Fault Tolerance

Distributed Systems [8]

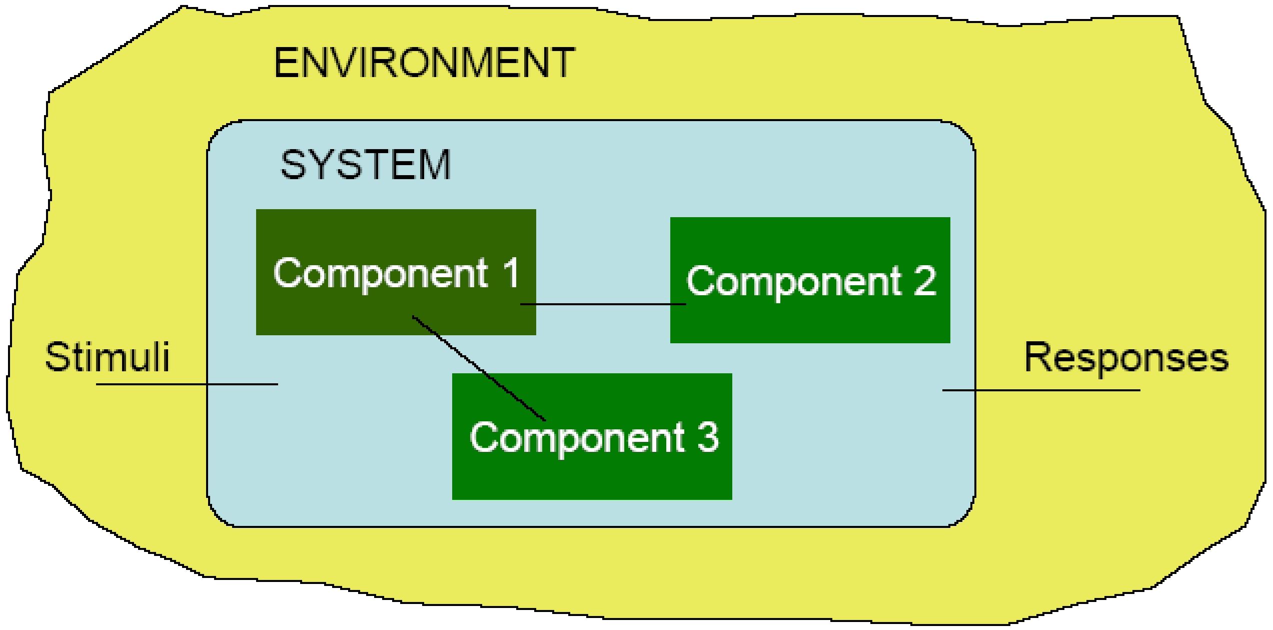
Basic Concepts

- Dependability Includes
  - Availability
  - Reliability
  - Safety
  - Maintainability

Dependability

- Reliability
  - A measure of success with which a system conforms to some authoritative specification of its behavior.
  - Probability that the system has not experienced any failures within a given time period.
  - Typically used to describe systems that cannot be repaired or where the continuous operation of the system is critical.
- Availability
  - The fraction of the time that a system meets its specification.
  - The probability that the system is operational at a given time *t*.
- Safety
  - When the system temporarily fails to conform to its specification, nothing catastrophic occurs.
- Maintainability
  - Measure of how easy it is to repair a system.

Basic System Concepts



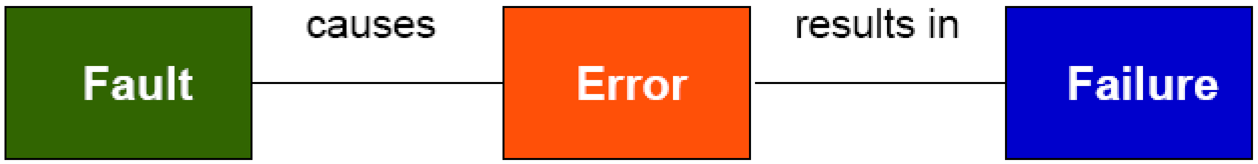
External state

Internal state

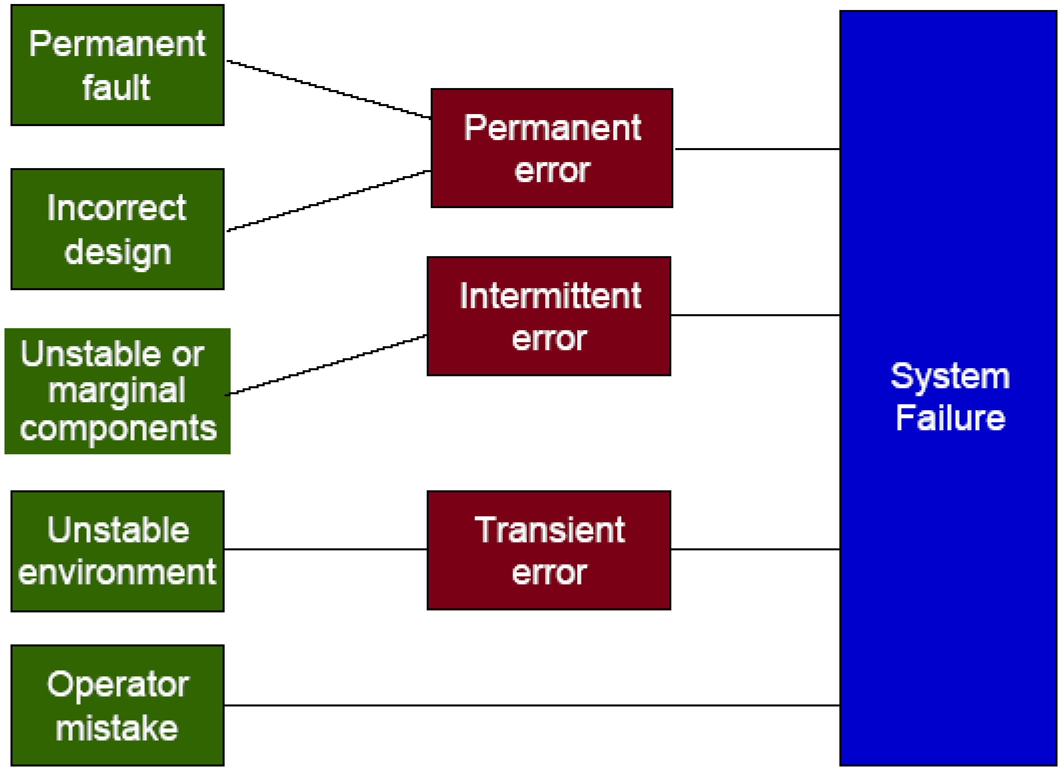
Fundamental Definitions

- Failure
  - The deviation of a system from the behavior that is described in its specification.
- Erroneous state
  - The internal state of a system such that there exist circumstances in which further processing, by the normal algorithms of the system, will lead to a failure which is not attributed to a subsequent fault.
- Error
  - The part of the state which is incorrect.
- Fault
  - An error in the internal states of the components of a system or in the design of a system.

Faults to Failures



Fault Classification



Failure Models

- 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 failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

How to Improve Dependability

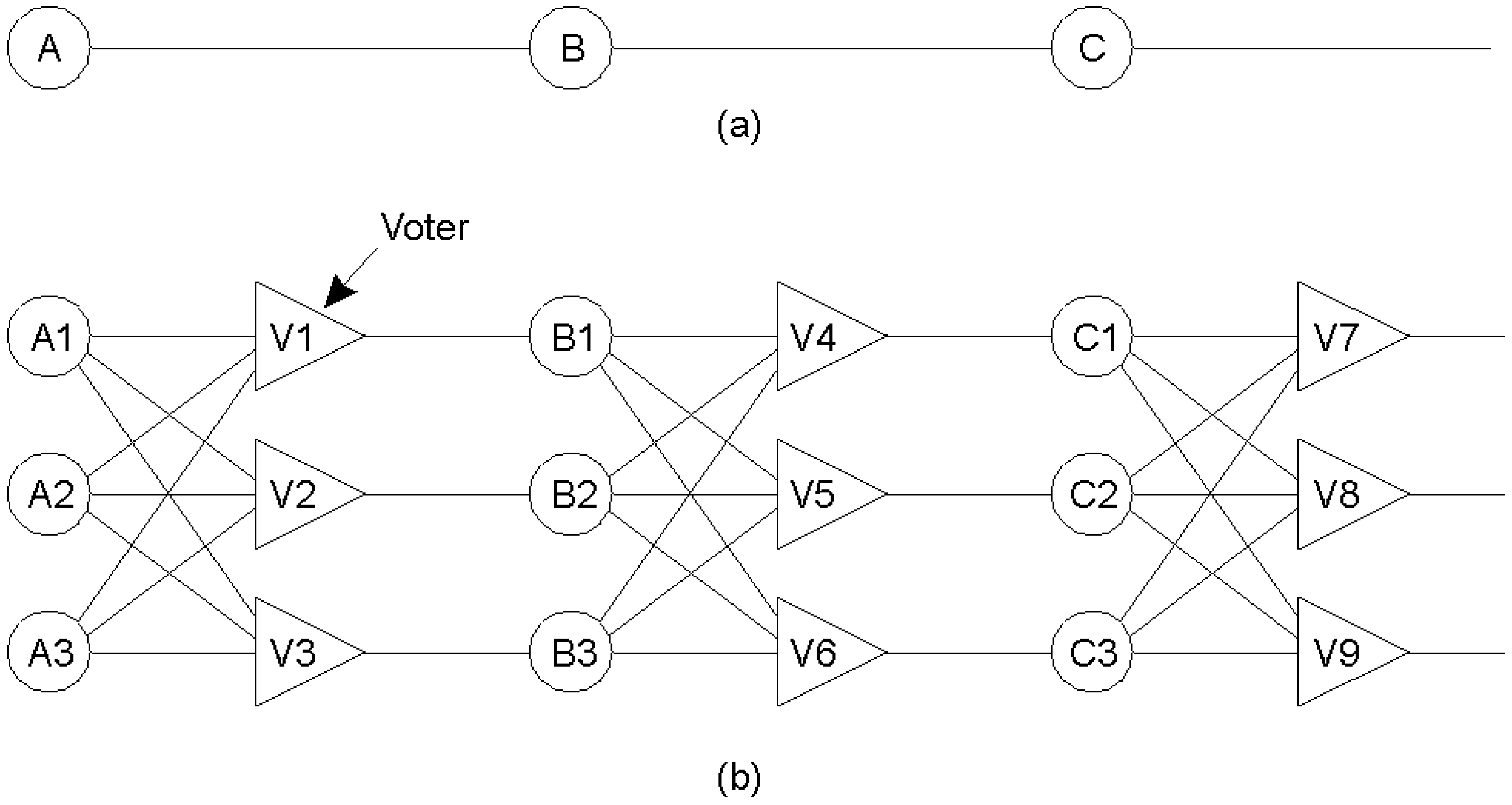
- Mask failures by redundancy
  - Information redundancy
    - E.g., add extra bits to detect and recovered data transmission errors
  - Time redundancy
    - Transactions; e.g., when a transaction aborts re-execute it without adverse effects.
  - Physical redundancy
    - Hardware redundancy
      - Take a distributed system with 4 file servers, each with a 0.95 chance of being up at any instant
      - The probability of all 4 being down simultaneously is 0.054 = 0.000006
      - So the probability of at least one being available (i.e., the reliability of the full system) is 0.999994, far better than 0.95
      - If there are 2 servers, then the reliability of the system is (1-0.052) = 0.9975
    - Software redundancy
      - Process redundancy with similar considerations
- A design that does not require simultaneous functioning of a substantial number of critical components.

