Fault Tolerance

Basic Concepts

Being fault tolerant is strongly related to what are called **dependable systems**.
Dependability includes:

- Availability
- Reliability
- Safety - when a system temporarily fails to operate correctly, no catastrophic event happens.
- Maintainability - refers how easily a failed system can be repaired.

Fault Tolerance

A system can provide its services even in the presence of faults.

Transient faults

They occur once and then disappear. If the operation is repeated, the fault goes away.

Intermittent faults

It vanishes of its own accord, and then reappears and so on. Example: A loose contact on a connector

Permanent faults

Continues to exist until the faulty component is replaced.

Different Type of Failures

Type of Failures	Description of server's behaviour
Crash Failure	Halts, but is working correctly until it halts
Omission Failure *Receive omission* *Send omission*	Fails to respond to incoming requests Fails to receive incoming requests Fails to send messages
Timing Failure	Response lies outside
Response Failure *Value Failure* *State-transition failure*	Response is incorrect The value of the response is wrong Deviates from the correct flow of control
Arbitrary Failure	May produce arbitrary responses at arbitrary time

If a process P does not receive a response from Q:

- If its an **asynchronous** system - no assumptions about execution speeds or message delivery times are made - so it cannot conclude the above.
- If its an **synchronous** system - Can conclude the above.

It is realistic to conclude that a distributed system is **partially synchronous**. For the asynchronous part, it can set a timeout to conclude that a process may have failed.

Halting failures

- **Fail-stop** failures refers to crash failures that can be reliably detected. When a process places a worst-case delay on another process.
- **Fail-noisy** failures - P will eventually come to the conclusion that Q has crashed. P's detections of the behaviour of Q is unreliable.
- **Fail-silent** P cannot distinguish crash failures from omission failures. (if it has crashed, or it could not simply send a response).
- **Fail-safe** these failures are benign; they cannot do any harm.
- **Fail-arbitrary** Q can fail in any possible way, failures may be unobservable, can cause strange behaviour or otherwise correct behaviour.

Failures masking by redundancy

Hide the occurence of failures by using redundancy

Information Redundancy

Extra bits are added to allow recovery from garbled bits.

Time Redundancy

An action is performed, and if need be, it is performed again. Example: Transactions in a database.

Physical Redundancy

Extra equipments or processes are added to make the system tolerable to the failure.

