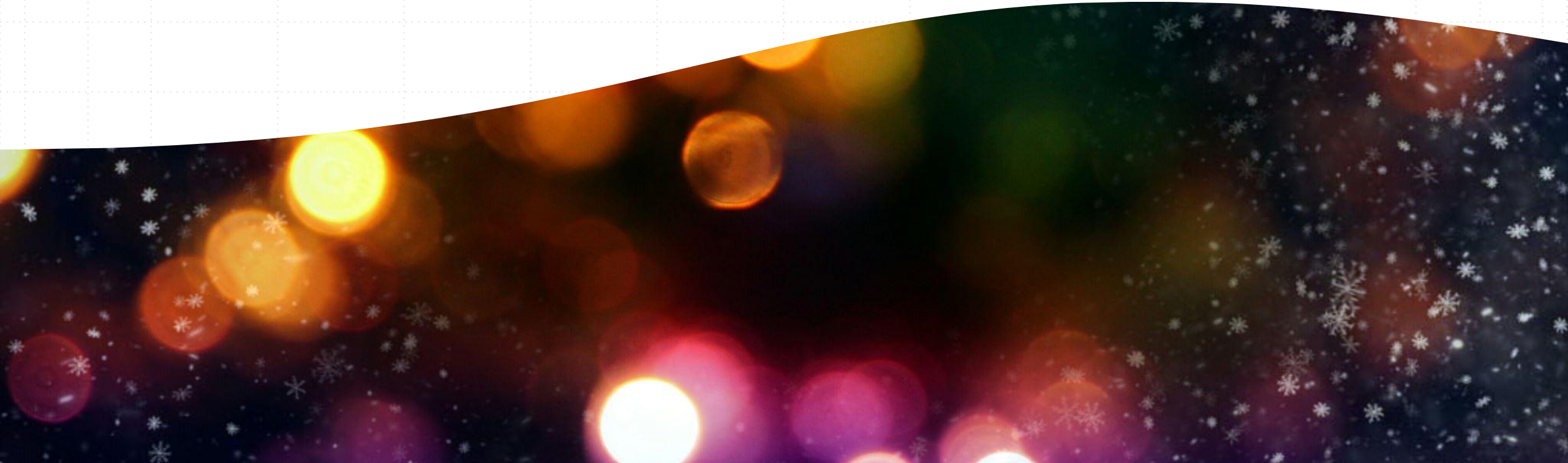
FAULT TOLERANCE

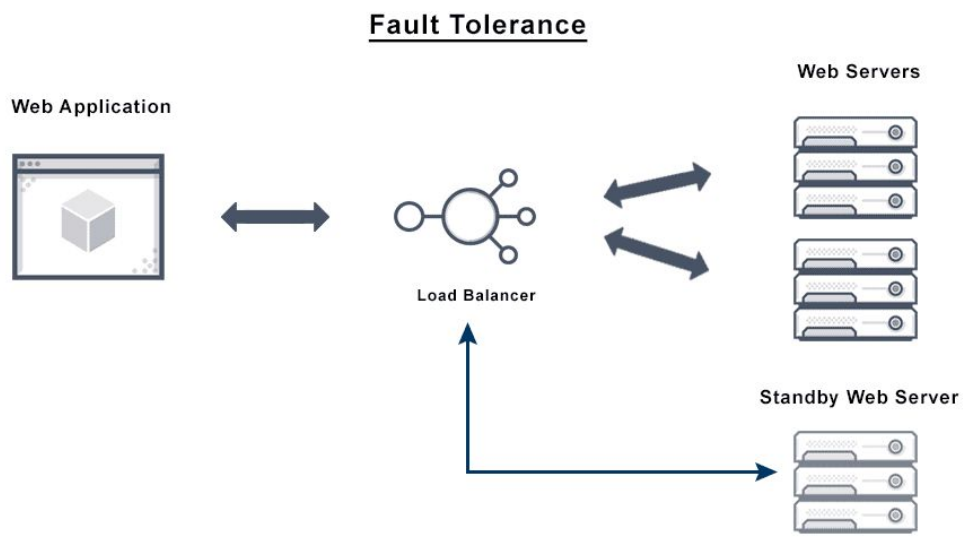
CHAPTER 8



Introduction to Fault Tolerance

- Fault Tolerance simply means **a system's ability to continue operating uninterrupted despite the failure of one or more of its components**.