Hardware Redundancy

- Two computers are employed for a single application, one acting as a standby
  - Very costly, but often very effective solution
- Redundancy can be planned at a finer grain
  - Individual servers can be replicated
  - Redundant hardware can be used for non-critical activities when no faults are present
  - Redundant routes in network

Failure Masking by Redundancy

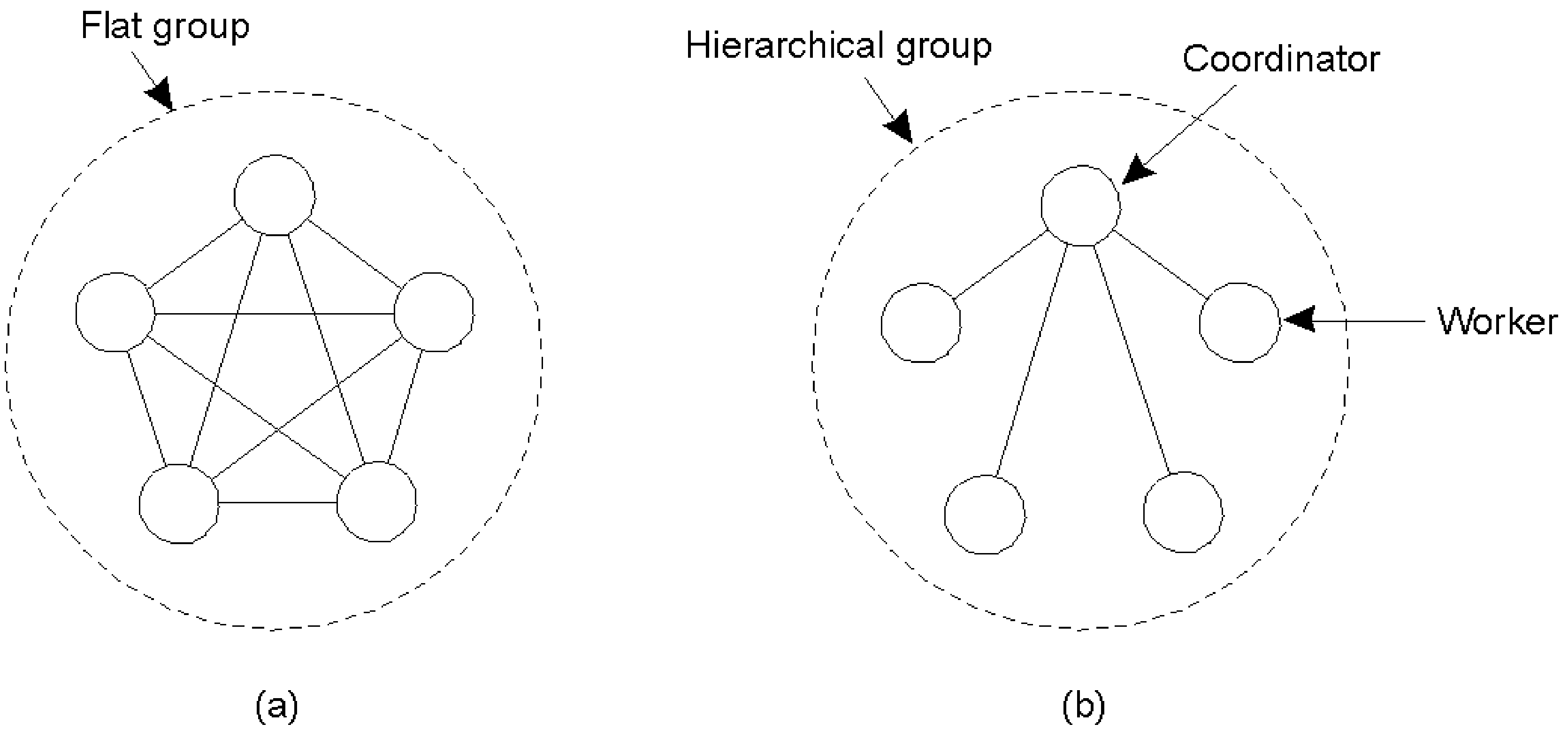
- Triple modular redundancy.



Process Redundancy

- Process groups
  - All members of a group receive a message to the group
  - If one process fails, others can take over
  - Can be dynamic; processes can have multiple memberships
  - Flat vs. hierarchical groups

Flat Groups versus Hierarchical Groups



a) Communication in a flat group.

b) Communication in a simple hierarchical group

Management of Replicated Processes

- Primary copy
  - Primary-backup setup
  - Coordinator is the primary that coordinates all updates
  - If coordinator fails, one backup takes over (usually through an election procedure)
  - Processes are organized hierarchically
- Replicated-writes
  - Active replication and quorum-based protocols
  - Flat group organization
  - No single points of failure

Process resilience

How can fault tolerance be achieved in distributed systems?

1. Key approach to tolerating a faulty process:

replicate the process and organize these identical process into a group.

$$\begin{tikzcd}
C \arrow[r, "m"'] & \text{process}
\end{tikzcd}$$

$$\begin{tikzcd}
 & \text{process} \\
C \arrow[ur, "m"'] \arrow[r, "m"'] \arrow[dr] & \text{process} \\
 & \text{process}
\end{tikzcd}$$

Some design issues

- How many replicas? For K fault tolerant,

  Fail-silent faults : K+1

  Byzantine faults : 2K+1 majority

- Structure of group: flat/hierarchical
- Need for managing groups and group membership
  - centralized: group server
  - distributed: totally-ordered reliable multicast

Agreement in faulty system

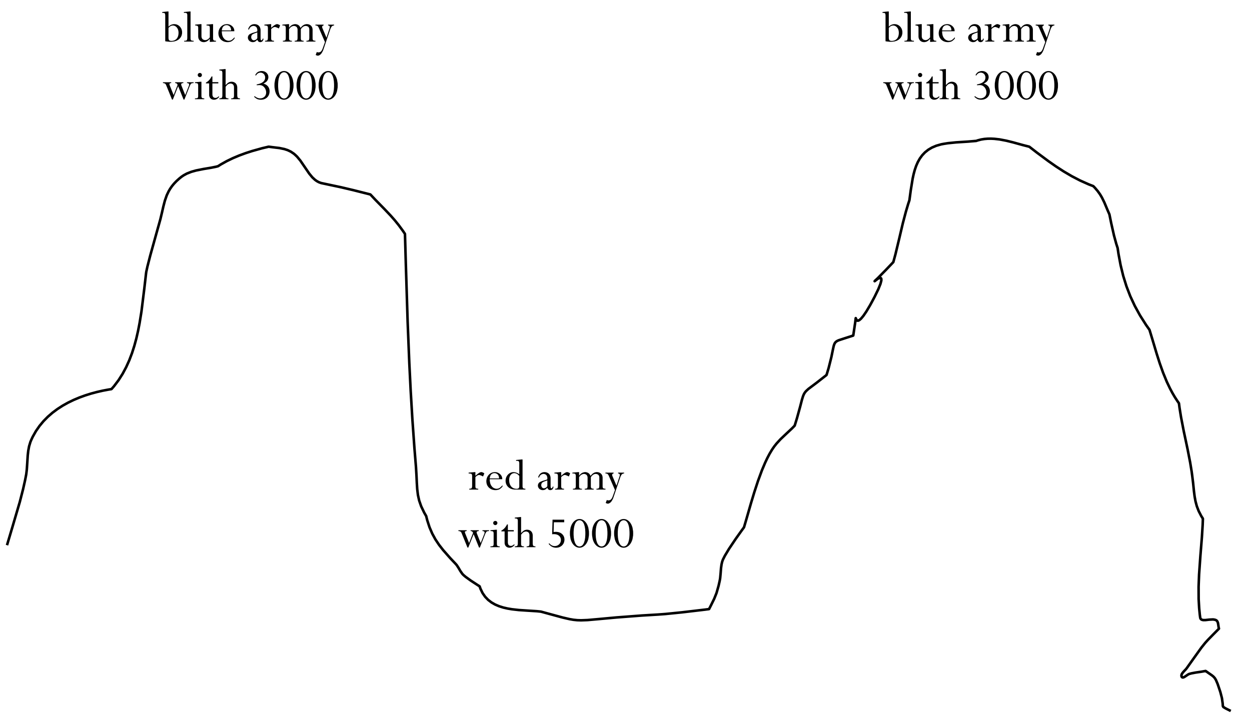
Introduction

There is a need to have processes agree on something.

