Fault Tolerance

Fall 2008

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

- Failure = toimintahäiriö
- Fault = vika
- Error = virhe(tila)
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>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 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)

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)

La Tryste
on a rainy day ...

"e-mail"

Requirement:
- an agreement
- within a bounded time

Faulty data communication: no agreement possible

Alice -> Bob  Let's meet at noon in front of La Tryste …
Alice <- Bob  OK!!
Alice:  If Bob doesn't know that I received his message, he will not come …
Alice -> Bob  I received your message, so it's OK.
Bob:  If Alice doesn't know that I received her message, she will not come …

...
Agreement in Faulty Systems (2)

Reliable data communication, unreliable nodes

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.
The same as in previous slide, except now with 2 loyal generals and one traitor.
An agreement can be achieved, when
- message delivery is reliable with a bounded delay
- processors are subject to Byzantine failures, but fewer than one third of them fail

An agreement cannot be achieved, if
- messages can be dropped (even if none of the processors fail)
- message delivery is reliable but with unbounded delays, and even one processor can fail

Further theoretical results are presented in the literature
Reliable Client-Server Communication

1. Point-to-Point Communication ("reliable")
   - masked: omission, value
   - not masked: crash, (timing)

2. RPC semantics
   - the client unable to locate the server
   - the message is lost (request / reply)
   - the server crashes (before / during / after service)
   - the client crashes
Server Crashes (1)

A server in client-server communication

a) Normal case
b) Crash after execution
c) Crash before execution
### Server Crashes (2)

Different combinations of client and server strategies in the presence of server crashes (client’s continuation after server’s recovery: reissue the request?)

- **M**: send the completion message
- **P**: print the text
- **C**: crash

#### Strategy M -> P

<table>
<thead>
<tr>
<th>Reissue strategy</th>
<th>MPC</th>
<th>MC(P)</th>
<th>C(MP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td>DUP</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Never</td>
<td>OK</td>
<td>ZERO</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>ZERO</td>
<td>OK</td>
</tr>
</tbody>
</table>

#### Strategy P -> M

<table>
<thead>
<tr>
<th>Reissue strategy</th>
<th>PMC</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>
Client Crashes

- Orphan: an active computation looking for a non-existing parent

- Solutions
  - extermination: the client stub records all calls, after crash recovery all orphans are killed
  - reincarnation: time is divided into epochs, client reboot => broadcast “new epoch” => servers kill orphans
  - gentle incarnation: “new epoch” => only “real orphans” are killed
  - expiration: a “time-to-live” for each RPC (+ possibility to request for a further time slice)

- New problems: grandorphans, reserved locks, entries in remote queues, ....
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
  
  \( \textit{acks from all} \implies \text{ok} \)

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

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

\( \text{An additional problem: the sender disappears (remember: multicast } \sim \text{ for (all } P_i \text{ in } G) \{\text{send } m \text{ to } P_i\} \) \)
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 \( Basic\_multicast(G, m) \):
1. for each \( p_i \) in \( G \): send\( (p_i, m) \) (a reliable one-to-one send)
2. on receive\( (m) \) at \( p_i \): 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 …)
Reliable Multicast and Group Changes

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

**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) => change of groupview
- Change of group view: update as a multicast $vc$
- **Concurrent group_change and multicast**
  => concurrent messages $m$ and $vc$

*Virtual synchrony*:
*all nonfaulty processes see $m$ and $vc$ in the same order*
Virtually Synchronous Reliable MC (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 \( G_i \) (=> ordering \( m, \; vc \))
- \( m \) is ignored by all other members of \( G_i \) (=> 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**
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
Implementing Virtual Synchrony (2)

- 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 $G_{i+1}$
Ordered Multicast

Need: all messages are delivered in the intended order

1. If \( p: \text{multicast}(G, m) \) 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 \( q \): \( \text{deliver}(m) < \text{deliver}(m') \)

2. 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.
Distributed Transactions

Atomic
Consistent
Isolated
Durable
A distributed banking transaction

\[ T = openTransaction \]
\[ a.\text{withdraw}(4); \]
\[ c.\text{deposit}(4); \]
\[ b.\text{withdraw}(3); \]
\[ d.\text{deposit}(3); \]
\[ closeTransaction \]

Note: the coordinator is in one of the servers, e.g. BranchX

Figure 13.3
Concurrency Control

- General organization of managers for handling distributed transactions.
Transaction Processing (1)

client

…. 
Open transaction
T_write F1,P1
T_write F2,P2
T_write F3,P3
Close transaction
…. 

S1

T_Id
flag: init
P1 27
y 1223

F1

S2

T_Id
flag: init
P2 27
ab 667

S3

F2

T_Id
flag: init
P3 2746

F3
Transaction Processing (2)

client

....
Open transaction
T_read F1,P1
T_write F2,P2
T_write F3,P3
Close transaction
....
Operations for Two-Phase Commit Protocol

`canCommit?(trans) -> Yes / No`
Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

`doCommit(trans)`
Call from coordinator to participant to tell participant to commit its part of a transaction.

`doAbort(trans)`
Call from coordinator to participant to tell participant to abort its part of a transaction.

`haveCommitted(trans, participant)` Call from participant to coordinator to confirm that it has committed the transaction.

`getDecision(trans) -> Yes / No`
Call from participant to coordinator to ask for the decision on a transaction after it has voted Yes but has still had no reply after some delay. Used to recover from server crash or delayed messages.

**Figure 13.4**
Communication in Two-phase Commit Protocol

![Diagram showing the process of communication in Two-phase Commit Protocol]

**Coordinator**

<table>
<thead>
<tr>
<th>step</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>tentative</td>
</tr>
<tr>
<td></td>
<td>prepared to commit</td>
</tr>
<tr>
<td></td>
<td>(wait)</td>
</tr>
<tr>
<td>3</td>
<td>committed</td>
</tr>
<tr>
<td></td>
<td>done</td>
</tr>
</tbody>
</table>

**Participant**

<table>
<thead>
<tr>
<th>step</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>tentative</td>
</tr>
<tr>
<td></td>
<td>prepared to commit</td>
</tr>
<tr>
<td></td>
<td>(ready)</td>
</tr>
<tr>
<td>4</td>
<td>committed</td>
</tr>
</tbody>
</table>

**Messages:**
- canCommit?
- doCommit
- haveCommitted
- prepared to commit
- committed
- done

Figure 13.6
The Two-Phase Commit protocol

Phase 1 (voting phase):
1. The coordinator sends a *canCommit* request to each of the participants in the transaction.
2. When a participant receives a *canCommit* request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

Phase 2 (completion according to outcome of vote):
3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Figure 13.5
Failures

- A message is lost
- Node crash and recovery (memory contents lost, disk contents preserved)
  - transaction data structures preserved (incl. the state)
  - process states are lost
- After a crash: transaction recovery
  - tentative => abort
  - aborted => abort
  - wait (coordinator) => abort (resend canCommit ? )
  - ready (participant) => ask for a decision
  - committed => do it!
Two-Phase Commit (1)

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 {
    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;
}
Two-Phase Commit (2)

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;
}

Steps taken by participant process in 2PC.
Two-Phase Commit (3)

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 */
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

(a) Stable Storage
(b) Crash after drive 1 is updated
(c) 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
Checking of Dependencies

Figure 10.14  Vector timestamps and variable values
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)

- P1
- P2
- P3

- 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.