Chapter 5: Distributed Systems: Fault Tolerance

Fall 2011
Jussi Kangasharju
Chapter Outline

- Fault tolerance
- Process resilience
- Reliable group communication
- Distributed commit
- Recovery
Basic Concepts

Dependability includes

- Availability
- Reliability
- Safety
- Maintainability
Fault, error, failure

client

server

failure

error

fault
Failure Model

- Challenge: independent failures
- Detection
  - which component?
  - what went wrong?
- Recovery
  - failure dependent
  - ignorance increases complexity
=> taxonomy of failures
Fault Tolerance

- Detection

- Recovery
  - mask the error  OR
  - fail predictably

- Designer
  - possible failure types?
  - recovery action  (for the possible failure types)

- A fault classification:
  - transient  (disappear)
  - intermittent (disappear and reappear)
  - 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></td>
<td><em>Receive omission</em> A server fails to receive incoming messages</td>
</tr>
<tr>
<td></td>
<td><em>Send omission</em> 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></td>
<td><em>Value failure</em> The value of the response is wrong</td>
</tr>
<tr>
<td></td>
<td><em>State transition failure</em> 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>

Crash: **fail-stop, fail-safe** (*detectable*), **fail-silent** (*seems to have crashed*)
Failure Masking (1)

Detection

- redundant information
  - error detecting codes (parity, checksums)
  - replicas
- redundant processing
  - groupwork and comparison
- control functions
  - timers
  - acknowledgements
Failure Masking (2)

Recovery

- redundant information
  - error correcting codes
  - replicas
- redundant processing
  - *time redundancy*
    - retrial
    - recomputation (checkpoint, log)
  - *physical redundancy*
    - groupwork and voting
    - tightly synchronized groups
Example: Physical Redundancy

(a)

(b)

Voter

A1 A2 A3 V1 V2 V3

B1 B2 B3 V4 V5 V6

C1 C2 C3 V7 V8 V9

Triple modular redundancy.
Failure Masking (3)

- Failure models vs. implementation issues:
  the (sub-)system belongs to a class
  => certain failures do not occur
  => easier detection & recovery
- A point of view: forward vs. backward recovery
- Issues:
  - process resilience
  - reliable communication
Process Resilience (1)

- Redundant processing: groups
  - Tightly synchronized
    - flat group: voting
    - hierarchical group:
      a primary and a hot standby (execution-level synchrony)
  - Loosely synchronized
    - hierarchical group:
      a primary and a cold standby (checkpoint, log)

- Technical basis
  - “group” – a single abstraction
  - reliable message passing
Flat and Hierarchical Groups (1)

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

Group management: a group server OR distributed management
Flat and Hierarchical Groups (2)

- Flat groups
  - symmetrical
  - no single point of failure
  - complicated decision making

- Hierarchical groups
  - the opposite properties

- Group management issues
  - join, leave;
  - crash (*no notification*)
Process Groups

- Communication vs management
  - application communication: message passing
  - group management: message passing
  - synchronization requirement: each group communication operation in a stable group

- Failure masking
  - **k fault tolerant**: tolerates k faulty members
    - fail silent:  \( k + 1 \) components needed
    - Byzantine: \( 2k + 1 \) components needed
  - a precondition: **atomic multicast**
  - in practice: the probability of a failure must be “small enough”
Agreement in Faulty Systems (1)

Let’s meet at noon in front of La Tryste …

Alice -> Bob
Alice <- Bob
Alice: If Bob doesn’t know that I received his message, he will not come …
Alice -> Bob
Bob: If Alice doesn’t know that I received her message, she will not come …

Requirement:
- an agreement
- within a bounded time

Faulty data communication: no agreement possible

La Tryste
on a rainy day …

Alice -> Bob
Let’s meet at noon in front of La Tryste …
Alice <- Bob
OK!!
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.

