Fault Tolerance
**Basic Concepts**

- **Dependability includes**
  - **Availability** = probability the system is operating correctly at any point in time
  - **Reliability** = time period the system is working correctly
    - Availability is different than reliability
  - **Safety** = are catastrophic events possible in case of failures
  - **Maintainability** = effort to repair failed system

- **A system fails when it does not meet its promises**
- **Error** = part of a system’s state that may lead to a failure
- **Fault** = the cause of an error
Fault types

Faults can be

- Transient faults that occur once and then disappear
- Intermittent faults that occur, disappear, and then reappear
- Permanent faults continue to exist until the system is repaired
## Failure Models

### Different types of failures.

<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>A server fails to respond to incoming requests</td>
</tr>
<tr>
<td>Receive omission</td>
<td>A server fails to receive incoming messages</td>
</tr>
<tr>
<td>Send omission</td>
<td>A server fails to send 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>The server's response is incorrect</td>
</tr>
<tr>
<td>Value failure</td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td>State transition failure</td>
<td>The server deviates from the correct flow of control</td>
</tr>
<tr>
<td>Arbitrary (Byzantine) failure</td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>

- Fail-stop failures = server simply stops responding
- Fail-silent servers do not inform others their intent to fail
- Fail-safe faults = arbitrary faults that others can easily detect
Failure Masking by Redundancy

- Triple Modular Redundancy (TRM)
Process Resiliency

- Organize multiple identical processes in a group for the purpose of tolerating faults
- Process groups can be dynamic and have their internal organization
  - Flat vs hierarchical groups
  - Managing the membership of a group is a critical issue
Flat versus Hierarchical Process Groups

(a) Flat group

(b) Hierarchical group

Coordinator

Worker
Fault Masking in process groups

- Process groups provide fault tolerance by replication
- How much replication is needed?
  - A system is $k$ fault tolerant if it can tolerate $k$ faults
- In Fail-stop and fail-silent systems $k+1$ replication tolerates $k$ faults
- In Byzantine failures $2k+1$ replicas are needed to tolerate $k$ faults
**Agreement in Faulty Systems**

- **Circumstances under which distributed agreement is possible**

<table>
<thead>
<tr>
<th></th>
<th>Unordered Msgs</th>
<th>Ordered Msgs</th>
<th>Bounded</th>
<th>Communication delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous processes</td>
<td></td>
<td>X</td>
<td></td>
<td>Bounded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>Unbounded</td>
<td></td>
</tr>
<tr>
<td>Asynchronous processes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>Bounded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>Unbounded</td>
</tr>
<tr>
<td>Unicast</td>
<td></td>
<td>Multicast</td>
<td>Unicast</td>
<td>Multicast</td>
</tr>
<tr>
<td>Message transmission</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Lamport’s algorithm for the Byzantine agreement problem tolerates k faults with 3k+1 replication**
Agreement in Faulty Systems

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

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

b) The vectors that each general assembles based on (a)

c) The vectors that each general receives in step 3.

d) Each general decides the troop strength of the others, using the majority value (or unknown if no majority exists), e.g., all decide/agree on (1, 2, ?, 4)
Agreement in Faulty Systems

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

- How to detect failures?
  - Actively ping a process
  - Passively wait until you hear from process
- Distinguish node failures from network failures
- How to inform non-faulty nodes of a failure?
Failures in RPCs

Types of failures and treatment

- Client can not locate the server
  - Raise an exception to handle it
- Client crashes
  - Need to deal with orphan calls
    - Orphan extermination: make log and kill all orphans upon recovery
    - Reincarnation: upon recovery start new epoch, kill all older calls
    - Gentle reincarnation: kill only orphan calls despite being old
    - Expiration: each call is given a quantum, and needs to request a another quantum from parent if needed
- Lost request messages
  - Retransmit request?
- Server crashes
- Lost replies
Server Crashes in an RPC call

- A server in client-server communication
  - a) Normal case
  - b) Crash after execution
  - c) Crash before execution
Server Crashes in an RPC call

- What to do?
  - At least once semantics
    - Reissue request
  - At most once semantics
    - Report failure but do not reissue request
  - Exactly once semantics
    - In general, impossible
    - Consider the case where client request server to “print”
**Server Crashes in RPC**

All the different combinations of client and server strategies in the presence of server crashes in an RPC call to “print”, where:
- M = send completion message
- P = print
- C = crash

