume also that the sender knows all receivers
  - Message numbering + history buffer at sender suffices

Weak reliable multicast



All receivers are known and are assumed not to fail

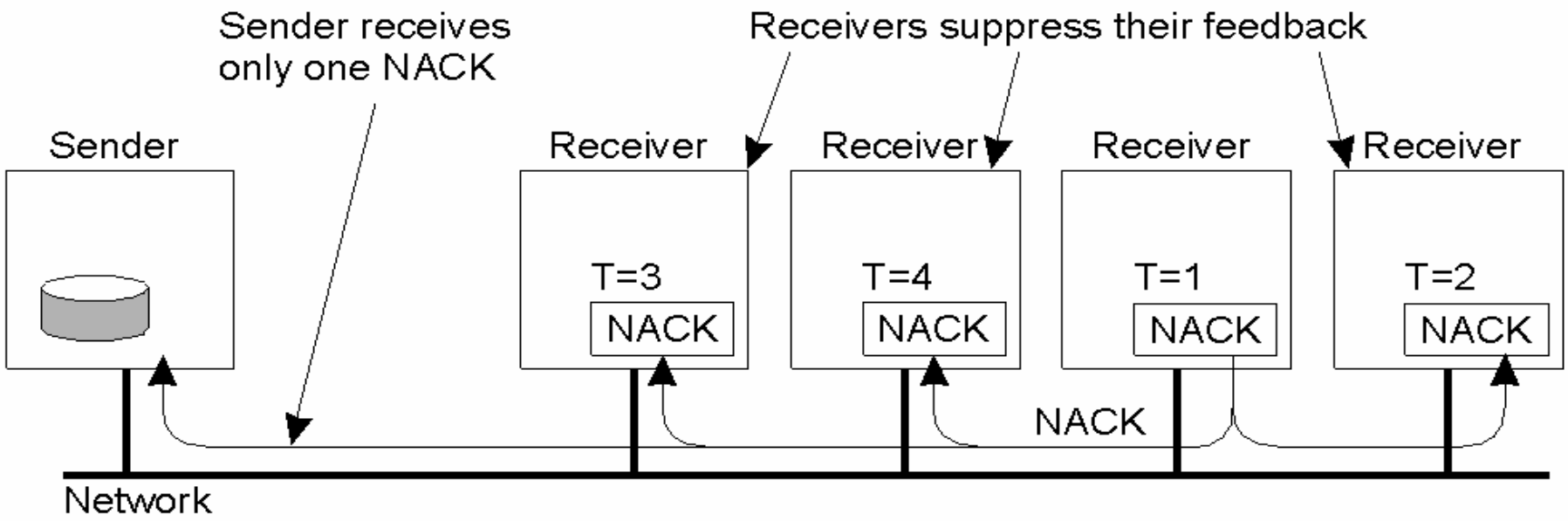
a) Message transmission

b) Reporting feedback

Scalability in Reliable Multicast

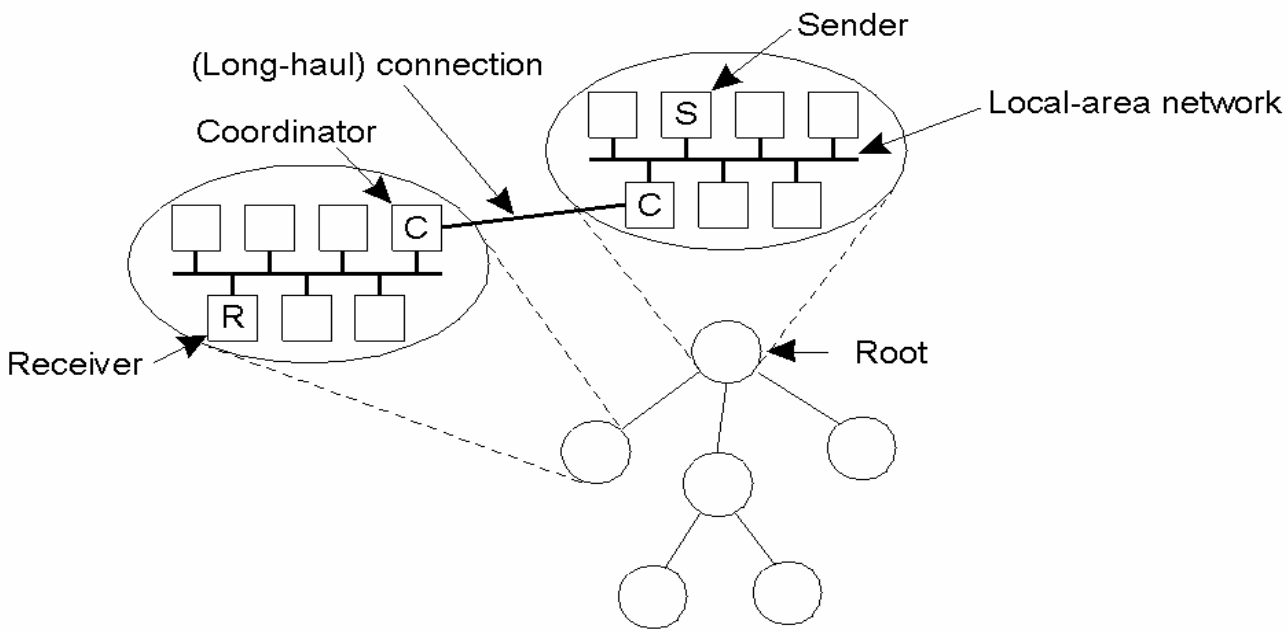
- Scalability problem
  - With many receivers the positive acknowledgments may generate too high load on the network + sender
- Negative acknowledgments (NAKs)
  - Load is smaller
  - Sender must store messages principally forever
- Nonhierarchical feedback control
  - Scalable Reliable Multicasting (SRM)
  - Feedback suppression
    - After a random delay *T*, NAKs are multicast to all members
    - NAK of the same message is transmitted only once – further load reduction

Nonhierarchical Feedback Control



- Several receivers have scheduled a request for retransmission
- The first retransmission request leads to the suppression of others