- This is true whether it is a computer system, a cloud cluster, a network, or something else.


Fault Tolerance
Web Application
Load Balancer
Web Servers
Standby Web Server

Dependability requirements of fault tolerant system

1. Availability (system should be available for use at any given time)
2. Reliability (system should run continuously without failure)
3. Safety (temporary failures should not result in a catastrophe)
4. Maintainability (a failed system should be easy to repair)
5. Security (avoidance or tolerance of deliberate attacks to the system)

Fault

- Fault means defects within hardware or software unit.
  - Error means deviation from accuracy.
  - Failure is the condition occured when error results system to function incorrectly.
  - Fault tolerance means ability to provide services despite of faults occur in some node of the system.
  - A system is said to be k-fault tolerant if it is able to function properly even if k nodes of the system suffer from concurrent failures.

Types of Fault

- Node Fault: A fault that occur at an individual node participating in the distributed system. Example: a machine in distributed system fails due to some error in database configuration.
- Program Fault: A fault that occur due to some logical error in the code. Eg: program that is designed to filter the data by category but instead filtered by other means.
- Communication Fault: A fault that occurred due to unreliable communication channles connecting the node. Eg: a node sending data 010110 to another node but due to some problem in communication channel, the receiver receives 010111.
- Timing Fault: The fault that occurred due to mismatch on timing of any particular response . Eg: a server replies too late or a server is provided with data too soon that it has no enough buffer to hold the data.

Failure Model

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

•Different fault management techniques:

- fault prevention: prevent the occurrence of a fault,
- fault tolerance: build a component in such a way that it can meet its specifications in the presence of faults (i.e., mask the presence of faults),
- fault removal: reduce the presence, number, seriousness of faults,
- fault forecasting: estimate the present number, future incidence, and the consequences of faults.

Process Resilience

Process Resilience is a mechanism to protect yourself against faulty processes by replicating and distributing computations in a group.

- To tolerate a faulty process, organize several identical processes into a group
- □ So we can send a message to a group without explicitly knowing who are they, how many are there, or where are they (e.g., e-mail groups, newsgroups)

