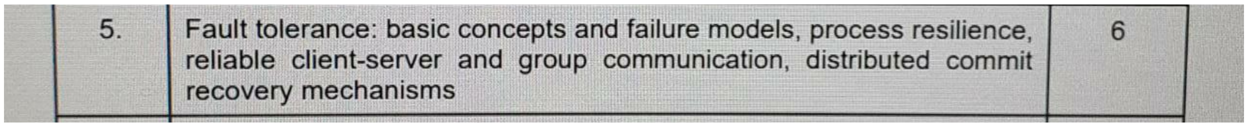
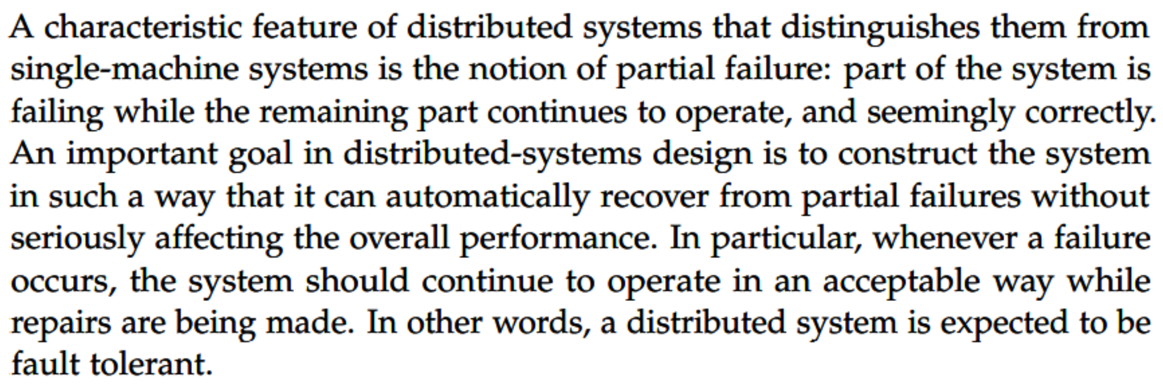
UNIT-5 DIS

Fault Tolerance



5.	Fault tolerance: basic concepts and failure models, process resilience, reliable client-server and group communication, distributed commit recovery mechanisms	6



A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of partial failure: part of the system is failing while the remaining part continues to operate, and seemingly correctly. An important goal in distributed-systems design is to construct the system in such a way that it can automatically recover from partial failures without seriously affecting the overall performance. In particular, whenever a failure occurs, the system should continue to operate in an acceptable way while repairs are being made. In other words, a distributed system is expected to be fault tolerant.

Being fault tolerant is directly proportional to dependable systems ;
Dependable system poses four characters i.e

A. Availability
B. reliability
C. safety
D. maintainability .

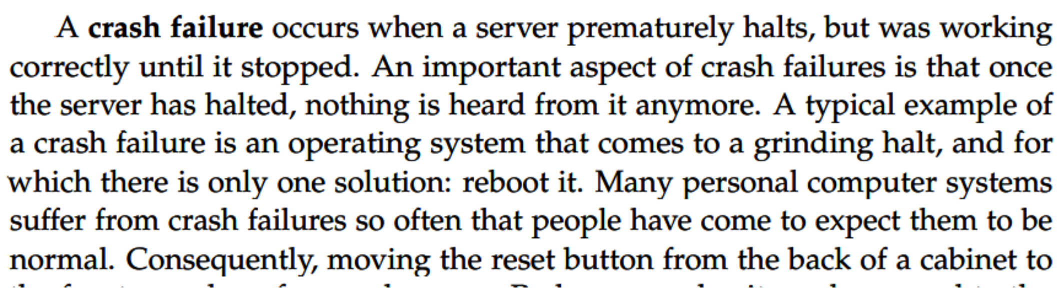
Types of Faults

- Transient Fault – appears once, then disappears. If the operation is repeated the fault goes away. Eg: a bird flying through a beam of microwave in a wireless network may cause loss of some bits .
- Intermittent Fault – occurs, vanishes, reappears; but: follows no real pattern (worst kind). Eg: a loose contact of connector .
- Permanent Fault – once it occurs, only the replacement/repair of a faulty component will allow the DS to function normally. Eg: burnt out chips , software crashes etc..