Hierarchical Feedback Control

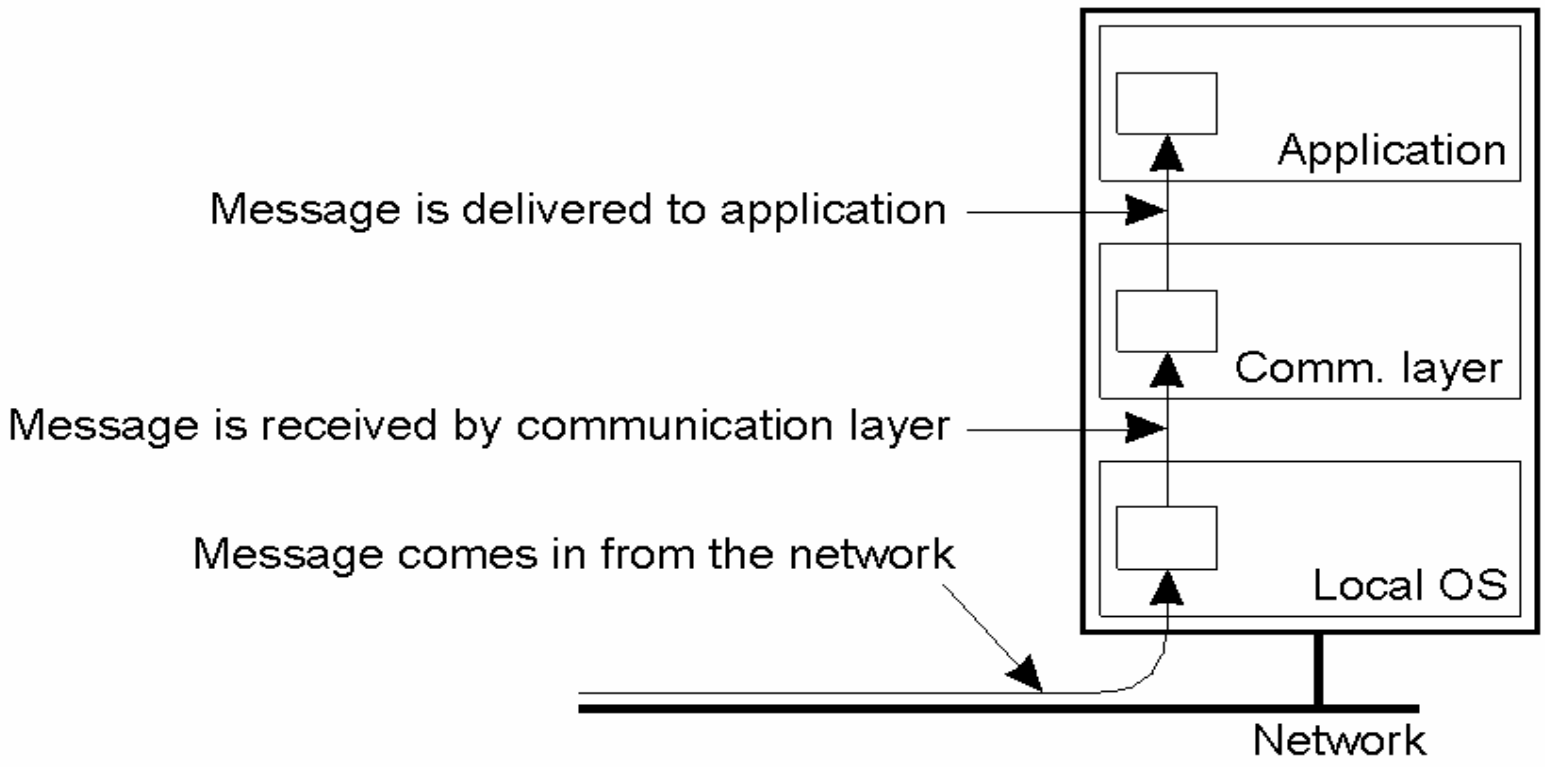


- The local coordinators form a tree
  - Tree creation may be difficult
- Local coordinator handles retransmission requests, own history buffer
  - On demand it requires message from father

