Fault Tolerance

- Fault tolerance is concerned with being able to provide correct services, even in the presence of faults.
- includes
  - preventing faults and failures from affecting other components of the system,
  - automatically recovering from partial failures, and doing so without seriously affecting performance.

Dependability

Dependability is the ability to avoid service failures that are more frequent.

A key requirement of most systems is to provide some level of dependability. A dependable systems has the following properties.

- Availability: system is ready to be used immediately
- Reliability: system can run continuously without failure
- Safety: when a system (temporarily) fails to operate correctly, nothing disastrous happens
- Maintainability: how easily a failed system can be repaired (sometimes, without its users noticing the failure).

Building a dependable system comes down to controlling failure and faults.

Cont.

- Dependability implies the following:
  1. Availability
  2. Reliability
  3. Safety
  4. Maintainability

Cont.

- *Availability* – the system is ready to be used immediately.
- *Reliability* – the system can run continuously without failure.
- *Safety* – if a system fails, nothing catastrophic will happen.
- *Maintainability* – when a system fails, it can be repaired easily and quickly (sometimes, without its users noticing the failure).

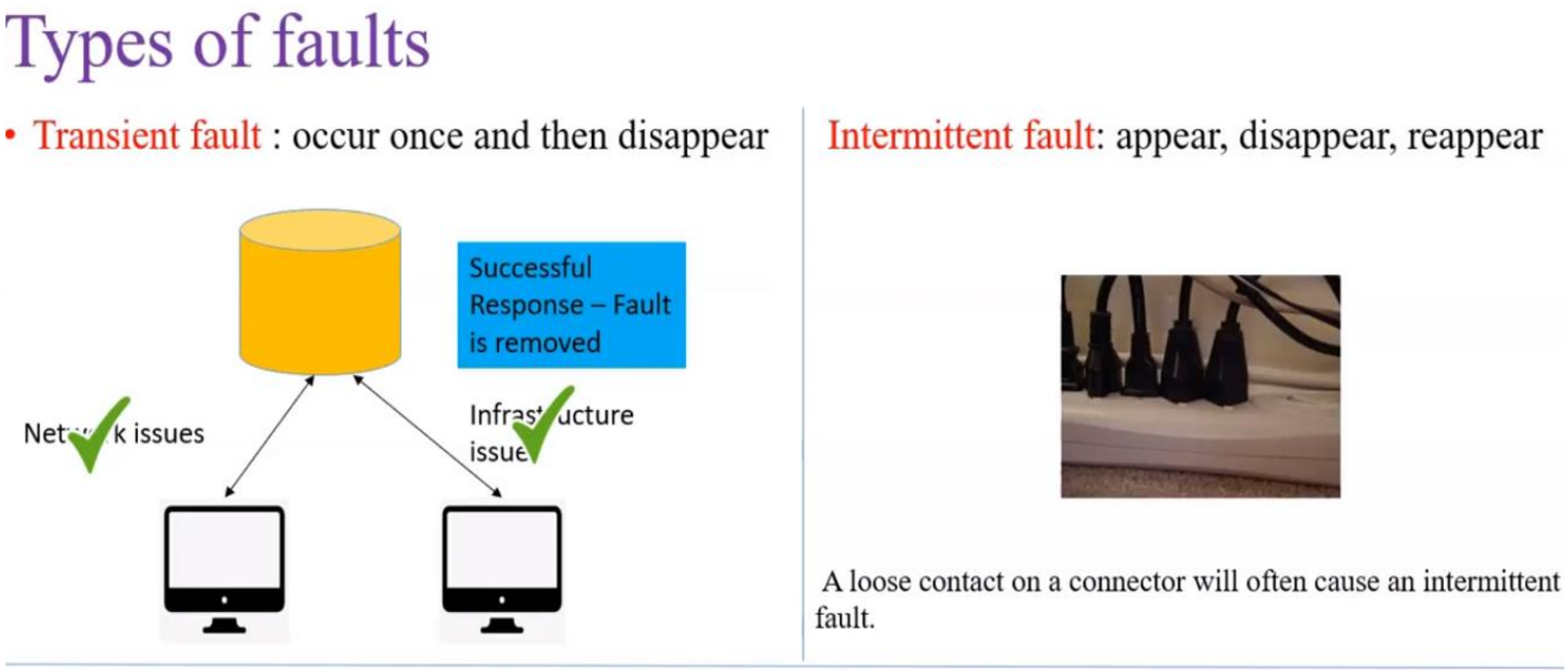
Fail/Error/Fault

- A system is said to “fail” when it *cannot meet* its promises.
- A failure is brought about by the *existence* of “errors” in the system.
- The *cause* of an error is a “fault”.

Types of Fault

- There are three main types of ‘fault’:

1. *Transient Fault*
2. *Intermittent Fault*
3. *Permanent Fault*



Types of faults

- Transient fault : occur once and then disappear

Intermittent fault: appear, disappear, reappear

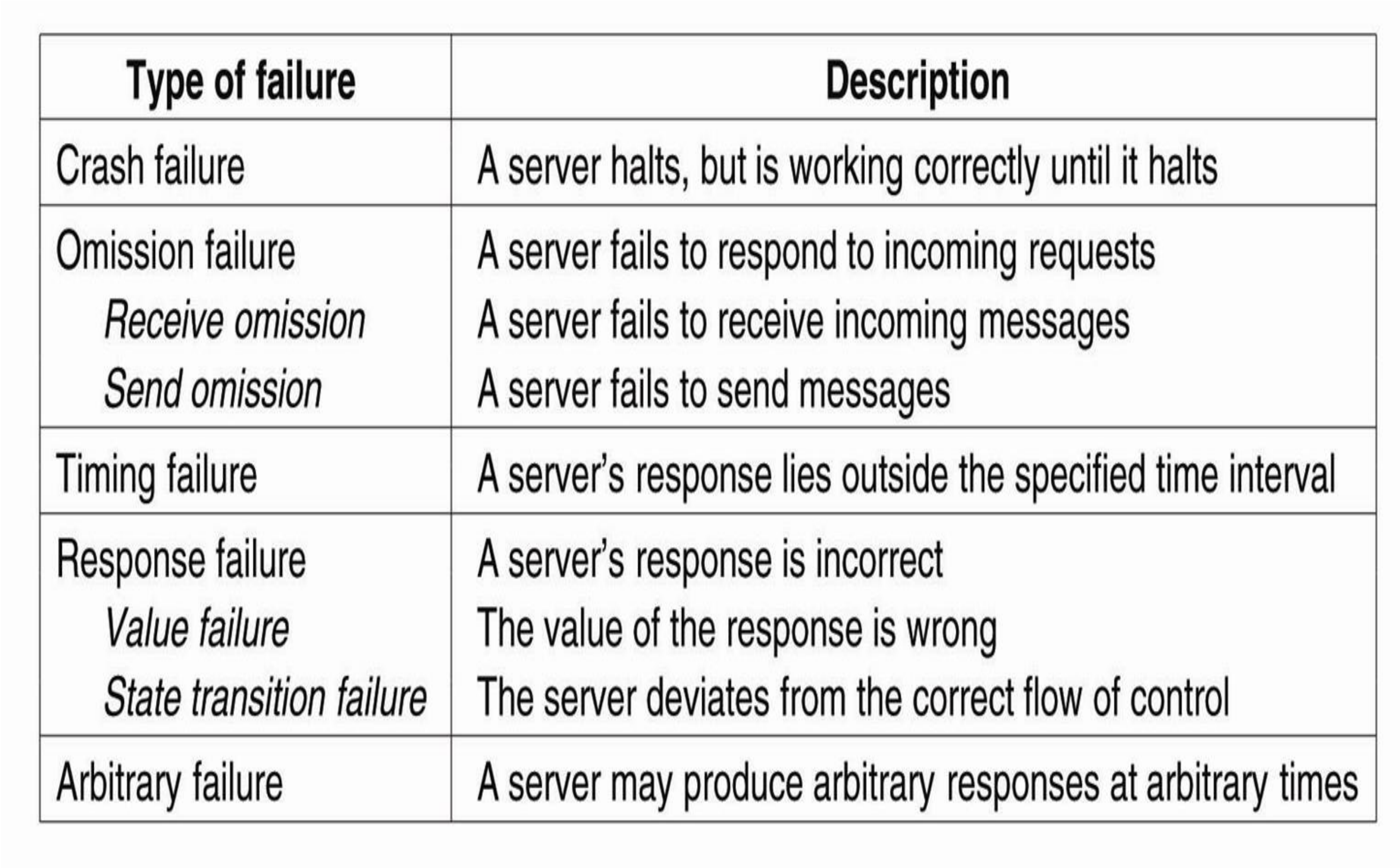
A loose contact on a connector will often cause an intermittent fault.

- *Permanent Fault* – continues until repaired

Burnt out chips

Disk head crash

Failure Models



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

Failure Masking by Redundancy

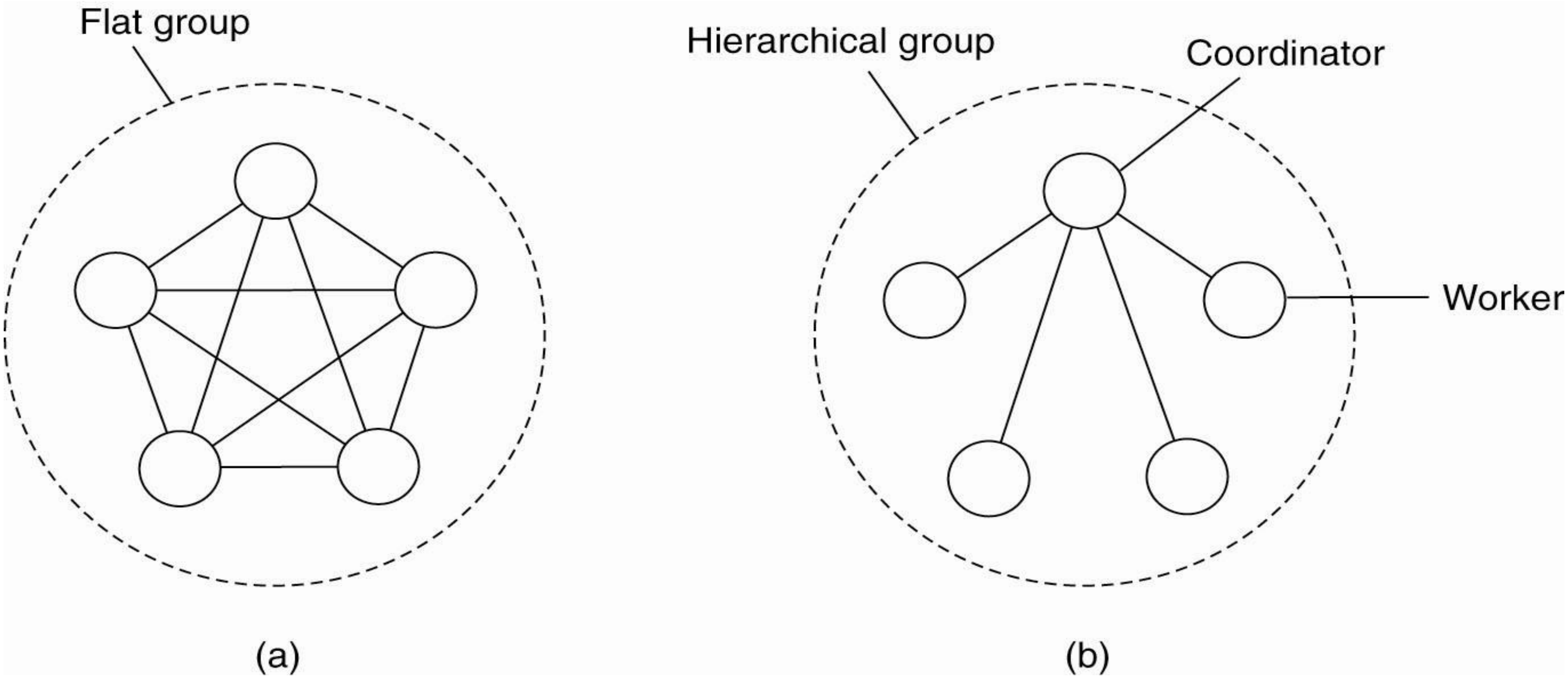
- **Strategy**: hide the occurrence of failure from other processes using *redundancy*.
- Three main types:

1. *Information Redundancy* – add extra bits to allow for error detection/recovery (e.g., Hamming codes and the like).
2. *Time Redundancy* – perform operation and, if needs be, perform it again. Think about how transactions work (BEGIN/END/COMMIT/ABORT).
3. *Physical Redundancy* – add extra (duplicate) hardware and/or software to the system.