Types of Failures

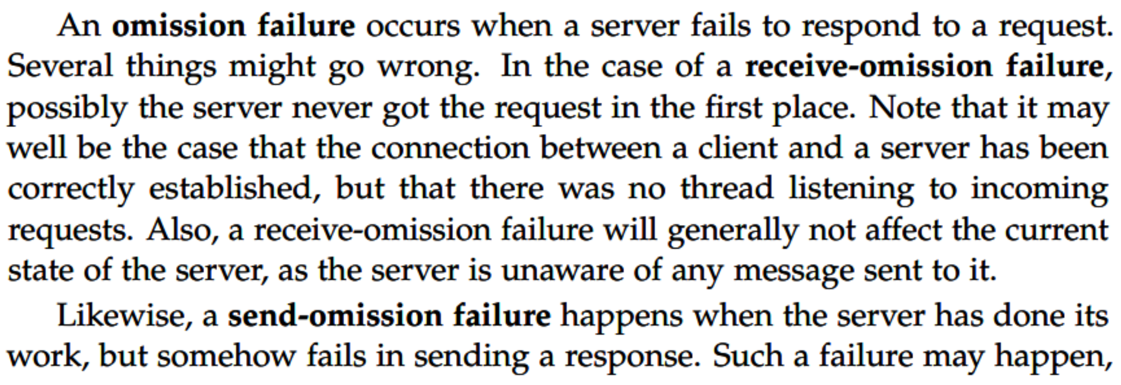
Type of failure	**Description**
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
• *Receive omission*	• A server fails to receive incoming messages
• *Send omission*	• A server fails to send messages
Timing failure	A 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

1. Crash failure :



A **crash failure** occurs when a server prematurely halts, but was working correctly until it stopped. An important aspect of crash failures is that once the server has halted, nothing is heard from it anymore. 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. Many personal computer systems suffer from crash failures so often that people have come to expect them to be normal. Consequently, moving the reset button from the back of a cabinet to

2. Omission failure



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. Also, a receive-omission failure will generally not affect the current state of the server, as the server is unaware of any message sent to it.

Likewise, a **send-omission failure** happens when the server has done its work, but somehow fails in sending a response. Such a failure may happen,

3. Timing failure

Another class of failures is related to timing. **Timing failures** occur when the response lies outside a specified real-time interval. For example, in the case of streaming video's, providing data too soon may easily cause trouble for a recipient if there is not enough buffer space to hold all the incoming

data. More common, however, is that a server responds too late, in which case a *performance* failure is said to occur.

4. Response failure

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. For example, a search engine that systematically returns Web pages not related to any of the search terms used, has failed.

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. In particular, a faulty server may incorrectly take default actions it should never have initiated.

5. Arbitrary failure

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.

Process resilience (It means how to make a process fault tolerant)

Resilience by process groups

The key approach to tolerating a faulty process is to organize several identical processes into a group. The key property that all groups have is that when a message is sent to the group itself, all members of the group receive it. In this way, if one process in a group fails, hopefully some other process can take over for it [Guerraoui and Schiper, 1997].

Process groups may be dynamic. New groups can be created and old groups can be destroyed. A process can join a group or leave one during system operation. A process can be a member of several groups at the same time. Consequently, mechanisms are needed for managing groups and group membership.

The purpose of introducing groups is to allow a process to deal with collections of other processes as a single abstraction. Thus a process P can send a message to a group $\mathbf{Q} = \{Q_1, \ldots, Q_N\}$ of servers without having to know who they are, how many there are, or where they are, which may change from one call to the next. To P, the group $\mathbf{Q}$ appears to be a single, logical process.

Group organization

An important distinction between different groups has to do with their internal structure. In some groups, all processes are equal. There is no distinctive leader and all decisions are made collectively. In other groups, some kind of hierarchy exists. For example, one process is the coordinator and all the others are workers. In this model, when a request for work is generated, either by an external client or by one of the workers, it is sent to the coordinator. The coordinator then decides which worker is best suited to carry it out, and forwards it there. More complex hierarchies are also possible, of course. These communication patterns are illustrated in Figure 8.4.

Each of these organizations has its own advantages and disadvantages. The flat group is symmetrical and has no single point of failure. If one of the processes crashes, the group simply becomes smaller, but can otherwise continue. A disadvantage is that decision making is more complicated. For example, to decide anything, a vote often has to be taken, incurring some delay and overhead.

