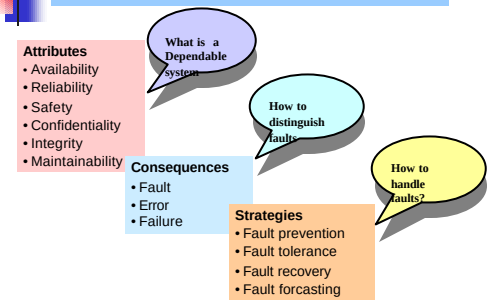
Chapter 9: Fault Tolerance

- Fault Tolerance Basics, Hardware and Software Faults
- Failure Models in Distributed Systems
- Hardware Reliability Modeling
- Fault Tolerance in Distributed Systems
- Static Redundancy: reliability models, TMR
- Agreement in Faulty Systems:
  - Byzantine Generals problem
- Fault Tolerant Services
- Reliable Client-Server Communication
- Reliable Group Communication
- Recovery
  - Checkpointing
  - Message Logging

Concepts of Fault Tolerance

- Hardware, software and networks cannot be totally free from failures
- Fault tolerance is a non-functional (QoS) requirement that requires a system to continue to operate, even in the presence of faults
- Fault tolerance should be achieved with minimal involvement of users or system administrators
- Distributed systems can be more fault tolerant than centralized systems, but with more processor hosts generally the occurrence of individual faults is likely to be more frequent
- Notion of a partial failure in a distributed system

Attributes, Consequences and Strategies



Attributes of a Dependable System

- **System attributes:**
- **Availability** – system always ready for use, or probability that system is ready or available at a given time
- **Reliability** – property that a system can run without failure, for a given time
- **Safety** – indicates the safety issues in the case the system fails
- **Maintainability** – refers to the ease of repair to a failed system
- Failure in a distributed system = when a service cannot be fully provided
- System failure may be partial
- A single failure may affect other parts of a system (failure escalation)

Terminology of Fault Tolerance

Fault – is a defect within the system

Error – is observed by a deviation from the expected behaviour of the system

Failure occurs when the system can no longer perform as required (does not meet spec)

Fault Tolerance – is ability of system to provide a service, even in the presence of errors

Types of Fault (*wrt* time)

Hard or Permanent – repeatable error, e.g. failed component, power fail, fire, flood, design error (usually software), sabotage

Soft Fault

Transient – occurs once or seldom, often due to unstable environment (e.g. bird flies past microwave transmitter)

Intermittent – occurs randomly, but where factors influencing fault are not clearly identified, e.g. unstable component

Operator error – human error

Types of Fault (*wrt* attributes)

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Amnesia crash	Lost all history, must be reboot
Pause crash	Still remember state before crash, can be recovered
Halting crash	Hardware failure, must be replaced or re-installed
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	The 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

Strategies to Handle Faults

- Fault avoidance
  - Techniques aim to **prevent** faults from entering the system during design stage
- Fault removal
  - Methods attempt to **find** faults within a system before it enters service
- Fault detection
  - Techniques used during service to **detect** faults within the operational system
- Fault tolerant
  - Techniques designed to **tolerant** faults, i.e. to allow the system operate correctly in the presence of faults.

Actions to identify and remove errors:
- **Design reviews**
- **Testing**
- **Use certified tools**
- **Analysis:**
  - **Hazard analysis**
  - **Formal methods - proof & refinement**

No non-trivial system can be guaranteed free from error
Must have an expectation of failure and make appropriate provision

Architectural approaches

Simplex systems
- highly reliable components

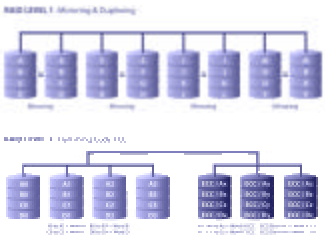
Dual Systems
- twin identical
- twin **dissimilar**
- control + monitor

N-way Redundant systems
- identical / dissimilar
- self-checking / voting

Dissimilar systems are also known as "diverse" systems in which an operation is performed in a different way in the hope that the same fault will not be present in different implementations.