Process Resilience

- Fault: It is an incorrect internal state in your system
- Failure: Inability of system to do its intended job
- Resilience: is about preventing Fault turning into failure
- Protection against process failures – achieved by replicating processes into groups
- Groups:
  - Organize identical processes into groups
  - Process groups are dynamic
  - Processes can be members of multiple groups
  - Mechanisms for managing groups and group membership
- Flat vs Hierarchical Groups:
  - Flat group: all decisions made collectively
    - There is no single point of failure
    - Decision making is difficult
  - Hierarchical group: coordinator makes decisions
    - Decision making process is much simpler
    - Single point of failure.

Flat Groups versus Hierarchical Groups



(a) Communication in a flat group.
(b) Communication in a simple hierarchical group.

Failure Masking and Replication

- Replicate processes and organize them into a group to replace a vulnerable process with a fault tolerant group
- There are two ways to approach such replication:

1. Primary (backup) Protocols(kind of client-server arch.):
   - primary coordinates all write operations.
   - Its role can be taken over by one of the backup, if need be.
   - When the primary crashes, the backups execute some lection algorithm to choose a new primary
2. Replicated-Write Protocols :
   - organizing identical processes into a flat group.
   - No single point of failure

Agreement in faulty systems

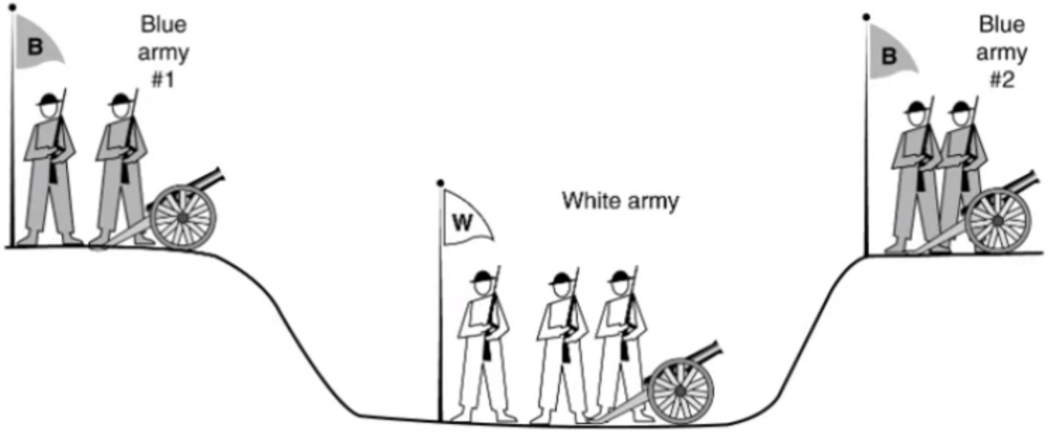
- Several sites communicating via messages, some type of trust bet then is required for this sites make some sort of agreements for mutual working.
- Examples: Election, transaction commit/abort, dividing tasks among workers, mutual exclusion
  - What happens when processes can fail?
  - What happens when communication can fail?

- We want all non-faulty processes to reach and establish agreement (within a finite number of steps)

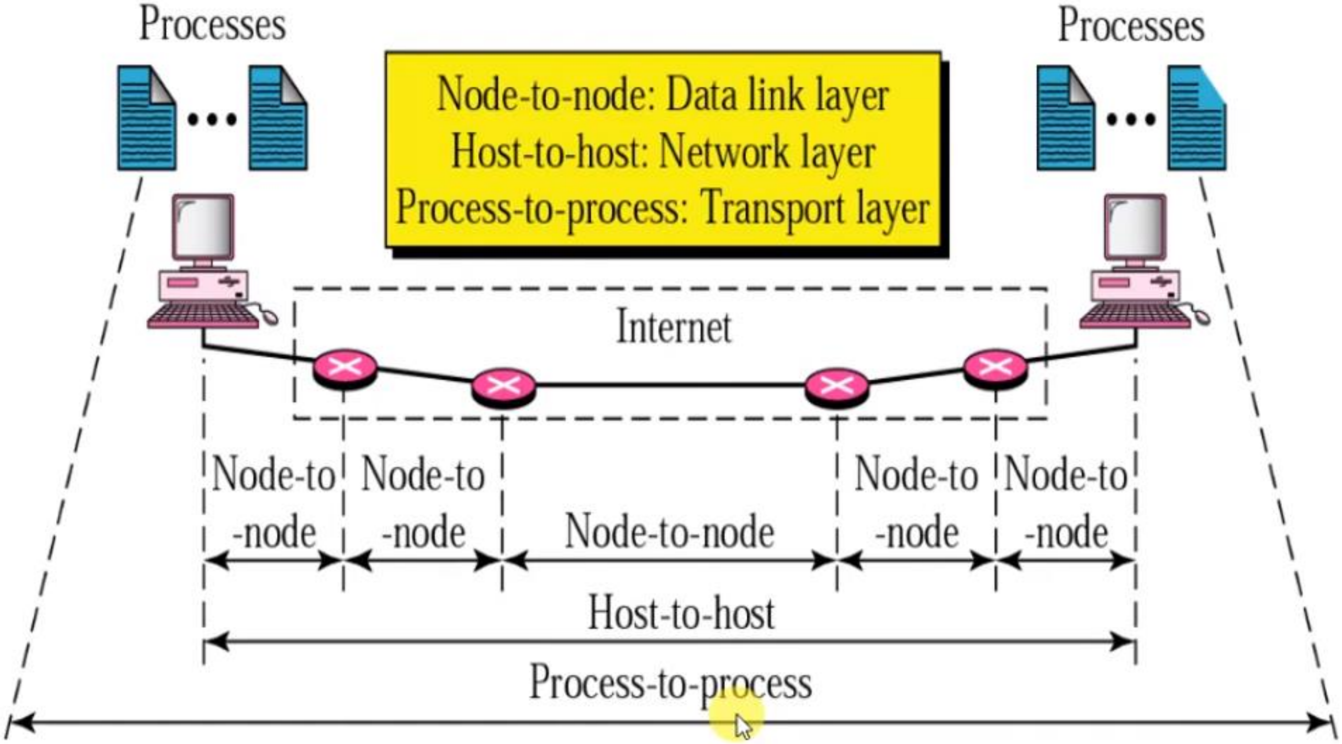
The Goal of Agreement Algorithms

- “To have all *non-faulty* processes reach consensus on some issue (quickly).”
- The **two-army problem -**with non-faulty processes, agreement between even two processes is not possible in the face of unreliable communication.
- **Byzantine generals problem** – communication is perfect but the processes are not.

The two-army problem.