Atomic Multicast

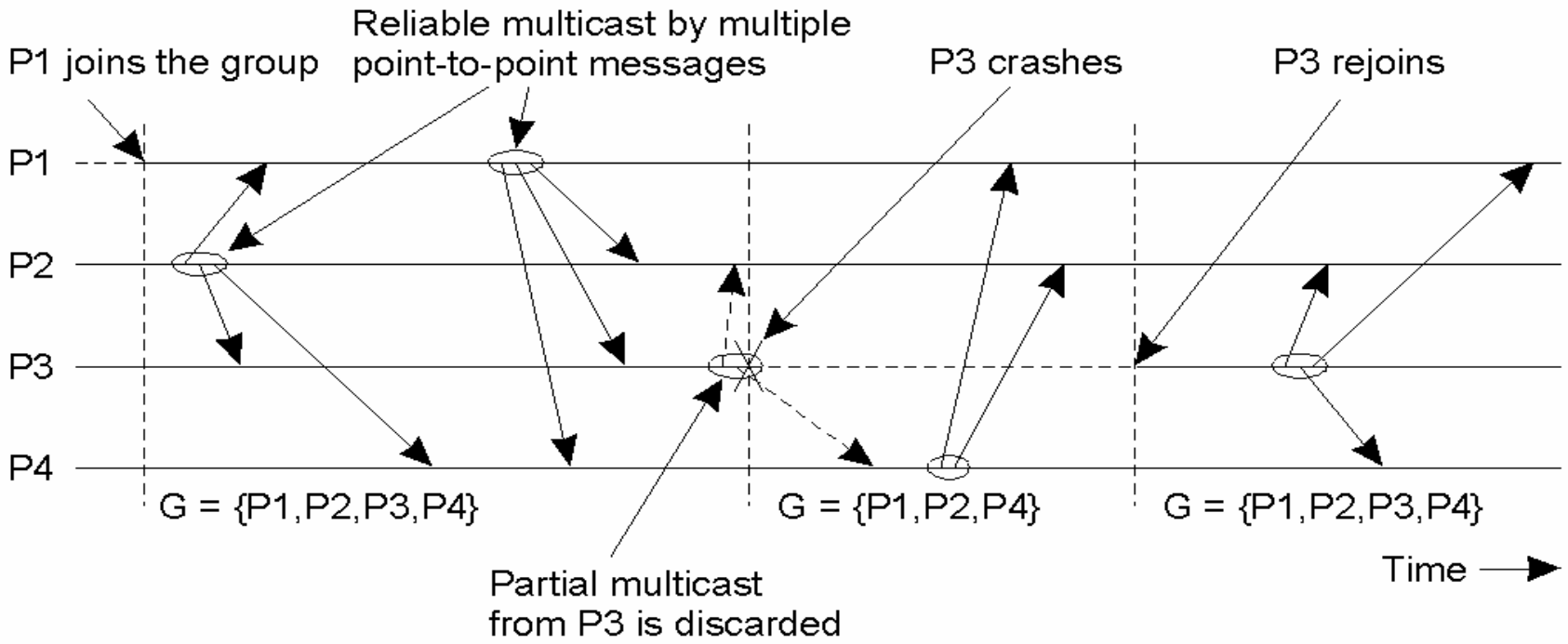
- All members of a group get all messages, even in the case of failures
- If the groups changes (join or leave): *view change*
- *Virtual Synchrony*
  - All multicast messages are delivered between view changes
  - Similar to the idea of consistent cuts
  - If a sender crashes, either all members get the message or nobody
- If in a virtual synchronous system all messages are received by all members in the same order: *atomic multicast*

Virtual Synchrony (1)



- The communication layer buffers out-or-order messages
- Delivery to the application may be deferred

Virtual Synchrony (2)



- Message *m* from P_3 could not be delivered *m* to P_1:
- the communication layer discards *m* in P_2 and P_4