The hierarchical group has the opposite properties. Loss of the coordinator brings the entire group to a grinding halt, but as long as it is running, it can make decisions without bothering everyone else. In practice, when the coordinator in a hierarchical group fails, its role will need to be taken over and one of the workers is elected as new coordinator. We discussed leader-election algorithms in Chapter 6.

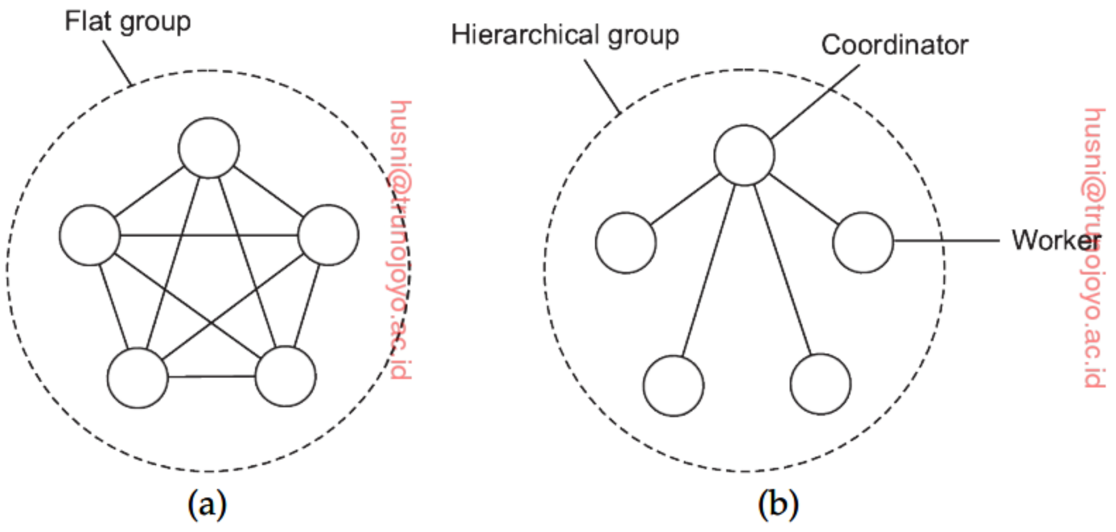
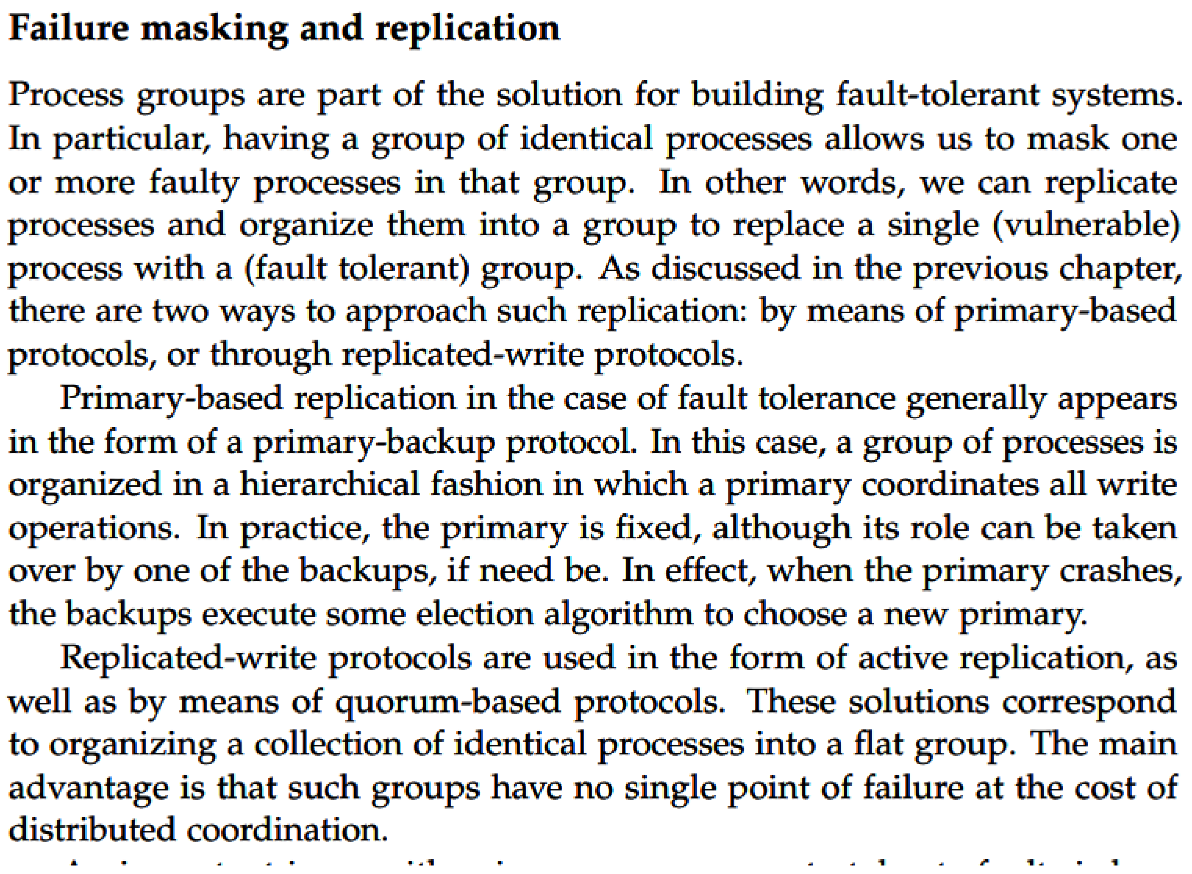


