Fault Tolerance


Hongfei Yan
School of EECS, Peking University
5/5/2008

“Failure is not an option. It comes bundled with your software.” (--unknown)

“You know you have [a distributed system] when the crash of a computer you've never heard of stops you from getting any work done.” (--Leslie Lamport)

Some real-world datapoints

Outline

Fault handling approaches

Design Goal

| Table 2: Node failures that were attributed to hardware in Mean in Fault. Bianca Schroeder problems broken down by the responsible hardware component. This includes all failures, not only those that required replacement of a hardware component. |

<table>
<thead>
<tr>
<th>Component</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>94</td>
</tr>
<tr>
<td>Memory</td>
<td>24</td>
</tr>
<tr>
<td>Hard drive</td>
<td>16</td>
</tr>
<tr>
<td>PCI motherboard</td>
<td>9</td>
</tr>
<tr>
<td>Power supply</td>
<td>2</td>
</tr>
</tbody>
</table>

Fault prevention: prevent the occurrence of a fault

Fault tolerance: build a component in such a way that it can meet its specifications in the presence of faults (i.e., mask 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

(with regard to fault tolerance):

Design a (distributed) system that can recover from partial failures without affecting correctness or significantly impacting overall performance
一个进程\( P \)可能依赖不同计算机上其他进程提供的服务，如果那些进程由于出现错误或故障而失去联系，则\( P \)无法正常运行。

计算机死机，或许网络断开，或许对方负载太重，暂时无法提供所需的服务。

如果选取一组计算机联合执行同一个任务，当个别或少数计算机出现错误和故障时，大多数计算机仍然能够正常地完成任务。

E.g., 分布式复制系统

Note: For distributed systems, components can be either processes or channels

Being fault tolerant is stringly related to what are called dependable systems.

Some properties of dependability:
- Availability
- Readiness for usage
- Reliability
- Continuity of service delivery
- Safety
- Low probability of catastrophes
- Maintainability
- How easy can a failed system be repaired

Failure: When a component is not living up to its specifications, a failure occurs.

Error: The part of a component's state that can lead to a failure.


Fault types generally (in terms of their properties):
- Transient: occur once and then disappear
- Intermittent: occur, then vanish, then reappear
- Permanent: continues to exist

Failure types generally (in terms of their properties):
- Transient: occur once and then disappear
- Intermittent: occur, then vanish, then reappear
- Permanent: continues to exist

Failure Models (in terms of their specifications)
- Crash failures: A component simply halts, but behaves correctly before halting.
- Omission failures: A component fails to respond to incoming requests.
- Receive omission: Fails to receive incoming messages.
- Send omission: Fails to send messages.
- Timing failures: The output of a component is correct, but lies outside a specified real-time interval.
- E.g., performance failures: too slow.
- Value failure: The wrong value is produced.
- State transition failure: Execution of the component's service brings it into a wrong state.
- Arbitrary (byzantine) failures: A component may produce arbitrary output and be subject to arbitrary timing failures.

Note: Crash failures are the least severe; arbitrary failures are the worst.

Problem: Clients cannot distinguish between a crashed component and one that is just a bit slow.

Examples: Consider a server from which a client is expecting output:
- Is the server perhaps exhibiting timing or omission failures?
- Is the channel between client and server faulty (crashed, or exhibiting timing or omission failures)?

Fail-stop: The component exhibits crash failures, but its failure can be detected (either through announcement or timeouts).

Fail-silent: The component exhibits omission or crash failures; clients cannot tell what went wrong.

Fail-safe: The component exhibits arbitrary; but benign failures (they can't do any harm).

Main approach: mask failures using redundancy.

- Information redundancy
  - E.g., a Hamming code can be added to transmitted data to recover from noise on the transmission line.
- Time redundancy
  - Is especially helpful for transient or intermittent faults.
  - E.g., using transactions
- Physical redundancy
  - E.g., 747s have four engines but can fly on three.
Outline

- Basic concepts
- Process resilience
- Reliable client-server communication
- Reliable group communication
- Distributed commit
- Recovery

Process Resilience

Basic issue: Protect against faulty processes

Solution: Process Groups

- Replicate and distribute computations in a group.
  - provide abstraction for a collection of processes
  - "identical" processes
  - all members receive all messages sent to the group

Process Groups

- Flat groups: Good for fault tolerance as information exchange immediately occurs with all group members.
  - however, may impose more overhead as control is completely distributed (hard to implement).
- Hierarchical groups: All communication through a single coordinator ⇒ not really fault tolerant and scalable, but relatively easy to implement.