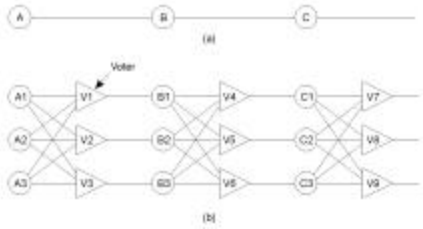
The basic approach to achieve fault tolerance is redundancy

Example: RAID (Redundant Array of Independent Disks)



RAID has been classified into several levels: 0, 1, 2, 3, 4, 5, 6, 10, 50, each level provides a different degree of fault tolerance

Failure Masking by TMR



(a) Original circuit
(b) Triple modular redundancy

Example: Space Shuttle

- Uses 5 identical computers which can be assigned to redundant operation under program control.
- During critical mission phases - boost, re-entry and loading - 4 of its 5 computers operate an NMR configuration, receiving the same inputs and executing identical tasks. When a failure is detected the computer concerned is switched out of the system leaving a TMR arrangement.
- The fifth computer is used to perform non-critical tasks in a simplex mode, however, under extreme cases may take over critical functions. The unit has "diverse" software and could be used if a systematic fault was discovered in the other four computers.
- The shuttle can tolerate up to two computer failures; after a second failure it operates as a duplex system and uses comparison and self-test techniques to survive a third fault.

Forms of redundancy

- Hardware redundancy
  - Use more hardware
- Software redundancy
  - Use more software
- Information redundancy, e.g.
  - Parity bits
  - Error detecting or correcting codes
  - Checksums
- Temporal (time) redundancy
  - Repeating calculations and comparing results
  - For detecting transient faults

Software Faults

❑ **Program code (may) contains bugs if actual behavior disagrees with the intended specification. These faults may occur from:**

- **specification error**
- **design error**
- **coding error, e.g. use on un-initialized variables**
- **integration error**
- **run time error e.g. operating system stack overflow, divide by zero**

❑ **Software failure is (usually) deterministic, i.e. predictable, based on the state of the system. There is no random element to the failure – unless the system state cannot be specified precisely. A non-deterministic fault behavior usually indicates that the relevant system state parameters have not been identified.**

❑ **Fault coverage – defines the fraction of possible faults that can be detected by testing (statement, condition or structural analysis)**

Software Fault Tolerance

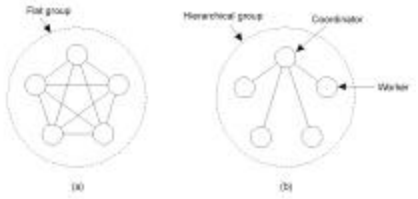
N-version programming

- Use several different implementations of the same specification
- The versions may run *sequentially* on one processor or *in parallel* on different processors.
- They use the same input and their results are compared.
  - In the absence of a disagreement, the result is output.
  - When produced different results:
    - If there are 2 routines:
      - the routines may be repeated in case this was a transient error;
      - to decide which routine is in error.
    - If there are 3 or more routines,
      - voting may be applied to mask the effects of the fault.

Process Groups

- Organize several identical processes into a group
- When a message is send to a group, all members of the group receives it
- If one process in a group fails (no matter what reason), hopefully some other process can take over for it
- The purpose of introducing groups is to allow processes to deal with collections of processes as a single abstraction.
- Important design issue is how to reach agreement within a process group when one or more of its members cannot be trusted to give correct answers.