Figure 8.4: Communication in a (a) flat group and in a (b) hierarchical group.

Failure Masking and Replication

By organizing a fault tolerant group of processes , we can protect a single vulnerable process.

Two approaches to arranging the replication of the group:



Failure masking and replication

Process groups are part of the solution for building fault-tolerant systems. In particular, having a group of identical processes allows us to mask one or more faulty processes in that group. In other words, we can replicate processes and organize them into a group to replace a single (vulnerable) process with a (fault tolerant) group. As discussed in the previous chapter, there are two ways to approach such replication: by means of primary-based protocols, or through replicated-write protocols.

Primary-based replication in the case of fault tolerance generally appears in the form of a primary-backup protocol. In this case, a group of processes is organized in a hierarchical fashion in which a primary coordinates all write operations. In practice, the primary is fixed, although its role can be taken over by one of the backups, if need be. In effect, when the primary crashes, the backups execute some election algorithm to choose a new primary.

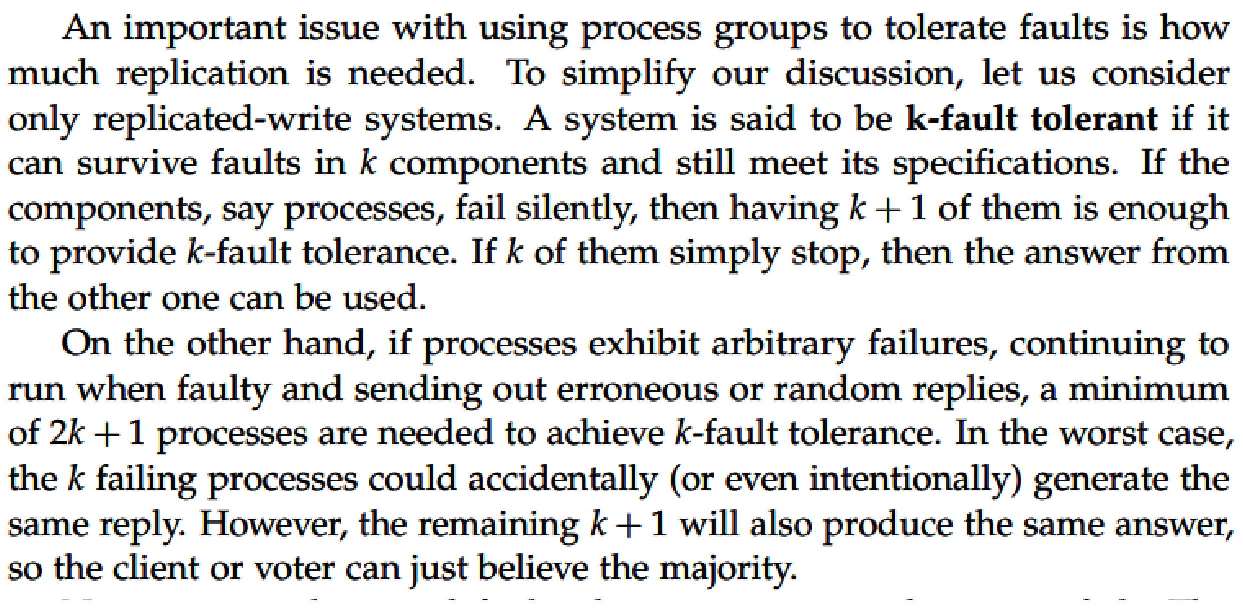
Replicated-write protocols are used in the form of active replication, as well as by means of quorum-based protocols. These solutions correspond to organizing a collection of identical processes into a flat group. The main advantage is that such groups have no single point of failure at the cost of distributed coordination.