Q: What consistency protocols fit best for each approach?

Issue: group membership

Groups and Failure Masking (1/4)

Terminology: a k-fault tolerant group can mask any k concurrent member failures (k is called degree of fault tolerance).

Problem: how large does a k-fault tolerant group need to be?

- From a client perspective:
  - Assume crash failure semantics ⇒ a total of k + 1 members are needed to survive k member failures.
  - Assume arbitrary (byzantine) failure semantics and group output defined by voting collected by the client ⇒ a total of 2k+1 members are needed to survive k member failures.

Assumption: all members are identical, and process all input in the same order (atomic multicast problem) ⇒ only then are we sure that they do exactly the same thing.

- From a process group perspective (reaching agreement) the problem is more complex

Groups and Failure Masking (2/4)

Assumption: Group members are not identical, i.e., we have a distributed computation

Problem: Non-faulty group members should reach agreement on the same value

Observation: Assuming arbitrary failure semantics, we need 3k + 1 group members to survive the attacks of k faulty members

Note: This is also known as Byzantine failures.

Essence: We are trying to reach a majority vote among the group of loyalists, in the presence of k traitors ⇒ need 2k + 1 loyalists.

Groups and Failure Masking (3/4)

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

- what they send to each other
- what each one got from the other
- what each one got in second step
Groups and Failure Masking (4/4)

**Issue:** What are the necessary conditions for reaching agreement?

**Process:** Synchronous ⇒ operate in lockstep

**Delays:** Are delays on communication bounded?

**Ordering:** Are messages delivered in the (real time) order they were sent?

**Transmission:** Are messages sent one-by-one, or multicast?

----

Summary so far

- Use replication to provide fault tolerance
- When process groups are used, reaching agreement is a key requirement.

**Main results:**
- Two army-problem
  - Impossible to design a protocol that guarantees that reach agreement is always reached with unreliable unicast communication
- Byzantine generals problem:
  - In a system with $k$ faulty processes, agreement can be achieved only if $2k+1$ correctly functioning processes are present. (Lamport, 1982)
  - If messages cannot be guaranteed to be delivered within a known, finite time, no agreement is possible even with one faulty process. (Fischer, 1985)

----

Failure Detection

**Essence:** Detect failures through timeout mechanisms

- Setting timeouts properly is difficult and application dependent
- You cannot distinguish process failures from network failures
- Need to consider failure notification throughout the system:
  - Gossiping (i.e., proactively disseminate a failure detection)

Outline

- Basic concepts
- Process resilience
- Reliable client-server communication
- Reliable group communication
- Distributed commit
- Recovery

----

Reliable Communication

**So far:** Concentrated on process resilience (by means of process groups).

**Q:** 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 so much redundancy that corrupted packets can be automatically corrected
- Request retransmission of lost, or last $N$ packets

**Observation:** Most of this work assumes point-to-point communication

Reliable RPC (1/3)

**What can go wrong?:**
1. Client cannot locate server
2. Client request is lost
3. Server crashes
4. Server response is lost
5. Client crashes

[1:] Relatively simple - just report back to client
[2:] Just resend message (and use messageID to uniquely identify messages)
Reliable RPC (2/3)

[3] Server crashes are harder as you don’t know what the server has already done:

- **Problem**: we need to decide on what to expect from the server
  - **At-least-once-semantics**: The server guarantees it will carry out an operation at least once, no matter what.
  - **At-most-once-semantics**: The server guarantees it will carry out an operation at most once, but possibly none at all.

Reliable RPC (3/3)

[4:] Lost replies ⇒ Detection hard: because it can also be that the server had crashed. You don’t know whether the server has carried out the operation

- **Solution**: None, (works sometimes) make your operations **idempotent**: repeatable without any harm done if it happened to be carried out before.

[5:] Client crashes ⇒ The server is doing work and holding resources for nothing (called doing an orphan computation).

