Fault Tolerance

Basics

✓ A component provides services to clients. To provide services, the component may require the services from other components ⇒ a component may depend on some other component.

✓ Specifically: A component C depends on C' if the correctness of C's behavior depends on the correctness of C'’s behavior.
Dependable Systems

Dependable systems are characterized by

- **Availability**
  - % of time system may be used immediately
- **Reliability**
  - Mean time between failures
- **Safety**
  - How serious is the impact of a failure
- **Maintainability**
  - How long does it take to repair the system
- **Security**

Definitions

- **A system fails** when it does not perform according to its specification.
- **An error** is part of a system state that may lead to a failure.
- **A fault** is the cause of an error.

**Note:** For distributed systems, components can be either processes or channels.

**A fault tolerant system does not fail in the presence of faults.**
Server Failure Models

<table>
<thead>
<tr>
<th>Type of failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>A server halts, but is working correctly until it halts</td>
</tr>
<tr>
<td>Omission failure</td>
<td></td>
</tr>
<tr>
<td>Receive omission</td>
<td>A server fails to respond to incoming requests</td>
</tr>
<tr>
<td>Send omission</td>
<td>A server fails to receive incoming messages</td>
</tr>
<tr>
<td>Timing failure</td>
<td>A server's response lies outside the specified time interval</td>
</tr>
<tr>
<td>Response failure</td>
<td></td>
</tr>
<tr>
<td>Value failure</td>
<td>The server's response is incorrect</td>
</tr>
<tr>
<td>State transition failure</td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td>Arbitrary failure</td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>

Observation: Crash failures are the least severe; arbitrary failures are the worst

Crash Failure (1)

 ✓ Problem: Clients cannot distinguish between a crashed component and one that is just a bit slow

 ✓ Examples: Consider a server from which a client is expecting output:
  o Is the server perhaps exhibiting timing or omission failures
  o Is the channel between client and server faulty (crashed, or exhibiting timing or omission failures)
Crash Failure (2)

✓ Fail-silent: The component exhibits omission or crash failures; clients cannot tell what went wrong

✓ Fail-stop: The component exhibits crash failures, but its failure can be detected (either through announcement or timeouts)

✓ Fail-safe: The component exhibits arbitrary, but benign failures (they can't do any harm)

Masking Errors with Redundancy

✓ Information redundancy - Hamming code

✓ Time redundancy - timeout and retry

✓ Physical redundancy - triple modular redundancy
Processes

- Process Groups
- Replicated processes
- Reaching agreement

**Basic issue:** Protect yourself against faulty processes by replicating and distributing computations in a group.

Flat and Hierarchical Groups

- **Flat group**
  - All processes are equal
  - Complex decision making
  - Requires multicast

- **Hierarchical group**
  - Communicate with coordinator
  - Coordinator makes decisions
  - Single point of failure
Replicated Processes

✓ Fault tolerance can be achieved by
  o Primary backup
  o Replicated write

✓ K fault tolerant
  o System can tolerate k faulty components
  o To tolerate fail-stop requires k+1
  o To tolerate byzantine requires 2k+1

Byzantine Generals

The generals announce their troop strengths (in units of 1 kilosoldiers).

The vectors that each general assembles

The vectors that each general receives in step 3.
Agreement in Faulty Systems

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

Reliable Communication

So far: Concentrated on process resilience (by means of process groups). What about reliable communication channels?

- **Error detection:**
  - Framing of packets to allow for bit error detection
  - Use of frame numbering to detect packet loss

- **Error correction:**
  - Add so much redundancy that corrupted packets can be automatically corrected
  - Request retransmission of lost, or last N packets

**Observation:** Most of this work assumes point-to-point communication
Reliable RPC/RMI?

✓ Client unable to locate the server
  o Relatively simple - just report back to client
✓ Invocation from client to server is lost
  o Just resend message
✓ Server crashes after receiving request
✓ Reply message from server is lost
✓ Client crashes after sending an invocation

Server Crashes

Server crashes are harder as you don’t what it had already done:

✓ At least once - retry until ack received
✓ At most once - report failure
✓ Exactly once - ideal target
Reliable Communication

- Client unable to locate the server
- Invocation from client to server is lost
- Server crashes after receiving request
- Reply message from server is lost
- Client crashes after sending an invocation

Reliable RPC/RMI

Detecting lost replies can be hard, because it can also be that the server had crashed. You don’t know whether the server has carried out the operation.

**Solution:** None, except that you can try to make your operations idempotent:
- repeatable without any harm done if it happened to be carried out before.
Reliable RPC/RMI

Problem: The server is doing work and holding resources for nothing (called doing an orphan computation).

