Introduction to fault tolerance

Availability

The availability of a system is the probability that the system will perform its required action.

Example:

- one workstation
- crashes once a week
- takes 2 minutes to recover

Availability:

\[1 - p_{\text{crash}} = 1 - 2 \cdot 10^{-4} = 0.9998\]
Availability

The availability of a system is the probability that the system will perform its required action.

Example:

- One workstations
- Crash once a week
- Take 2 minutes to recover

Availability:

\[(1 - p_{\text{crash}})^{30} \approx 1 - n \cdot p_{\text{crash}} = 0.994\]

Increasing availability

- Avoid a single point of failure
- Use replication (in time, or space)
- Replicas communicate through narrow interface (e.g. send/receive)

Example

- Replicate the workstation 30 times

Availability:
Increasing availability

- Avoid a single point of failure
  - use replication (in time, or space)
  - replicas communicate through narrow interface (e.g. send/receive)
- Example
  - Replicate the workstation 30 times

Availability:
\[ 1 - p^n_{\text{crash}} = 1 - (2 \cdot 10^{-4})^{30} = 1 - 10^{111} \]

Modeling faults

- Mean Time To Failure/ Mean Time To Recover
  - close to hardware
- Threshold: \( f \) out of \( n \)
  - makes condition for correct operation explicit
  - measures fault-tolerance of architecture, not single components
- Set of explicit failure scenarios

A hierarchy of failure models

- Crash

A hierarchy of failure models

- Fail-stop
- Crash
A hierarchy of failure models

Fail-stop \rightarrow Crash

Send Omission \rightarrow Receive Omission

benign failures

Arbitrary failures with message authentication

Arbitrary (Byzantine) failures
Replication in space

- Run parallel copies of a unit
- Vote on replica output
- Failures are masked
- High availability, but at high cost

Replication in time

- When a replica fails, restart it (or replace it)
- Failures are detected, not masked
- Lower maintenance, lower availability
- Tolerates only benign failures
- Better than you think...

Non-determinism

An event is non-deterministic if the state that it produces is not uniquely determined by the state in which it is executed

Handling non-deterministic events at different replicas is challenging

- Replication in time requires to reproduce during recovery the original outcome of all non-deterministic events
- Replication in space requires each replica to handle non-deterministic events identically

Primary-Backup
The Idea

- Clients communicate with a single replica (the primary)
- The primary updates the other replicas (backups)
- Backups detect the failure of the primary using a timeout mechanism,
- Clients fail over to a backup

Note: Non-deterministic events are executed only at the primary

PB: A specification
(Budhiraja, Marzullo, Schneider, Toueg)

PB1: There exists a local predicate $Prm_i$ on the state of each server $s$. At any time, there is at most one server $s$ whose state satisfies $Prm_i$.

PB2: Each client $i$ maintains a server identity $Dest_i$ such that to make a request, client $i$ sends a message to $Dest_i$.

PB3: If a client request arrives at a server that is not the current primary, then that request is not enqueued (and therefore is not processed)

PB4: There exist fixed values $k$ and $\Delta$ such that the service behaves like a single $(k,\Delta)$-bofo server

Terminology

- The failover time of a primary-backup service is the longest time the service can be without a primary
- The service has a server outage at $t$ if some correct client sends a request at time $t$ to the service, but does not receive a response
- A $(k,\Delta)$-bofo service is one in which all server outages can be grouped into at most $k$ intervals of time, each of at most length $\Delta$

A simple example: system model

- point-to-point communication
- non-faulty channels
- upper bound $\delta$ on message delivery time
- at most one server crashes
A simple example: system model

- point-to-point communication
- non-faulty channels
- upper bound $\delta$ on message delivery time
- at most one server crashes

Two processes:
- the primary $p_1$
- the backup $p_2$

A simple example: $p_1$’s protocol

- Upon receipt of a client request, process $p_1$
  - consumes request and updates its state
  - sends state update message to $p_2$
  - sends response to client without waiting for ack from $p_2$
- $p_1$ sends heartbeat message to $p_2$ every $\tau$ seconds

A simple example: $p_2$’s protocol

- Upon receiving a state update from $p_1$, $p_2$
  - updates its state
- If $p_2$ does not receive a heartbeat for $\tau + \delta$ seconds,
  - $p_2$ declares itself primary
  - it informs the clients
  - it begins consuming subsequent requests from clients

It meets the spec...