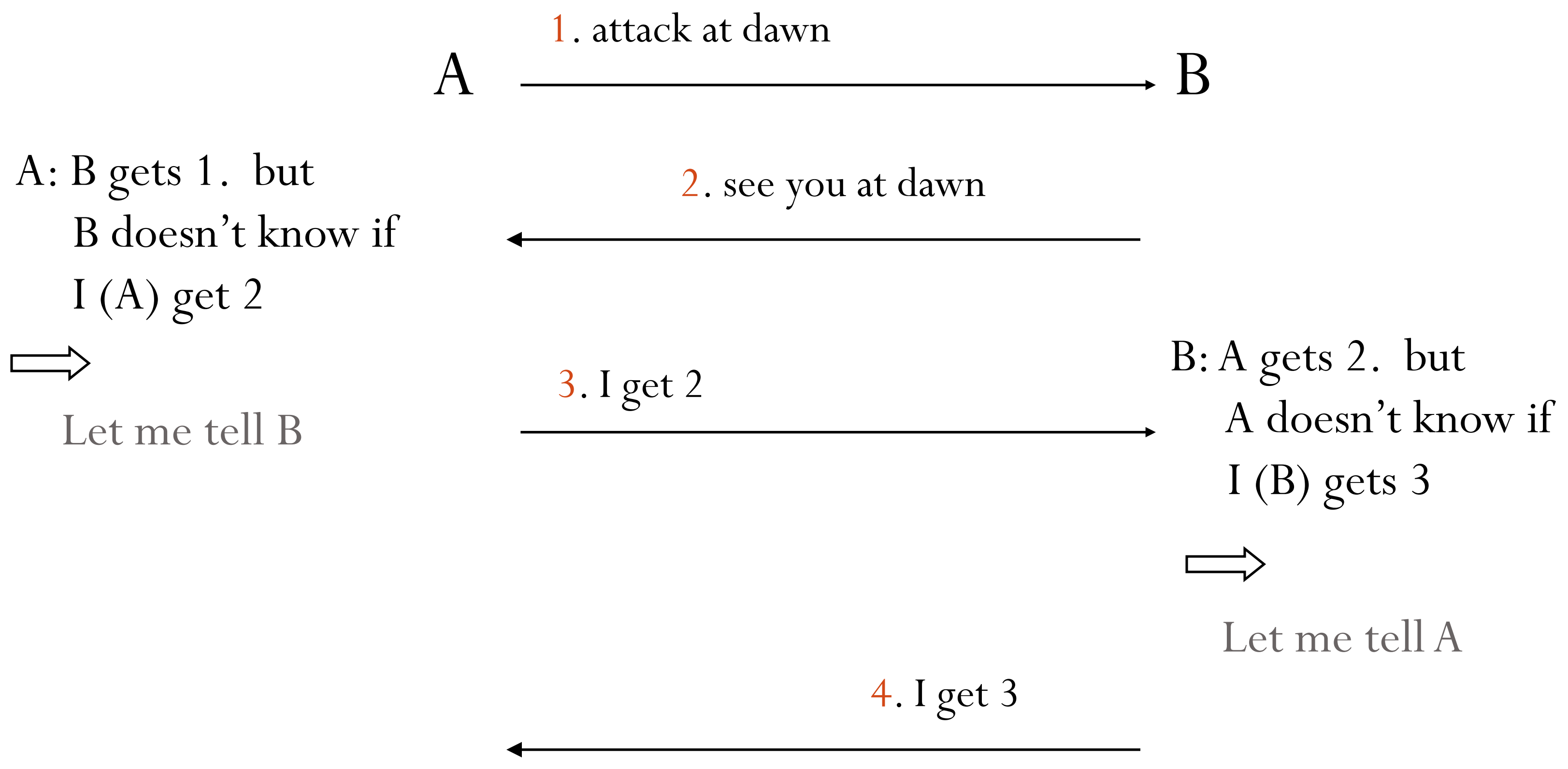
- Goal: all non-faulty processors reach consensus on some issue within a finite number of steps
- Two kinds of fault:
  - communication fault : two-army problem
  - processor fault : Byzantine generals problem

Two-army problem

- Problem:



Two blue armies want to coordinate their attacks on the red army. But they can only send messenger



A and B will never reach agreement

Because sender of the last message doesn’t know if the last message arrived.

- Conclusion

  agreement between two processes is not possible in the case unreliable communication

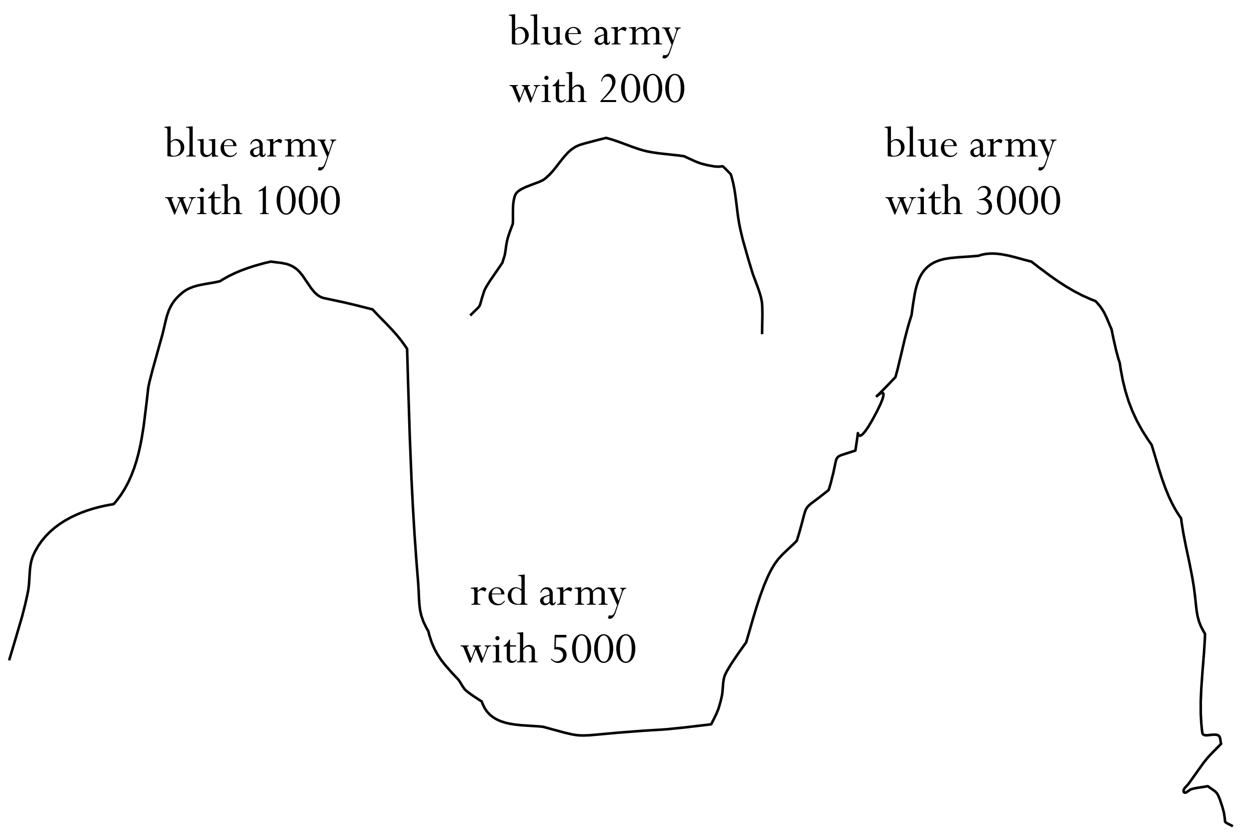
- How does TCP deal with this problem when two computer want to establish a TCP connection ?

Byzantine generals problem

Communication is perfect but the processors are not (in Byzantine faults) .

- Problem:

  n blue generals want to coordinate their attacks on the red army. But *m* of them are traitors.



Question:

Whether the loyal generals can still reach agreement?

Generality

Generals exchange troop strengths, at the end, each general has a vector of length n corresponding to all the armies.