Primary (backup) Protocols

- A group of processes is organized in a hierarchical fashion in which a primary coordinates all write operations.
- When the primary crashes, the backups execute some election algorithm to choose a new primary.

Replicated-Write Protocols

- Replicated-write protocols are used in the form of active replication, as well as by means of quorum-based protocols.
- Solutions correspond to organizing a collection of identical processes into a flat group.
- Adv. - these groups have no single point of failure, at the cost of distributed coordination.


An important issue with using process groups to tolerate faults is how much replication is needed. To simplify our discussion, let us consider only replicated-write systems. A system is said to be **k-fault tolerant** if it can survive faults in k components and still meet its specifications. If the components, say processes, fail silently, then having $k+1$ of them is enough to provide k-fault tolerance. If k of them simply stop, then the answer from the other one can be used.

On the other hand, if processes exhibit arbitrary failures, continuing to run when faulty and sending out erroneous or random replies, a minimum of $2k+1$ processes are needed to achieve k-fault tolerance. In the worst case, the k failing processes could accidentally (or even intentionally) generate the same reply. However, the remaining $k+1$ will also produce the same answer, so the client or voter can just believe the majority.

RELIABLE CLIENT-SERVER COMMUNICATION

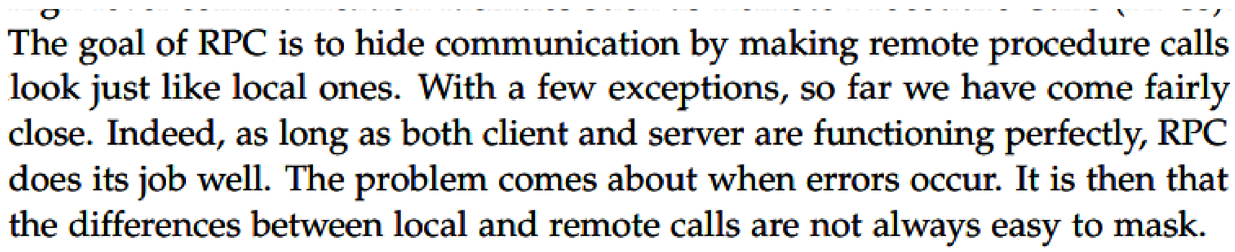
Kinds of Failures in Client-server communication.

- *Crash* (system halts);
- *Omission* (incoming request ignored);
- *Timing* (responding too soon or too late);
- *Response* (getting the order wrong);
- *Arbitrary/Byzantine* (indeterminate, unpredictable).

What is a Remote Procedure Call (RPC)?

Remote Procedure Call is a software communication protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details. RPC is used to call other processes on remote systems like a local system. A procedure call is also sometimes known as a *function call* or a *subroutine call*.

RPC uses the client-server model. The requesting program is a client, and the service-providing program is the server. Like a local procedure call, an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned. However, the use of lightweight processes or threads that share the same address space enables multiple RPCs to be performed concurrently.


The goal of RPC is to hide communication by making remote procedure calls look just like local ones. With a few exceptions, so far we have come fairly close. Indeed, 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.

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.