PB1: \( Prmy_{p_1} \land Prmy_{p_2} = \text{false} \)

Failover: Time during which

\[ \neg Prmy_{p_1} \land \neg Prmy_{p_2} \]

\( Prmy_{p_1} \equiv p_1 \) has not crashed

\( Prmy_{p_2} \equiv p_2 \) has not received a message from $p_1$ for $\tau + \delta$ seconds
It meets the spec...

PB1: \( Prmyp_1 \land Prmyp_2 = false \)

Failover: Time during which
\( \neg Prmyp_1 \land \neg Prmyp_2 \)

Prmyp_1 \equiv \text{p1 has not crashed}
Prmyp_2 \equiv \text{p2 has not received a message from p1 for } \tau + \delta \text{ seconds}
...indeed, it does!

- PB2, PB3: Follow immediately from protocol
- PB4: Find \( k, \Delta \) to implement \((k,\Delta)\)-bofo server

\[ \Delta = \tau + 4\delta \]

Some like it hot

- **Hot** Backups process information from the primary as soon as they receive it
- **Cold** Backups log information received from primary, and process it only if primary fails
- **Rollback Recovery** implements cold backups cheaply:
  - the primary logs directly to stable storage the information needed by backups
  - if the primary crashes, a newly initialized process is given content of logs—backups are generated "on demand"
State-Machine Replication

The Problem

Clients → Server

Solution: replicate server!

The Solution
1. Make server deterministic (state machine)

The Solution
1. Make server deterministic (state machine)
2. Replicate server
The Solution
1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions
4. Vote on replica outputs for fault-tolerance

A conundrum
A: voter and client share fate!

Clients
State machines

A solution diagram showing the flow of commands and state transitions from clients to state machines, and the role of a voter in fault-tolerance.
State Machines
- Set of state variables + Sequence of commands
- A command
  - Reads its read set values (opt. environment)
  - Writes to its write set values (opt. environment)
- A deterministic command
  - Produces deterministic wsvs and outputs on given rsv
- A deterministic state machine
  - Reads a fixed sequence of deterministic commands

Replica Coordination
- All non-faulty state machines receive all commands in the same order
  - Agreement: Every non-faulty state machine receives every command
  - Order: Every non-faulty state machine processes the commands it receives in the same order

Where should RC be implemented?
- In hardware
  - sensitive to architecture changes
- At the OS level
  - state transitions hard to track and coordinate
- At the application level
  - requires sophisticated application programmers

Hypervisor-based Fault-tolerance
- Implement RC at a virtual machine running on the same instruction-set as underlying hardware
- Undetectable by higher layers of software
- One of the great come-backs in systems research!
  - CP-67 for IBM 369 [1970]
  - Xen [SOSP 2003], VMware
The Hypervisor as a State Machine

- Two types of commands
  - virtual-machine instructions
  - virtual-machine interrupts (with DMA input)
- State transition must be deterministic
  - ...but some VM instructions are not (e.g. time-of-day)
- Interrupts must be delivered at the same point in command sequence

The Architecture

- Good-ol’ Primary-Backup
- Primary makes all non-deterministic choices
- I/O Accessibility Assumption
  Primary and backup have access to same I/O operations

Ensuring identical command sequences

- Ordinary (deterministic) instructions
- Environment (nondeterministic) instructions

Ensuring identical command sequences

- Ordinary (deterministic) instructions
- Environment (nondeterministic) instructions
- Environment Instruction Assumption
  Hypervisor captures all environmental instructions, simulates them, and ensures they have the same effect at all state machines
Ensuring identical command sequences

