Distributed System UNIT 3

Introduction to fault Tolerance

Fault tolerant is strongly related to what are called dependable systems. Dependability is a term that covers a number of useful requirements for distributed systems including the following :

1.Availability
2.Reliability
3.Safety
4. Maintainability

Availability :

Avail ability is defined as the property that a system is ready to be used immediately. In other words, a highly available system is one that will most likely be working at a given instant in time.

Reliability :

Reliability refers to the property that a system can run continuously without failure. In contrast to availability, reliability is defined in terms of a time interval instead of an instant in time. A highly-reliable system is one that will most likely continue to work without interruption during a relatively long period of time. This is a subtle but important difference when compared to availability

Safety :

Safety refers to the situation that when a system temporarily fails to operate correctly, nothing catastrophic happens.

Maintainability :

Maintainability refers to how easy a failed system can be repaired.

Faults are generally classified as transient, intermittent, or permanent.

1.TransientFaults

♦ It can occur suddenly, disappear, and may not occur again if the operation is repeated. For example: During heavy traffic in network, a telephone call is miss route but if call is retried, it will reach destination correctly.

2.Intermittent faults :

These faults occur often, but may not be periodic in nature. ForExample:

♦ The loose connection to network switch may cause intermittent connection problems.
♦ SuchFaultsAreDifficultToDiagnose.
♦ During debugging process, it is possible that the fault may not be detected at all.

3.PermanentFaults

♦ These faults can be easily identified and the components can be replaced. ForExample:

♦ Software bug or disk head crash causes permanent faults.
♦ These faults can be identified and corresponding action can be taken.

(OR)

A permanent fault is one that continues to exist until the faulty component is replaced. Burnt-out chips, software bugs, and disk head crashes are examples of permanent faults.

FAILURE MODELS :

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*	A 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

Figure 8-1. Different types of failures.

Different types of failures.

♦ A crash failure occurs when a server prematurely halts, but was working correctly until it stopped. A typical example of a crash failure is an operating system that comes to a grinding halt, and for which there is only one solution: reboot it.

♦ An omission failure occurs when a server fails to respond to a request. Several things might go wrong. In the case of a receive omission failure, possibly the server never got the request in the first place. Note that it may well be the case that the connection between a client and a server has been correctly established, but that there was no thread listening to incoming requests. Likewise, a send omission failure happens when the server has done its work, but somehow fails in sending a response.

♦ Timing failures occur when the response lies outside a specified real-time interval.

♦ A serious type of failure is a response failure, by which the server's response is simply incorrect. Two kinds of response failures may happen. In the case of a value failure, a server simply provides the wrong reply to a request.

♦ The other type of response failure is known as a state transition failure. This kind of failure happens when the server reacts unexpectedly to an incoming request. For example, if a server receives a message it cannot recognize, a state transition failure happens if no measures have been taken to handle such messages.

♦ The most serious are arbitrary failures, also known as Byzantine failures. In effect, when arbitrary failures occur, clients should be prepared for the worst.In particular, it may happen that a server is producing output it should never have produced, but which cannot be detected as being incorrect Worse yet a faulty server may even be maliciously working together with other servers to produce intentionally wrong answers. This situation illustrates why security is also considered an important requirement when talking about dependable systems.

FAILURE MASKING :

The key technique for masking faults is to use redundancy. Three kinds are possible: information redundancy, time redundancy, and physical redundancy.

Types of Redundancy are as follows:

1.InformationRedundancy
Extra bits are added to data to handle fault tolerance by detecting errors.

2.TimeRedundancy
♦ An action performed once is repeated if needed after a specific time period. For example, if an atomic transaction aborts, it can be executed without any side effects.
♦ Time redundancy is helpful when the faults are transient or intermittent.