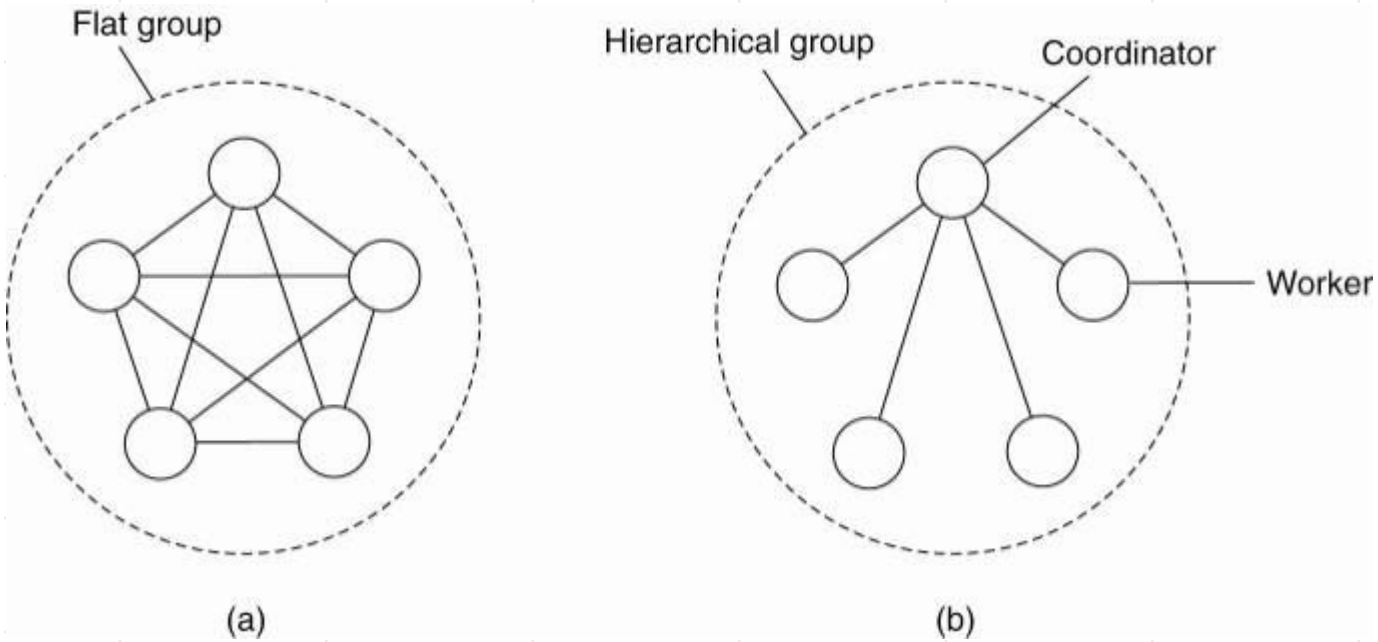
  □ Key property: When a message is sent, all members of the group must receive it. So if one fails, the others can take over for it.

- Groups could be dynamic
- □ So we need mechanisms to manage groups and membership (e.g., join, leave, be part of two groups)

•The group can be of two types:

- a. flat groups: good for fault tolerance as information exchange immediately occurs with all group members.
- All process with in the group have equal roles and control is completely distributed to all process.
- May impose more overhead as control is completely distributed (hard to implement).
- b. hierarchical groups: all communication through a single coordinator

  ⇒ not really fault tolerant and scalable, but relatively easy to implement.


Flat group
(a)
Hierarchical group
Coordinator
Worker
(b)

Group Membership

- Centralized: have a group server to maintain a database for each group and get these requests
- Efficient, easy to implement, but single point of failure
- Distributed: to join a group, a new process can send a message to all group members that it wishes to join the group (Assume that reliable multicasting is available)
- To leave, a process can ideally send a goodbye msg to all, but if it crashes (not just slow) then the others should discover that and remove it from the group!

Reliable Client-Server Communication

- A client communicates **a request to the server for operations on the resource**, and the server communicates the outcome of the operations to the client by a response.
- This request-response style of exchange is fundamental to client-server communications (also referred to as client-server interaction)

•Detecting process failures:

- Processes actively send "are you alive?" messages to each other (for which they obviously expect an answer)
- Makes sense only when it can be guaranteed that there is enough communication between processes.
- Processes passively wait until messages come in from different processes.

- **Example: RPC Semantics and Failures**
- Remote Procedure Call (RPC) mechanism works well as long as both the client and server function perfectly!!!

- Five classes of RPC failure can be identified:
  - *The client cannot locate the server*, so no request can be sent.
  - *The client's request to the server is lost*, so no response is returned by the server to the waiting client.
  - *The server crashes after receiving the request*, and the service request is left acknowledged, but undone.
  - *The server's reply is lost on its way to the client*, the service has completed, but the results never arrive at the client
  - *The client crashes after sending its request*, and the server sends a reply to a newly-restarted client that may not be expecting it.

- **A server in client-server communication.**
- (a). A request arrives, is carried out, and a reply is sent.
- (b). A request arrives and is carried out, just as before, but the server crashes before it can send the reply.
- (c). Again a request arrives, but this time the server crashes before it can even be carried out. And, no reply is sent back.
- **Client Crashes**
- When a client crashes, and when an ‘old’ reply arrives, such a reply is known as an *orphan*.