- Orphan is killed by client when it reboots
- Broadcast new epoch number when recovering ⇒ servers kill orphans
- Require computations to complete in a T time units. Old ones are simply removed.

**Outline**

- Basic concepts
- Process resilience
- Reliable client-server communication
- Reliable group communication
- Distributed commit
- Recovery

Reliable Multicasting (1/2)

Model: a multicast channel $c$ with two (possibly overlapping) groups:

- **The sender group** $SND(c)$ of processes that submit messages to channel $c$
- **The receiver group** $RCV(c)$ of processes that receive messages from channel $c$

Possible reliability requirements:

- **Simple reliability**: No messages lost
  - 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 $P$

- **Virtually synchronous multicast**: All active processes receive the same thing
  - Ensure that a message $m$ submitted to channel $c$ is delivered to process $P \in RCV(c)$ only if $m$ is delivered to all members of $RCV(c)$

Reliable Multicasting (2/2)

**Observation**: If we can stick to a local-area network, reliable multicasting is ‘easy’

**Principle**: 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

**Question**: Why doesn’t this scale?

Scalable Reliable Multicasting: Feedback Suppression

**Basic idea**: Let a process $P$ suppress its own feedback when it notices another process $Q$ is already asking for a retransmission

**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

**Question**: Why is the random schedule so important?
Basic solution: Construct a hierarchical feedback channel in which all submitted messages are sent only to the root. Intermediate nodes aggregate feedback messages before passing them on.

Question: What's the main problem with this solution?
Observation: Intermediate nodes can easily be used for retransmission purposes.

Virtual Synchronous Multicast

Idea: Formulate reliable multicasting in the presence of process failures in terms of process groups and changes to group membership.

Guarantee: A message is delivered only to the non-faulty members of the current group. All members should agree on the current group membership.

Virtual Synchrony – Notes (1/2)

Essence: Consider views $V = RCV(c) \cup SND(c)$

Properties of virtually synchronous multicast:
(1) For each consistent state, there is a unique view on which all its members agree.
   - Note: implies that all non-faulty processes see all view changes in the same order
(2) If message $m$ is sent to $V$ before a view change $vc$ to $V'$, then either all $P \in V$ that execute $vc$ receive $m$, or no processes $P \in V$ that execute $vc$ receive $m$.
   - Note: all non-faulty members in the same view get to see the same set of multicast messages.
(3) A message sent to view $V$ can be delivered only to processes in $V$, and is discarded by the following views

Virtual Synchrony – Notes (2/2)

A sender to a view $V$ need not be member of $V$

If a sender $S \in V$ crashes, its multicast message $m$ is flushed before $S$ is removed from $V$. $m$ will never be delivered after the point that $V$ changes

- Notes: Messages from $S$ may still be delivered to all, or none (nonfaulty) processes in $V$ before they all agree on a new view to which $S$ does not belong.
- If a receiver $P$ fails, a message $m$ may be lost but can be recovered as we know exactly what has been received in $V$. Alternatively, we may decide to deliver $m$ to members in $V - \{P\}$

Message Ordering (1/2)

Observation: Virtually synchronous behavior is independent from the ordering of message delivery.
- The only issue is that messages are delivered to an agreed upon group of receivers.

<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 m2</td>
<td>receives m1</td>
<td>sends m4</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.
Message ordering (2/2)

- 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 (1/3)

- Assumptions:
  - Point-to-point communication in the underlying network:
    - Reliable, in-order delivery (TCP-like semantics)
  - Multicast implemented as a sequence of point-to-point transmissions
    - But sender can fail before sending to all receivers
- Requirements
  - All messages send while in view Gi are delivered to all non-faulty processes in Gi before the next group membership change
  - Gi+1 might be installed before m is delivered

Implementing Virtual Synchrony (2/3)

- Solution clue:
  - Every process in Gi keeps m until it knows for sure that all other members in Gi have received m
  - Terminology: m is stable if received by all processes in Gi
  - Only stable messages are delivered

Implementing Virtual Synchrony (3/3)

