Chapter 8
Fault Tolerance
Fault Tolerance Basic Concepts

- Being fault tolerant is strongly related to what are called dependable systems
- Dependability implies the following:
  - Availability
  - Reliability
  - Safety
  - Maintainability

Types of faults: *Transient, Intermittent, Permanent*
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>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>A 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 failure</td>
<td>A server may produce arbitrary responses at arbitrary times</td>
</tr>
</tbody>
</table>

Figure 8-1. Different types of failures.
Failure Masking by Redundancy

- **Information Redundancy.** For example, adding extra bits (like in Hamming Codes, see the book *Coding and Information Theory*) to allow recovery from garbled bits

- **Time Redundancy.** Repeat actions if need be

- **Physical Redundancy.** Extra equipment or processes are added to make the system tolerate loss of some components
Failure Masking by Physical Redundancy

Figure 8-2. Triple modular redundancy.
Process Resilience

Achieved by replicating processes into groups.

- How to design fault-tolerant groups?
- How to reach an agreement within a group when some members cannot be trusted to give correct answers?
Flat Groups versus Hierarchical Groups

Figure 8-3. (a) Communication in a flat group. (b) Communication in a simple hierarchical group.
Failure Masking and Replication

- **Primary-backup protocol.** A primary coordinates all write operations. If it fails, then the others hold an election to replace the primary

- **Replicated-write protocols.** Active replication as well as quorum based protocols. Corresponds to a flat group

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

- For fail-silent components, $k+1$ are enough to be $k$ fault tolerant
- For Byzantine failures, at least $2k+1$ extra components are needed to achieve $k$ fault tolerance
- Requires **atomic multicasting**: all requests arrive at all servers in same order
Agreement in Faulty Systems (1)

Possible cases:

- Synchronous versus asynchronous systems
- Communication delay is bounded or not
- Message delivery is ordered or not
- Message transmission is done through unicasting or multicasting
### Agreement in Faulty Systems (2)

**Figure 8-4.** Circumstances under which distributed agreement can be reached.

<table>
<thead>
<tr>
<th>Process behavior</th>
<th>Message ordering</th>
<th>Communication delay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unordered</td>
<td>Ordered</td>
</tr>
<tr>
<td>Synchronous</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asynchronous</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Unicast</td>
<td>Bounded</td>
<td></td>
</tr>
<tr>
<td>Multicast</td>
<td>Unbounded</td>
<td></td>
</tr>
</tbody>
</table>

This diagram illustrates the conditions under which distributed agreement can be reached, considering the type of process behavior (synchronous or asynchronous), message ordering (unordered or ordered), and message transmission (unicast or multicast). The communication delay is also indicated as bounded or unbounded.
Two Army Problem

Non-faulty generals with unreliable communication
Byzantine Generals problem

- Red army in the valley, \( n \) blue generals each with their own army surrounding them.

- Communication is pairwise, instantaneous and perfect.

- However \( m \) of the blue generals are traitors (faulty processes) and are actively trying to prevent the loyal generals from reaching agreement. The generals know the value \( m \).

**Goal:** The generals need to exchange their troop strengths. At the end of the algorithm, each general has a vector of length \( n \). If \( i \)th general is loyal, then the \( i \)th element has their troop strength otherwise it is undefined.
Conditions for a Solution

- All loyal generals decide upon the same plan of action
- A small number of traitors cannot cause the loyal generals to adopt a bad plan
Agreement in Faulty Systems (3)

Figure 8-5. The Byzantine agreement problem for three non-faulty and one faulty process. (a) Each process sends their value to the others.
Byzantine Example

The Byzantine generals problem for 3 loyal generals and 1 traitor:
- The generals announce their troop strengths (in units of 1 kilo soldiers)
- The vectors that each general assembles based on previous step
- The vectors that each general receives
- If a value has a majority, then we know it correctly, else it is unknown
Byzantine Example (2)

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

For $m$ faulty processes, we need a total of $3m+1$ processes to reach agreement.
Reliable Client-Server Communication

*Example:* TCP masks omission failures (e.g. lost messages) but crash failures of connections are often not masked. The distributed system may automatically try to set up a new connection...

RMI semantics in the presence of failures.

- 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 to the client is lost
- The client crashes after sending a request
RPC semantics in the presence of failures

- RPC failures
  - *Client cannot locate server*: Raise an exception or send a signal to client leading to loss in transparency
  - *Lost request messages*: Start a timer when sending a request. If timer expires before a reply is received, send the request again. Server would need to detect duplicate requests
  - *Server crashes*: Server crashes before or after executing the request is indistinguishable from the client side...