<table>
<thead>
<tr>
<th>General</th>
<th>Troop Strengths</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Got(1, 2, x, 4)</td>
</tr>
<tr>
<td>2</td>
<td>Got(1, 2, y, 4)</td>
</tr>
<tr>
<td>3</td>
<td>Got(1, 2, 3, 4)</td>
</tr>
<tr>
<td>4</td>
<td>Got(1, 2, z, 4)</td>
</tr>
</tbody>
</table>

Faulty process

Reliable data communication, unreliable nodes

![Diagram of the Byzantine generals problem](image)
The same as in previous slide, except now with 2 loyal generals and one traitor.
Reliable Group Communication

- Lower-level data communication support
  - unreliable multicast (LAN)
  - reliable point-to-point channels
  - unreliable point-to-point channels

- Group communication
  - individual point-to-point message passing
  - implemented in middleware or in application

- Reliability
  - acks: lost messages, lost members
  - communication consistency?
Reliability of Group Communication?

- A sent message is received by all members
  \[(\text{acks from all} \Rightarrow \text{ok})\]

- Problem: during a multicast operation
  - an old member disappears from the group
  - a new member joins the group

- Solution
  - membership changes synchronize multicasting
    \[\Rightarrow \text{during an MC operation no membership changes}\]

An additional problem: the sender disappears (remember: multicast ~ for (all \(P_i \in G\) \{send m to \(P_i\}\})
Basic Reliable-Multicasting Scheme

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

Scalability?

Feedback implosion!
Scalability: Feedback Suppression

1. Never acknowledge successful delivery.

2. Multicast negative acknowledgements – suppress redundant NACKs
   Problem: detection of lost messages and lost group members
Hierarchical Feedback Control

The essence of hierarchical reliable multicasting.

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

**Guarantee:**
the message will eventually be delivered to all member of the group (during the multicast: a fixed membership)

**Group view:** \( G = \{p_i\} \)
“delivery list”

Implementation of \( \text{Basic\_multicast}(G, m) \):
1. for each \( p_i \) in \( G \): \( \text{send}(p_i, m) \) (a reliable one-to-one send)
2. on \( \text{receive}(m) \) at \( p_i \): \( \text{deliver}(m) \) at \( p_i \)
Message Delivery

Delivery of messages
- new message => HBQ
- decision making
  - delivery order
  - deliver or not to deliver?
- the message is allowed to be delivered: HBQ => DQ
- when at the head of DQ: message => application (application: receive ...)

Application

delivery

hold-back queue
delivery queue

Message passing system
Reliable Multicast and Group Changes

Assume
- reliable point-to-point communication
- group $G = \{ p_i \}$: each $p_i$ : groupview

**Reliable multicast** $(G, m)$:
if a message is delivered to one in $G$,
then it is delivered to all in $G$

- Group change (join, leave) $\Rightarrow$ change of groupview
- Change of group view: update as a multicast $vc$
- **Concurrent group_change and multicast**
  $\Rightarrow$ concurrent messages $m$ and $vc$

**Virtual synchrony:**
all nonfaulty processes see $m$ and $vc$ in the same order
Virtually Synchronous Reliable MC (1)

Group change: $G_i = G_{i+1}$

Virtual synchrony: “all” processes see $m$ and $vc$ in the same order
- $m, vc \Rightarrow m$ is delivered to all nonfaulty processes in $G_i$ (alternative: this order is not allowed!)
- $vc, m \Rightarrow m$ is delivered to all processes in $G_{i+1}$ (what is the difference?)

Problem: the sender fails (during the multicast – why is it a problem?)