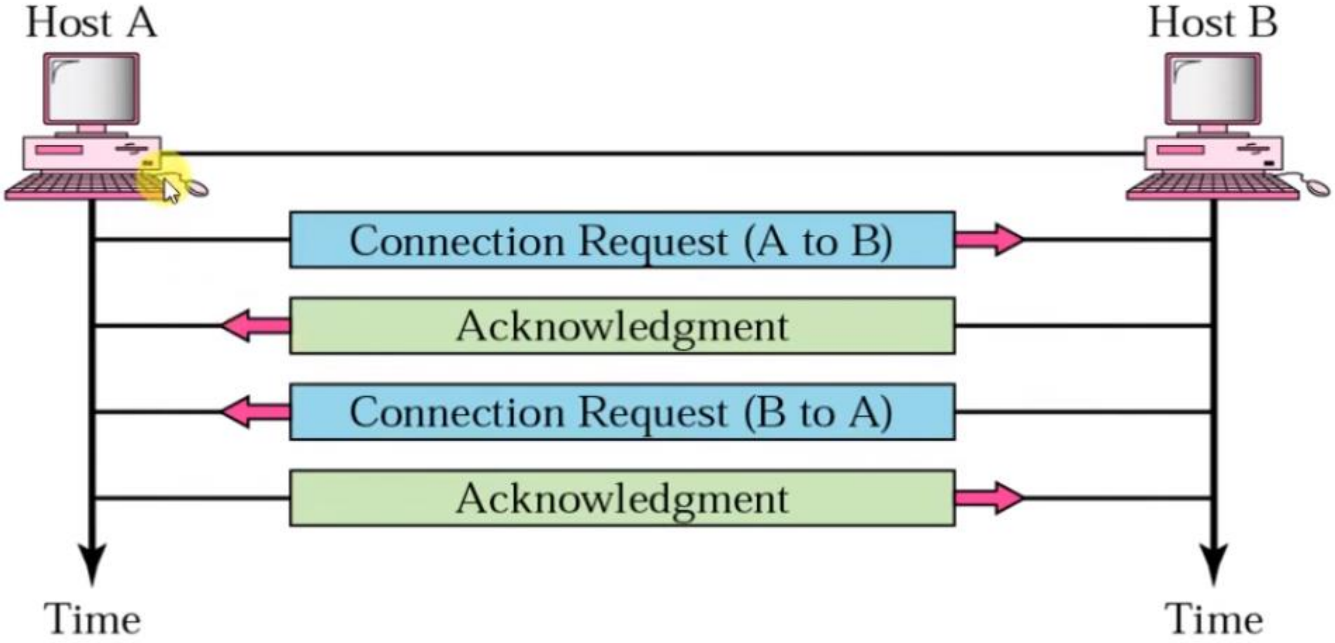


Types of data deliveries

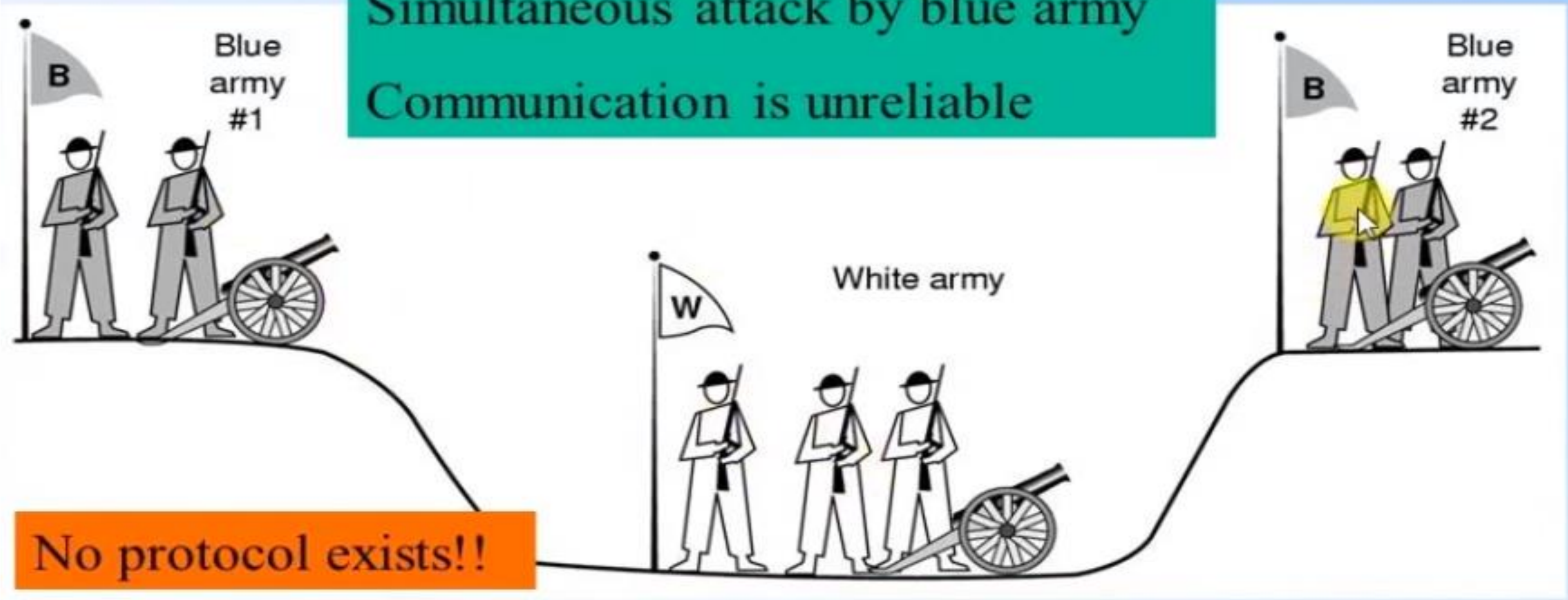


Node-to-node: Data link layer
Host-to-host: Network layer
Process-to-process: Transport layer

Connection Establishment



❑ Symmetric: two-army-problem

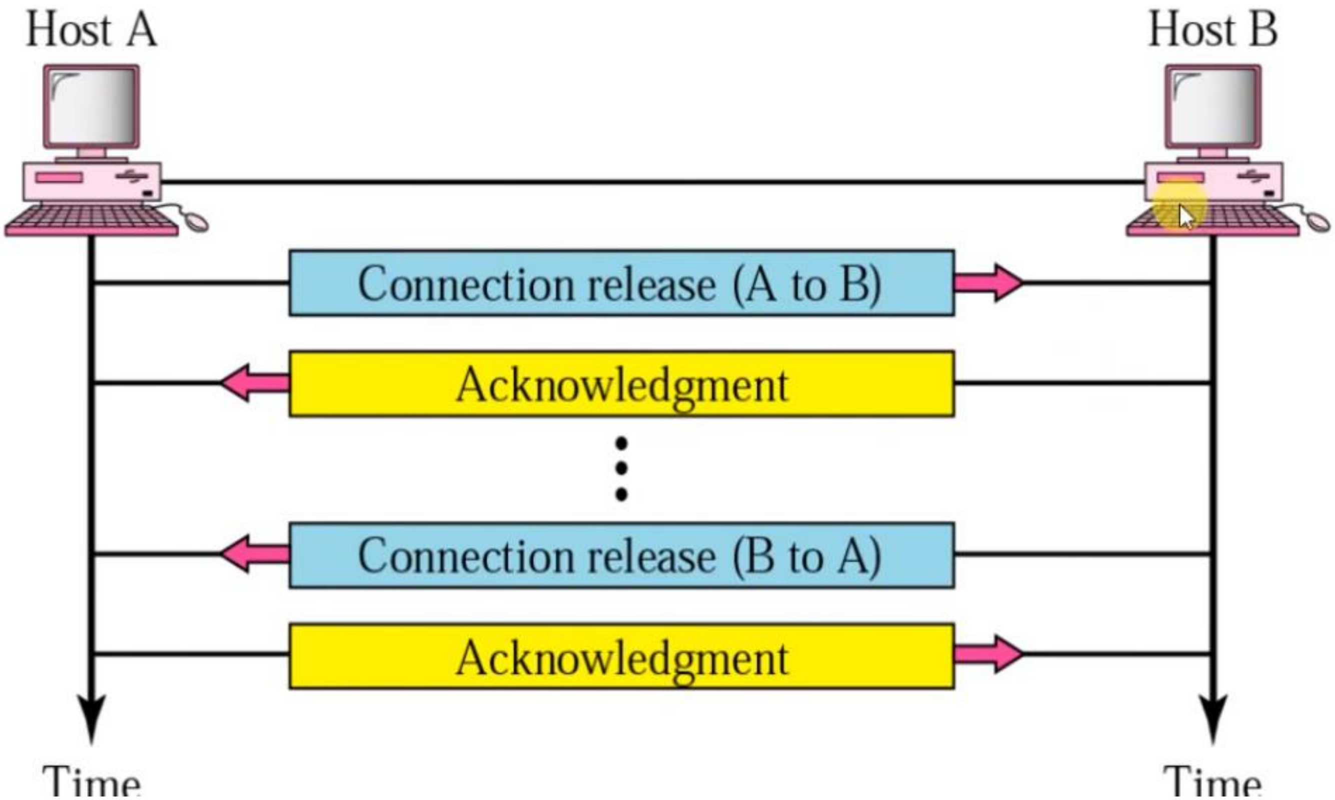


Simultaneous attack by blue army

Communication is unreliable

No protocol exists!!

Connection Release



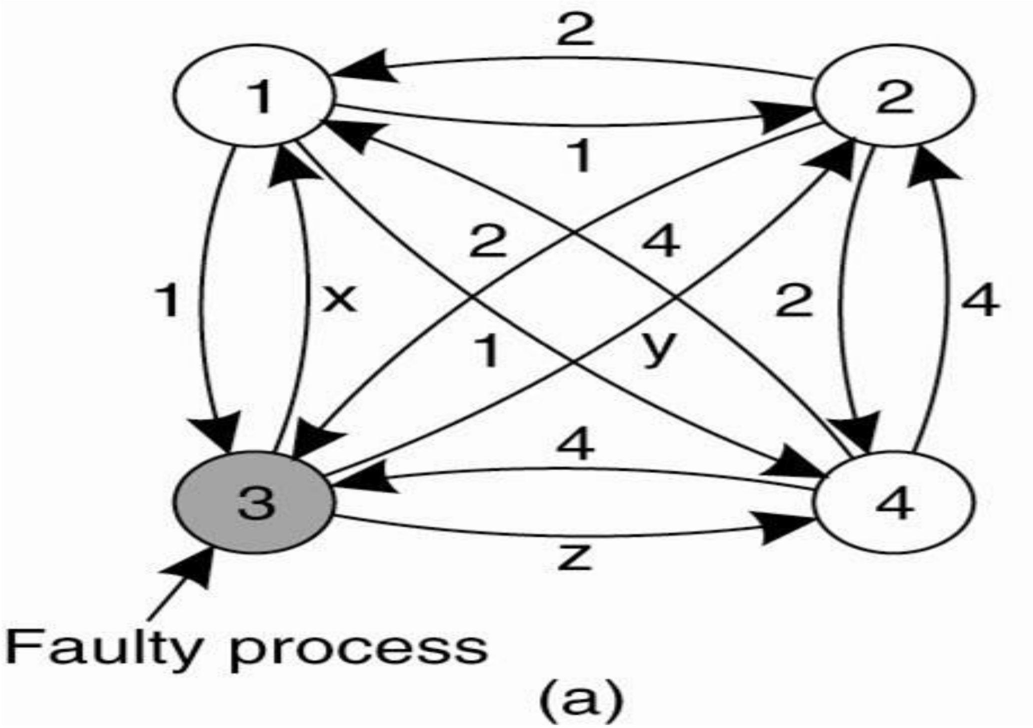
Byzantine Generals Problem

- it was typical for intentionally wrong and malicious activity to occur among the ruling group.
  A similar occurrence can surface in a DS, and is known as ‘Byzantine failure’.
- *Question*: how do we deal with such malicious group members within a distributed system?

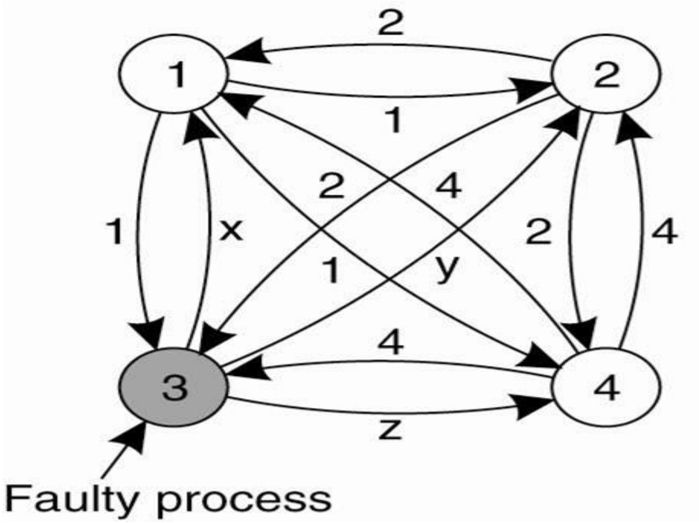
Agreement in Faulty Systems

- Recursive algorithm was devised by Lamport et. al in 1982
- Step 1: every general sends a (reliable) message to every other general announcing his troop strength
- Step 2: results of announcements of step 1 are collected together in the form of the vectors
- Step 3: every general passing vectors to other generals
- Step 4: each general examines the *i*th element of each of the newly received vectors. If no majority, corresponding element will be considered to be UNKNOWN

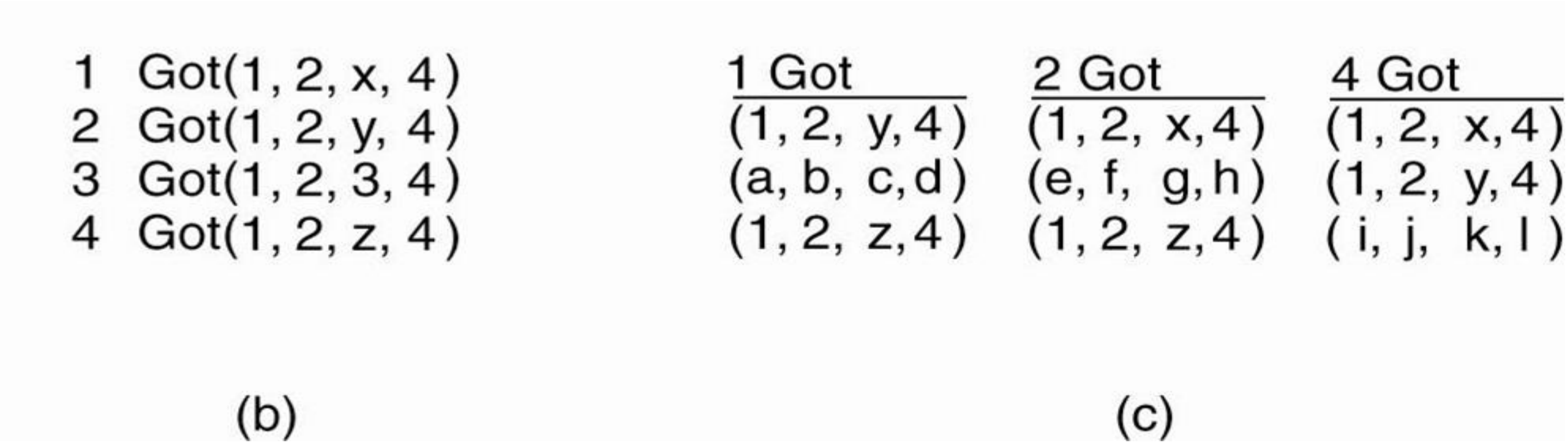
Agreement in Faulty Systems



The Byzantine agreement problem for three non-faulty and one faulty process. (a) Each process sends their value to the others.