- Each of these categories posed different problems and requires different solutions . The solutions are:
- 1. Report back to client
- 2. Resend the message
- 4. Operations should be implemented (send multiple request)
- 5. kill the orphan communication

Reliable Group Communication

- reliable group communication to mean that **at least one member of the group receives the message and replies to it**.

Distributed Commit

- The distributed commit problem involves having an operation being performed by each member of a process group, or none at all.

- Distributed commit is often established by means of a coordinator.
- In a simple scheme, this coordinator tells all other processes that are also involved, called participants, whether or not to perform the operation .
-

- In a distributed system, either all sites commit or none of them does.
- The different distributed commit protocols are −
- One-phase commit
- Two-phase commit
- Three-phase commit

Distributed One-phase Commit

- Distributed one-phase commit is the simplest commit protocol.
- Let us consider that there is a controlling site and a number of slave sites where the transaction is being executed. The steps in distributed commit are −
- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site.
- The slaves wait for “Commit” or “Abort” message from the controlling site. This waiting time is called **window of vulnerability**.
- When the controlling site receives “DONE” message from each slave, it makes a decision to commit or abort. This is called the commit point. Then, it sends this message to all the slaves.
- On receiving this message, a slave either commits or aborts and then sends an acknowledgement message to the controlling site.

Distributed Two-phase Commit

- Distributed two-phase commit reduces the vulnerability of one-phase commit protocols. The steps performed in the two phases are as follows −
- **Phase 1: Prepare Phase**
- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site.
- When the controlling site has received “DONE” message from all slaves, it sends a “Prepare” message to the slaves.
- The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a “Ready” message.
- A slave that does not want to commit sends a “Not Ready” message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

- **Phase 2: Commit/Abort Phase**
- After the controlling site has received “Ready” message from all the slaves −
  - The controlling site sends a “Global Commit” message to the slaves.
  - The slaves apply the transaction and send a “Commit ACK” message to the controlling site.
  - When the controlling site receives “Commit ACK” message from all the slaves, it considers the transaction as committed.
- After the controlling site has received the first “Not Ready” message from any slave −
  - The controlling site sends a “Global Abort” message to the slaves.
  - The slaves abort the transaction and send a “Abort ACK” message to the controlling site.
  - When the controlling site receives “Abort ACK” message from all the slaves, it considers the transaction as aborted.

Distributed Three-phase Commit

- The steps in distributed three-phase commit are as follows −
- **Phase 1: Prepare Phase**
- The steps are same as in distributed two-phase commit.
- **Phase 2: Prepare to Commit Phase**
- The controlling site issues an “Enter Prepared State” broadcast message.
- The slave sites vote “OK” in response.
- **Phase 3: Commit / Abort Phase**
- The steps are same as two-phase commit except that “Commit ACK”/”Abort ACK” message is not required.

RECOVERY

-Recovery is the mechanism to handle failure.

-It helps to recover correct state of system after failure.

- Fundamental to fault tolerance is recovery from error.

-The whole idea of recovery is to replace an erroneous state with an error free state.

Recovery in distributed systems focuses on maintaining functionality and data integrity despite failures.

It involves strategies for detecting faults, restoring state, and ensuring continuity across interconnected nodes.

Importance of Recovery in Distributed Systems

- Effective recovery in distributed systems is crucial for ensuring system reliability, availability, and fault tolerance.
- When a component fails or an error occurs, the system must recover quickly and correctly to minimize downtime and data loss.
- Effective recovery mechanisms, such as checkpointing, rollback, and forward recovery, help maintain system consistency, prevent cascading failures, and ensure that the system can continue to function even in the presence of faults.