Process Resilience

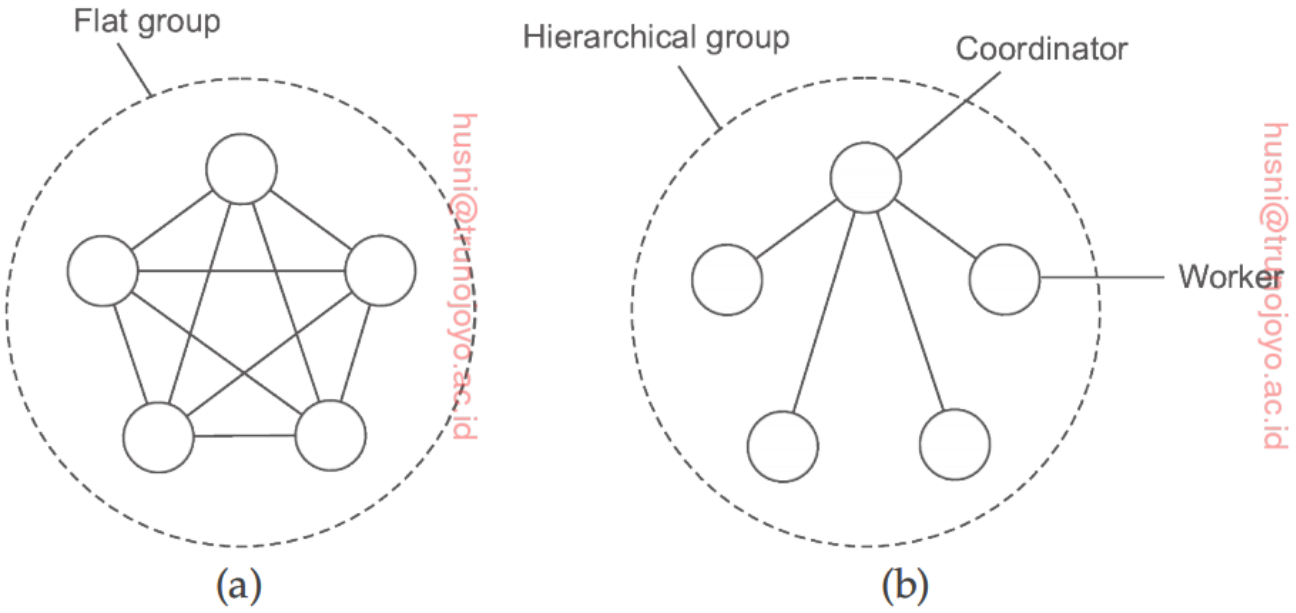
Resilience by process groups

Organize several identical process into a group. A message is sent to all processes in a group. If one process fails, another can take over it. This process is dynamic. New groups can be created or old groups can be destroyed. A process can enter a group or leave a group.

Group Organisation

In some groups, all processes are equal. In others, there may exists a heirarchy. A request coming through is sent to the coordinator, which decides best worker to carry it out.

Flat group = Every process is equal. No coordinator. Voting is done, incurring some delay and overhead. Hierarchical Group = Coordinator. Loss of a coordinator brings the group to a grinding halt.



Membership Management

Group server to which all requests (creating, deleting, joining, etc) are sent. Maintains a complete database of all the groups and their exact management.

Failure masking and replication

We can replace a single process with a fault tolerant group.
Ways of replication:

- Primary based
- Replicated write

A system is said to be k**-fault tolerant** if it can survive faults in k components and still meet its specifications.

Consensus in faulty system with crash failures

In a fault-tolerant process group, each nonfaulty process executes the same commands, in the same order, as every other nonfaulty process.

Formally, this means that the group need to reach a **consensus** on which command to execute. If failures cannot happen, reaching consensus is easy.

$2K + 1$ processes are needed to survive K crashed members.

FLOODING CONSENSUS

Executes in rounds. A process P_i sends its list of proposed command it has seen so far every other process in P. At the end of the round, each process merges all recieved proposed commands into a new list and selects deterministically which command to execute, if possible.

If a process P_2 receives some commands from P_1 and P_1 crashes, P_3 and P_4 are left out to dry. The best they can do is postpone their decision but P_2 can come to a decision and subsequently broadcasts its decision and in the next round P_3 and P_4 can also make a decision then.

Consensus in arbitrary systems

A group in which K members fail due to arbitrary failures, we need atleast $3K + 1$ members to reach consensus.

n processes, 1 primary and $n - 1$ backups

Failures can be detected and a receiver can identify its sender.

In order to achieve **Byzantine agreement**:

- **BA1**: Every nonfaulty backup process stores the same value.
- **BA2**: If the primary is nonfaulty then every nonfaulty backup process stores exactly what the primary sent.

Why $3K$ **processes are not enough**: Suppose $k = 1$
and P is primary and two backups : B_1 and B_2, primary (faulty) may sent 2 different values (T and F) to the backups and from that it is impossible to reach a consensus.

$3K + 1$ **processes are enough**: there can be three backups, the faulty primary has to send two same processes the same value and one can be different (T, T, F) from which a consensus can be reached.

Limitations on realising fault tolerance

3 requirements on reaching consensus:

- Processes must produce the same value
- Every output value must be valid
- Every process must eventually provide an output.

CAP Theorem: Any netwoked system providing shared data can provide only 2 of the below 3 properties:
C: **Consistency**
A: **availability**
P: **Tolerant to partitioning** of a group

Failure Detection

Either processes can send out "Are you alive?" messages to each other or wait for messages to come in from different processes. Can use timeouts or probes (a process probing another process to see if its alive).

Reliable Client-server communication

Point to Point communication

- Established using a reliable transport protocol, such as TCP.
- TCP can mask omission failures in the form of lost messages by sending out acknowledgements and retransmissions
- TCP connection can also fail. What it can do mask it? Establish a new connection.

RPC Semantics

5 different classes of failures in RPC systems:

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

Recovery

Backward Recovery - Main issue is to bring the state from its present erraneous state back to into previously correct state. Each time part of the system's present state is recorded, a **checkpoint** is made.

Forward Recovery - When the system has entered an erraneous state, instead of moving back to the previously checkpointed state, an attemp is made to bring the system in a correct new state from which it can continue to execute. It has to be known which errors may occur in advance. Every error can have a different recovery procedure.

Checkpointing is combined with **message logging**. Becomes less expensive to store state.