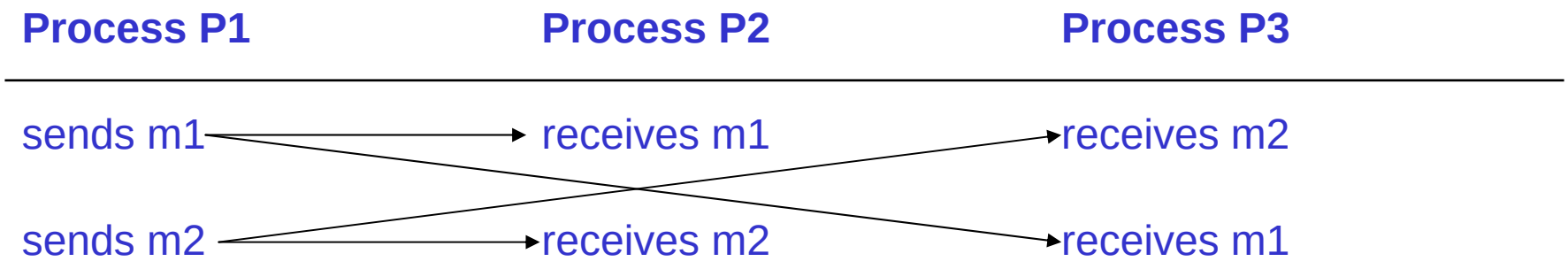
Message Ordering

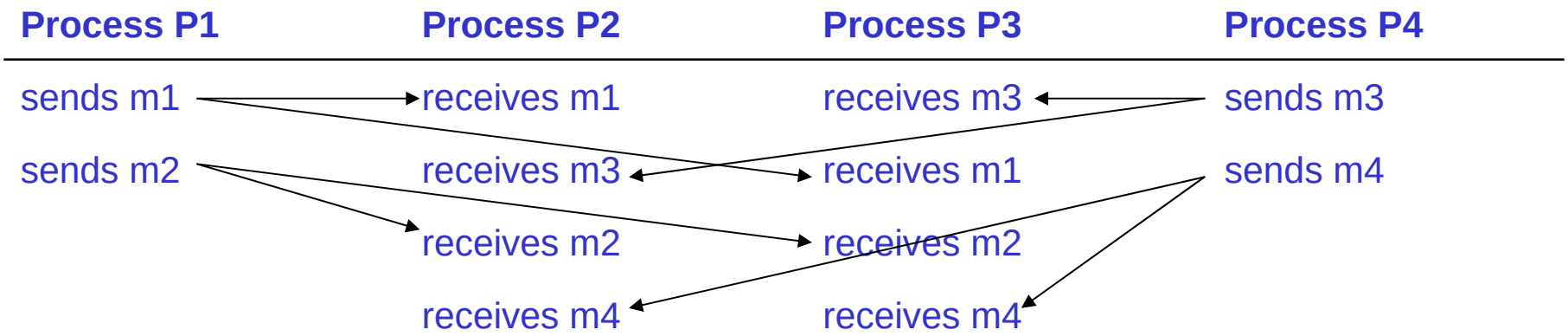
1. Unordered multicast
   - Arbitrary message order is accepted
2. FIFO-ordered multicast
   - Messages from the same sender are received in the same order
3. Causally-ordered multicast
   - Causal chains are preserved

Totally-ordered multicast

- All messages are received by all members in the same order
- This is an additional requirement to the basic ordering
- Combined with virtual synch: *atomic multicasting*

Unordered and FIFO-ordered multicast





Versions on virtual synch. reliable multicast

Multicast	Basic Message Ordering	Total-ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Broadcast in ISIS

- The ISIS group communication system
  - Implements different kinds of broadcast semantics
  - Assumes TCP based reliable point-to-point communication
- ABCAST
  - Loosely synchronous communication
    - All messages are delivered in the same order
  - Used for data transmission between members
  - Implemented by a two-phase commit protocol
  - Correct, but expensive
- GBCAST
  - Similar to ABCAST
  - Used for group management
- CBCAST
  - Virtually synchronous communication
  - Ensures causally ordered reliable multicast
  - Implementation is based on vector time stamps

CBCAST in ISIS

- Each process maintains a vector of size *n* (*n* members) containing the last message-number from $member_i$
- Each message also delivers such a vector
- If $process_i$ sends a message it increments $slot_i$
- If $process_i$ receives a message “too early” then it buffers the message, until the missing messages arrive
- V_i : i^{th} number of the vector in the incoming message
- L_i : i^{th} number of the vector stored at the receiver
- A message, sent by $member_j$ is immediately accepted if
- $V_j = L_j + 1$ (this is the next message from $node_j$) and
- $V_i \leq L_i$ ($\forall$ i $\neq$ j, i.e. the sender has not seen any message that the receiver has missed)