Recovery Techniques in Distributed Systems

- **Checkpointing**: Periodically saving the system's state to a stable storage, so that in the event of a failure, the system can be restored to the last known good state. Checkpointing is a key aspect of backward recovery.
- **Rollback Recovery**: Involves reverting the system to a previous checkpointed state upon detecting an error. This technique is useful for undoing the effects of errors and is often combined with checkpointing.
- **Forward Recovery**: Instead of reverting to a previous state, forward recovery attempts to move the system from an erroneous state to a new, correct state.
- **Logging and Replay**: Keeping logs of system operations and replaying them from a certain point to recover the system's state. This is useful in scenarios where a complete rollback might not be feasible.

- **Replication**: Maintaining multiple copies of data or system components across different nodes. If one component fails, another can take over, ensuring continuity of service.
- **Error Detection and Correction**: Incorporating mechanisms that detect errors and automatically correct them before they lead to system failure. This is a proactive approach that enhances system resilience.

TWO FORM OF ERROR RECOVERY

1. BACKWARD RECOVERY (UNDO)

   -In this recovery the main issue is to bring the system from its present erroneous state back to previously correct state.

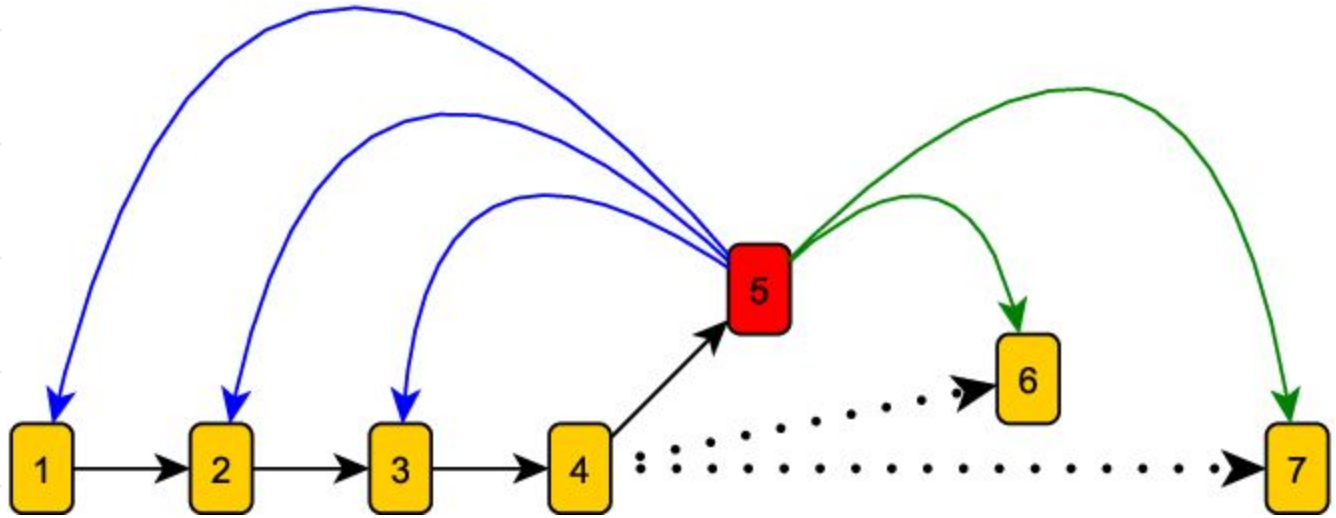
   -To do so it will be necessary to record the systems state from time to time and to restore such a recorded state when things goes wrong.

   -Each time the systems present state is recorded , a checkpoint is said to be made.

2. FORWARD RECOVERY(REDO)

– In this recovery , when a system has entered an erroneous state , instead of moving back to previous state , a checkpointed state, an attempt is made to bring the system in a correct new state from which it can continue to execute.

-the main problem with forward recovery is that it has to be known in advance which error may occur


1
2
3
4
5
6
7

Recovery Actions in Fault-Tolerant Distributed Systems

In fault-tolerant distributed systems, there are multiple techniques to ensure the **undo** and **redo** actions are applied correctly to ensure **consistency** and **availability**.