- Ordinary (deterministic) instructions
- Environment (nondeterministic) instructions
- Environment Instruction Assumption
- VM interrupts must be delivered at same point in instruction sequence at all replicas

Instruction Stream Interrupt Assumption

The failure-free protocol

P0: On processing environment instruction \( i \) at \( pc \), HV of primary \( p \):
- sends \([e_p, pc, Val]\) to backup \( b \)
- waits for ack
P1: If \( b \) receives \( Int \) from its VM:
- \( p \) buffers \( Int \) until epoch ends
P2: If epoch ends at \( p \):
- \( p \) sends to \( b \) all buffered \( Int \) in \( e_p \)
- \( p \) waits for ack
- \( p \) delivers all VM, \( Int \) in \( e_p \)
- \( e_p := e_p + 1 \)
- \( p \) starts \( e_p \)

P3: If \( b \) HV processes environment instruction \( i \) at \( pc \):
- \( b \) waits for \([e_b, pc, Val]\) from \( p \)
- \( b \) returns \( Val \)
- If \( b \) receives \([E, pc, Val]\) from \( p \):
  - \( b \) sends ack to \( p \)
  - \( b \) buffers \( Val \) for delivery at \( E, pc \)
P4: If \( b \) HV receives \( Int \) from its VM
  - \( b \) ignores \( Int \)
P5: If epoch ends at \( b \):
  - \( b \) waits from \( p \) for interrupts for \( e_b \)
  - \( b \) sends ack to \( p \)
  - \( b \) delivers all VM, \( Int \) buffered in \( e_b \)
  - \( e_b := e_b + 1 \)
  - \( b \) starts \( e_b \)
If the primary fails...

**P6:** If \( b \) receives a failure notification instead of \([e_b, pc, Val]\), \( b \) executes \( i \)

If in **P5** \( b \) receives failure notification instead of \( Int \):

\[ e_b := e_b + 1 \]

\( b \) starts \( e_b \) \( \leftarrow \) failover epoch

\( b \) is promoted primary for epoch \( e_b + 1 \)

**If** \( p \) **crashes before sending** \( Int \) **to** \( b \),

\( Int \) **is lost!**

---

SMR and the environment

- On outputs, no exactly-once guarantee on outputs
- On primary failure, avoid input inconsistencies
  - time must increase monotonically
    - at epoch, primary informs backup of value of its clock
  - interrupts must be delivered as a fault-free processor would
    - but interrupts can be lost...
    - weaken constraints on I/O interrupts

---

On I/O device drivers

**IO1:** If an I/O instruction is executed and the I/O operation performed, the processor issuing the instruction delivers a completion interrupt, unless it fails. Either way, the I/O device is unaffected.

**IO2:** An I/O device may cause an uncertain interrupt (indicating the operation has been terminated) to be delivered by the processor issuing the I/O instruction. The instruction could have been in progress, completed, or not even started.

On an uncertain interrupt, the device driver reissues the corresponding I/O instruction—not all devices though are idempotent or testable

---

Backup promotion and uncertain interrupts

**P7:** The backup’s VM generates an uncertain interrupt for each I/O operation that is outstanding right before the backup is promoted primary (at the end of the failover epoch)
The Hypervisor prototype

- Supports only one VM to eliminate issues of address translation
- Exploits unused privileged levels in HP's PA-RISC architecture (HV runs at level 1)
- To prevent software to detect HV, hacks one assembly HP-UX boot instruction

RC in the Hypervisor

- Nondeterministic ordinary instructions (Surprise!)
  - TLB replacement policy non-deterministic
  - TLB misses handled by software
  - Primary and backup may execute a different number of instructions!

- HV takes over TLB replacement

Optimizations

- $p$ sends $Int$ immediately
- $p$ blocks for acks only before output commit
Ahhh, Java...