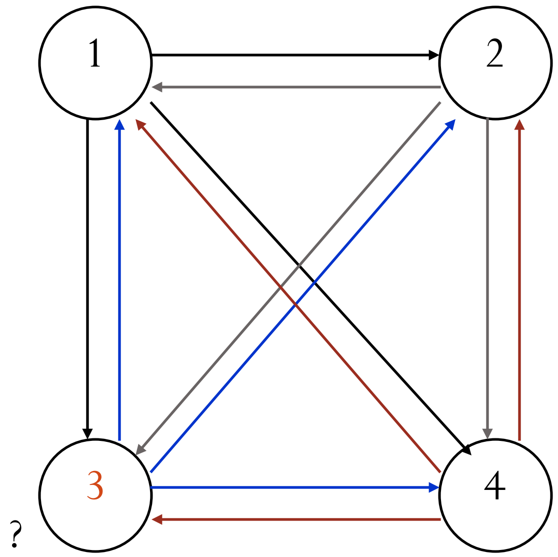
Let n=4, m=1

general 1 has 1K troop

general 2 has 2K troop

general 3 is traitor

general 4 has 4K troop



1 is not sure if 2 sends him true message

1 is not sure if 3 sends him true message

1 is not sure if 4 sends him true message

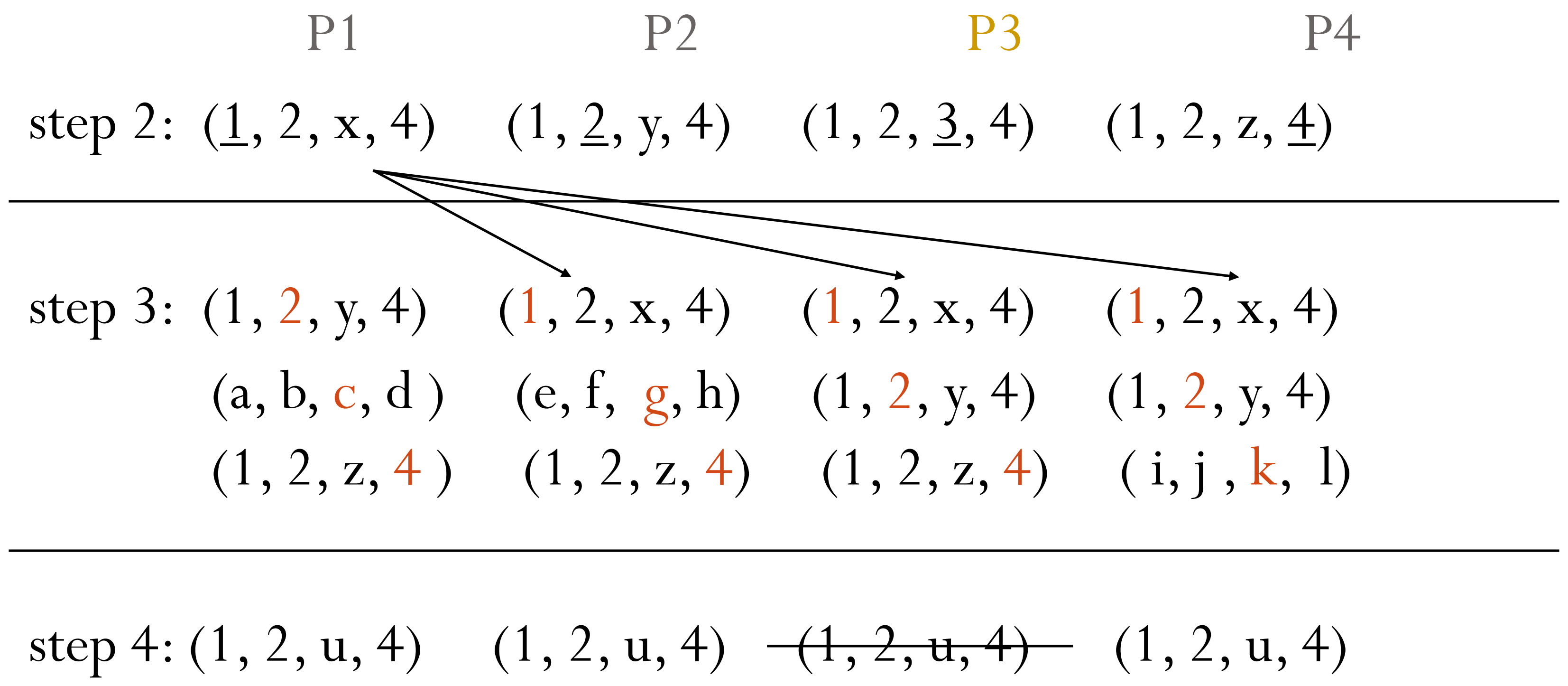
- Algorithm to reach agreement. They perform the following :

step 1: every general sends a message to every other general telling his strength (true or lie)

step 2: each general collects received messages to form a vector

step 3: every general passes his vector to every other general

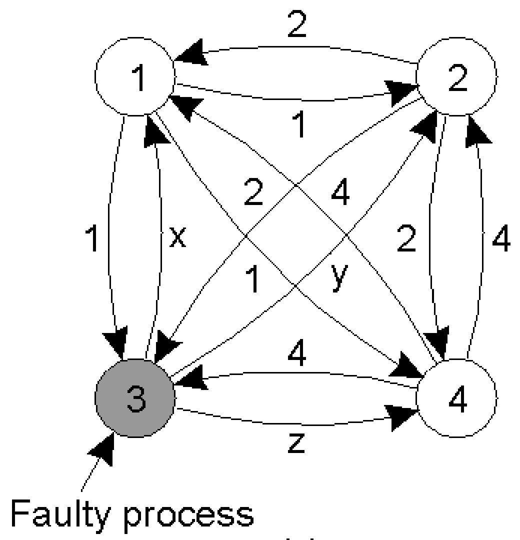
step 4: each general examines the ith element of each of the newly received vectors. If any value has a majority, that value is put into the result vector



	P1	P2	P3	P4
step 2:	($\underline{1}$, 2, x, 4)	(1, $\underline{2}$, y, 4)	(1, 2, $\underline{3}$, 4)	(1, 2, z, $\underline{4}$)
step 3:	(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)	(1, 2, x, 4)
	(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)	(1, 2, y, 4)
	(1, 2, z, 4)	(1, 2, z, 4)	(1, 2, z, 4)	(i, j , k, l)
step 4:	(1, 2, u, 4)	(1, 2, u, 4)	~~(1, 2, u, 4)~~	(1, 2, u, 4)

	P1	P2	P3	P4
step 2:	(_, 2, x, 4)	(1, _, y, 4)	(1, 2, _, 4)	(1, 2, z, _)
step 3:	(1, _, y, 4)	(_, 2, x, 4)	(_, 2, x, 4)	(_, 2, x, 4)
	(a, b, _, d)	(e, f, _, h)	(1, _, y, 4)	(1, _, y, 4)
	(1, 2, z, _)	(1, 2, z, _)	(1, 2, z, _)	(i, j, _, l)
step 4:	(1, 2, u, 4)	(1, 2, u, 4)	~~(1, 2, u, 4)~~	(1, 2, u, 4)

Agreement in Faulty Systems (1)



(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)

- The Byzantine generals problem for 3 loyal generals and1 traitor.

a) The generals announce their troop strengths (in units of 1 kilosoldiers).

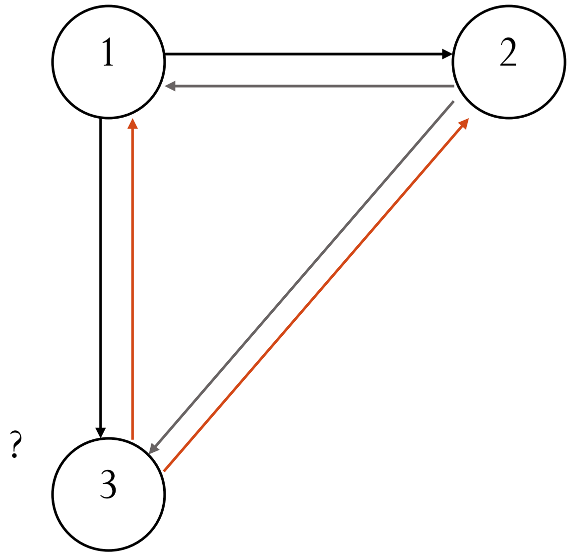
b) The vectors that each general assembles based on (a)

c) The vectors that each general receives in step 3.

- ③ is Byzantine faulty processor

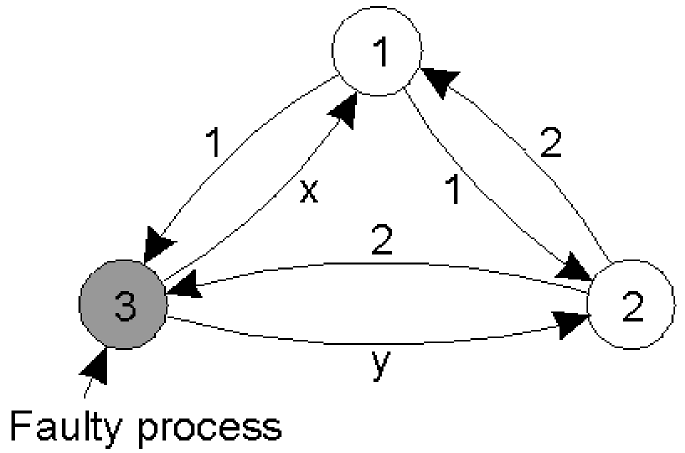
A system with m faulty processors, agreement can be achieved only if 2m+1 processors work properly, for a total of 3m+1. i.e. >2n/3

For example, let n=3, m=1



	P1	P2	P3
step 2:	($\underline{1}$, 2, x)	(1, $\underline{2}$, y)	(1, 2, $\underline{3}$)
step 3:	(1, 2, y)	(1, 2, x)	(1, 2, x)
	(a, b, c)	(d, e, f)	(1, 2, y)
step 4:	(u, u, u)	(u, u, u,)	~~(1, u, u)~~

Agreement in Faulty Systems (2)



(a)

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

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

- The same as in previous slide, except now with 2 loyal generals and one traitor.

Distributed Consensus

- Consider a set of ***n*** isolated processors, of which it is known that no more than ***m*** are faulty. It is not known, however, which processors are faulty. Suppose that the processors can communicate only by means of two-party messages. The communication medium is presumed to be fail-safe and of negligible delay. The sender of a message, moreover, is always identifiable by the receiver. Suppose also that each processor p has some private value of information Vp (such as its clock value or its reading of some sensor).

Interactive Consistency(IC)

- The question is whether for given m, n > 0, it is possible to devise an algorithm based on an exchange of messages that will allow each nonfaulty processors to compute a vector of values with an element for each of the n processors, such that

- (1) the nonfaulty processors compute exactly the ***same vector***;
- (2) the element of this vector corresponding to a given nonfaulty processor is the private value of that processor.

Requirements

- The generals must have an algorithm to guarantee that:
  - A. All loyal generals decide upon the same plan of action.
  - B. A small number of traitors cannot cause the loyal generals to adopt a bad plan.