- Possible RPC semantics
  - Exactly once semantics
  - At once semantics
  - At most once semantics
  - Guarantee nothing semantics
Lost Request Messages
Server Crashes (1)

A server in client-server communication
a. Normal case
b. Crash after execution
c. Crash before execution
Server Crashes (1)

**Server**: Prints text on receiving request from client and sends message to client after text is printed.

- Send a completion message just before it actually tells the printer to do its work
- Or after the text has been printed

**Client**:

- Never to reissue a request.
- Always reissue a request.
- Reissue a request only if it did not receive an acknowledgement of its request being delivered to the server.
- Reissue a request only if it has not received an acknowledgement of its print request.
Three events that can happen at the server:

- Send the completion message (M)
- Print the text (P)
- Crash (C)
Server Crashes (3)

These events can occur in six different orderings:

1. \( M \rightarrow P \rightarrow C \): A crash occurs after sending the completion message and printing the text.

2. \( M \rightarrow C \rightarrow P \): A crash happens after sending the completion message, but before the text could be printed.

3. \( P \rightarrow M \rightarrow C \): A crash occurs after sending the completion message and printing the text.

4. \( P \rightarrow C \rightarrow M \): The text printed, after which a crash occurs before the completion message could be sent.

5. \( C \rightarrow P \rightarrow M \): A crash happens before the server could do anything.

6. \( C \rightarrow M \rightarrow P \): A crash happens before the server could do anything.
Server Crashes (4)

M: send the completion message  
P: print the text  
C: server crash

Different combinations of client and server strategies in the presence of server crashes.
Lost Reply Messages. Set a timer on client. If it expires without a reply, then send the request again. If requests are idempotent, then they can be repeated again without ill-effects

Client Crashes. Creates orphans. An orphan is an active computation on the server for which there is no client waiting. How to deal with orphans:

- Extermination. Client logs each request in a file before sending it. After a reboot the file is checked and the orphan is explicitly killed off. Expensive, cannot locate grand-orphans etc

- Reincarnation. Divide time into sequentially numbered epochs. When a client reboots, it broadcasts a message declaring a new epoch. This allows servers to terminate orphan computations

- Gentle reincarnation. A server tries to locate the owner of orphans before killing the computation

- Expiration. Each RPC is given a quantum of time to finish its job. If it cannot finish, then it asks for another quantum. After a crash, a client need only wait for a quantum to make sure all orphans are gone
Idempotent Operations

- An *idempotent* operation is one that can be repeated as often as necessary without any harm being done. E.g. reading a block from a file.

- In general, try to make RPC/RMI methods be idempotent if possible. If not, it can be dealt with in a couple of ways.
  - Use a sequence number with each request so server can detect duplicates. But now the server needs to keep state for each client.
  - Have a bit in the message to distinguish between original and duplicate transmission.
Reliable Group Communication

Reliable multicasting guarantees that messages are delivered to all members in a process group. Reliable multicasting turns out to be surprisingly tricky.

- Basic reliable multicasting schemes
- Scalability in reliable multicasting
  - Non-hierarchical feedback control
  - Hierarchical feedback control
- Atomic multicasting using *Virtual Synchrony*
Basic Reliable-Multicasting Schemes (1)

Reliable point-to-point channels are available but reliable communication to a group of processes is rarely built-in to the transport layer. For example, multicasting uses datagrams, which are not reliable.

- Few processes: Set up reliable point-to-point channels. Inefficient for more processes
- What does reliable multicasting mean?
  - What if a process joins during the communication?
  - What happens if a sending process crashes during communication?
- How to reach agreement on what does the group look like?
Basic Reliable-Multicasting Schemes (2)

Assume that processes do not fail and join or leave the group during communication.

- Sending process assigns a sequence number to each message it multicasts. Messages are received in the order which they were sent.
- Each multicast message is kept in a history buffer at the sender. Assuming that the sender knows the receivers, the sender simply keeps the message until all receivers have returned the acknowledgement (Ack).
- Sender retransmits on a negative ack or on timeout before all acks were received.
- Ack can be piggy-backed. Retransmissions can be done with point-to-point communication.
Basic Reliable-Multicasting Schemes (3)

Figure 8-9. A simple solution to reliable multicasting when all receivers are known and are assumed not to fail.
(a) Message transmission. (b) Reporting feedback.
Scalability in Reliable Multicasting

- **Negative acknowledgements**: A receiver returns feedback only if it is missing a message. This improves scalability by cutting down on the number of messages. However, this forces the sender to keep a message in its buffer forever (so we need to use timeouts for the buffer).

- **Nonhierarchical feedback control**: feedback suppression via multicasting of negative feedback.