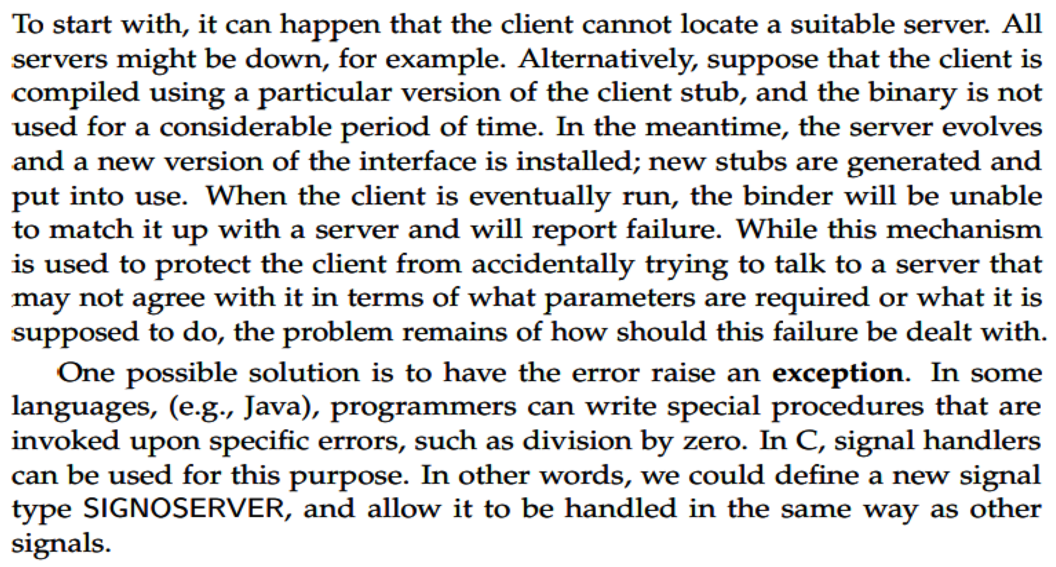
- In practice, actively pinging processes is usually followed.

Five classes of RPC failure can be identified:

1. The client cannot locate the server, so no request can be sent.

   Y and how did this happened ? How can this failure be overcomed ?

   Maybe all servers are down so it can't be located. One solution is to write exception.



To start with, it can happen that the client cannot locate a suitable server. All servers might be down, for example. Alternatively, suppose that the client is compiled using a particular version of the client stub, and the binary is not used for a considerable period of time. In the meantime, the server evolves and a new version of the interface is installed; new stubs are generated and put into use. When the client is eventually run, the binder will be unable to match it up with a server and will report failure. While this mechanism is used to protect the client from accidentally trying to talk to a server that may not agree with it in terms of what parameters are required or what it is supposed to do, the problem remains of how should this failure be dealt with.

One possible solution is to have the error raise an **exception**. In some languages, (e.g., Java), programmers can write special procedures that are invoked upon specific errors, such as division by zero. In C, signal handlers can be used for this purpose. In other words, we could define a new signal type SIGNOSERVER, and allow it to be handled in the same way as other signals.

2. The client's request to the server is lost, so no response is returned by the server to the waiting client.

   → easiest way to handle .

   → just set a timer when a request is sent , if u dont receive a response in the specified time , retransmit it .

Lost request messages

The second item on the list is dealing with lost request messages. This is the easiest one to deal with: just have the operating system or client stub start a timer when sending the request. If the timer expires before a reply or acknowledgment comes back, the message is sent again. If the message was truly lost, the server will not be able to tell the difference between the retransmission and the original, and everything will work fine. Unless, of course, so many request messages are lost that the client gives up and falsely concludes that the server is down, in which case we are back to "Cannot locate server." If the request was not lost, the only thing we need to do is let the server be able to detect it is dealing with a retransmission. Unfortunately, doing so is not so simple, as we explain when discussing lost replies.

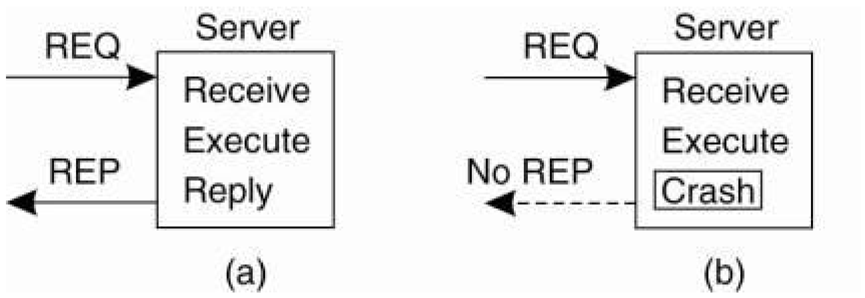
3. The server crashes after receiving the request, and the service request is left acknowledged, but undone.

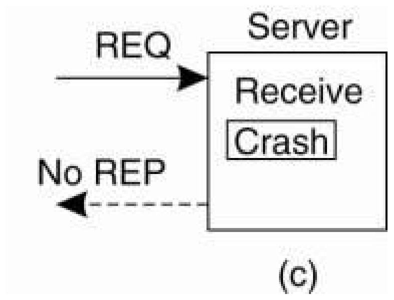
A server in client-server communication.