For condition A to be satisfied, the following must be true:

- 1. Every loyal general must obtain the same information v (1) . . . . , v (n).

  1'. Any two loyal generals use the same value of *v(i)*.
- 2. If the ith general is loyal, then the value that he sends must be used by every loyal general as the value of v (i)

Byzantine Generals Problem

A commanding general must send an order to his n - 1 lieutenant generals such that

- IC1. All loyal lieutenants obey the same order.
- IC2. If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

Oral Message Algorithm

Algorithm OM(0):

1. Commander sends his value to every lieutenant
2. Each lieutenant uses the value received or “retreat” if no value received

Algorithm OM(m), m > 0:

1. Commander sends his value to every lieutenant
2. For each *i*, let v_i be the value that lieutenant *i* receives from the commander or “retreat”. Lieutenant *i* acts as the commander in OM(m-1) to send the value v_i to each of the other n-2 other lieutenants
3. For each *i*, and each *j* <> *i*, let v_j be the value lieutenant *i* received from lieutenant *j* in step 2. Lieutenant *i* uses the value *majority* $(v_1, \ldots, v_{n-1})$

Signed Messages

- A1. Every message that is sent is delivered correctly.
- A2. The receiver of a message knows who sent it.
- A3. The absence of a message can be detected.
- A4. (a) A loyal general's signature cannot be forged, and any alteration of the contents of his signed messages can be detected.

  (b) Anyone can verify the authenticity of a general's signature.

Reliable Client-Server Communication

Example, RPC

1. Normal jobs of c_stub and s_stub

- C_stub: pack, send, recv, unpack
- S_stub: recv, unpack, call, send

2. Fault tolerant RPC

- C_stub Client
- S_stub Server

} What should they do if …?

Five different classes of failures

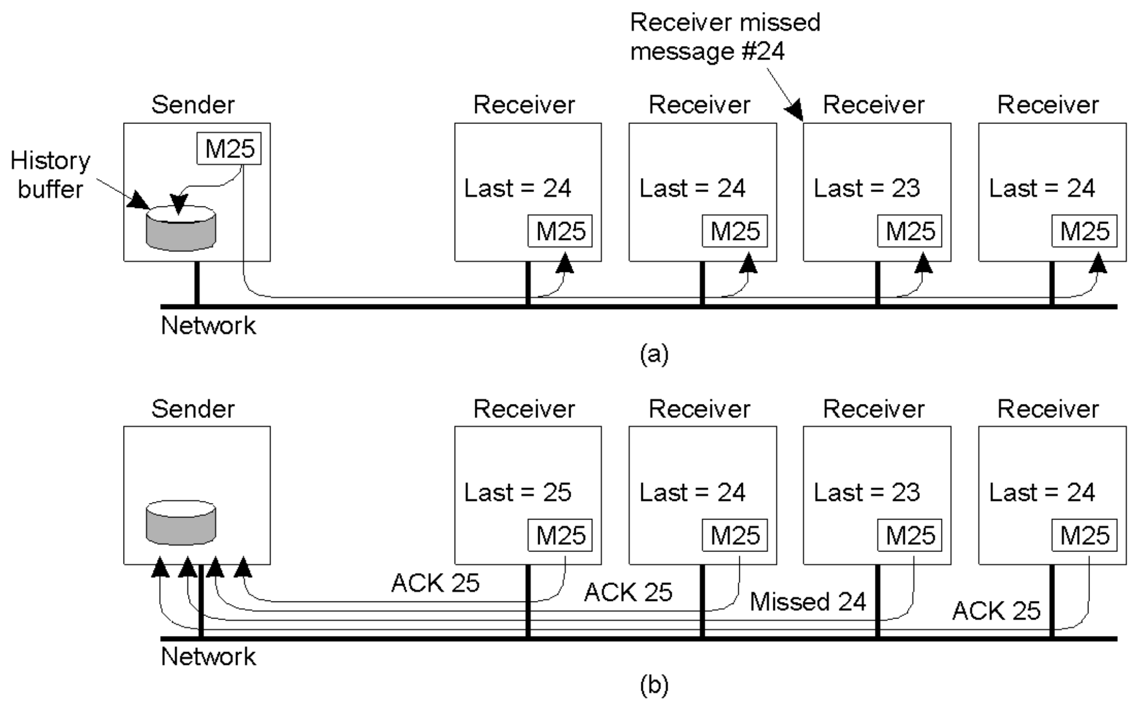
- The client is unable to locate the server
- The request message from client to server is lost
- The server crash after receiving a request
- The reply message from server to client is lost
- The client crashes after sending a request

Reliable Multicast

1. Basic reliable-multicasting schemes

2. Scalability in reliable multicasting
    - Nonhierarchical Feedback Control
    - Hierarchical Feedback Control

Basic Reliable-Multicasting Schemes

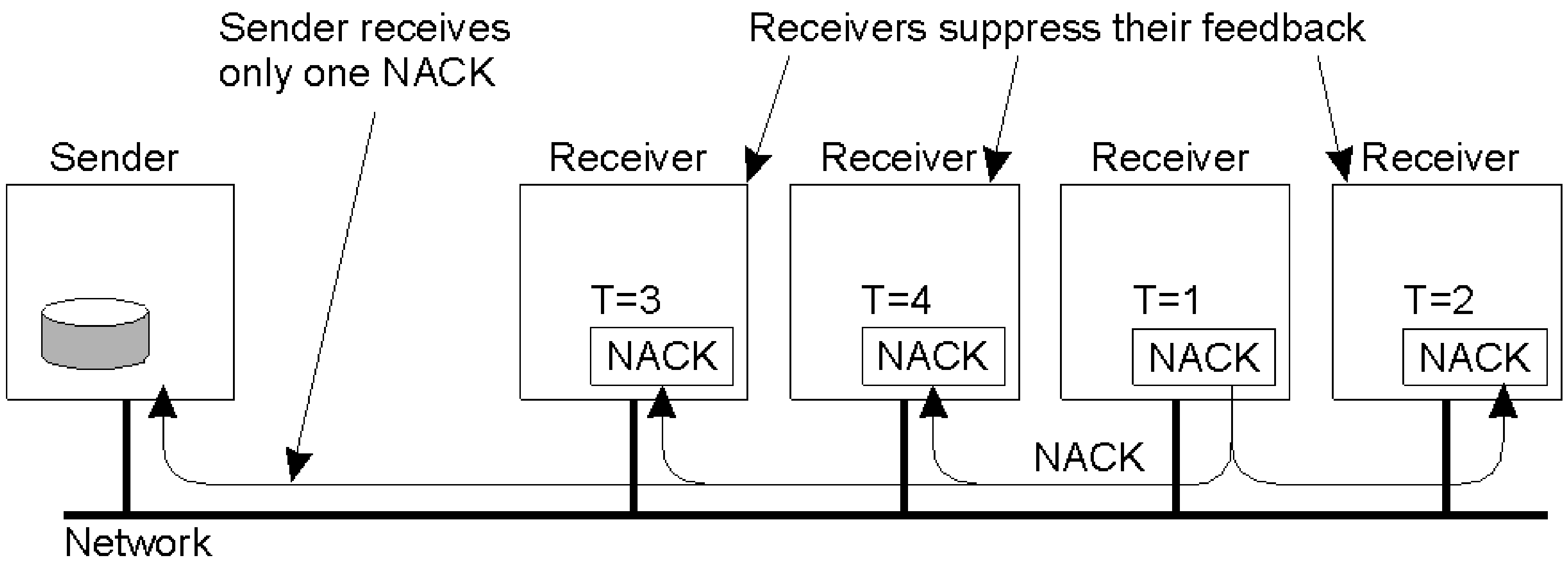


- A simple solution to reliable multicasting when all receivers are known and are assumed not to fail

