Fault Tolerance

Today

- Introduction to fault tolerance
- Process resilience
- Communication resilience
- Distributed commit
- Recovery

Hurry, it's having a nervous breakdown
Dependability

- Fault tolerance ~ dependability
- Components provide/require services from/to other components ⇒ a component may depend on another
- Some properties of dependability
  - Availability – Readiness for usage (probability of operating correctly at any moment)
  - Reliability – Continuity of service delivery (rather than probability, uptime)
  - Safety – If there’s a failure, low probability of catastrophes
  - Maintainability – How easy can a failed system be repaired

- For distributed systems, components can be either processes or channels
Terminology

- Failure – System cannot meet its promises
- Error – Part of system’s state that can lead to a failure
- Fault – Cause of an error

- Fault tolerance – Build a system so that it can meet its specifications in the presence of faults
- Fault removal – Reduce the presence, number, seriousness of faults
- Fault forecasting – Estimate the present number, future incidence, and the consequences of faults
Failure models

- **Crash failures** – a component simply halts, but behaves correctly before halting
  - Fail-stop – system exhibits crash failures, but its failure can be detected (either through announcement or timeouts)

- **Omission failures** – … fails to respond to incoming requests
  - Receive or send omission
    - Fail-silent – system exhibits omission or crash failures; hard to tell what went wrong

- **Timing failures** – output is correct, but lies outside a specified real-time interval
Failure models

- Response failures – output is incorrect
  - Value failure: The wrong value is produced
  - State transition failure: Execution of the component’s service brings it into a wrong state
  - Fail-safe –system exhibits arbitrary, but benign failures (generating random output)

- Arbitrary/byzantine failures – may produce arbitrary output and be subject to arbitrary timing failures

- Basic approach to masking faults – redundancy
  - Information – add extra bits to a message for reconstruction
  - Time – do something more than once if needed
  - Physical – add copies of software/hardware
Process resilience

- Replicates processes in a group (a message to a process is set to the group)

- Flat or hierarchical
  - Flat – symmetric w/o single point of failure; complicated decision making
  - Hierarchical – i.e. somebody (coordinator) in charge; opposite properties

- Dynamic or static

- Group membership - creating and deleting groups
  - Group server – single point of failure
  - Distributed with group communication - reliable multicast a join request (catching up on messages?)
  - …
Groups and failure masking

- A group that can mask $k$ concurrent member failures, is $k$-fault tolerant ($k$ is called degree of fault tolerance)

- How large does a $k$-fault tolerant group need to be?
  - Assume crash/performance failure semantics $\Rightarrow k + 1$ members are needed to survive $k$ member failures
  - Assume arbitrary/Byzantine failure semantics, and group output defined by voting $\Rightarrow 2k+1$

- Reaching agreement is another matter
  - Byzantine agreement problem
  - Lamport et al. solution for $N$ processes
    - Synchronous processes
    - Messages are unicast
    - Message preserve ordering
    - Communication delay is bounded
Groups and failure masking

- Each process $i$ provides a value $v_i$ to the other $N-1$
- Goal: Each process constructs vector $V$ of length $N$, such that if process $i$ is not faulty, $V[i] = i$, otherwise is undef
- The algorithm operates in four steps
  1. Every non-faulty process $i$ sends $v_i$ to every other using reliable unicast

$3k+1$ (1 faulty: 4 total)
Groups and failure masking

- ... Algorithm’s four steps
  1. Every non-faulty process $i$ sends $v_i$ to every other using reliable unicast
  2. Results are collected into a vector
  3. Processes exchange their vectors
  4. Result vector is computed with majority value or unknown

1. Got (1, 2, x, 4)
2. Got (1, 2, y, 4)
3. Got (1, 2, 3, 4)
4. Got (1, 2, z, 4)

1. Got (1, 2, y, 4)
(a, b, c, d)
(1, 2, z, 4)

2. Got (1, 2, x, 4)
(e, f, g, h)
(1, 2, z, 4)

4. Got (1, 2, x, 4)
(1, 2, y, 4)
(i, j, k, l)

1. Got (1, 2, U, 4)
2. Got (1, 2, U, 4)

4. Got (1, 2, U, 4)
Groups and failure masking

- What are the necessary conditions for reaching agreement?