Alternative solutions:
- $m$ is delivered to all other members of $G_i$ ($\Rightarrow$ ordering $m$, $vc$)
- $m$ is ignored by all other members of $G_i$ ($\Rightarrow$ ordering $vc$, $m$)
The principle of virtual synchronous multicast:

- a **reliable multicast**, and if the sender crashes
- the message may be **delivered to all or ignored by each**
Implementing Virtual Synchrony

- Communication: reliable, order-preserving, point-to-point
- Requirement: all messages are delivered to all nonfaulty processes in G
- Solution
  - each \( p_j \) in G keeps a message in the hold-back queue until it knows that all \( p_j \) in G have received it
  - a message received by all is called stable
  - only stable messages are allowed to be delivered
  - view change \( G_i \Rightarrow G_{i+1} \):
    - multicast all unstable messages to all \( p_j \) in \( G_{i+1} \)
    - multicast a flush message to all \( p_j \) in \( G_{i+1} \)
    - after having received a flush message from all: install the new view \( G_{i+1} \)
Implementing Virtual Synchrony

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
Ordered Multicast

Need:
all messages are delivered in the intended order

If $p$: multicast$(G,m)$ and if (for any $m'$)

- for **FIFO** $\text{multicast}(G, m) < \text{multicast}(G, m')$
- for **causal** $\text{multicast}(G, m) \rightarrow \text{multicast}(G, m')$
- for **total** if at any $q$: $\text{deliver}(m) < \text{deliver}(m')$

then for all $q$ in $G$: $\text{deliver}(m) < \text{deliver}(m')$
### Reliable FIFO-Ordered Multicast

<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>receives m2</td>
<td>receives m2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>receives m4</td>
<td>receives m4</td>
<td></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
### Virtually Synchronous Multicasting

<table>
<thead>
<tr>
<th>Virtually synchronous 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>

**Six different versions of virtually synchronous reliable multicasting**

- **virtually synchronous**: everybody or nobody (members of the group) (sender fails: either everybody else or nobody)
- **atomic multicasting**: virtually synchronous reliable multicasting with totally-ordered delivery.
Recovery

- Fault tolerance: recovery from an error (erroneous state => error-free state)
- Two approaches
  - backward recovery: back into a previous correct state
  - forward recovery:
    - detect that the new state is erroneous
    - bring the system in a correct new state
    challenge: the possible errors must be known in advance
- forward: continuous need for redundancy
- backward:
  - expensive when needed
  - recovery after a failure is not always possible
Recovery Stable Storage

Stable Storage

Crash after drive 1 is updated

Bad spot
Implementing Stable Storage

- Careful block operations (fault tolerance: transient faults)
  - careful_read: \{get_block, check_parity, error=> N retries\}
  - careful_write: \{write_block, get_block, compare, error=> N retries\}
  - irrecoverable failure => report to the “client”

- Stable Storage operations (fault tolerance: data storage errors)
  - stable_get:
    \{careful_read(replica_1), if failure then careful_read(replica_2)\}
  - stable_put: \{careful_write(replica_1), careful_write(replica_2)\}
  - error/failure recovery: read both replicas and compare
    - both good and the same => ok
    - both good and different => replace replica_2 with replica_1
    - one good, one bad => replace the bad block with the good block
Checkpointing

Needed: a consistent global state to be used as a recovery line

A recovery line: the most recent distributed snapshot
Independent Checkpointing

Each process records its local state from time to time ⇒ difficult to find a recovery line

If the most recently saved states do not form a recovery line ⇒ rollback to a previous saved state (threat: the domino effect).

A solution: coordinated checkpointing
Coordinated Checkpointing (1)

- Nonblocking checkpointing
  - see: distributed snapshot (Ch. 5.3)
- Blocking checkpointing
  - **coordinator**: multicast CHECKPOINT_REQ
  - **partner**:
    - take a local checkpoint
    - acknowledge the coordinator
    - wait (and queue any subsequent messages)
  - **coordinator**:
    - wait for all acknowledgements
    - multicast CHECKPOINT_DONE
  - **coordinator, partner**: continue
Coordinated Checkpointing (2)

checkpoint request
ack
checkpoint done
local checkpoint
message
Message Logging

Improving efficiency: checkpointing and message logging

Recovery: most recent checkpoint + replay of messages

Problem: Incorrect replay of messages after recovery may lead to orphan processes.
Chapter Summary

- Fault tolerance
- Process resilience
- Reliable group communication
- Distributed commit
- Recovery