(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.





Server crashes are dealt with by implementing one of three possible implementation philosophies:

At least once semantics: a guarantee is given that the RPC occurred at least once, but (also) possibly more than once.

At most once semantics: a guarantee is given that the RPC occurred at most once, but possibly not at all.

No semantics: nothing is guaranteed, and clients and servers take their chances!

4. The server's reply is lost on its way to the client, the service has completed, but the results never arrive at the client.

A request that can be repeated any number of times without any nasty side-effects is said to be idempotent.

- (For example: a read of a static web-page is said to be idempotent).

Non Idempotent requests (for example, the electronic transfer of funds) are a little harder to deal with.

Solution to deal with this failure is :

- A common solution is to employ *unique sequence numbers*.
- Another technique is the inclusion of additional bits in a retransmission to identify it as such to the server.

5. The client crashes after sending its request, and the server sends a reply to a newly-restarted client that may not be expecting it.

When a client crashes, and when an 'old' reply arrives, such a reply is known as an *orphan*.

Four orphan solutions have been proposed:

1. ***extermination*** (the orphan is simply killed-off),
2. ***reincarnation*** (each client session has an *epoch* associated with it, making orphans easy to spot),
3. ***gentle reincarnation*** (when a new epoch is identified, an attempt is made to locate a requests owner, otherwise the orphan is killed),
4. ***expiration*** (if the RPC cannot be completed within a standard amount of time, it is assumed to have expired).

In practice, however, none of these methods are desirable for dealing with orphans.

Reliable group Communication.

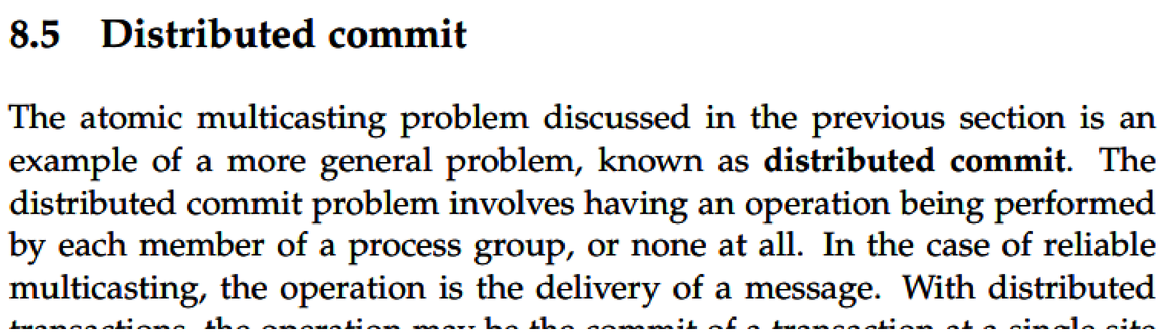
Distributed Commit

Atomic Multicast

Atomic multicast problem:

- A requirement where the system needs to ensure that all processes get the message, or that none of them get it.
- An additional requirement is that all messages arrive at all processes in sequential order.

- Atomic multicasting ensures that non faulty processes maintain a consistent view of the database, and forces reconciliation when a replica recovers and rejoins the group.



8.5 Distributed commit

The atomic multicasting problem discussed in the previous section 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. With distributed transactions, the operation may be the commit of a transaction at a single site

General Goal: *We want 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 and three-phase commit.

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.*

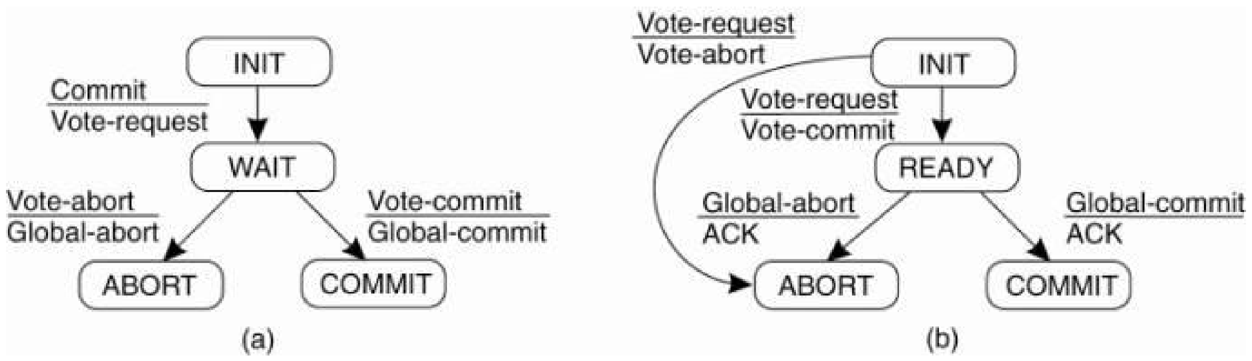
1. The coordinator sends a *VOTE_REQUEST* message to all group members.
2. A group member returns *VOTE_COMMIT* if it can commit locally, otherwise *VOTE_ABORT*.
3. All votes are collected by the coordinator.