In practice, most distributed systems assume …

- Process: Synchronous ⇒ operate in lockstep
- Delays: Are delays on communication bounded?
- Ordering: Are messages delivered in the order they were sent?
- Transmission: Are messages sent one-by-one, or multicast?
Failure detection

- Failure detection is key to fault tolerance
- How do we detect process failures?
  - Keep alive messages
  - Passively wait for a sign
- Basically, detect failures through timeout mechanisms
  - Setting timeouts properly is very difficult and application dependent
  - You cannot distinguish process failures from network failures
  - We need to consider failure notification throughout the system:
    - Gossiping (i.e., proactively disseminate a failure detection)
    - On failure detection, pretend you failed as well
Reliable communication

• 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 enough redundancy so corrupted packets can be automatically corrected
  – Request retransmission of lost, or last $N$ packets
• Most of this work assumes point-to-point communication
Reliable RPC

- What can go wrong with a remote procedure call?
- 1: Client cannot locate server
  - Either went down or has a new version of the interface; relatively simple – just report back to client (of course, that’s not too transparent)
- 2: Client request is lost
  - Just resend message after a timeout
- 3: Server crashes
  - Harder to handle – we don’t know how far it went
  - What should we expect from the server?
    - At-least-once – guarantees an operation at least once, but perhaps more
    - At-most-once – guarantees an operation at most once
    - Exactly-once – *no way to arrange this*!

- ...
Reliable RPC

- Exactly-once semantics
  - Client asks to print text, server sends completion
  - Server can
    - Send completion before (M→P) or after printing (P→M)
    - If it crashes and comes back up again, announces it
  - Client can
    - Always reissue, never reissue, reissue request only when ACK, reissue only when not ACK
  - *Not good solution for any situation!*

### MPC: crashed after sending ack and printing
### MC(P): crashed after sending ack but before printing
### C(MP): crashed before sending ack and printing

<table>
<thead>
<tr>
<th>OK</th>
<th>Text is printed once</th>
<th>MPC</th>
<th>MC(P)</th>
<th>C(MP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUP</td>
<td>Text is printed twice</td>
<td>DUP</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>ZERO</td>
<td>Text is not printed at all</td>
<td>DUP</td>
<td>DUK</td>
<td>OK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client</th>
<th>Strategy M → P</th>
<th>Server</th>
<th>Strategy P → M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Never</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Only when ACKed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Only when not ACKed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reliable RPC

- 4: Server response is lost
  - Hard to detect, the server could also have crashed. Did it get it done? Solution: No much, try making operations idempotent

- 5: Client crashes
  - Server is doing work and holding resources for nothing (doing an orphan computation)
    - Orphan is killed (or rolled back) by client when it reboots, but client have to write log to disk (orphan extermination)
    - Broadcast new epoch number when recovering ⇒ servers kill orphans (reincarnation)
    - Require computations to complete in a $T$ time units (expiration)
  - Old ones are simply removed
Reliable group communication

- Reliable multicast – guarantee that msgs are delivered to all members of a group
- Basic model: A multicast channel $c$ with two (possibly overlapping) groups:
  - Sender group $SND(c)$ of processes that submit msgs to $c$
  - Receiver group $RCV(c)$ that can receive messages from $c$
- Simple reliability (non-faulty processes) & agreement on $RCV$
  - If process $P \in RCV(c)$ at the time message $m$ was submitted to $c$, and $P$ does not leave $RCV(c)$, $m$ should be delivered to $P$
Reliable group communication

- Observation: If we can stick to a local-area network, reliable multicasting is “easy”
- Let the sender log messages submitted to channel $c$:
  - If $P$ sends message $m$, $m$ is stored in a history buffer
  - Each receiver acknowledges the receipt of $m$, or requests retransmission at $P$ when noticing message lost
  - Sender $P$ removes $m$ from history buffer when everyone has acknowledged receipt
- Why doesn’t this scale?
  - N acks!
- Solution – use NACKs instead
  - Issue – how long should you keep the msg in the buffer?
Scalable reliable multicast – SRM

- Feedback suppression – Let a process $P$ suppress its own feedback when it notices process $Q$ is already asking for a retransmission (Floyd et al.’s SRM)

- 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

- A few issues
  - The random interval is key
  - Multicasting feedback also interrupt processes that got the request
  - Other receivers can also help in the recovery
Scalable reliable multicast – hierarchical

- Add hierarchy for scalability – a hierarchical feedback channel in which all submitted messages are sent only to the root
  - Within a subgroup, any reliable multicasting scheme is OK

- Intermediate nodes aggregate feedback messages before passing them on

- Main problem – tree construction
Atomic multicast

- Atomic multicast – the msg is delivered to all or none
  - A msg is associated with a group of processes, a group view
  - What if the group view changes?