Agreement in Faulty Systems



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)

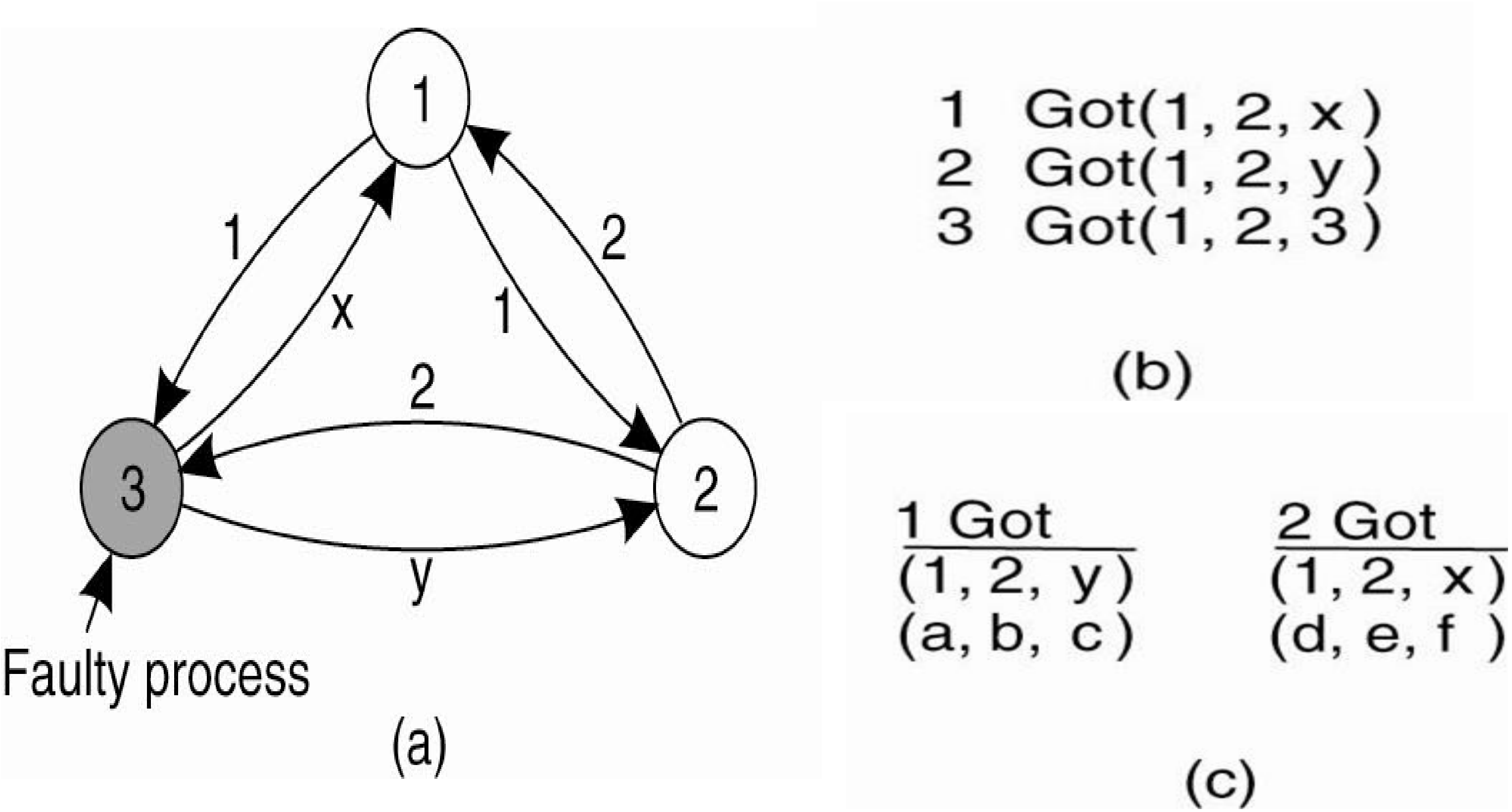
(c)

The Byzantine agreement problem for three non-faulty and one faulty process.

(b) The vectors that each process assembles based on (a).

(c) The vectors that each process receives in step 3.

Agreement in Faulty Systems



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 before, except now with two correct process and one faulty process

Reliable Client/Server Communications

- In addition to process failure a communication channel may exhibit crash, omission, timing, and arbitrary failures.
- Focus is on masking *crash* and *omission* failures.
- Example: In DS, reliable point-to-point communication is established using TCP which masks omission failures by guarding against lost messages using ACKs and retransmissions.
- It performs poorly when a crash occurs (DS may try to mask a TCP crash by automatically re-establishing the lost connection).

RPC Semantics and Failures

- The RPC mechanism works well as long as both the client and server function perfectly.
- Five classes of RPC failure can be identified:

1. The client cannot locate the server
2. The client’s request to the server is lost
3. The server crashes after receiving the request
4. The server’s reply is lost on its way to the client
5. The client crashes after sending its request

The Five Classes of Failure

(1) Client cannot locate the server

- An appropriate exception handling mechanism can deal with a missing server.
- Assume the server is done and client is requesting for the website then instead of loading infinitely. We show an exception stating page cannot be reached.

(2) Lost request message

- It can be dealt with easily using timeouts.
- If no ACK arrives in time, the message is resent.
- Server needs to be able to deal with the possibility of duplicate requests.

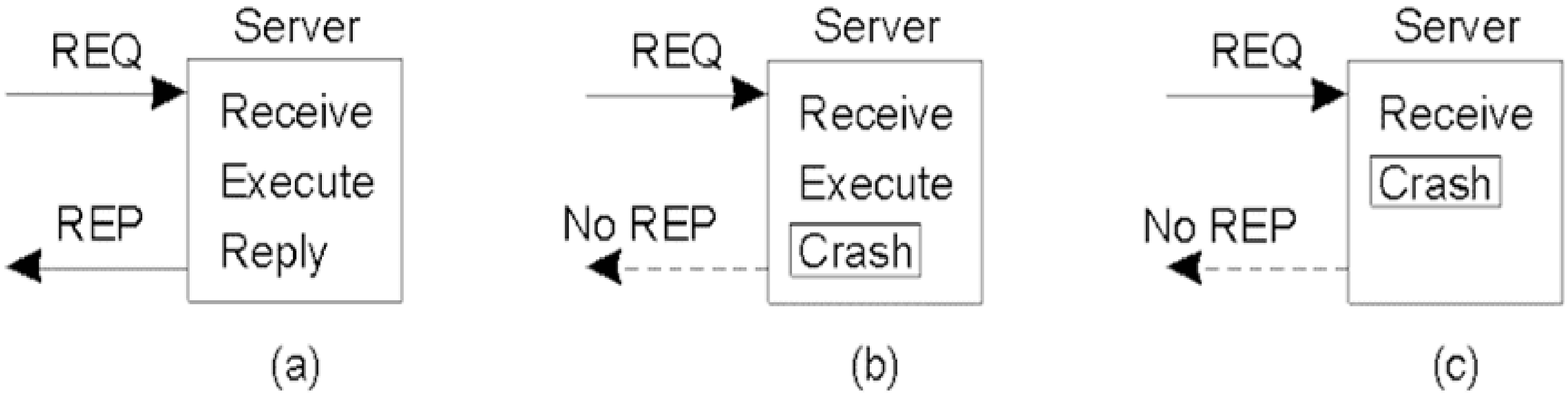
The Five Classes of Failure

(3) Server crashes

a) The normal case(success)

b) Crash *after* service execution.

c) Crash *before* service execution.



The Five Classes of Failure

- Server crashes are dealt with by implementing one of three possible implementation philosophies:
  1. *At least once semantics*: keep trying until a reply is received. Guarantee is given that the RPC occurred at least once, but (also) possibly more that once.
  2. *At most once semantics*: gives up immediately and reports back failure. Guarantee is given that the RPC occurred at most once, but possibly not at all.
  3. *No semantics*: When a server crashes, client gets no indication. Nothing is guaranteed, and client and servers take their chances. Easy to implement
- It has proved difficult to provide *exactly once semantics*.

Server Crashes

- Remote operation: print some text and (when done) send a completion message.
- Three events that can happen at the server:

1. Send the completion message (M)
2. Print the text (P)
3. Crash (C)

Server Crashes

- These three events can occur in six different orderings:

1. M →P →C: A crash occurs after sending the completion message and printing the text.
2. M →C (→P): A crash happens after sending the completion message, but before the text could be printed.
3. P →M →C: A crash occurs after sending the completion message and printing the text.
4. P→C(→M): The text printed, after which a crash occurs before the completion message could be sent.
5. C (→P →M): A crash happens before the server could do anything.
6. C (→M →P): A crash happens before the server could do anything.

- Parentheses indicate an event that can no longer happen because the server already crashed

The Five Classes of Failure

(4) Lost reply messages

- *Why* was there no reply? Is the server *dead*, *slow*, or did the reply just go *missing*?
- A request that can be repeated any number of times without any side-effects is said to be *idempotent*. (Example: a read of a static web-page is said to be idempotent)
- *Nonidempotent* requests (for example, the electronic transfer of funds) are a little harder to deal with.
- The obvious solution is just relay on timer again that has been set by the client operating system.
- If no reply then just send request one more time