- 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.

4. 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

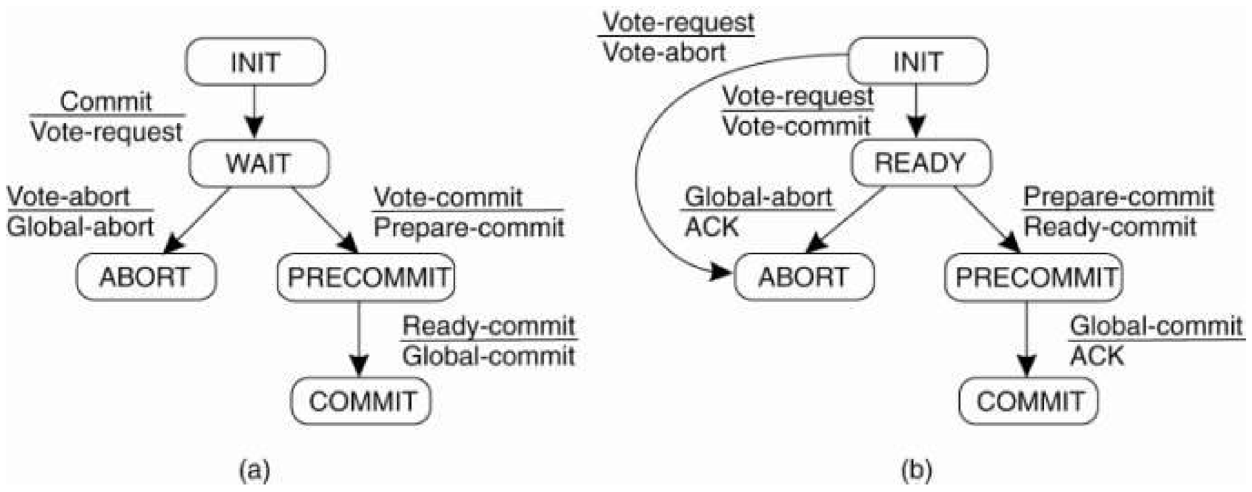
- It can lead to both the coordinator and the group members **blocking**, which may lead to the dreaded *deadlock*.
- If the coordinator crashes, the group members may not be able to *reach a final decision*, and they may, therefore, block until the coordinator *recovers …*
- Two-Phase Commit is known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

Three-Phase Commit Protocol:

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.


INIT
Commit / Vote-request
WAIT
Vote-abort / Global-abort
Vote-commit / Prepare-commit
ABORT
PRECOMMIT
Ready-commit / Global-commit
COMMIT
(a)
Vote-request / Vote-abort
INIT
Vote-request / Vote-commit
READY
Global-abort / ACK
Prepare-commit / Ready-commit
ABORT
PRECOMMIT
Global-commit / ACK
COMMIT
(b)

RECOVERY MECHANISMS.

- 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.
- **Checkpoints**: This is a record of a consistent global state of a distributed system stored in stable storage, also known as **distributed snapshot.** Whenever an error occurs at any stage we can use these in order to restore the system to its previous consistent state.
- **Message logging:**

    - Checkpoint is expensive operation therefore message logging is used to reduce number of checkpoints and still enable recovery
    - Idea is that if the transmission of messages can be replayed, we can still reach a globally consistent state but without having to restore that state from stable storage. Instead,a checkpointed state is taken as the starting point and all messages that have been sent since are simply retransmitted and handled accordingly.
- **Backward Recovery**:
    1. 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.
    2. Disadvantages:
        - Checkpointing (can be very expensive (especially when errors are very rare).
        - 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.