- Virtual synchronous
  - A msg $m$ is delivered to each non-faulty process in $G$
  - If the group changes, send a msg $vc$ announcing the joining/leaving of a process – a view change/new epoch
  - So, ensure that either all processes in $G$ received $m$ before receiving $vc$ or none receives it
  - if the sender crashes, $m$ can either be delivered to all or be ignored by all
Atomic multicast

- Virtual synchrony lets us see multicast as happening in epochs separated by group memberships.
Message ordering

- What about order of messages?
  - Unordered – virtual synchronous w/o order guarantees
    - Process P1
      Sends m1
      Sends m2
    - Process P2
      Receives m1
      Receives m2
    - Process P3
      Receives m2
      Receives m1
  - FIFO-ordered – from the same process in the same order
    - Process P1
      Sends m1
      Sends m2
    - Process P2
      Receives m1
      Receives m3
      Receives m2
      Receives m4
    - Process P3
      Receives m2
      Receives m1
      Receives m2
      Receives m4
    - Process P4
      Sends m3
      Sends m4
  - Causally-ordered – preserving potential causality bet/ different messages
    - If m1 causally precedes m3

Violates causal order
Causally ordered multicasting

- Ensure that a msg is delivered only if all causally preceding msgs have already been delivered
- Clock adjustment only when sending/receiving messages:
  - \( p_i \) increments \( V_{ij} \) only when sending a message
  - \( p_j \) “adjusts” \( V_j \) when receiving a message
- \( p_j \) postpones delivery of \( m \) until:
  - \( ts(m)[i] = V_{ij} + 1 \)
    - \( m \) is next msg \( p_j \) was expecting from \( p_i \)
  - \( ts(m)[k] \leq V_{jk} \) for \( k \neq j \)
    - \( p_j \) has seen all msgs seen by \( p_i \) when it sent the message
Causally ordered multicasting

- Suppose $P_j$ receives $m$ from $P_i$ with timestamp $ts(m)$
- $P_j$ postpones delivery of $m$ until:
  - $ts(m)[i] = VC_j[i] + 1$
  - $ts(m)[k] \leq VC_j[k]$ for $k \neq j$
Message ordering

- What about order of messages?
  - In addition, we may require total ordering
  - Totally-ordered – whether unordered, FIFO or causally ordered, msgs are delivered in same order to all processes

- Totally ordered
  - \( p_i \) sends timestamped \( msg_i \) to all others
  - Message is put in a local queue \( queue_i \)
  - Any incoming message at \( p_j \) is queued in \( queue_j \), by its timestamp, and ack to every other process
  - \( p_j \) passes \( msg_i \) to its application if:
    1. \( msg_i \) is at the head of \( queue_j \)
    2. for each \( p_k \), there is a \( msg_k \) in \( queue_j \) with a larger timestamp

Violates total order
### Message ordering

- Virtual synchronous reliable multicasting with totally-ordered delivery – atomic multicasting
  - e.g. causal multicast and causal atomic multicast – causal-ordered without/with total-ordered delivery

<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</td>
<td>No</td>
</tr>
<tr>
<td>Causal multicast</td>
<td>Causal-ordered</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</td>
<td>Yes</td>
</tr>
<tr>
<td>Causal atomic multicast</td>
<td>Causal-ordered</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Virtual synchronous multicast in ISIS

- Relies on reliable, ordered, unicast – TCP
  - Multicast – reliable unicast each member in the group
- Problem to solve – guarantee that all msgs sent to view G are delivery to all non-faulty processes in G before a membership change
- To deal with crashed sender, every process in G keeps the message until it is sure everybody got it – i.e. message is stable
- Only stable messages can be delivered
Virtual synchronous multicast in ISIS

- When a process P receives view-change msg for $G_{i+1}$,
  - Forwards a copy of any unstable message from $G_i$ to all processes in $G_{i+1}$
    - When Q receives a copy of m sent in $G_i$, it delivers m (discards it if dup)
  - Marks message as stable (remember – reliable point-to-point, so no forward is lost)
  - To indicate it has no unstable messages left, so it’s ready to install new view, mcast a flush message
  - When it receives a flush message from all, installs new view
Distributed commit

- Atomic multicast – a form of distributed commit
- Given a computation distributed across a process group, ensure that either all processes commit to the final result, or none of them do
  - One-phase commit
    - Coordinator tells everyone what to do – no way to know if they did it or not
  - Two-phase commit
    - Coordinator makes sure everybody is going to do it
    - It can’t handle coordinator failure
  - Three-phase commit
    - Avoids blocking all until coordinator recovers, rarely used in practice
Two-phase commit

- Client that initiates computation acts as coordinator (C); processes required to commit are participants (P)
- Phases
  - 1a: C sends vote-request to all (a pre-write)
  - 1b: When P receives vote-request it returns either vote-commit or vote-abort to C; if it sends vote-abort, it aborts its local computation
  - 2a: C collects all votes; if all are vote-commit, it sends global-commit to all, otherwise it sends global-abort
  - 2b: Each P waits for global-commit or global-abort and handles accordingly
2PC and failures