3.PhysicalRedundancy
♦ Extra equipment's are added to enable the system to tolerate faults due to loss or malfunctionofsomecomponents.
For example, extra stand by processors can be used in the system

Process Resilience

Process Resilience :

Overview of Process Resilience

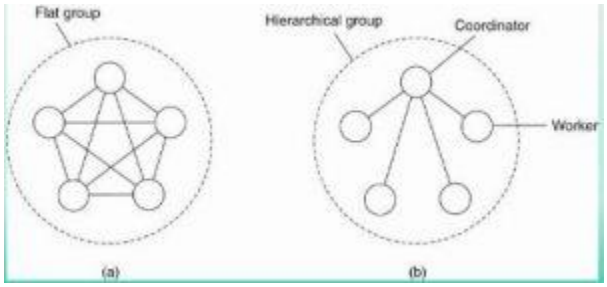
- Design Issues
- Failure Masking and Replication
- Agreement in Faulty Systems
- Failure Detection

The following are the ways to achieve fault tolerance in distributed systems and especially against process failures.

- Replicating processes into groups.
- Consider collections of process as a single abstraction
- All members of the group receive the same message, if one process fails, the others can take over for it.
- Process groups are dynamic and a Process can be member of several groups.
- Hence we need some mechanisms to manage the groups.

DESIGN ISSUES

Flat Group vs. Hierarchical Group



a) Flat Group
b) Hierarchical Group

Flat Group	Hierarchical Group
They are symmetric.	These groups are asymmetric.
In flat group all the process are equal and the decisions made in the flat group are collective.	One of the process is elected to be the coordinator, which selects another process(a worker) to perform the operation.
No single point-of-failure is seen but the decision making is complicated as consensus is required.	Single point failure is observed in this case. Decisions are easily and quickly made by the coordinator without having to get consensus.

Group Members

A process can join or leave the group depending on the work load. A group can be broken and created. So it is necessary to keep the track of these groups.

♦ Hierarchical Groups

Less complex, simple and easy to execute.

Most important downfall of this group is Single point of failure.

♦ Flat Groups

In order to join or leave a group each process has to broadcast a message.

♦ Fault is occurred because of a dead member or a very slow member.

♦ Synchronization is required while joining and leaving the group. When a new process joins the group all the messages which were exchanged previously are sent to the newly joint group.

The following is the procedure describing how groups are formed.

♦ The system call setside() is used to create a new session containing a single new process group. The groups are identified by a positive integer, process group ID is used to identify the groups.

♦ The system call setpgid() is used to set the process group ID.

REPLICATING AND MASKING FAILURES :

- Process are replicated and organized into groups.
- A single vulnerable process in the whole fault tolerant Group is replaced.
- A system is said to be K fault tolerant if it can survive faults in K components and still meet its specifications.
- A system is K fault tolerant if it can have K faulty components and still meet its specification.
- K+1 or 2K+1, is the replication needed to support K Fault Tolerance
- Case:If K processes stop, then the answer from the other one can be used.
  □K+1

  If it meets Byzantine failure, the number is

  □ 2K+1

Agreement in Faulty Systems :

Goal of Agreement

- Making all the non-faulty processes reach consensus on some issue
- Establishing that consensus within a finite number of steps.

A process group typically requires reaching an agreement in:

-Dividing tasks among workers.

-Electing a coordinator.

-Deciding whether or not to commit a transaction.

-Synchronization.

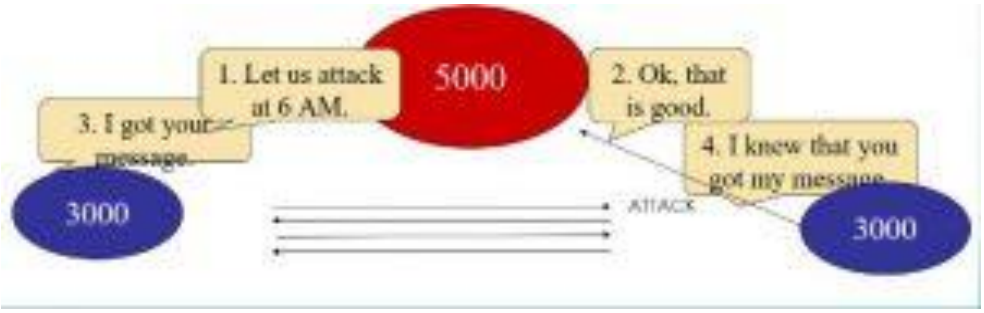
-When the communication and processes: are good, agreement is reached in a simple and a straightforward way. Whereas when it is bad, reaching to an agreement is troublesome.