- Algorithm sketch
  - P detects a view change
  - Forwards any unstable message in Gi to all processes in Gi
  - P sends a ‘flush’ message
  - P collects a ‘flush response’ from everyone
  - P installs new view
  - Q (another process)
    - When receiving m in the view Gi it believes, it delivers m
      - (after ordering)
    - When receiving a flush message
      - Multicasts all its unstable messages
      - Sends the ‘flush response’
      - Installs new view
  - Control messages so that each process knows what are the messages received by everyone else

Implementing Virtual Synchrony - Example

- Process 4 notices that process 7 has crashed, sends a view change
- Process 6 sends out all its unstable messages, followed by a flush message
- Process 6 installs the new view when it has received a flush message from everyone else

Virtual Synchrony Implementation (1/3)

- The current view is known at each P by means of a delivery list dest(P)
  - If P ∈ dest(Q) then Q ∈ dest(P)
  - Messages received by P are queued in queue[P]
  - If P fails, the group view must change, but not before all messages from P have been flushed
  - Each P attaches a (stepwise increasing) timestamp with each message it sends
- Assume FIFO-ordered delivery; the highest numbered message from Q that has been received by P is recorded in rcvd[P][Q]
- The vector rcvd[P][Q] is sent (as a control message) to all members in dest(P)
- Each P records rcvd[O][Q] in remote[P][Q]
Virtual Synchrony Implementation (2/3)

- **Observation:** remote[P][Q] shows what P knows about message arrival at Q
  
  \[
  \begin{array}{cccc}
  1 & 2 & 3 & 4 \\
  2 & 2 & 2 & 4 \\
  3 & 3 & 1 & 4 \\
  4 & 4 & 2 & 4 \\
  \end{array}
  \]
  
  \[\text{min} = 2 1 1 4\]

- A message is **stable** if it has been received by all \(Q \in \text{dest}[P]\) (shown as the \(\text{min}\) vector)
- Stable messages can be delivered to the next layer (which may deal with ordering). **Note:** Causal message delivery comes for free
- As soon as all messages from the faulty process have been flushed, that process can be removed from the (local) views

Virtual Synchrony Implementation (3/3)

- **Remains:** What if a sender \(P\) failed and not all its messages made it to the nonfaulty members of the current view?
- **Solution:** Select a coordinator which has all (unstable) messages from \(P\), and forward those to the other group members.
- **Note:** Member failure is assumed to be detected and subsequently multicast to the current view as a view change. That view change will not be carried out before all messages in the current view have been delivered.

Distributed Commit

- Two-phase commit
- Three-phase commit

- **Essential issue:** Given a computation distributed across a process group, how can we ensure that either all processes commit to the final result, or none of them do (atomicity)?

Two-Phase Commit

- **Model:** The client who initiated the computation acts as coordinator; processes required to commit are the participants
- **Phase 1a:** Coordinator sends vote-request to participants (also called a pre-write)
- **Phase 1b:** When participant receives vote-request it returns either vote-commit or vote-abort to coordinator. If it sends vote-abort, it aborts its local computation
- **Phase 2a:** Coordinator collects all votes; if all are vote-commit, it sends global-commit to all participants, otherwise it sends global-abort
- **Phase 2b:** Each participant waits for global-commit or global-abort and handles accordingly.

Two-Phase Commit FSMs

- Where does the waiting/blocking occur?
  - Coordinator-WAIT
  - Participant-INIT
  - Participant-READY

Two-Phase Commit Recovery (1/2)

- What happens in case of a crash? How do we detect a crash?
  - If timeout in Coordinator-WAIT, then abort.
  - If timeout in Participant-INIT, then abort.
  - If timeout in Participant-READY, then need to find out if globally committed or aborted.
  - Just wait for Coordinator to recover.
  - Check with others.
Two-Phase Commit Recovery (2/2)

- If in Participant-READY, and we wish to check with others:
  - If Q is in COMMIT, then commit. If Q is in ABORT, then ABORT.
  - If Q in INIT, then can safely ABORT.
- If all in READY, nothing can be done.

Three-Phase Commit

The states of the coordinator and each participant satisfy the following two conditions:
1. There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.
2. 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 (1/2)

(a) The finite state machine for the coordinator in 3PC.
(b) The finite state machine for a participant.