Process Group Architectures



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

Fault Tolerant in Process Group

- A system is said to be k fault tolerant if it can survive faults in k components and still meets its specification.
- If the components (processes) fail silently, then having k + 1 of them is enough to provide k fault tolerant.
- If processes exhibit Byzantine failures (continuing to run when sick and sending out erroneous or random replies, a minimum 2k + 1 processes are needed.
- If we demand that a process group reaches an agreement, such as electing a coordinator, synchronization, etc., we need even more processes to tolerate faults .

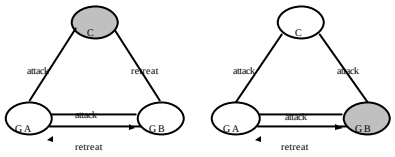
Distributed Agreement

- A Distributed Agreement is a process to reach an agreement among non-faulty processes within finite steps.
- Suppose that n processes, $P = \{p_1, p_2, ..., p_n\}$, try to establish an agreement. let $p_i \in P$ has an initial value V_i? If the set F contains all the faulty processes, then $(P - F)$ is the set of non-faulty processes?
- The goal of a distributed agreement algorithm is to let every process p_i of P to calculate an agreement value A_i, such that the following conditions hold:
- (1)Let p_i, p_j $(i \neq j; i, j = 1..n)$ be arbitrary processes, if $p_i, p_j \in (P - F)$, then $A_i = A_j$, and we call this value as an agreement value.
- (2) An agreement value of $(P - F)$ if the function of $\{V_i\}$.

Byzantine Generals Problem

- N armies were called to meet the invading enemy. Each of the armies was led by its own general.Each general had his own preference about whether to attack. Since the attacks had to be coordinated, the generals sent messengers to each other.
- The disloyal generals would attempt to deceive the loyal generals to prevent a coordinated attack. The loyal generals therefore agreed to follow a protocol that ensure a distributed agreement.
- Each general p_i will transmit its opinion V_i to others subject to the following interactive consistency conditions:
- (1) if sender p_s is loyal, the loyal generals will agree V_s;
  (2) if the sender p_s a traitor, the loyal generals will agree on the same value for V_s.

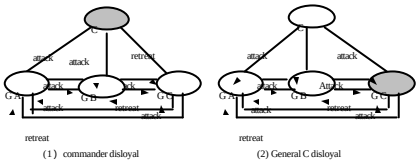
Three Generals



(1) commander disloyal (2) G B disloyal

- Since the generals can come to agreement if each can reliably broadcast its opinion, we will focus on a single broadcast, and let the sender be the commander general. To solve the Byzantine generals problem, each general takes his turn as commander.
- G A can't follow C's order, because if C is disloyal, A and B would obey different orders; G A can't disobey the order either, because C might be loyal. There, G A can not make a decision.

Four Generals



(1) commander disloyal (2) General C disloyal

- Suppose there are four generals where one is disloyal. The commander broadcasts his order to each general. Next, each general tells the other two the order he received from C. The general then obeys the majority opinion.
- Suppose that there are t traitors among M generals, then we can prove: if M <= 3t, the system can not reach agreement.

Reliable Communication

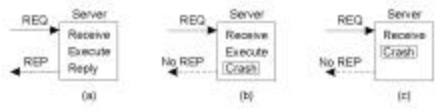
- Fault Tolerance in Distributed system must consider communication failures.
- A communication channel may exhibit crash, omission, timing, and arbitrary failures.
- Reliable P2P communication is established by a reliable transport protocol, such as TCP.
- In client/server model, RPC/RMI semantics must be satisfied in the presence of failures.
- In process group architecture or distributed replication systems, a reliable multicast/broadcast service is very important.