Problems of two cases

Good process, but unreliable communication :Example: Two-army problem

Good communication, but crashed process: Example: Byzantine generals problem

Two-army problem

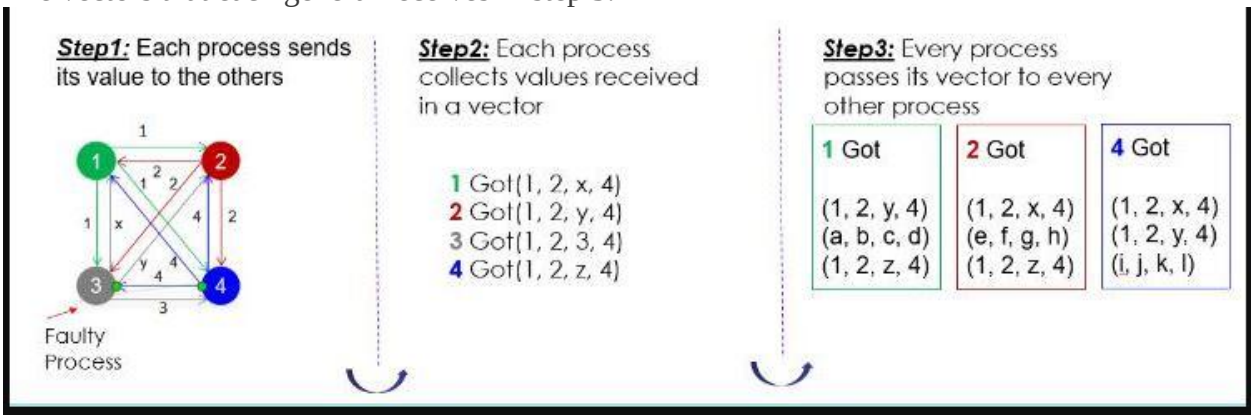


This problem is classically stated as the two-army problem, and is insoluble. The agreed upon action will never take place, because the last sender will never be certain that the last confirmation went through.(Due to unreliable communication)

Byzantine generals problem :

The Byzantine generals problem for 3 loyal generals and 1 traitor.

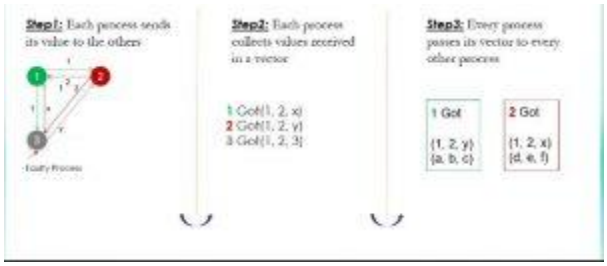
- The generals announce their troop strengths (in units of 1 thousand soldiers).
- The vectors that each general assembles based on (a)
- The vectors that each general receives in step 3.



Step 4:

- Each process 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
- If no value has a majority, the corresponding element of the result vector is marked UNKNOWN

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



Step 4:

- Each process 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
- If no value has a majority, the corresponding element of the result vector is
- marked UNKNOWN

Result Vector:
(UNKNOWN, UNKNOWN, UNKNOWN).

Concluding Remarks on the Byzantine Agreement Problem

- Lamport et al proved that in a system with k faulty processes, an agreement can be achieved only if 2k+1 correctly functioning processes are present, for a total of 3k+1. i.e., An agreement is possible only if more than two-thirds of the processes are working properly.
- Fisher et al. proved that in a distributed system in which ordering of messages cannot be guaranteed to be delivered within a known, finite time, no agreement is possible even if only one process is faulty.