- Many distributed systems, especially **distributed databases**, use a **log** to store all the actions (or transactions) in a **write-ahead log** (WAL). This log is maintained to facilitate recovery after crashes or failures.
    - **Undo Operation**: If a failure happens after a transaction has been initiated but before it is fully committed, the system will use the log to **undo** the transaction (rolling back to the previous consistent state).
    - **Redo Operation**: After a failure or crash, the system can replay the log (or parts of it) to **redo** any operations that were successfully committed but not yet fully replicated across all nodes or systems.

RECOVERY TECHNIQUES

-COORDINATED CHECKPOINT

-INDEPENDENT CHECKPOINT

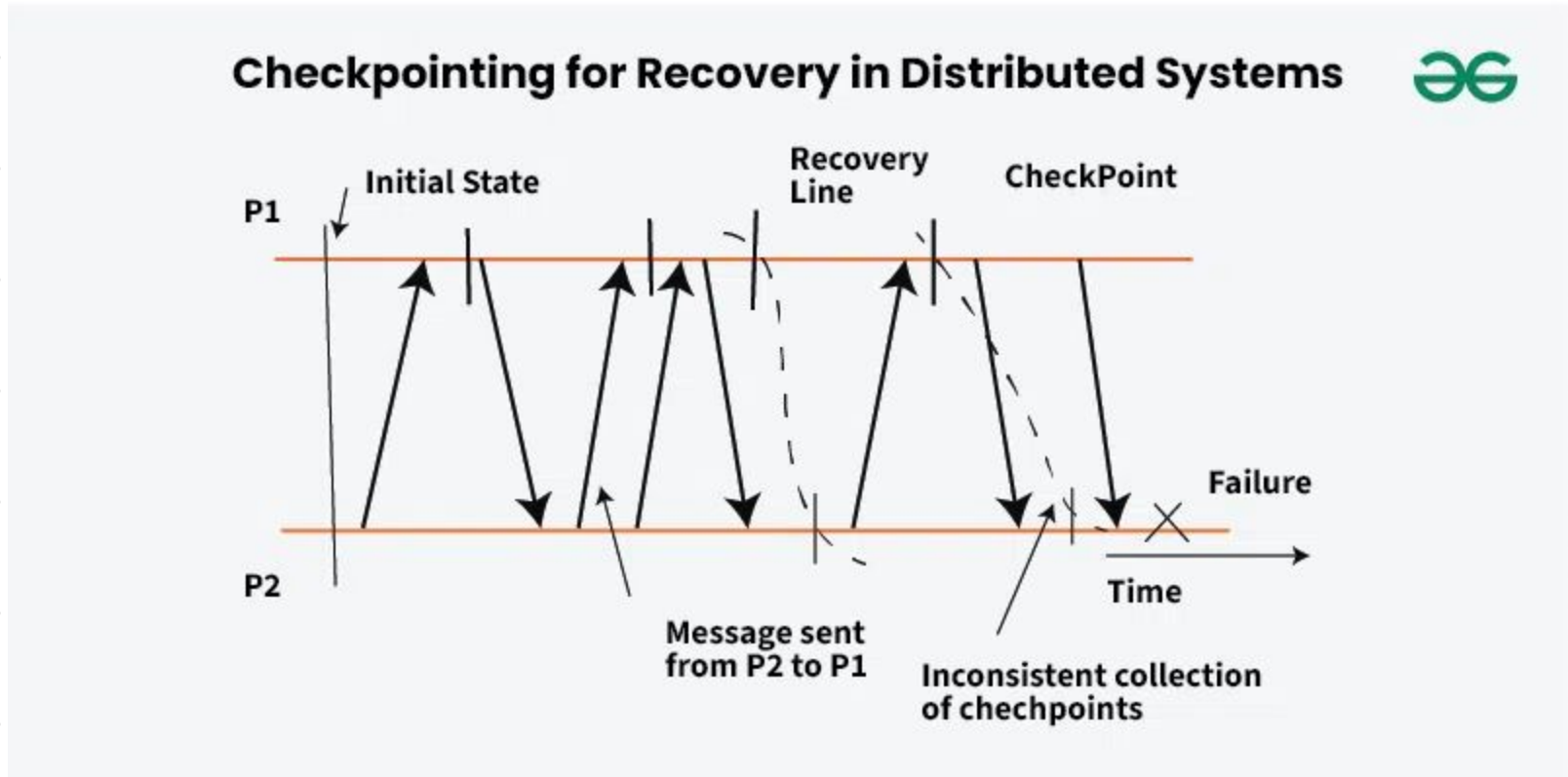
-MESSAGE LOGGING

Checkpointing for Recovery in Distributed Systems

Backward error recovery in a fault-tolerant distributed system involves regularly saving the system's state to stable storage.

To do this, we need to take a distributed snapshot, also known as a consistent global state.

- Checkpointing is most typically used to provide fault tolerance to applications.
- In this case, the state of the entire application is periodically saved to some kind of stable storage, e.g., disk, and can be retrieved in case the original application crashes due to a failure in the underlying system.
- The application is then restarted (or recovered) from the checkpoint that was created last and continued from that point on, thereby minimizing the time lost due to the failure.


Checkpointing for Recovery in Distributed Systems
Initial State
Recovery Line
CheckPoint
P1
P2
Failure
Time