- Participant
  - Initial state – no problem, P was unaware of the protocol
  - Ready state – waiting to either commit/abort, ask other P what to do
  - Abort state – make intro into abort state idempotent
  - Commit state – also make entry into commit state idempotent

- Coordinator
  - Record that it is entering WAIT so that it can possible retransmit the VOTE_REQUEST after recovering
  - If it has decided either ABORT or COMMIT, retransmit it when recovered

- If coordinator crashed when all participants have received and process the VOTE_REQUEST, everybody blocks!
Recovery

- When a failure occurs, bring system to error-free state
  - Forward error recovery – find a new state from which the system can continue operation, e.g. erasure code
    - Errors must be known in advance
  - Backward error recovery – bring system back into a previous error-free state, e.g. checkpointing & rollback
    - Application independent
    - Use backward error recovery, requires establishing recovery points (kept in stable storage)
    - Not everything can be rollback (ATM withdraw)
    - Performance hit – combine checkpointing with logging

- Recovery in distributed systems – processes need to cooperate in identifying a consistent state from where to recover – a distributed snapshot
Consistent recovery state

- Every message received is also shown to have been sent in the state of the sender
- Recovery line – assuming processes regularly checkpoint their state, the most recent consistent global checkpoint

If checkpointing is done at the “wrong” times, the recovery line may lie at system startup time ⇒ cascaded rollback
Coordinated checkpointing

- Independent checkpointing
  - Mayor problem – computing the recovery line

- Coordinated checkpointing
  - Each process takes checkpoint after a globally coordinated action
    - Simple solution: Use a two-phase blocking protocol
      - A coordinator multicasts a *checkpoint request* msg
      - When a participant receives this msg, it takes a checkpoint, stops sending (application) msgs, and reports back that it has taken a checkpoint
      - When all checkpoints have been confirmed at the coordinator, the latter broadcasts a *checkpoint done* msg to allow all processes to continue

- It is possible to consider only processes that depend on the recovery of the coordinator, and ignore the rest
Message logging

- Instead of taking an (expensive) checkpoint, try to replay your (communication) behavior from the most recent checkpoint ⇒ store messages in a log

- Assume a piecewise deterministic execution model:
  - The execution of each process can be considered as a sequence of state intervals
  - Each state interval starts with a nondeterministic event (e.g., message receipt)
  - Execution in a state interval is deterministic

- If we record nondeterministic events (for later replay), we obtain a deterministic execution model that will allow a complete replay
Message logging and consistency

- When should we actually log messages?
- Issue: Avoid orphans:
  - Q has just received and subsequently delivered $m1$ and $m2$
  - Assume that $m2$ is never logged
  - After delivering $m1$ and $m2$, Q sends msg $m3$ to process R
  - R receives and subsequently delivers $m3$

- When Q recovers, it doesn’t replay $m2$, so $m3$ may not take place!
- We need message logging schemes in which orphans do not occur
Message-logging schemes

- **Hdr[^m]** – header of msg contains src, dest, seq #, …
  - All what’s needed to resend and deliver it in the correct order
  - A msg \( m \) is stable if Hdr[^m] cannot be lost (in stable storage)
- **DEP[^m]** – set of processes to which \( m \), or another msg causally depending on \( m \), has been delivered
- **COPY[^m]** – set of processes that have a copy of Hdr[^m] in their volatile memory
- If C is a collection of crashed processes, then Q is an orphan if there’s a msg \( m \) such that Q is in DEP[^m]
  and every process in COPY[^m] has crashed (i.e. \( \subseteq C \))
  - That is, Q depends on \( m \) but there’s no way to replay \( m \)’s transmission
Message-logging schemes

- Goal: No orphans means that for each msg \( m \), \( DEP(m) \subseteq COPY(m) \)

- Pessimistic protocol: for each non-stable msg \( m \), there is at most one process dependent on \( m \), \( |DEP(m)| \leq 1 \)
  - An unstable msg must be made stable before sending another

- Optimistic protocol: for each unstable message \( m \), we ensure that if \( COPY(m) \subseteq C \), then eventually also \( DEP(m) \subseteq C \), where \( C \) denotes a set of processes that have been marked as faulty
  - To guarantee that \( DEP(m) \subseteq C \), we generally rollback each orphan process \( Q \) until \( Q \) not-in \( DEP(m) \)
Summary

- Fault tolerant becomes increasingly important for distributed systems
- Redundancy is the key technique to achieve fault tolerance
- With process redundancy, you need agreement
- And, of course, once a failure has occurred, there’s nothing to do but to recover to a correct state