Process Failure Detection

- Before mask failures are properly masked, there is a need to detect them
- For a group of processes, non-faulty members should be able to decide who is still a member and who is not
- Two policies:

♦ Processes actively send "are you alive?" messages to each other (i.e., pinging each other)

♦ Processes passively wait until messages come in from different processes

Failure Considerations

There are various issues that need to be taken into account when designing a failure detection subsystem:

- Failure detection can be done as a side-effect of regularly exchanging information with neighbors.
- Network failure and node failures must be distinguished by the fault detecting system.

Reliable Client-server Communication.

Reliable Client-server Communication.

Point-to-Point Communication :

In many distributed systems, reliable point-to-point communication is established by making use of a reliable transport protocol, such as TCP. TCP masks omission failures, which occur in the form of lost messages, by using acknowledgments and re-transmissions. Such failures are completely hidden from a TCP client.

However, crash failures of connections are not masked. A crash failure may occur when (for whatever reason) a TCP connection is abruptly broken so that no more messages can be transmitted through the channel. In most cases, the client is informed that the channel has crashed by raising an exception. The only way to mask such failures is to let the distributed system attempt to automatically set up a new connection, by simply resending a connection request. The underlying assumptioriis that the other side is still, or again, responsive to such requests.

RPC Semantics in the Presence of Failures :

As long as both client and server are functioning perfectly, RPC does its job well. The problem comes about when errors occur. It is then that the differences between local and remote calls are not always easy to mask five different classes of failures that can occur in RPC systems, as follows:
1. Client cannot locate server
2. Server crashes after receiving a request
3. Client request is lost
4. Server response is lost
5. Client crashes after sending a request
Each of these categories poses different problems and requires different solutions.

Client Cannot Locate the Server :

– Solution: The RPC system informs the client of the failure.

Server Crashes after Receiving a Request:
• The client cannot tell if the crash occurred before or after the request is carried

out
• Three possible semantics
– At-least-once: keep trying until a reply is received
– At-most-once: give up immediately and report back failure
– Exactly once: desirable but not achievable.

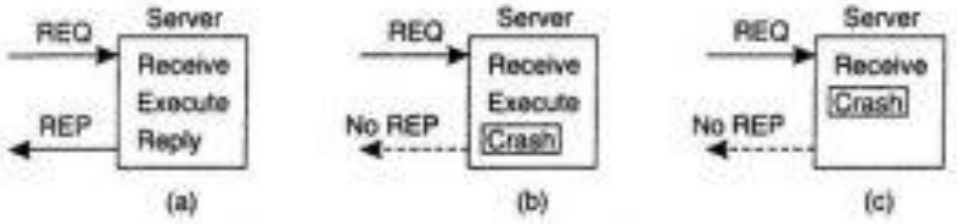


Figure 8-7. A server in client-server communication. (a) The normal case. (b) Crash after execution. (c) Crash before execution.

A server in client-server communication.

(a) The normal case.
(b) Crash after execution. (c) Crash before execution.

Lost Request/Reply Messages :
• Client waits for reply message, resents the request upon timeout
– Problem: Upon timeout, client cannot tell whether the request was lost or the reply was lost
• Client can safely resend the request for idempotent operations
– An idempotent operation is an operation that can be safely repeated
– E.g., reading the first line of a file is idempotent, transferring money is not
• For non idempotent operations, client can add sequence numbers to requests so that the server can distinguish a re transmitted request from an original request
– Server need keep track of the most recently received sequence number from each client
– Server will not carry out a re transmitted request, but will send a reply to the client.

Client Crashes after Sending a Request :
• What happens to the server computation, referred to as an orphan?

– Extermination: Client explicitly kills off the orphan when it comes back up
• Client stub makes a log entry on disk before sending an RPC message
– Reincarnation: When a client reboots, it broadcasts a new epoch number; when server receives the broadcast, it kills the computations that were running on behalf of the client
– Expiration: each RPC is associated with an expiration time T
• The call is aborted when the expiration time is reached
• If RPC cannot finish within T, the client must asks for another quantum
• If after a crash the client waits a time T before rebooting, all orphans are sure to be gone.