Message sent from P2 to P1
Inconsistent collection of chechpoints

In backward error recovery, each process periodically saves its state to stable storage that it can access locally.

To recover from a system or process failure, we need to piece together a stable global state from these local states.

The goal is to recover to the most recent distributed snapshot, known as the recovery line.

This recovery line represents the latest stable group of saved checkpoints.

Coordinated Checkpoint

A **coordinated checkpoint** is a fault tolerance mechanism where **all processes in a distributed system synchronize** to create a **consistent global checkpoint**.

- **Coordination**: One process (often called the **coordinator**) initiates the checkpointing process and ensures all others take checkpoints in a **globally consistent manner**.

- **Consistent global state**: Ensures that no message is “in-flight” (i.e., sent but not yet received) in the saved state, which avoids inconsistencies.

How It Works (Simplified)

1. The **coordinator** sends a **checkpoint request** to all processes.
2. Each process:
    - Stops its execution temporarily (or buffers messages).
    - Takes a local checkpoint.
    - Acknowledges completion.
3. Only after **all processes complete** their checkpoints, the system resumes normal operation.

To avoid stopping the system entirely, **non-blocking coordinated checkpointing** protocols have been developed.

Independent Checkpoint

An **independent checkpoint** in a distributed system refers to a type of checkpointing mechanism where each process in the system **saves its state independently**, without coordinating with other processes.