The Five Classes of Failure

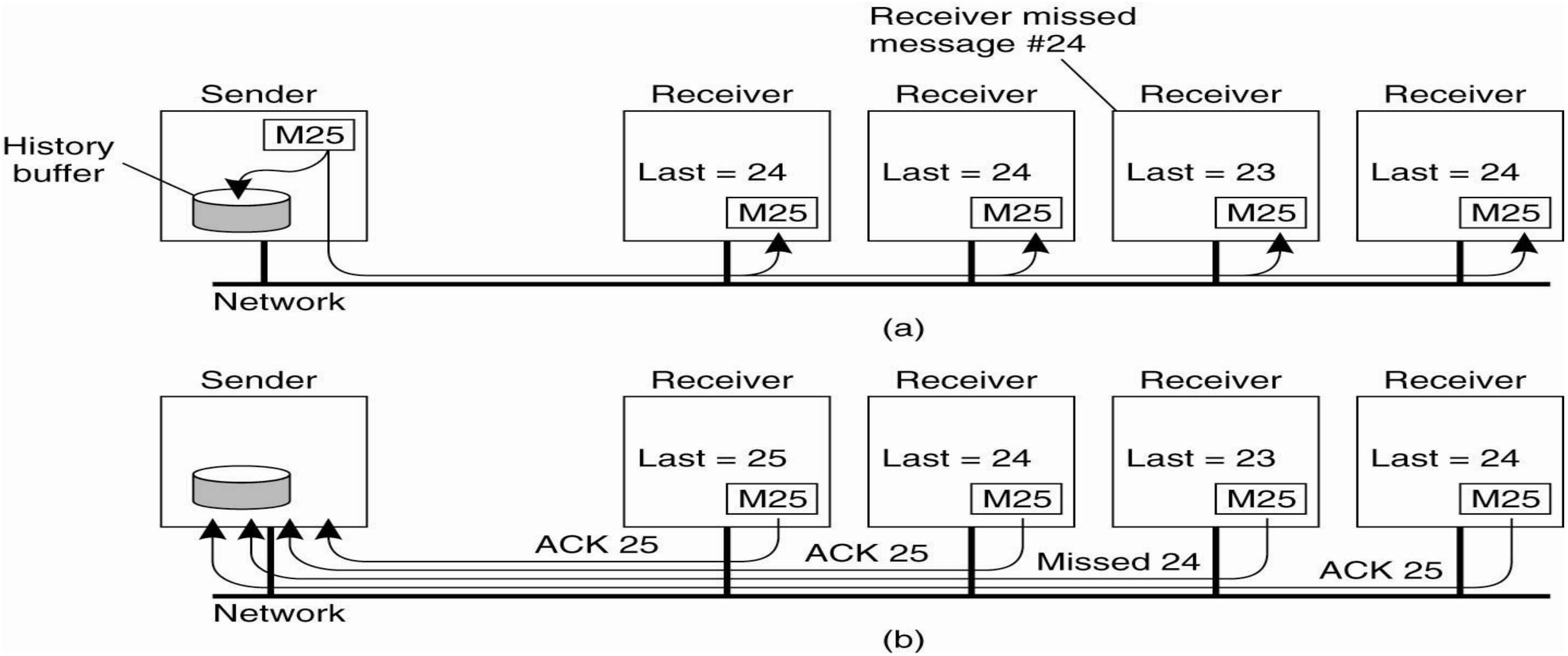
(5) Client crashes after request

- What happens if a client sends a request to a server to do some work and crashes before the server replies.
- At this point a computation is active and no parent is waiting for result hence it is called orphan
- Orphans can cause many problems like creating load on system/server/waiting CPU cycles etc

Reliable Group Communication

- Reliable multicast services guarantee that all messages are delivered to all members of a process group.
- But it is surprisingly *tricky* (as multicasting services tend to be *inherently* unreliable).
- For a small group, multiple, reliable point-to-point channels will do the job, however, such a solution *scales poorly* as the group membership grows. Also:
  - What happens if a process *joins* the group during communication?
  - Worse: what happens if the sender of the multiple, reliable point-to-point channels *crashes* half way through sending the messages?

Basic Reliable-Multicasting Schemes



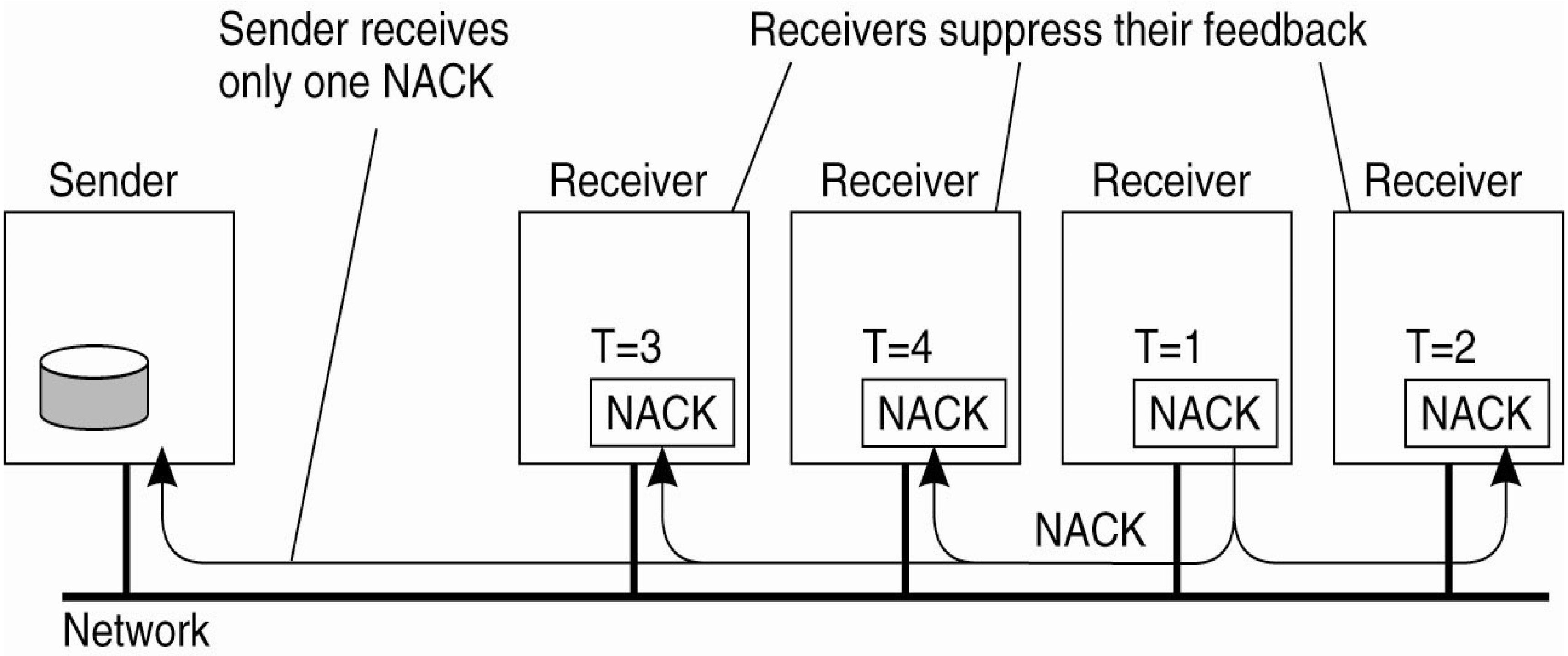
A simple solution to reliable multicasting when all receivers are known and are assumed not to fail.

(a) Message transmission. (b) Reporting feedback.

SRM: Scalable Reliable Multicasting

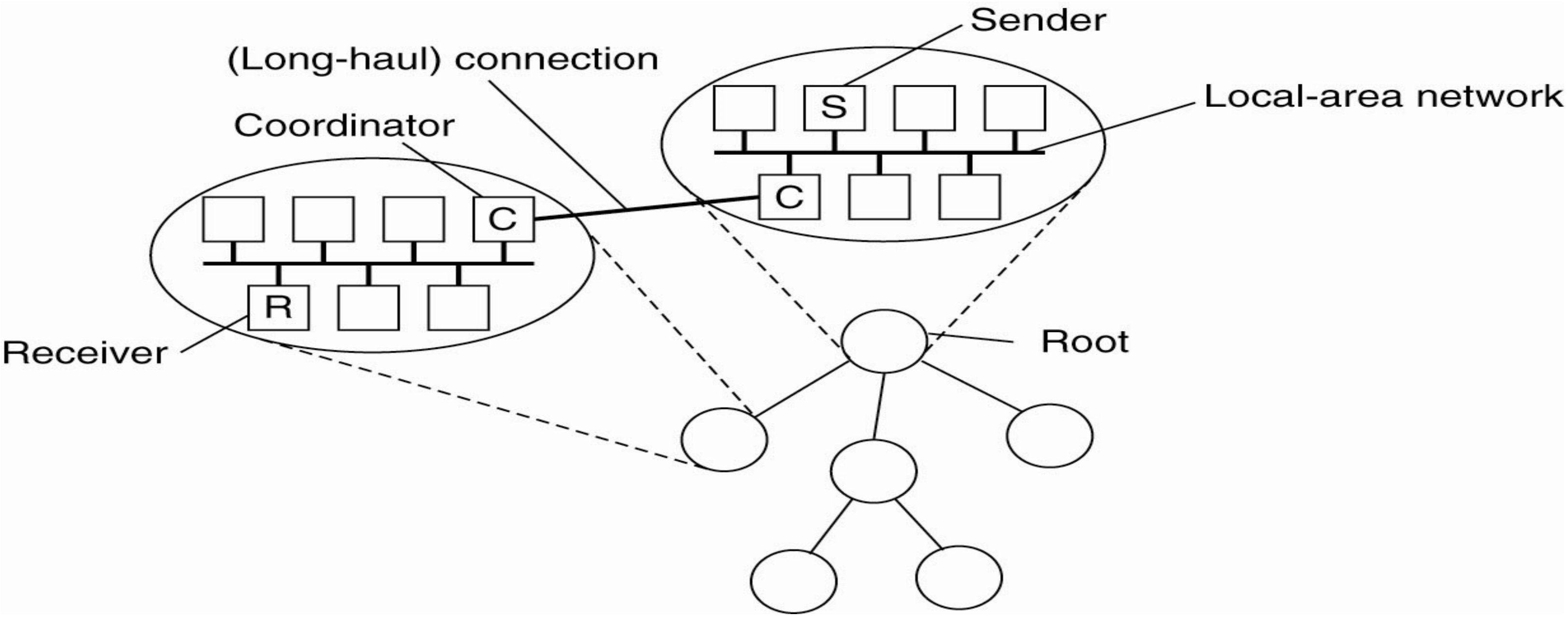
- Based on feedback suppression
- Receivers *never* acknowledge successful delivery.
- **Only missing messages are reported.**
- NACKs(negative ack) are multicast to all group members.
- This allows other members to suppress their feedback, if necessary.
- To avoid “retransmission clashes”, each member is required to wait a random delay prior to NACKing.

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.

Each local coordinator forwards the message to its children and later handles retransmission requests.

Atomic Multicast

- Reliable group communication in the face of
  - possibly faulty processes, it is useful to look at the atomic multicast problem.
- A message is delivered to either all processes, or none
- Process Group:
  - Group view: view of the group (list of processes) sender had when message sent
  - Each message uniquely associated with a group
  - All processes in group have the same view

Failure Recovery

- Recovery refers- to the process of restoring a (failed) system/process to a normal state of operation.
- Recovery can apply
  - to the complete system (involving rebooting a failed computer) or
  - to a particular application (involving restarting of failed process(es)).
- Recovery from an error is essential to fault tolerance, and error is a component of a system that could result in failure.
- The whole idea of error recovery is to replace an erroneous state with an error-free state.

Cont…

- Issues:

1) Reclamation of resources: a process may hold resources, such as locks or buffers, on a remote node.

   Naively restarting the process or its host will lead to resource leaks and possibly deadlocks.

2) Consistency: Naively restarting one part of a distributed computation will lead to a local state that is inconsistent with the rest of the computation.

   In order to achieve consistency it is, in general, necessary to undo partially completed operations on other nodes prior to restarting.

3) Efficiency: One way to avoid the above problems would be to restart the complete computation whenever one part fails.

   However, this is obviously very inefficient, as a significant amount of work may be discarded unnecessarily.

Forward error recovery(Repair and go ahead)

- If the nature of the error and damages can be completely accessed, then remove those errors in the process 's state and go forward.

Backward recovery (**Repairing is not possible so Go back)**

- If the nature of the error and damages **can not be** completely accessed, and also it is **not possible** to remove those errors, then the process should restore to the previous error – free state.