Reliable Group Communication.

Reliable Group Communication.

Considering how important process resilience by replication is, it is not surprising that reliable multicast services are important as well. Such services guarantee that messages are delivered to all members in a process group.

Basic Reliable-Multicasting Schemes :

Reliable multicasting means that
– If all processes are nonfaulty, every message should be delivered to each group member
– In the presence of faulty processes, every message should be delivered to each non faulty group member.

Reliable Multicasting When Processes are Nonfaulty :
• Assumption: the underlying communication system offers only unreliable multicasting
• A solution:
– Sender assigns a sequence number to each message
– When sender sends message M, it stores M in a history buffer
– Each receiver acknowledges the receipt of M, or requests retransmission when noticing message loss
– Sender removes M from history buffer when everyone has returned an acknowledgement (ACK)
• This solution does not scale: The sender may be swamped with ACKs when the number of receivers is large.

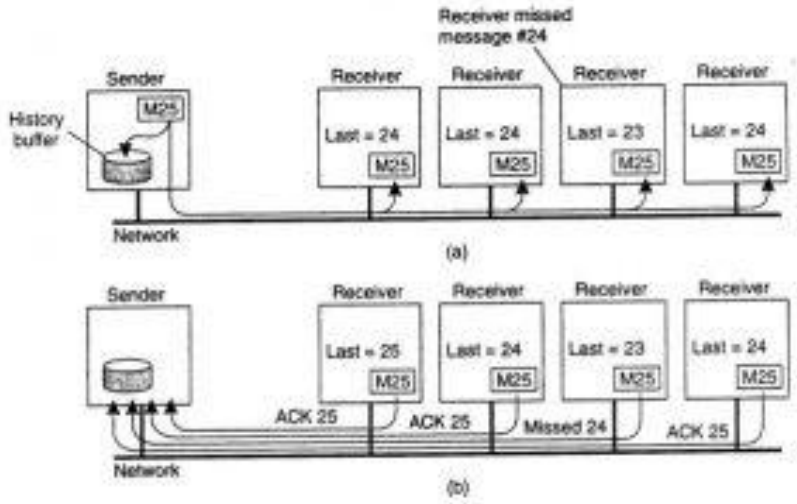


Figure 8-9. A simple solution to reliable multicasting when all receivers are known and are assumed not to fail. (a) Message transmission. (b) Reporting feedback.

Distributed Commit

♦ **The atomic multicasting problem is an example of a more general problem known as distributed commit.**

♦ **The distributed commit problem involves having an operation being performed by each member of a process group, or none at all. In the case of reliable multicasting, the operation is the delivery of a message.**

- **General Goal:**

The general goal is to make an operation to be performed by all group members or none at all.

[In the case of atomic multicasting, the operation is the delivery of the message.]

- **There are three types of "commit protocol":**

Single-phase.

Two-phase.

Three-phase.

One-Phase Commit Protocol:

- **An elected co-ordinator tells all the other processes to perform the operation in question.**
- **But, what if a process cannot perform the operation?**
- **There's no way to tell the coordinator!**
- **The solutions: T*wo-Phase* and *Three-Phase Commit Protocols***

Two-Phase Commit Protocol:

- **First developed in 1978.**
- ***Summarized: GET READY, OK, GO AHEAD.***

1. **The coordinator sends a *VOTE_REQUEST* message to all group members.**
2. **A group member returns *VOTE_COMMIT* if it can commit locally, otherwise *VOTE_ABORT*.**
3. **All votes are collected by the coordinator.**

- **A *GLOBAL_COMMIT* is sent if all the group members voted to commit.**
- **If one group member voted to abort, a *GLOBAL_ABORT* is sent.**

1. **Group members then COMMIT or ABORT based on the last message received from the coordinator.**