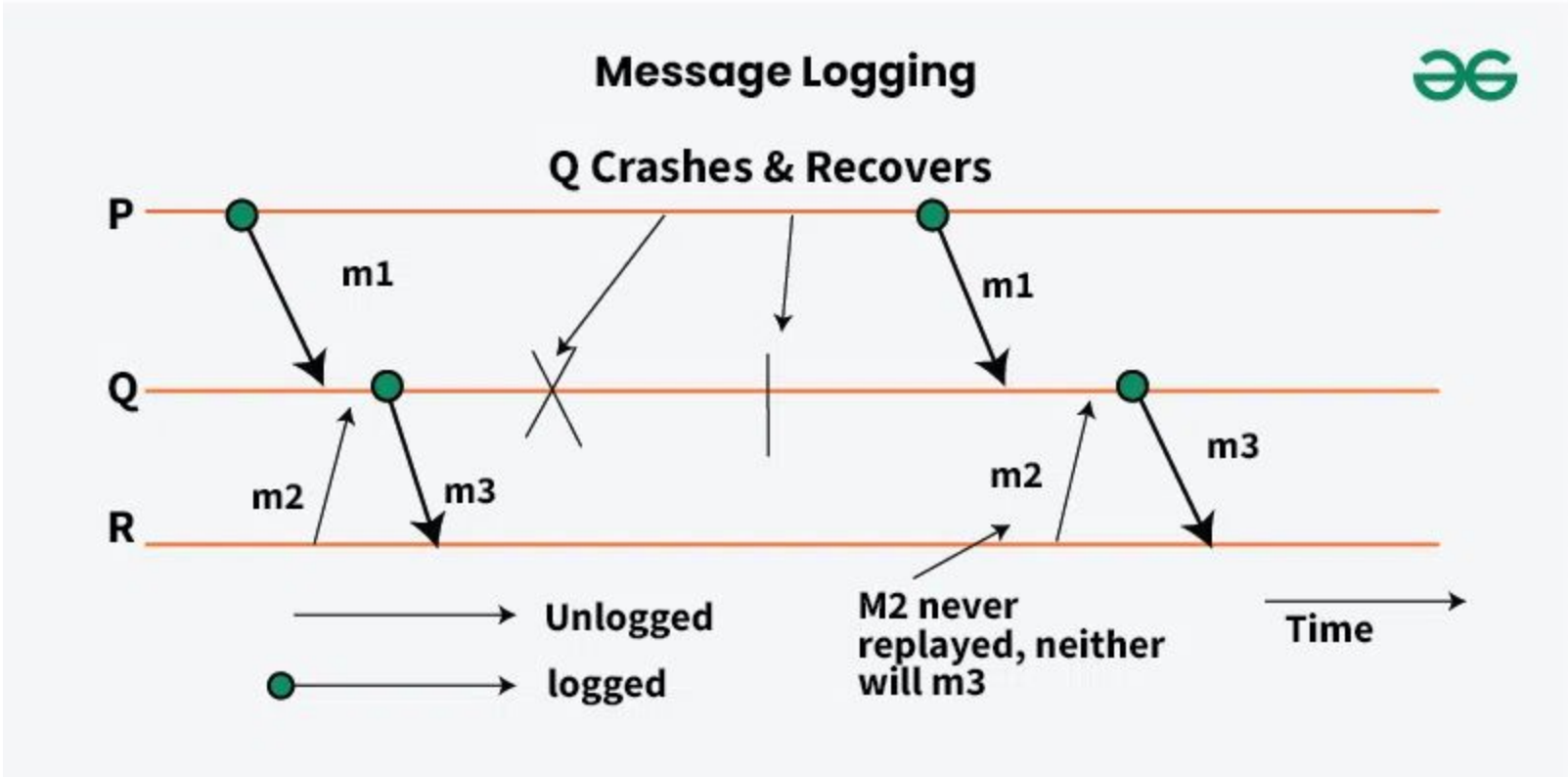
In a distributed system:

- A **checkpoint** is a saved local state of a process that can be used to recover from failures.
- An **independent checkpoint** is taken **without synchronization** or communication between the distributed processes.

Each process decides **when** to take its own checkpoint based on its own criteria (e.g., time interval, number of messages sent/received, etc.).

Message Logging

- Message logging is a common technique used to build systems that can tolerate process crash failures.
- These protocols require that each process periodically records its local state and log the messages received since recording that state.
- When a process crashes, a new process is created in its place: the new process is given the appropriate recorded local state, and then it replays the logged messages in the order they were originally received.


Message Logging
Q Crashes & Recovers
P
Q
R
m1
m1
m2
m3
m2
m3
Unlogged
logged
M2 never
replayed, neither
will m3
Time

What is orphan process example?

- An orphan process is **a process whose parent has finished**.

- Suppose P1 and P2 are two process such that P1 is the parent process and P2 is the child process of P1. Now, if P1 finishes before P2 finishes, then P2 becomes an orphan process.