- Error recovery can be broadly divided into two categories.

1. **Backward Recovery:**

- Moving the system from its current state back into a formerly accurate condition from an incorrect one is the main challenge in backward recovery.
- It will be required to accomplish this by periodically recording the system’s state and restoring it when something goes wrong.
- A checkpoint is deemed to have been reached each time (part of) the system’s current state is noted.

2. Forward Recovery:

- Instead of returning the system to a previous, checkpointed state in this instance when it has entered an incorrect state, an effort is made to place the system in a correct new state from which it can continue to operate.
- The fundamental issue with forward error recovery techniques is that potential errors must be anticipated in advance. Only then is it feasible to change those mistakes and transfer to a new state.

These two types of possible recoveries are done in fault tolerance in distributed system.

Stable Storage :

- A pair of regular discs can be used to implement stable storage.
- Each block on drive 2 is a duplicate of the corresponding block on drive 1, with no differences.
- The block on drive 1 is updated and confirmed first whenever a block is updated, then the identical block on drive 2 is finished.

Stable Storage

- Suppose that the system crashes after drive 1 is updated but before the update on drive 2.
- Upon recovery, the disk can be compared with blocks. Since drive 1 is always updated before drive 2, the new block is copied from drive 1 to drive 2 whenever two comparable blocks differ, it is safe to believe that drive 1 is the correct one.
- Both drives will be identical once the recovery process is finished.

Checkpointing

- It is a mechanism where all the previous logs are removed from the system and stored permanently in storage disk

Checkpoint

The basic idea behind **checkpoint-recover** is the saving and restoration of the system state. By saving the current state of the system periodically, it provides the information needed for the restoration of the lost state , when the system fails.

- This checkpointing mechanism takes a snapshot of the system state and stores the data on some non-volatile storage medium.
- In the event of a system failure, the internal state of the system can be restored, and it can continue service from the point at which its state was last saved
- After the state has been restored, the system can continue normal execution

How does a rollback-recovery scheme work?

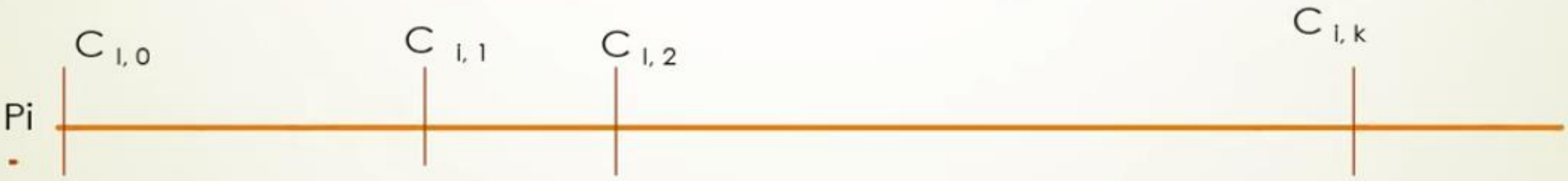
- Whenever a process rolls back to its checkpoint, it notifies all other processes also to roll back to their respective checkpoints. It then installs its checkpointed state and resumes execution.

What is a local check-point?

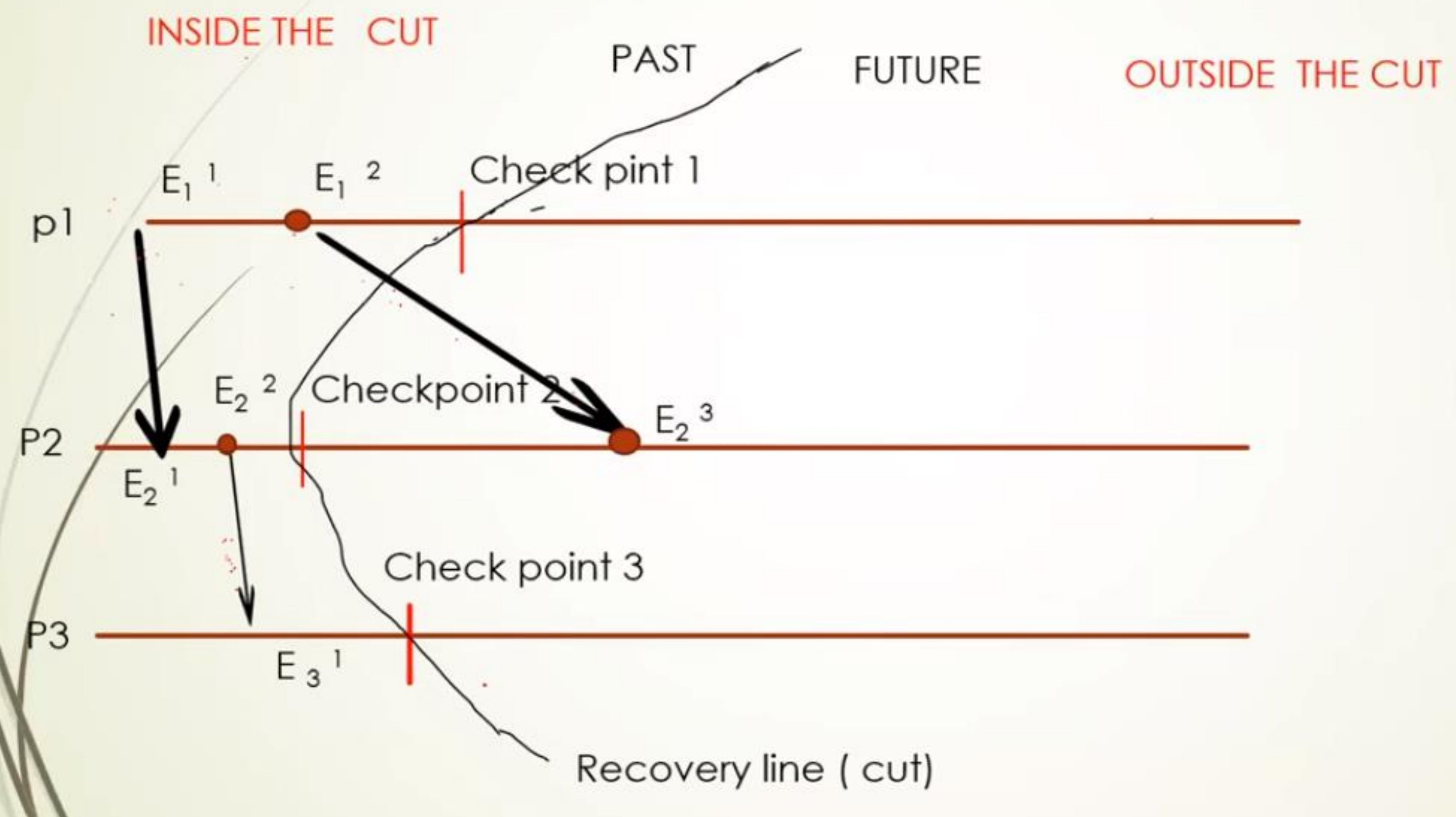
- All processes save their local states at certain instants of time . A local checkpoint is a snapshot of the state of the process at a given instance
- What is a global check point?
  - A set (or collection) of local checkpoints, one taken from each process.
- What is a global state of a distributed system?
- A collection of the individual states of all participating processes and the states of the communication channels

- Assumption of local check points
- – A process stores all local checkpoints on the stable storage
- – A process is able to roll back to any of its existing local checkpoints
- • $C_{i,0}$ – A process *Pi* takes a checkpoint $C_{i,0}$ before it starts execution
- • $C_{i, 2}$ – The 2th local checkpoint at process *Pi*
- • $C_{i, 5}$ – The 5th local checkpoint at process *Pi*
- …

$C_{i, k}$ – The kth local checkpoint at process *Pi*



Consistent checkpoints and consistent state



Consistent state

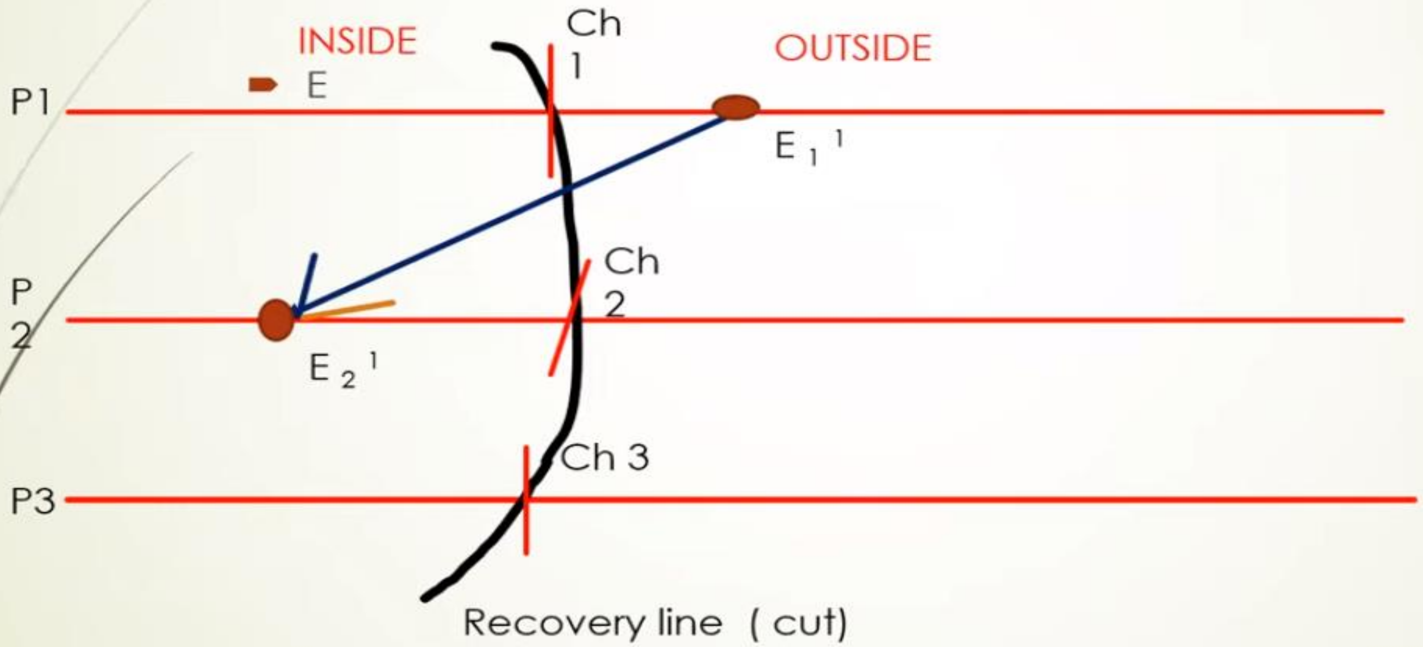
- In the above state transition diagram, note all receiving events and the corresponding sending events . for example,

- E_2^1 is the receiving event and E_1^1 is the sending event. Both are inside the cut region. Similarly,, E_3^1 is the receiving event and E_2^2 is the sending event. Both are also inside the cut region.

- E_2^3 is the receiving event and it is outside of the cut region, but the sending event E_1^3 is inside the cut region..

- Here for all receiving events, the corresponding all sending events are inside the cut region. So it is a consistent state.