- **Hierarchical feedback control**: use subgroups and coordinators in each subgroup.
Figure 8-10. Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others.
Hierarchical Feedback Control

Figure 8-11. The essence of hierarchical reliable multicasting. Each local coordinator forwards the message to its children and later handles retransmission requests.
Atomic Multicasting

The *atomic multicast* setup to achieve reliable multicasting in the presence of process failures requires the following conditions:

- A message is delivered to all processes or to none at all
- All messages are delivered in the same order to all processes

For example, this solves the problem of a replicated database on top of a distributed system.

*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.*
Virtual Synchrony (1)

Figure 8-12. The logical organization of a distributed system to distinguish between message receipt and message delivery.
A multicast message $m$ is uniquely associated with a list of processes to which it should be delivered. This delivery list corresponds to a group view. Each process on the list has the same view.

A view change takes place by multicasting a message $vc$ announcing the joining or leaving of a process.

Suppose $m$ and $vc$ are simultaneously in transit. We need to guarantee that $m$ is either delivered to all processes in the group view $G$ before each of them is delivered message $vc$, or $m$ is not delivered at all.

Note that $m$ being not delivered is because the sender of $m$ crashed.
A reliable multicast is said to be *virtually synchronous* if it has the following properties:

- A message multicast to group view G is delivered to each non-faulty process in G
- If a sender of the message crashes during the multicast, the message may either be delivered to all processes, or ignored by each of them

The principle is that all multicasts take place between view changes. All multicasts that are in transit when a view change takes place are completed before the view change comes into effect.
Figure 8-13. The principle of virtual synchronous multicast.
Virtual synchrony allows us to think about multicasts as taking place in epochs. But we can have several possible orderings of the multicasts:

- Unordered multicasts
- FIFO-ordered multicasts
- Causally-ordered multicasts (requires vector timestamps)
- Totally-ordered multicasts
### Message Ordering (2)

<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>

Figure 8-14. Three communicating processes in the same group. The ordering of events per process is shown along the vertical axis. This shows unordered multicasts.
### Message Ordering (3)

#### Figure 8-15. Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting

<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>
### Comparison of Message Orderings

<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>

Figure 8-16. Six different versions of virtually synchronous reliable multicasting.
Implementing Virtual Synchrony (1)

- Assume that we have reliable point to point communication (e.g. TCP) and that messages from the same source are received in the same order as sent (e.g. TCP)

- Every process in $G$ keeps message $m$ until it knows for sure that all member in $G$ have received it. If $m$ has been received by all members in $G$, then $m$ is said to be stable. Only stable messages are allowed to be delivered.

- To ensure stability, it is sufficient to pick an arbitrary process in $G$ and request it to send $m$ to all other processes. That arbitrary process can be the *coordinator*.

- Assumes that no process crashes during a view change although it can be generalized to handle that as well.
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
Distributed Commit

The *distributed commit* problem involves having an operation being performed by each member of a group or none at all. Examples:

- Reliable multicasting is a specific example with the operation being the delivery of a message
- With distributed transactions, the operation may be the commit of a transaction at a single site that takes part in the transaction

Solutions:

- *One-phase commit*, *two-phase commit* and *three-phase commit*
One-Phase Commit

- The coordinator tells all processes whether or not to (locally) perform the operation in question

- If one of the participants cannot perform the operation, then there is no way to tell the coordinator
Two-Phase Commit (1)

Figure 8-18.
(a) The finite state machine for the coordinator in 2PC.
(b) The finite state machine for a participant.
Two-Phase Commit (2)

- The protocol can fail if a process crashes for other processes may be waiting indefinitely for a message from the crashed process. We deal with this using timeouts

- **Participant blocked in INIT:** A participant is waiting for a VOTE_REQUEST message from the coordinator. On a timeout, it can locally abort the transaction and thus send a VOTE_ABORT message to the coordinator

- **Coordinator blocked in WAIT:** If it doesn’t get all the votes, it votes for an abort and sends a GLOBAL_ABORT to all participants

- **Participant blocked in READY:** Participant cannot simply decide to abort. It needs to know what message was sent by the coordinator.
  - We can simply block until the coordinator recovers
  - Or contact another participant Q to see if it can decide from Q’s state what to do. Four cases to deal with here that are summarized on next slide
Two-Phase Commit (3)

<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 (4)

actions by coordinator:

write START_2PC local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write 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 (5)

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;
}
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 (1)

- If the coordinator crashes in two-phase commit, the processes may not be able to reach a final decision and have to block until the coordinator recovers.