Example CBCAST in ISIS

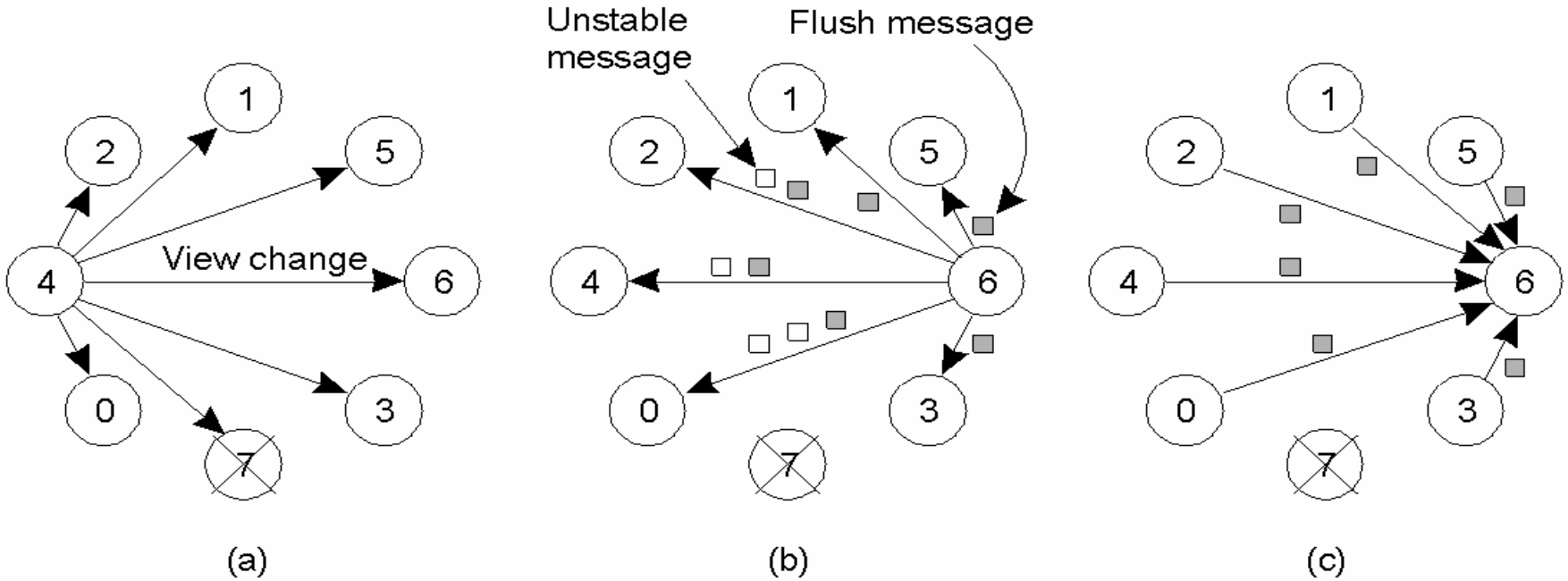
- $Process_0$: sent a message with vector (4, 6, 8, 2, 1, 5)
- $Process_1$: $V_0 = L_0 + 1$, $\forall\ i \neq j$: $V_i \leq L_i \rightarrow$ accept
- P_2: missed $message_6$ sent by P_1 ($V_1 > L_1$); P_3: has seen everything the sender has seen; P_4: missed the previous message from P_0; P_5: slightly ahead of P_0

P_0 (V)	P_1 (L)	P_2 (L)	P_3 (L)	P_4 (L)	P_5 (L)
4	3	3	3	2	3
6	7	5	7	6	7
8	8	8	8	8	8
2	2	2	3	2	3
1	1	1	1	1	1
5	5	5	5	5	5
sent	accept	delay	accept	delay	accept

Handling of crashed senders in ISIS

- If the sender process crashes during multicast
  - Some processes may not get the message *m* from it
  - They may get *m* from elsewhere
- Every process stores *m* until all members in a group *G* have received it
- If *m* has been received by all members: *stable*
  - An arbitrary process may send *m* to ensure stability
- Let call the current view G_i, the next veiw G_{i+1}
- If a process *P* receives a view change request
  - *P* forwards all unstable messages from G_i to every process in G_{i+1}
  - *P* sends a *flush* message to every process in G_{i+1} at the end
  - The point-to-point channels are reliable and keep order (TCP)
  - This protocol cannot handle process failures during view change

Handling of sender crash in CBCAST



a) P_4 notices that P_7 has crashed → sends a view change
b) P_6 sends out all its unstable messages, followed by a flush message
c) P_6 installs the new view when it has received a flush message from everyone else