Inconsistent state and cut

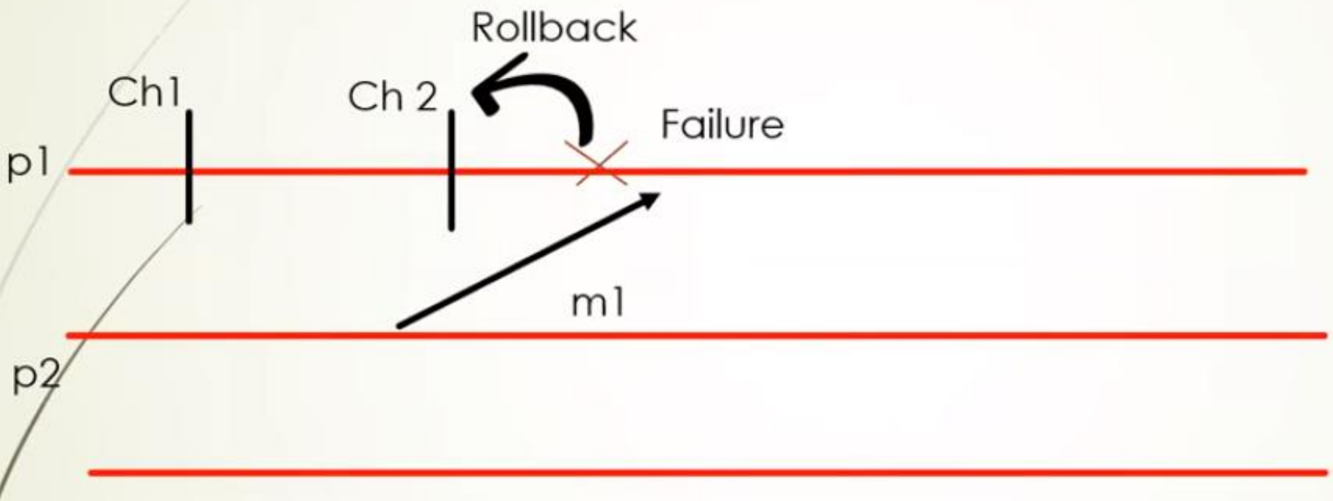


Inconsistent state

- In the above diagram, for the receiving event E_2^1, there is no sending event inside the cut so it is an inconsistent state / inconsistent checkpoints/ inconsistent cut.. i.e the event E_1^1 is outside of the cut.

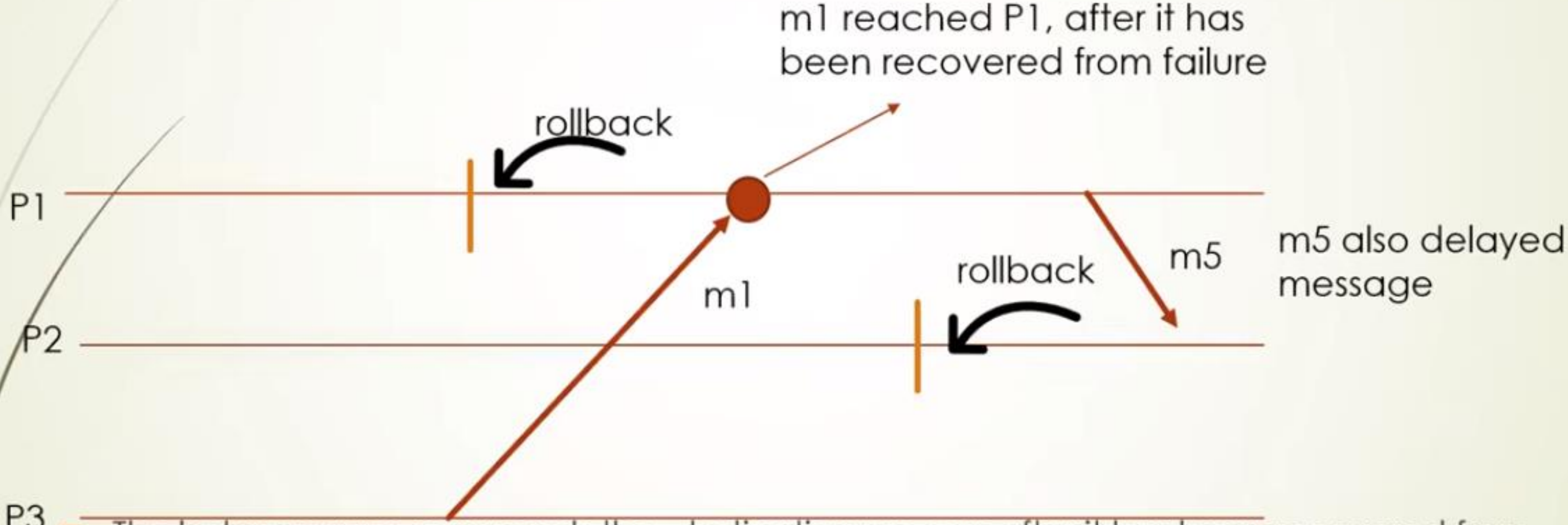
- E

Lost message



m1 is the message sent by process P2 but due to the failure in process P1, m1 could not reach P1 and so it is lost on the way. Then the process P1 has to rollback to the last checkpoint (ch2) to recover its old status to continue the task.

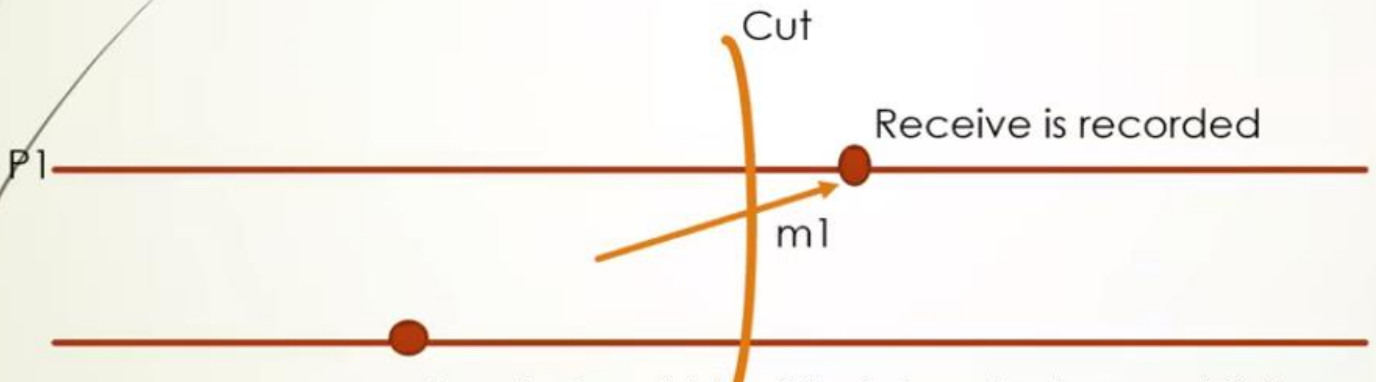
Delayed messages



- The lost messages may reach the destination process after it has been recovered from failure. In that case , these messages are said as "delayed messages " i.e messages arrived after rollback.

Orphan messages

- A Message that is received but the sender of the message has no record or proof of it. So sender can not be identified.

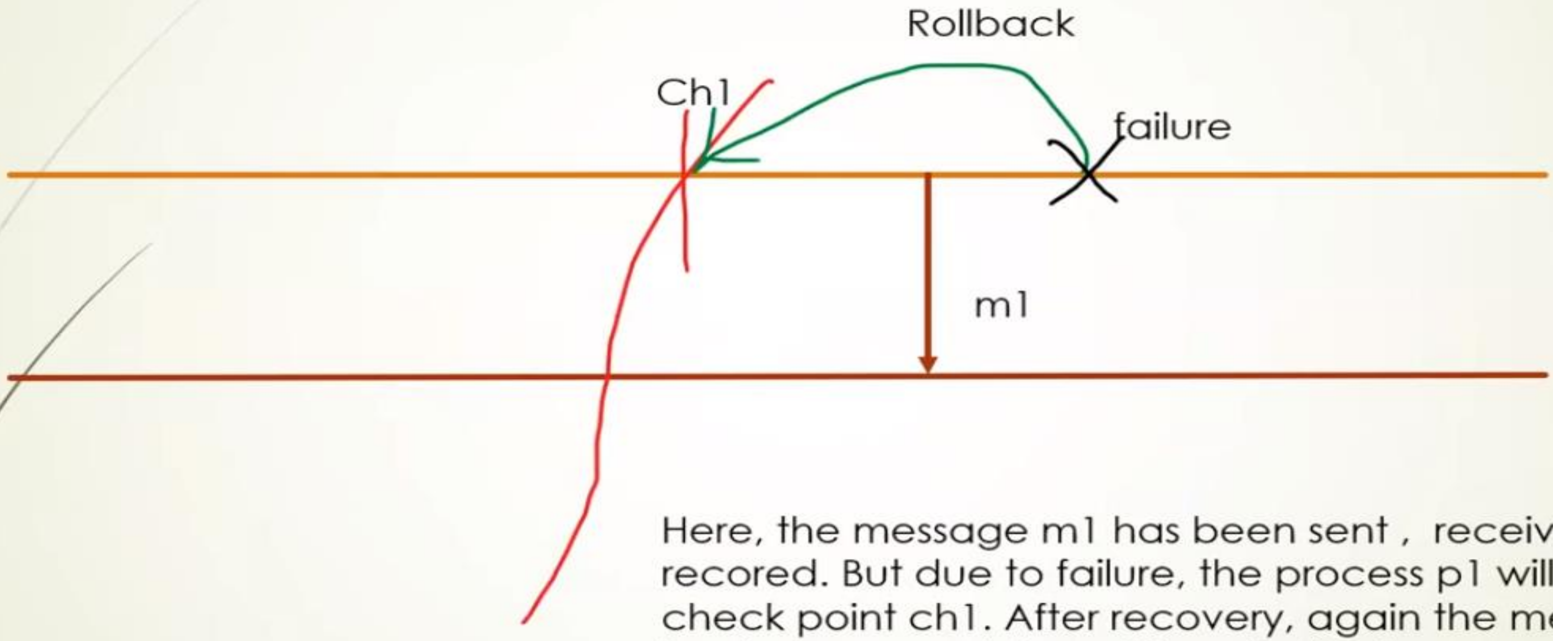


Sender is not identified. i.e who has sent it.?