- Three-phase commit protocol avoids blocking processes in presence of crashes. The states of the coordinator and each participant satisfy the following two conditions:
  - There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.
  - 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.
Three-Phase Commit (2)

Figure 8-22. (a) The finite state machine for the coordinator in 3PC. (b) The finite state machine for a participant.
Recovery

**Backward recovery.** Roll back the system from erroneous state to a previously correct state. This requires system to be *checkpointing*, which has the following issues:

- Relatively costly to checkpoint. Often combined with *message logging* for better performance. Messages are logged before sending or before receiving. Combined with checkpoints to makes recovery possible. Checkpoints alone cannot solve the issue of replaying all messages in the right order.

- Backward recovery requires a loop of recovery so failure transparency cannot be guaranteed. Some states can never be rolled back to...

**Forward recovery.** Bring the system to a correct new state from which it can continue execution. E.g. In an \((n,k)\) block erasure code, a set of \(k\) source packets is encoded into a set of \(n\) encoded packets, such that any set of \(k\) encoded packets is enough to reconstruct the original \(k\) source packets.
We need fault-tolerant disk storage for the checkpoints and message logs. Examples are various RAID (Redundant Array of Independent Disks) schemes (although they are used for both improved fault tolerance as well as improved performance). Some common schemes:

- RAID-0 (block-level striping)
- RAID-1 (mirroring)
- RAID-5 (block-level striping with distributed parity)
- RAID-6 (block-level striping with double distributed parity)
Figure 8-23. (a) Stable storage. (b) Crash after drive 1 is updated. (c) Bad spot due to spontaneous decay can be dealt with.
Checkpointing

- Backward error recovery schemes require that a distributed system regularly records a consistent global state to stable storage. This is known as a **distributed snapshot**.

- In a distributed snapshot, if a process P has recorded the receipt of a message, then there is also a process Q that has recorded the sending of that message.

- To recover after a process or system failure, it is best to recover to the most recent distributed snapshot, also known as the **recovery line**.

- Independent checkpointing:

- Coordinated checkpointing:

- Message logging:
  - Optimistic message logging
  - Pessimistic message logging
Checkpointing

Figure 8-24. A recovery line.
Independent Checkpointing

Figure 8-25. The domino effect.
Coordinated Checkpointing

All processes synchronize to jointly write their state to local stable storage, which implies that the saved state is automatically consistent.

- **Simple Coordinated Checkpointing.** Coordinator multicasts a `CHECKPOINT_REQUEST` to all processes. When a process receives the request, it takes a local checkpoint, queues any subsequent messages handed to it by the application it is executing, and acknowledges to the coordinator. When the coordinator has received an acknowledgement from all processes, it multicasts a `CHECKPOINT_DONE` message to allow the blocked processes to continue.

- **Incremental snapshot.** The coordinator multicasts a checkpoint request only to those processes it had sent a message to since it last took a checkpoint. When a process P receives such a request, it forwards it to all those processes to which P itself had sent a message since the last checkpoint and so on. A process forwards the request only once. When all processes have been identified, then a second message is multicast to trigger checkpointing and to allow the processes to continue.
Message Logging

- If the transmission of messages can be replayed, we can still reach a globally consistent state by starting from a checkpointed state and retransmitting all messages sent since. Helps in reducing the number of checkpoints.

- Assumes a \textit{piecewise deterministic model}, where deterministic intervals occur between sending/receiving messages.

- An orphan process is a process that has survived the crash of another process, but whose state is inconsistent with the crashed process after its recovery.
Message Logging

Incorrect replay of messages after recovery, leading to an *orphan process*. 
Message Logging Schemes

- A message is said to be *stable* if it can no longer be lost, because it has been written to stable storage. Stable messages can be used for recovery by replaying their transmission
  - \( DEP(m) \): A set of processes that depend upon the delivery of message \( m \)
  - \( COPY(m) \): A set of processes that have a copy of \( m \) but not yet in their local stable storage
- A process \( Q \) is an orphan process if there is a message \( m \) such that \( Q \) is contained in \( DEP(m) \), while at the same time all processes in \( COPY(m) \) have crashed. We want to avoid this scenario

- **Pessimistic logging protocol**: For each non-stable message \( m \), there is at most one process dependent upon \( m \), which means that this process is in \( COPY(m) \). Basically, a process \( P \) is not allowed to send any messages after delivery of \( m \) without first storing it in stable storage

- **Optimistic logging protocol**: After a crash, orphan processes are rolled back until they are not in \( DEP(m) \). Much more complicated than pessimistic logging
The CAP Theorem (1)

**CAP Theorem** (aka Brewer's Theorem): states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- **Consistency** (all nodes see the same data at the same time)
- **Availability** (a guarantee that every request receives a response about whether it was successful or failed)
- **Partition Tolerance** (the system continues to operate despite arbitrary message loss or failure of part of the system)
The CAP Theorem (2)