First phase – voting phase – steps 1 and 2.Second phase – decision phase steps 3 and 4.
(a) The finite state machine for the coordinator in 2PC.(b) The finite state machine for a participant. Big Problem with Two-Phase Commit

- **It can lead to both the coordinator and the group members blocking, which may lead to the dreaded *deadlock*.**
- **If the coordinator crashes, the group members may not be able to *reach a final decision*, and they may, therefore, block until the coordinator *recovers* …**
- **Two-Phase Commit is known as a blocking-commit protocol for this reason.**
- **The solution is *The Three-Phase Commit Protocol***

<u>Three-Phase Commit Protocol</u>:

Essence: 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.**

(a) The finite state machine for the coordinator in 3PC.

(b) The finite state machine for a participant.

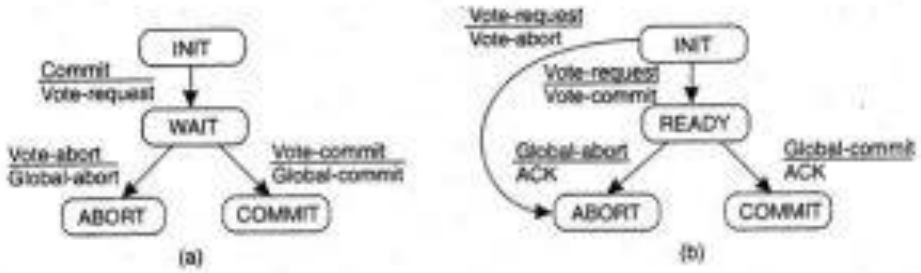


Figure 8-18. (a) The finite state machine for the coordinator in 2PC. (b) The finite state machine for a participant.

Recovery.

Recovery.

- Once a failure has occurred, it is essential that the process where the failure happened *recovers* to a correct state.
- Recovery from an error is *fundamental* to fault tolerance.
- Two main forms of recovery:

1. **Backward Recovery**: return the system to some previous correct state (using *checkpoints*), then continue executing.
2. **Forward Recovery**: bring the system into a correct state, from which it can then continue to execute.

Forward and Backward Recovery

Backward Recovery:

Advantages

- Generally applicable independent of any specific system or process.
- It can be integrated into (the middleware layer) of a distributed system as a general-purpose service.

Disadvantages:

- Checkpointing (can be very expensive (especially when errors are very rare).

[Despite the cost, backward recovery is implemented more often. The "logging" of information can be thought of as a type of checkpointing.].

- Recovery mechanisms are independent of the distributed application for which they are actually used – thus no guarantees can be given that once recovery has taken place, the same or similar failure will not happen again.

Disadvantage of Forward Recovery:

- In order to work, all potential errors need to be accounted for *up-front*.
- When an error occurs, the recovery mechanism then knows what to do to bring the system *forward* to a correct state.

Consider as an example:

Reliable Communications.

Retransmission of a lost/damaged packet – backward recovery technique.*Erasure Correction* – When a lost/damaged packet can be reconstructed as a result of the receipt of other successfully delivered packets – forward recovery technique. [see Rizzo (1997)]

- Elnozahy et al. (2002) and (Elnozahy and Planck, 2004) provide a survey of checkpointing and logging in distributed systems.
- See also Alvisi and Marzullo (1998) for message-logging schemes.

Recovery-Oriented Computing

Recovery-oriented computing – Start over again (Candea et al., 2004a).

- Underlying principle – it may be much cheaper to optimize for recovery, then it is aiming for systems that are free from failures for a long time.

Different flavors:

- Simply reboot (part of a system)
- e.g. restart Internet servers (Candea et al., 2004, 2006).
- To reboot only a part of the system – i the fault is properly localized.
- means deleting all instances of the identified components, along with the threads operating on them, and (often) to just restart the associated requests.
- Apply checkpointing and recovery techniques, but to continue execution in a changed environment.
- Basic idea – many failures can be simply avoided if programs are given extra buffer space, memory is zeroed before allocated, changing the ordering

of message delivery (as long as this does not affect semantics), and so on (Qin et al., 2005).