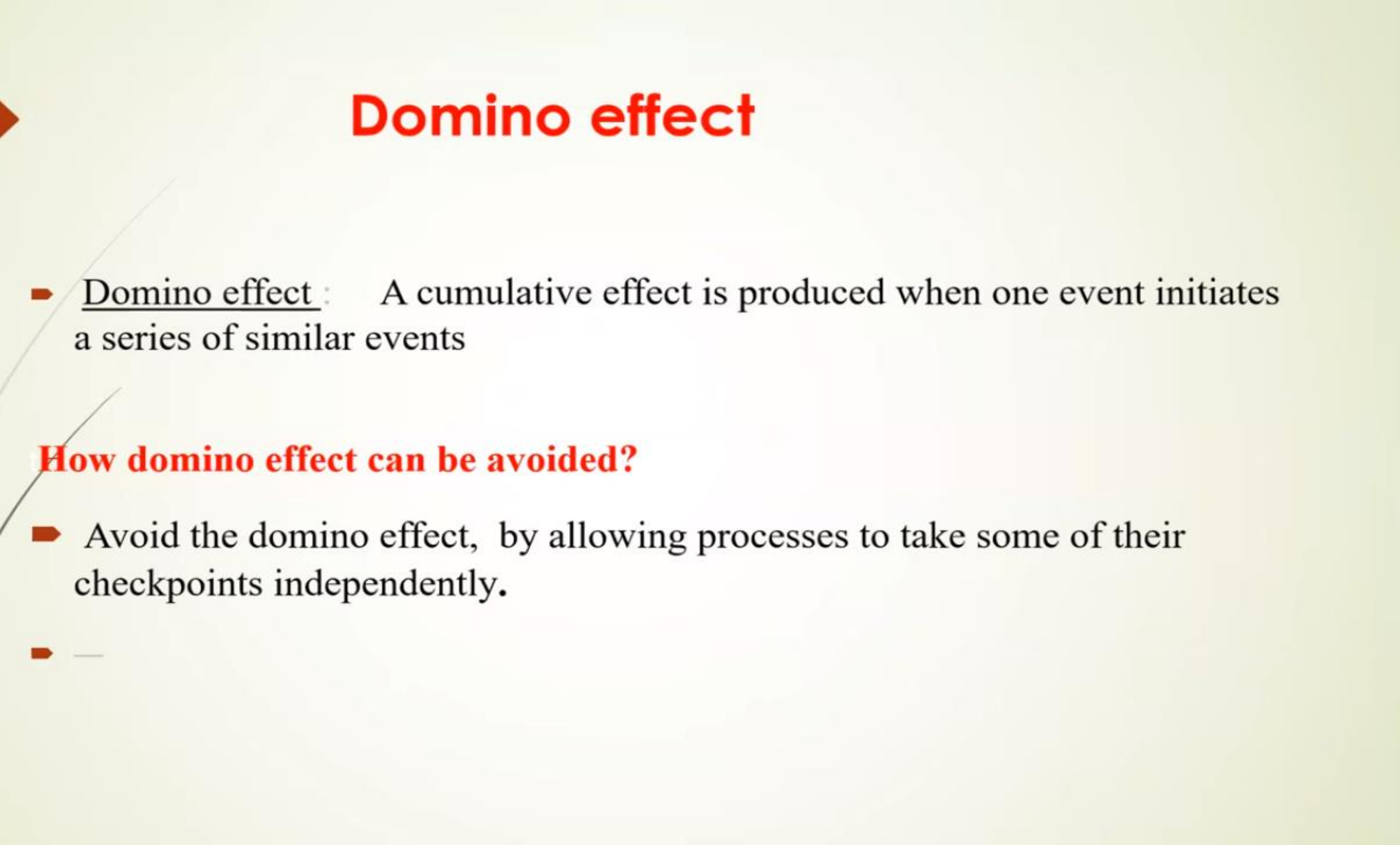
or The sender has no record of the sending of that message.

Here, m1 is orphan message

Duplicate messages



Here, the message m1 has been sent , received and recored. But due to failure, the process p1 will rollback to check point ch1. After recovery, again the message m1 has to be re-sent. So m1 is a duplicate message.



Domino effect

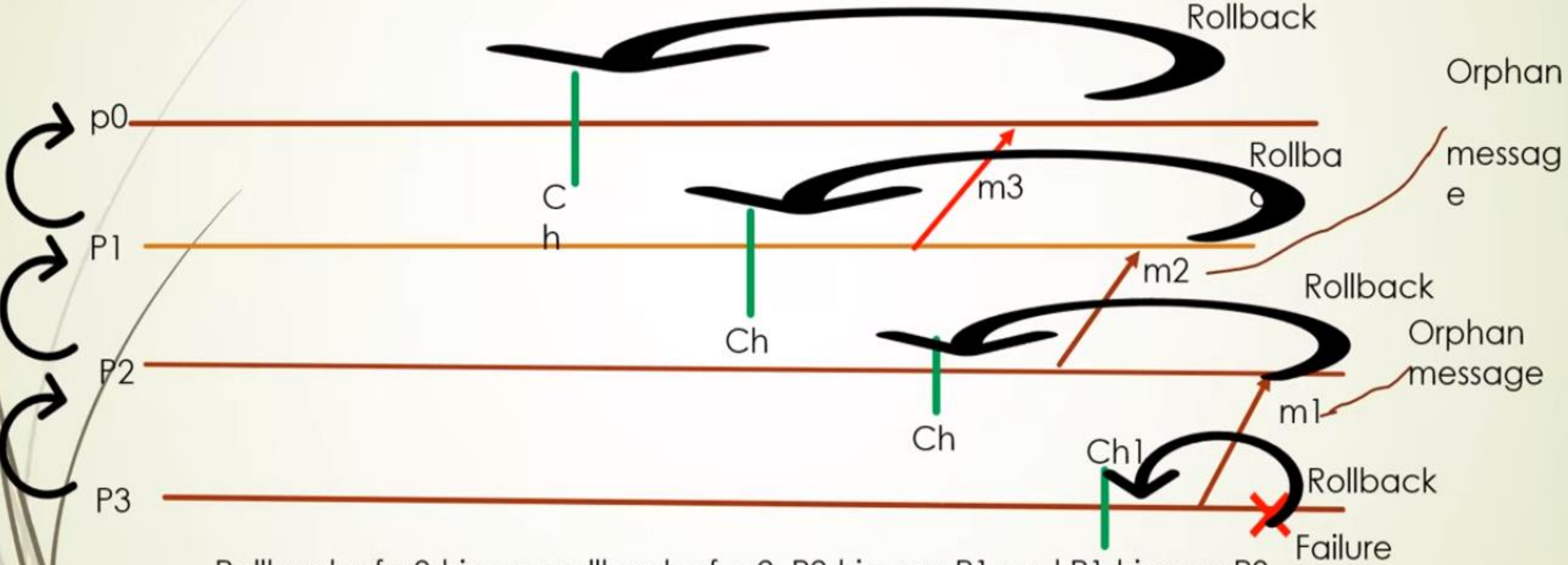
- Domino effect : A cumulative effect is produced when one event initiates a series of similar events

How domino effect can be avoided?

- Avoid the domino effect, by allowing processes to take some of their checkpoints independently.

- —

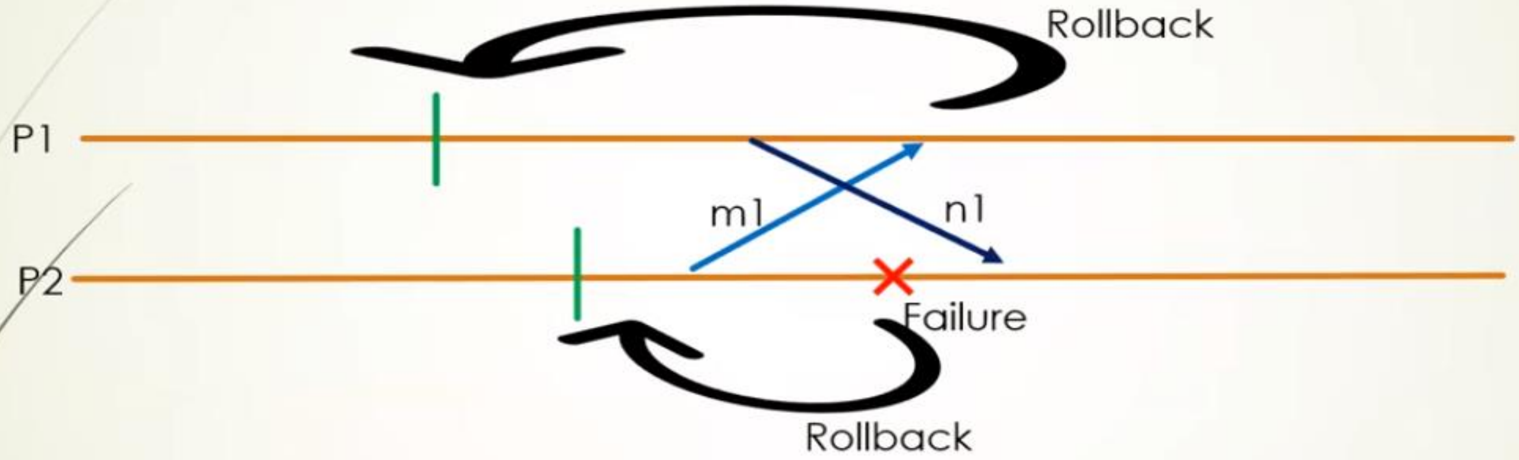
Cascading rollback and domino effect



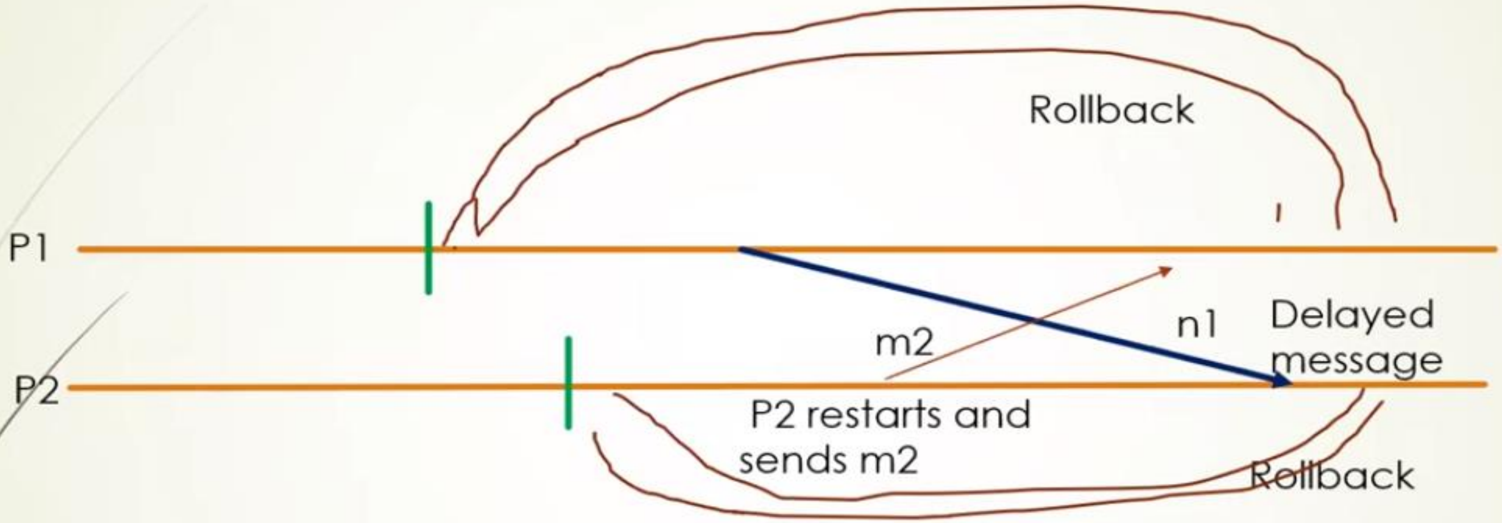
Rollback of p3 triggers rollback of p2, P2 triggers P1 and P1 triggers P0. M1, m2 and m3 becomes orphan message due to rollback This effect is called **domino effect**.

Live Lock

- Livelock: A single failure causing an infinite number of rollbacks.



Live Lock...............



Now n1 is received as a delayed message but there is no record of who sent it. So P2 has to rollback again.

P1 receives m2 but since P2 has been rollbacked, P1 also has to be rollbacked because of no record of sender of m2.
The above sequence will repeat indefinitely. This situation is called livelock