a) Message transmission

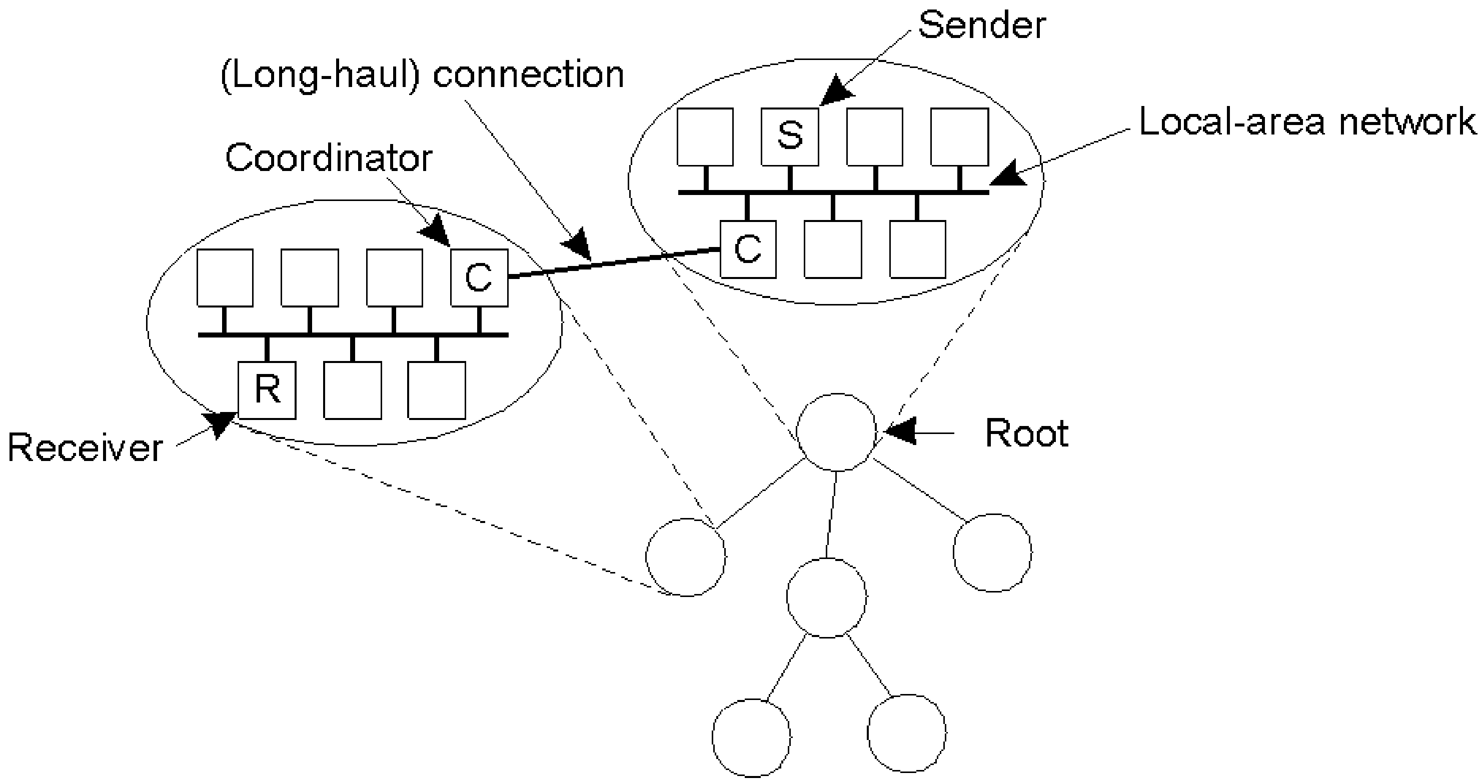
b) Reporting feedback

Nonhierarchical Feedback Control



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

Hierarchical Feedback Control



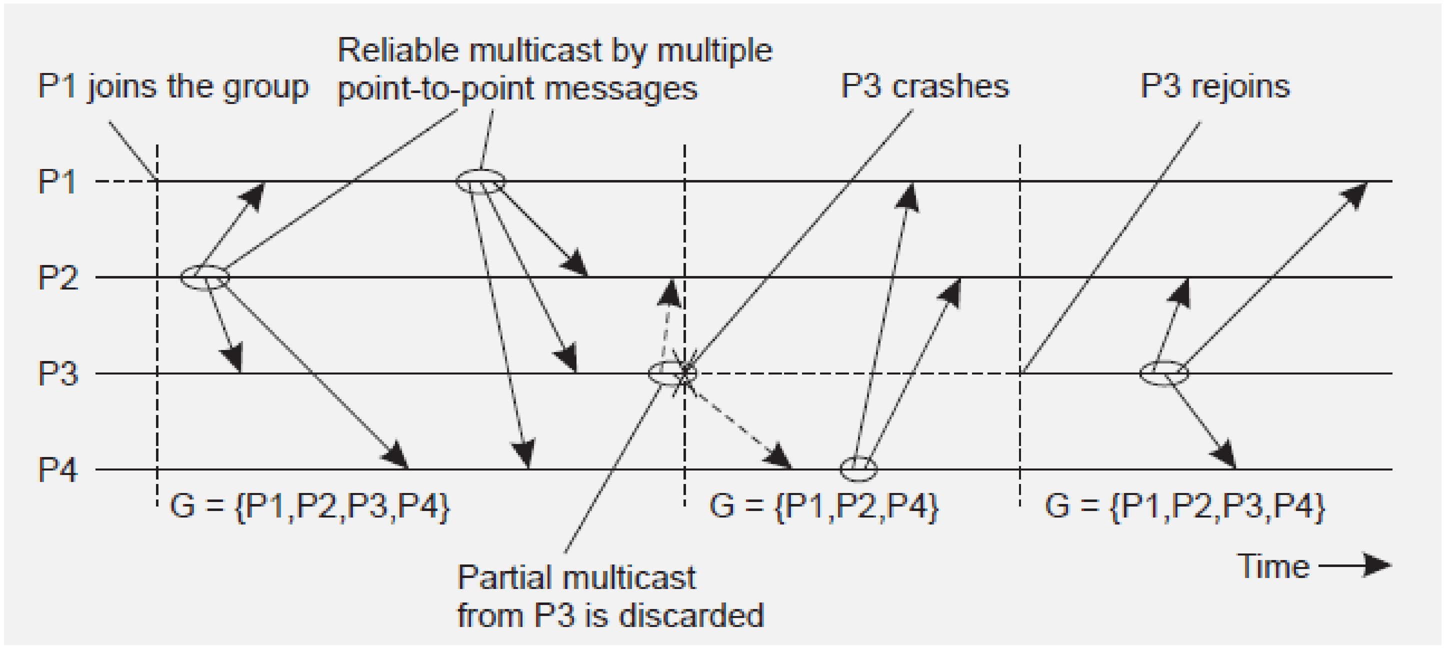
- The essence of hierarchical reliable multicasting.

a) Each local coordinator forwards the message to its children.

b) A local coordinator handles retransmission requests.

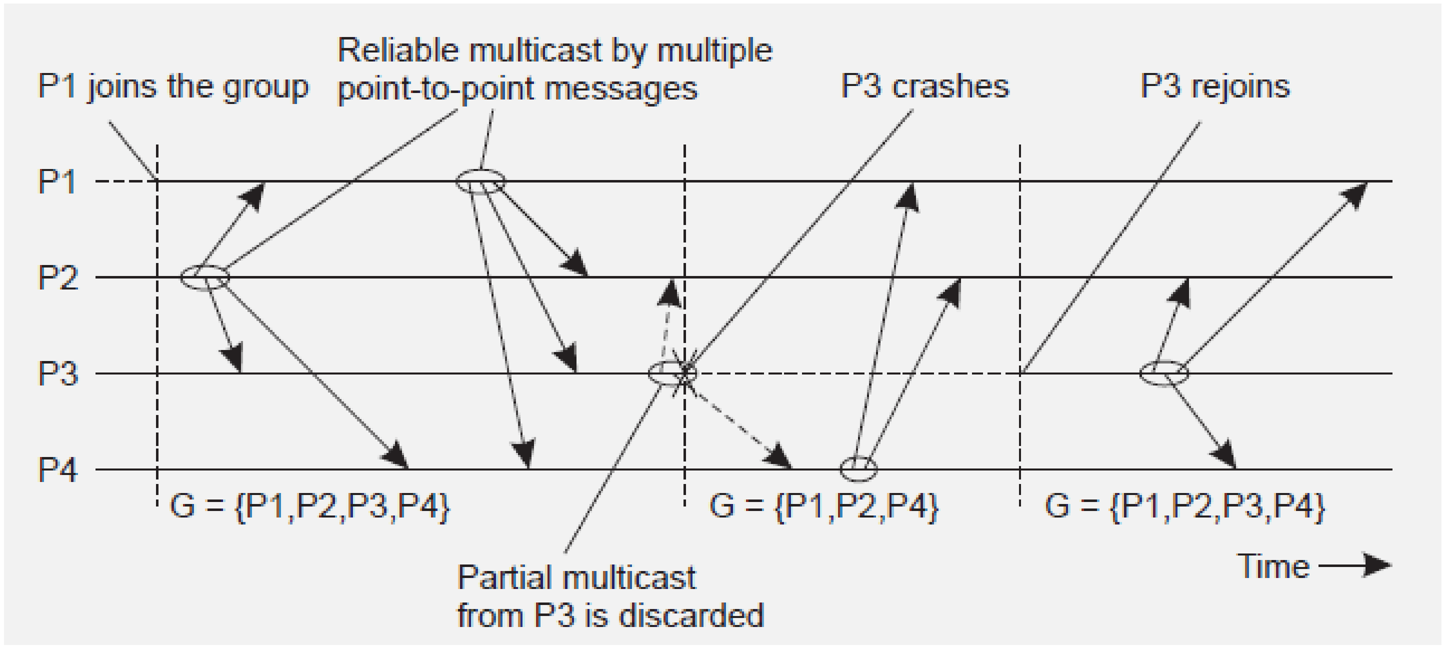
Atomic multicast

- Formulate reliable multicasting in the presence of process failures in terms of process groups and changes to group membership.



Atomic multicast

- A message is delivered only to the nonfaulty members of the current group. All members should agree on the current group membership ⇒ Virtually synchronous multicast.



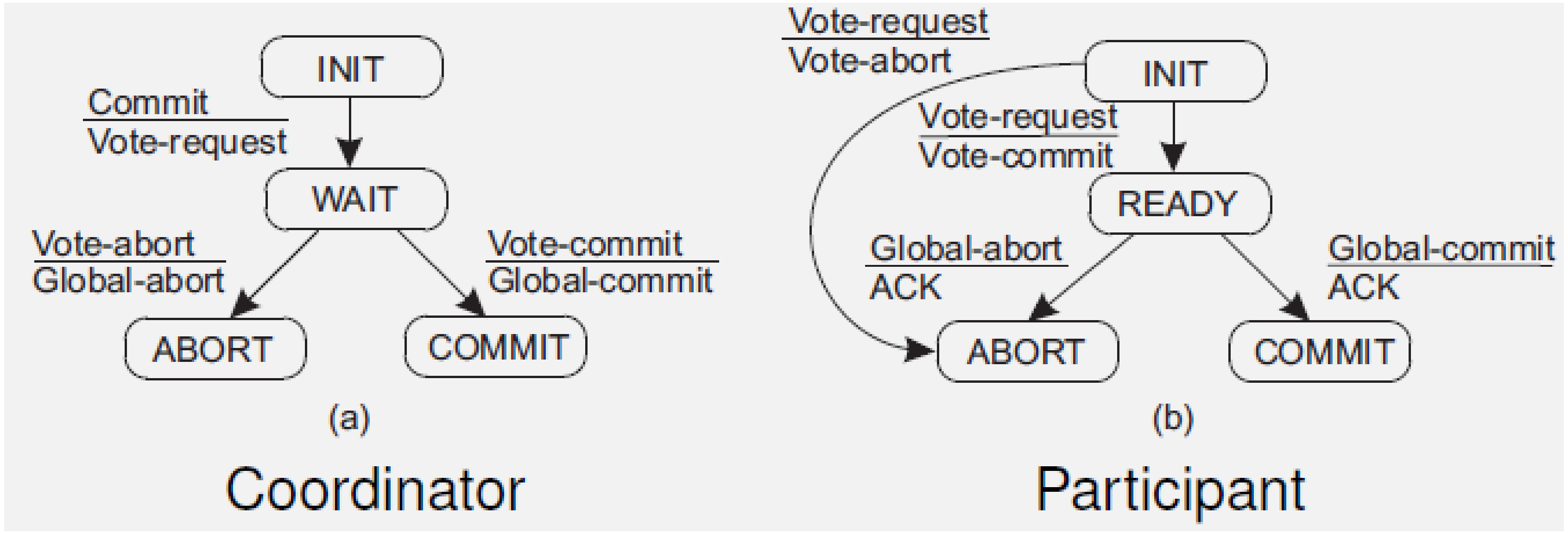
Distributed commit

- Two-phase commit
- Three-phase commit
- Essential issue
  - Given a computation distributed across a process group, how can we ensure that either all processes commit to the final result, or none of them do (atomicity)?