Three-Phase Commit (2/2)

Recovery

- Introduction
- Checkpointing
- Message logging

Recovery: Background

- **Essence:** When a failure occurs, we need to bring the system into an error-free state:
  - Forward error recovery: Find a new state from which the system can continue operation
  - Backward error recovery: Bring the system back into a previous error-free state
- **Practice:** Use backward error recovery, requiring that we establish recovery points
- **Observation:** Recovery in distributed systems is complicated by the fact that processes need to cooperate in identifying a consistent state from where to recover
Consistent Checkpoints

- **Requirement**: Every message that has been 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.
- **Observations**: If and only if the system provides reliable communication, should sent messages also be received in a consistent state.

Cascaded Rollback

- **Observation**: If checkpointing is done at the “wrong” instants, the recovery line may lie at system startup time ⇒ cascaded rollback.
- **Known as**: The “domino effect”.

Checkpointing: Stable Storage

- **Principle**: Replicate all data on at least two disks, and keep one copy “correct” at all times.

- **After a crash**:
  - If both disks are identical: you’re in good shape.
  - If one is bad, but the other is okay (checksums): choose the good one.
  - If both seem okay, but are different: choose the main disk.
  - If both aren’t good: you’re not in a good shape.

Independent Checkpointing

- **Essence**: Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.
  - Let $CP_i(m)$ denote the $m$th checkpoint of process $Pi$ and $INT_i(m)$ the interval between $CP_i(m−1)$ and $CP_i(m)$.
  - When process $Pi$ sends a message in interval $INT_i(m)$, it piggybacks $(i,m)$.
  - When process $Pj$ receives a message in interval $INT_j(n)$, it records the dependency $INT_i(m) → INT_j(n)$.
  - The dependency $INT_i(m) → INT_j(n)$ is saved to stable storage when taking checkpoint $CP_j(n)$.

- **Observation**: If process $Pi$ rolls back to $CP_i(m−1)$, $Pj$ must roll back to $CP_j(n−1)$.

Coordinated Checkpointing

- **There are distributed snapshot techniques that can help, but complex**.
- **An alternative is to use a global coordinator**.
  - Multicast a $CHECKPOINT\_REQUEST$ message.
  - Upon receipt, take a local checkpoint, block any new messages the application gives, and sends an ACK.
  - When coordinator gets an ACK from all processes, it sends back $CHECKPOINT\_DONE$.

Message Logging

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

- **Assumption**: We 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).

- **Conclusions**: When we record nondeterministic events (to replay them later), we obtain a deterministic execution model that will allow us to do a complete replay.

- **Questions**:
  - Why is logging only messages not enough?
  - Is logging only nondeterministic events enough?
Message Logging and Consistency

Problem: When should we actually log messages?

Issue: Avoid orphans.
- Process Q has just received and subsequently delivered messages m1 and m2.
- Assume that m2 is never logged.
- Process R receives and subsequently delivers m3.

Goal: Devise message logging schemes in which orphans do not occur.

Message-Logging Schemes (1/2)

- HDR[m]: The header of message m containing its source, destination, sequence number, and delivery number.
  - The header contains all information for resending a message and delivering it in the correct order (assume data is reproduced by the application).
  - A message m is stable if HDR[m] cannot be lost (e.g., because it has been written to stable storage).
- DEP[m]: The set of processes to which message m has been delivered, as well as any message that causally depends on delivery of m.
- COPY[m]: The set of processes that have a copy of HDR[m] in their volatile memory.

Goal: No orphans means that for each message m, DEP[m] ⊆ COPY[m].

Pessimistic protocol: for each nonstable message m, there is at most one process dependent on m, that is |DEP[m]| ≤ 1.

Consequence: An unstable message in a pessimistic protocol must be made stable before sending a next message.

Optimistic protocol: for each unstable message m, we ensure that if COPY[m] ⊆ C, then eventually also DEP[m] ⊆ C, where C denotes a set of processes that have been marked as faulty.

Consequence: To guarantee that DEP[m] ⊆ C, we generally rollback each orphan process Q until Q ∈ DEP[m].

References

- Chapter 7 of [Tanenbaum, 2002]
- Chapter 9 of [Xining Li, 2006]
  - book1, [Coulouri, 2005]
  - book2, [Birman, 2005]
  - book3, [Tanenbaum, 2006]
  - book4, [Tanenbaum, 2002]
  - Book5, [Xining Li, 2006]