Reliable Client-Server Communication

In the case of process failure the following situations need to be dealt with:

- **Client unable to locate server**
- **Client request to server is lost**
- **Server crash after receiving client request**
- **Server reply to client is lost**
- **Client crash after sending server request**

Lost Request Messages when Server Crashes



A server in client-server communication

a) Normal case
b) Crash after execution
c) Crash before execution

Solutions to Handle Server Failures (1)

➢ **Client unable to locate server, e.g. server down, or server has changed**

Solution:

- Use an exception handler – but this is not always possible in the programming language used

➢ **Client request to server is lost**

Solution:

- Use a timeout to await server reply, then re-send – but be careful about idempotent operations (no side effects when re-send)

- If multiple requests appear to get lost assume 'cannot locate server' error

Solutions to Handle Server Failures (2)

➢ **Server crash after receiving client request**

Problem may be not being able to tell if request was carried out (e.g. client requests print page, server may stop before or after printing, before acknowledgement)

Solutions:

- rebuild server and retry client request (assuming 'at least once' semantics for request)

- give up and report request failure (assuming 'at most once' semantics), what is usually required is exactly once semantics, but this difficult to guarantee

➢ **Server reply to client is lost**

Client can simply set timer and if no reply in time assume server down, request lost or server crashed during processing request.

Solutions to Handle Client Failures

➢ **Client crash after sending server request** : Server unable to reply to client (orphan request)

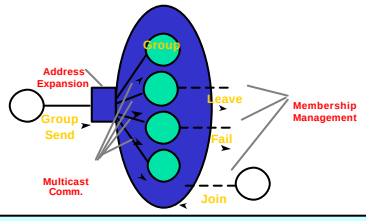
Options and Issues:

- **Extermination**: client makes a log of each RPC, and kills orphan after reboot. Expensive.

- **Reincarnation**. Time divided into epochs (large intervals). When client restarts it broadcasts to all, and starts a new time epoch. Servers dealing with client requests from a previous epoch can be terminated. Also unreachable servers (e.g. in different network areas) may later reply, but will refer to obsolete epoch numbers.

- **Gentle reincarnation**, as above but an attempt is made to contact the client owner (e.g. who may be logged out) to take action Expiration, server times out if client cannot be reached to return reply

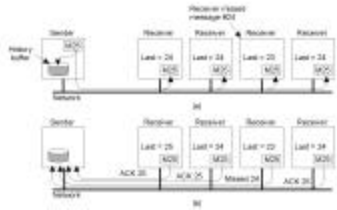
Group Communication



Static Groups: group membership is pre-defined
Dynamic Groups: Members may join and leave, as necessary

Member = process (or coordinator or RM Replica Manager)

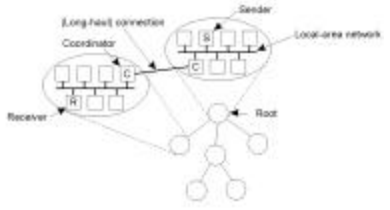
Basic Reliable-Multicasting



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

a) Message transmission
b) Reporting feedback

Hierarchical Feedback Control



The essence of hierarchical reliable multicasting (best for large process groups.

a) Each local coordinator forwards the message to its children.
b) A local coordinator handles retransmission requests.

Group View (1)

- A *group membership service* maintains group *views*, which are lists of current group members.
  - This is NOT a list maintained by a one member, but…
  - *Each member maintains its own view (thus, views may be different across members)*
- A view $V_p(g)$ is process p's understanding of its group (list of members)
  - Example: $V_{p,0}(g) = \{p\}$, $V_{p,1}(g) = \{p, q\}$, $V_{p,2}(g) = \{p, q, r\}$, $V_{p,3}(g) = \{p,r\}$
- A new group view is generated, throughout the group, whenever a member joins or leaves.
  - Member detecting failure of another member reliable multicasts a "view change" message (causal-total order)

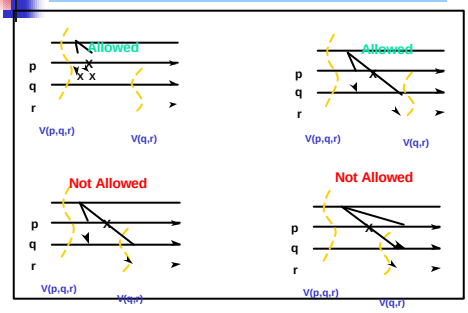
Group View (2)