- Orphan is killed (or rolled back) by client when it reboots
- Broadcast new epoch number when recovering ⇒ servers kill orphans
- Require computations to complete in a time $T$ units. Old ones are simply removed.

Reliable Multicast

- Multicast can be useful to groups of replicas
- What is reliable multicast?
  - Message order
  - Interaction with group membership
  - Failing and recovering members
**Basic Reliable Multicast**

- Known fixed group
- No failures
- Underlying unreliable multicast

![Diagram of Basic Reliable Multicast](image)

**Scalable Reliable Multicasting: Feedback Suppression**

**Basic idea:** Let a process $P$ suppress its own feedback when it notices another process $Q$ is already asking for a retransmission

**Assumptions:**
- All receivers listen to a common feedback channel to which feedback messages are submitted
- Process $P$ schedules its own feedback message randomly, and suppresses it when observing another feedback message

![Diagram of Scalable Reliable Multicasting](image)
**Multicast Feedback**

- Only negative acknowledgements
- Multicast NACKs suppress same
- Retransmit also multicast

**Hierarchical Feedback Control**

- Local coordinator
  - Forwards the message to its children.
  - Handles retransmission requests.
**Atomic Multicast**

- Message is delivered to all members of a group or to none.
- Messages are delivered in same order to all members.
- What does it mean for a member to join or leave the group?
- Simplifies higher level applications.

---

**Virtual Synchrony**

- Message is delivered to application
- Message is received by communication layer
- Message comes in from the network

Distinguish between message receipt and message delivery
Virtual Synchrony

P1 joins  P3 crashes  P3 rejoins

G={P1,P2,P3,P4}  G={P1,P2,P4}  G={P1,P2,P3,P4}

Principle of virtual synchronous multicast

Ordered Multicast

- **FIFO ordering**: If a correct process issues `multicast(g, m_1)` followed by `multicast(g,m_2)` then every correct process that delivers `m_2` will deliver `m_1` before `m_2`.
- **Causal ordering**: If `multicast(g, m_1)` happened before `multicast(g, m_2)` then any correct process that delivers `m_2` will deliver `m_1` before `m_2`.
- **Total ordering**: If a correct process delivers `m_1` before it delivers `m_2`, then any other correct process that delivers `m_2` will deliver `m_1` before `m_2`.

Ordered Multicast
FIFO Ordering

Causal Ordering
Total Causal Ordering

Six Reliable Multicasts

<table>
<thead>
<tr>
<th>Multicast</th>
<th>Basic Message Ordering</th>
<th>Total-ordered Delivery?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliable multicast</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>FIFO multicast</td>
<td>FIFO-ordered delivery</td>
<td>No</td>
</tr>
<tr>
<td>Causal multicast</td>
<td>Causal-ordered delivery</td>
<td>No</td>
</tr>
<tr>
<td>Atomic multicast</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>FIFO atomic multicast</td>
<td>FIFO-ordered delivery</td>
<td>Yes</td>
</tr>
<tr>
<td>Causal atomic multicast</td>
<td>Causal-ordered delivery</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**Totally Ordered Requests (ISIS)**

- **P1**
- **P2**
- **P3**
- **P4**

1. **1 Message**
2. **2 Proposed Seq**
3. **3 Agreed Seq**

---

**Implementing Virtual Synchrony**

1. **Process 4 notices that process 7 has crashed, sends a view change**
2. **Process 6 sends out all its unstable messages, followed by a flush message**
3. **Process 6 installs the new view when it has received a flush message from everyone else**
Two-Phase Commit (1)

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

Two-Phase Commit (2)

<table>
<thead>
<tr>
<th>State of Q</th>
<th>Action by P</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Make transition to COMMIT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>INIT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>READY</td>
<td>Contact another participant</td>
</tr>
</tbody>
</table>

Actions taken by a participant $P$ when residing in state $READY$ and having contacted another participant $Q$. 
Two-Phase Commit (3)

Outline of the steps taken by the coordinator in a two phase commit protocol

Two-Phase Commit (4)

Steps taken by participant process in 2PC.
Two-Phase Commit (5)

actions for handling decision requests: /* executed by separate thread */
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip; /* participant remains blocked */
}

Steps taken for handling incoming decision requests.

Three-Phase Commit

a) Finite state machine for the coordinator in 3PC
b) Finite state machine for a participant
Recovery Stable Storage

(a) Stable Storage
(b) Crash after drive 1 is updated
(c) Bad spot

Checkpointing

A recovery line.
**Independent Checkpointing**

The domino effect.

**Message Logging**

Incorrect replay of messages after recovery, leading to an orphan process.