Two-phase commit

- The client who initiated the computation acts as coordinator; processes required to commit are the participants
  - Phase 1a: Coordinator sends vote-request to participants (also called a pre-write)
  - Phase 1b: When participant receives vote-request it returns either vote-commit or vote-abort to coordinator. If it sends vote-abort, it aborts its local computation
  - Phase 2a: Coordinator collects all votes; if all are vote-commit, it sends global-commit to all participants, otherwise it sends global-abort
  - Phase 2b: Each participant waits for global-commit or global-abort and handles accordingly.

Two-phase commit



2PC – Failing participant

- Participant crashes in state S, and recovers to S
  - Initial state: No problem: participant was unaware of protocol
  - Ready state: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make ⇒ log the coordinator's decision
  - Abort state: Merely make entry into abort state idempotent, e.g., removing the workspace of results
  - Commit state: Also make entry into commit state idempotent, e.g., copying workspace to storage.
- When distributed commit is required, having participants use temporary workspaces to keep their results allows for simple recovery in the presence of failures.

2PC – Failing participant

- When a recovery is needed to READY state, check state of other participants ⇒ no need to log coordinator’s decision.
- Recovering participant P contacts another participant Q

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

- If all participants are in the READY state, the protocol blocks. Apparently, the coordinator is failing. Note: The protocol prescribes that we need the decision from the coordinator.

2PC – Failing participant

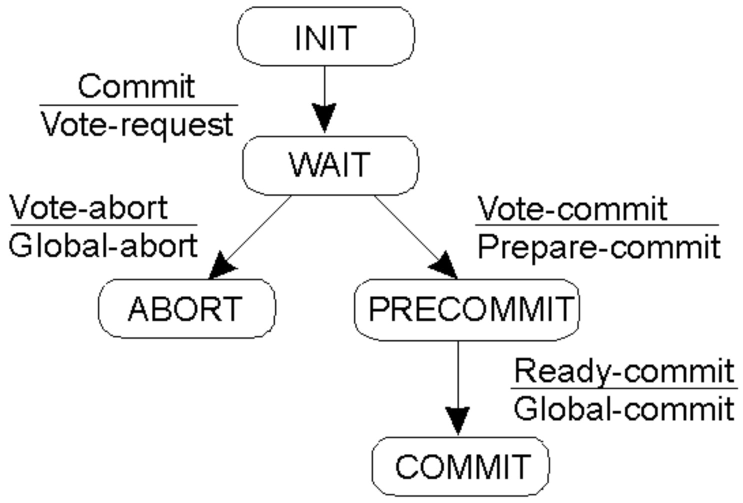
- The real problem lies in the fact that the coordinator’s final decision may not be available for some time (or actually lost).
- Let a participant P in the READY state timeout when it hasn’t received the coordinator’s decision; P tries to find out what other participants know (as discussed).
- Essence of the problem is that a recovering participant cannot make a local decision: it is dependent on other (possibly failed) processes

Three-Phase Commit

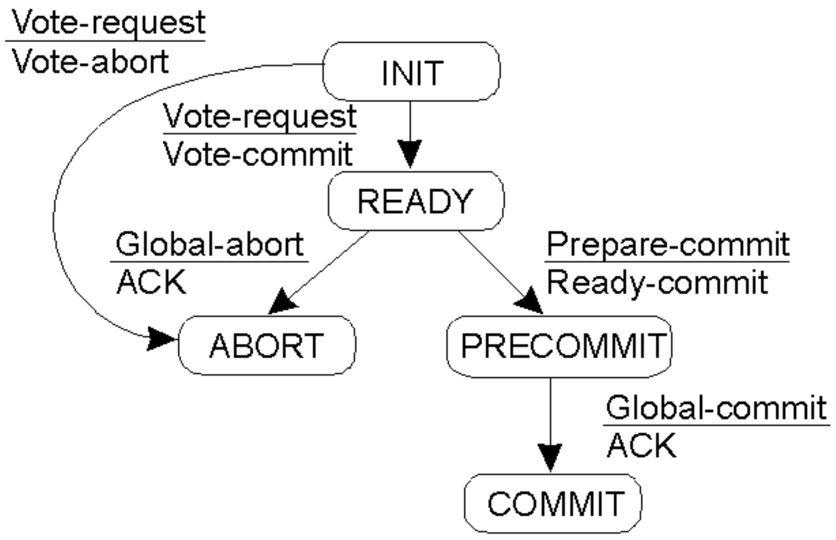
- The states of the coordinator and each participant satisfy the following two conditions:

1. There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.
2. There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made.

Three-Phase Commit



(a)



(b)

Recovery

1. Introduction

- Recovery:

  A process where a failure happened can recover to a correct state.

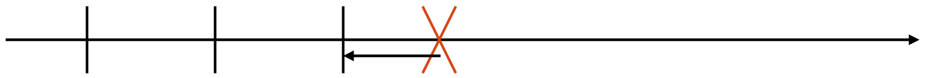
- What do we need for recovery?

  record states of a distributed system

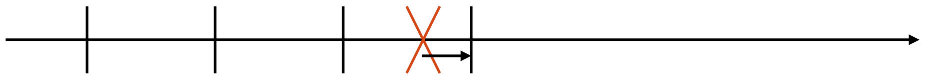
  when and how?