- An event is said to occur in a view $v_{p,i}(g)$ *if the event occurs at p, and at the time of event occurrence, p has delivered* $v_{p,i}(g)$ *but has not yet delivered* $v_{p,i+1}(g)$.
- Messages sent out in a view *i* need to be delivered in that view at *all* members in the group ("What happens in the View, stays in the View")
- Requirements for view delivery
  - Order: If *p* delivers $v_i(g)$ and then $v_{i+1}(g)$, then no other process *q* delivers $v_{i+1}(g)$ before $v_i(g)$.
  - Integrity: If *p* delivers $v_i(g)$, then *p* is in $v_i(g)$.
  - Non-triviality: if process q joins a group and becomes reachable from process p, then eventually q will always be present in the views that delivered at p.

Virtual Synchronous Communication (1)

- Virtual Synchronous Communication = Reliable multicast + Group Membership
- The following guarantees are provided for multicast messages:
  - Integrity : If *p* delivers message *m*, *p* does not deliver *m* again. Also $p \in group(m)$.
  - Validity : Correct processes always deliver all messages. That is, if p delivers message m in view $v(g)$, and some process q $\in v(g)$ does not deliver m in view $v(g)$, then the next view $v'(g)$ delivered at *p* will exclude *q*.
  - Agreement: Correct processes deliver the same set of messages in any view.
  - All View Delivery conditions (Order, Integrity and Non-triviality conditions, from last slide) are satisfied
- "What happens in the View, stays in the View"

Virtual Synchronous Communication (2)



Virtual Synchronous Communication (3)

Multicast	Basic Message Ordering	Total-ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Six different versions of virtually synchronous reliable multicasting

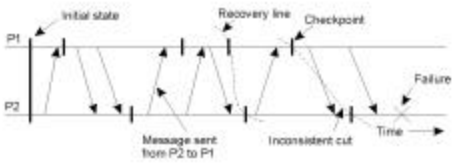
Recovery Techniques

- Once failure has occurred in many cases it is important to recover critical processes to a known state in order to resume processing
- Problem is compounded in distributed systems

Two Approaches:

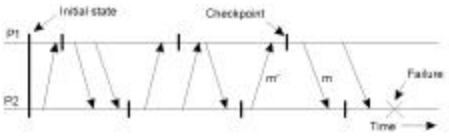
- Backward recovery, by use of checkpointing (global snapshot of distributed system status) to record the system state but checkpointing is costly (performance degradation)
- Forward recovery, attempt to bring system to a new stable state from which it is possible to proceed (applied in situations where the nature if errors is known and a reset can be applied)

Checkpointing



A recovery line is a distributed snapshot which records a consistent global state of the system

Independent Checkpointing



If these local checkpoints jointly do not form a distributed snapshot, the cascaded rollback of recovery process may lead to what is called the domino effect.

Possible solution is to use globally coordinated checkpointing – which requires global time synchronization rather than independent (per processor) checkpointing

Backward Recovery

- most extensively used in distributed systems and generally safest
- can be incorporated into middleware layers
- no guarantee that same fault may occur again (deterministic view – affects failure transparency properties)
- can not be applied to irreversible (non-idempotent) operations, e.g. ATM withdrawal or UNIX **rm ***

Forward Recovery (Exception)

- Exceptions
  - System states that should not occur
  - Exceptions can be defined either
    - predefined (e.g. array-index out of bounds, divide by zero)
    - explicitly declared by the programmer
- Raising an exception
  - When such a state is detected in the execution of the program
  - The action of indicating occurrence of such as state
- Exception handler
  - Code to be executed when an exception is raised
  - Declared by the programmer
  - For recovery action
- Supported by several programming languages
  - Ada, ISO Modula-2, Delphi, Java, C++.