<table>
<thead>
<tr>
<th>Reissue strategy</th>
<th>Client</th>
<th>Strategy M -&gt; P</th>
<th>Server</th>
<th>Strategy P -&gt; M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MPC</td>
<td>MC(P)</td>
<td>PC(M)</td>
</tr>
<tr>
<td>Always</td>
<td></td>
<td>DUP</td>
<td>OK</td>
<td>DUP</td>
</tr>
<tr>
<td>Never</td>
<td></td>
<td>OK</td>
<td>ZERO</td>
<td>OK</td>
</tr>
<tr>
<td>Only when ACKed</td>
<td></td>
<td>DUP</td>
<td>OK</td>
<td>ZERO</td>
</tr>
<tr>
<td>Only when not ACKed</td>
<td></td>
<td>OK</td>
<td>ZERO</td>
<td>OK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>MPC</th>
<th>PC(M)</th>
<th>C(PM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td>DUP</td>
<td>DUP</td>
<td>OK</td>
</tr>
<tr>
<td>Never</td>
<td>OK</td>
<td>OK</td>
<td>ZERO</td>
</tr>
<tr>
<td>Only when ACKed</td>
<td>DUP</td>
<td>OK</td>
<td>ZERO</td>
</tr>
<tr>
<td>Only when not ACKed</td>
<td>OK</td>
<td>DUP</td>
<td>OK</td>
</tr>
</tbody>
</table>
Basic Reliable-Multicasting Schemes

- A simple solution to reliable multicasting when all receivers are known and are assumed not to fail
  a) Message transmission
  b) Reporting feedback
Reliable Group Communications

- Reliable delivery of a message to a group
  - Need to decide group membership
  - Handle node failures during message transmission
- In the basic scheme, we assume that no processes fail and that no processes leave/join the process group
  - Sender logs the msg and waits for ACKs or NACKs
  - Server discards msg if it received ACK from everybody, and retransmits when it receives a NACK
  - However sender receives too many feedback msgs (feedback implosion)
- Use feedback supresion (by multicasting NACK to the group, and having group members refrain from sending yet another NACK upon receiving a NACK)
Nonhierarchical Feedback Control

Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others.
Hierarchical Feedback Control

The essence of hierarchical reliable multicasting.

a) Each local coordinator forwards the message to its children in its process subgroup and any children coordinators.

b) A local coordinator handles retransmission requests.
Virtual Synchrony (1)

- The logical organization of a distributed system to distinguish between message receipt and message delivery

Diagram:
- Message comes in from the network
- Message is received by communication layer
- Message is delivered to application
- Network
- Local OS
- Communication layer
- Application
**Virtual Synchrony (2)**

- The principle of virtual synchronous multicast.
### Message Ordering (1)

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sends m1</td>
<td>receives m1</td>
<td>receives m2</td>
</tr>
<tr>
<td>sends m2</td>
<td>receives m2</td>
<td>receives m1</td>
</tr>
</tbody>
</table>

Three communicating processes in the same group. The ordering of events per process is shown along the vertical axis.
### Message Ordering (2)

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
<th>Process P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>sends m1</td>
<td>receives m1</td>
<td>receives m3</td>
<td>sends m3</td>
</tr>
<tr>
<td>sends m2</td>
<td>receives m3</td>
<td>receives m1</td>
<td>sends m4</td>
</tr>
<tr>
<td></td>
<td>receives m2</td>
<td>receives m2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>receives m4</td>
<td>receives m4</td>
<td></td>
</tr>
</tbody>
</table>

- Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting
Implementing Virtual Synchrony (1)

Six different versions of virtually synchronous reliable multicasting.

<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>
Implementing Virtual Synchrony (2)

(a) Process 4 notices that process 7 has crashed, sends a view change
(b) Process 6 sends out all its unstable messages, followed by a flush message
(c) 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 $Q$. 

actions by coordinator:

while START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        while GLOBAL_ABORT to local log;
        multicast GLOBAL_ABORT to all participants;
        exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT{
    write GLOBAL_COMMIT to local log;
multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT to local log;
multicast GLOBAL_ABORT to all participants;
}

- Outline of the steps taken by the coordinator in a two phase commit protocol
Two-Phase Commit (4)

actions by participant:

write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
        write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
    send VOTE_ABORT to coordinator;
}
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.