- Two forms of error recovery

backward recovery



forward recovery

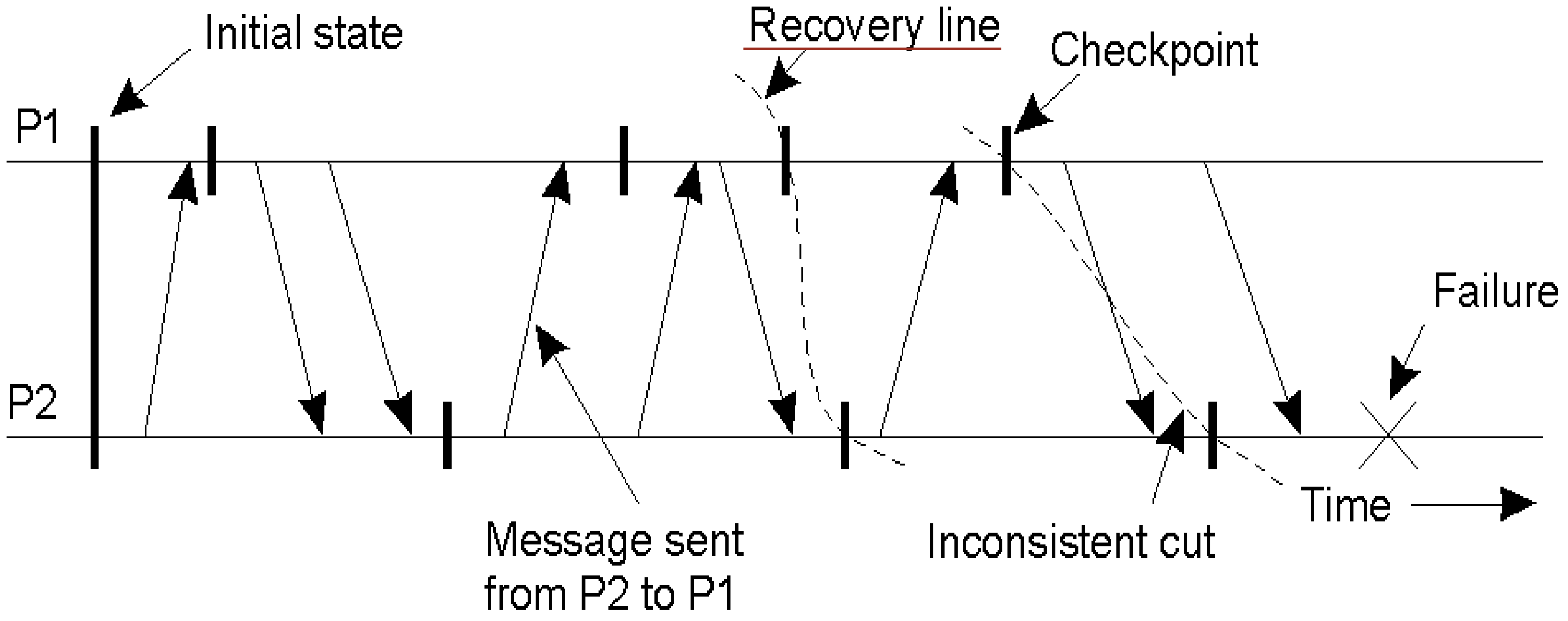


For example, reliable communication

a packet is lost $\xrightarrow{\hspace{2cm}}$ retransmission

Checkpointing

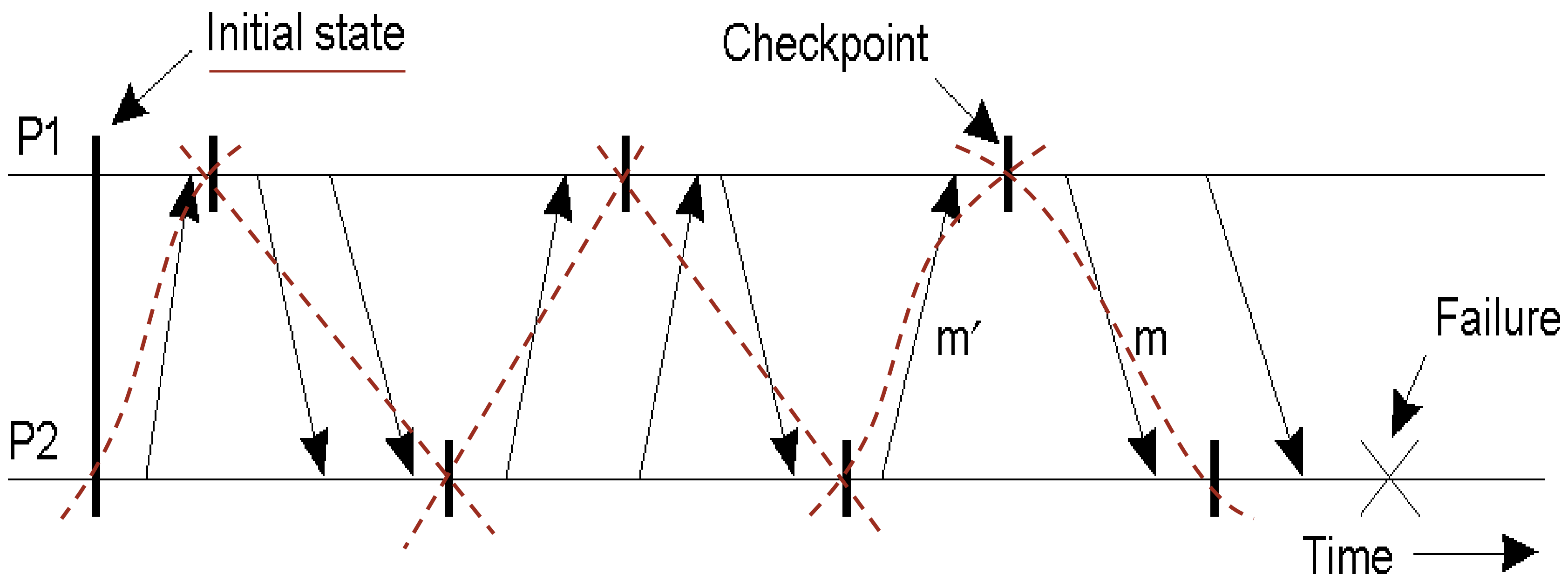
- System regularly saves its state onto stable storage



A recovery line.

- Recovery

  construct a consistent global state from local states.



To recover to most recently saved state, it requires that all processes coordinate checkpointing.

1) Independent checkpointing

- processes take local checkpoints independent of each other
- dependencies are recorded in such a way that processes can jointly roll back to a consistent global state

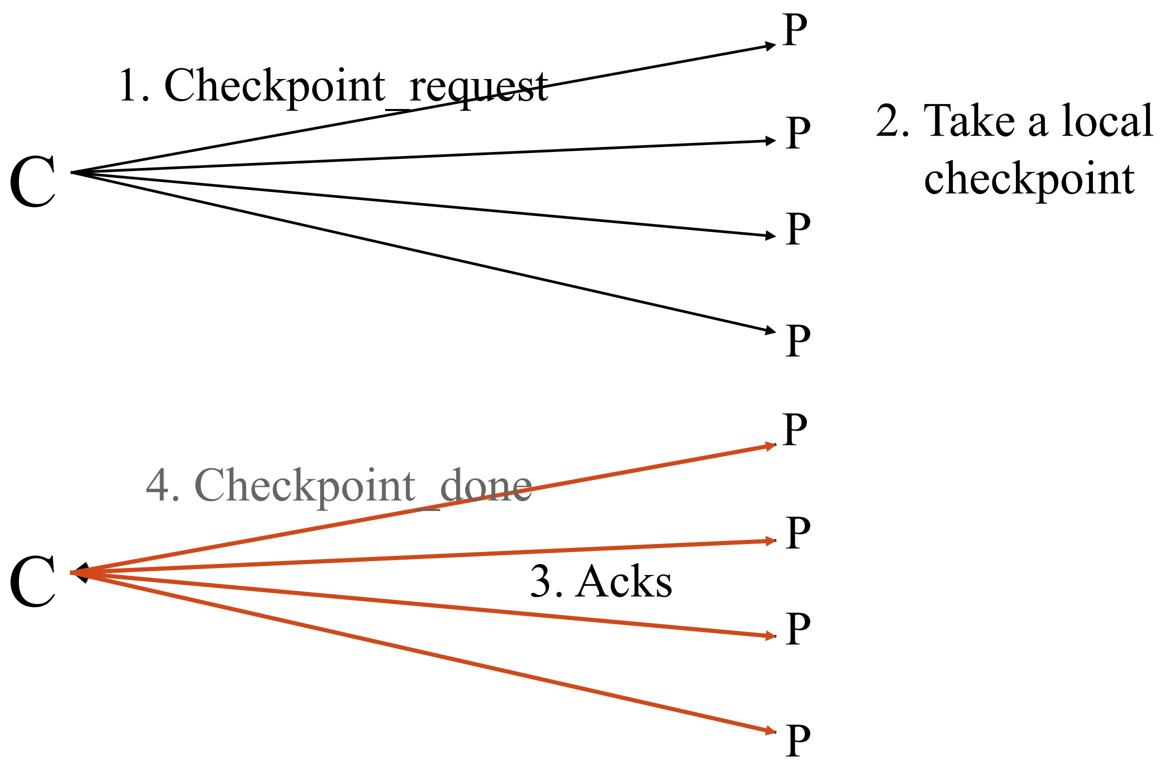
2) Coordinated checkpointing

All processes synchronize to jointly write their state to local stable storage which form a global consistent state.

Two algorithms:

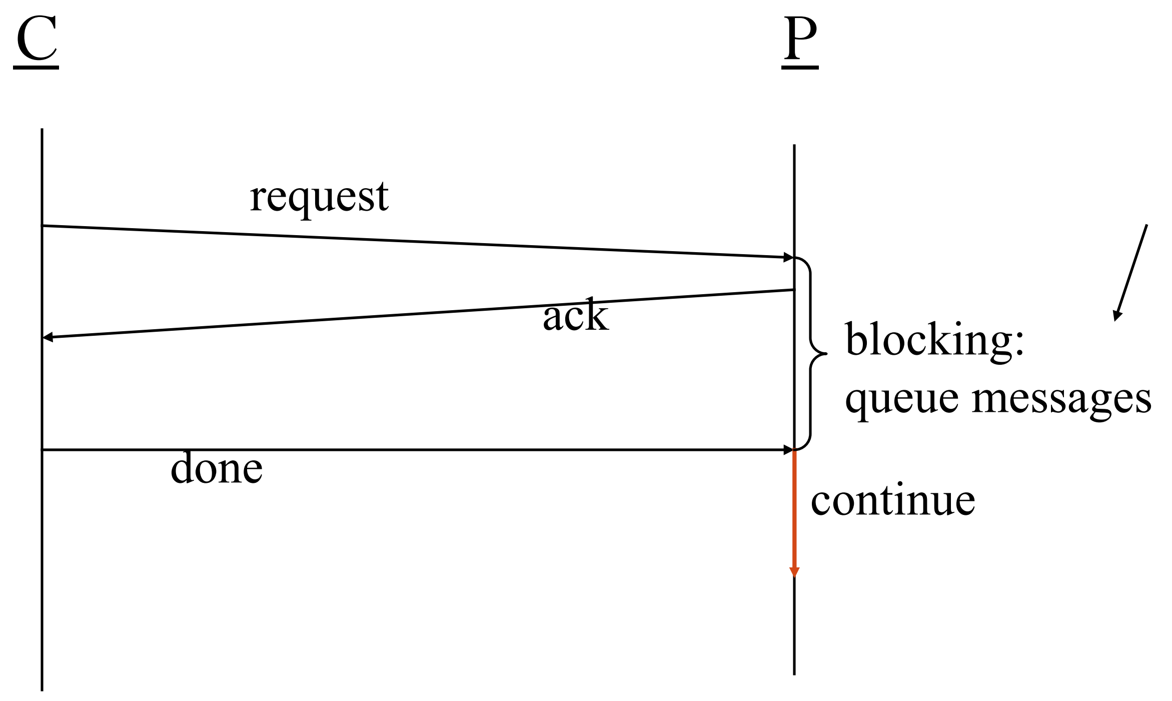
- distributed snapshot algorithm --- nonblocking one
- two-phase blocking protocol

Simple two-phase blocking protocol



Algorithm description

- A coordinator multicasts a Checkpoint_request message to all processes
- when a process receives such a message, it takes a local checkpoint, queue any subsequent message handed to it by the application it is executing, and acknowledges to the coordinator
- when the coordinator has received all acks, it multicasts a Checkpoint_done message to allow the (blocked) processes to continue



Explain that this approach will lead to a globally consistent state

Summary

Fault tolerance

- introduction

  category: component, system(fail-silent, Byzantine fault)

  K fault tolerant

Different from single machine system:

need record a consistent global state --- distributed snapshot

- Approaches

  redundancy

- Agreement in faulty system
  - Two-army problem
  - Byzantine generals problem