Introduction to Fault-Tolerance

**Definition:** *Availability* is the probability that a system/subsystem/processor will perform its required action.

**Example**
- One workstation
- Crashes once a week
- Takes two minutes to recover

What is the availability?

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

Problem becomes worse when some processors have lower availability than others

What if we have 30 workstations?

\[
(1 - \cdot 2 \cdot 10^{-4})^{30} = 0.994
\]

(unavailable approximately 1 hour/week)

How do we build a highly available system?

- Avoid single point of failure
  - Use replication
- Keep units isolated
  - Allow units to communicate only through a narrow interface (send/receive)

**Example**
Suppose we replicate a workstation 30 times…

The availability of this system is the probability that at least one is up

\[
(1 - \cdot 2 \cdot 10^{-4})^{30} = 1 \cdot 10^{28}
\]

Is this good enough?

Age of Earth (in ns) \(10^{28}\)
Failure Models

- Mean Time To Failure/ Mean Time To Recover
  - close to hardware

- $f$ out of $n$
  - makes condition for correct operation explicit
  - measures fault-tolerance of architecture, not single components

Failure Hierarchies

- Crash
- Send Omission
- Receive Omission
- General Omission
- Arbitrary failures with message authentication
- Arbitrary (Byzantine) failures

Replication

Replication in Space
- run parallel copies of potentially faulty units
- very high availability
- if failures are rare, can be an overkill

Replication in Time
- when a unit fails, restart (or replace) it
- to be effective, failure must be intermittent

Detection and Masking

Two major approaches for fault-tolerance:
- Detect the failure of a unit and take appropriate action
  - roll back (and try again)—assumes intermittent failures
  - roll forward (discard computation; reset)
- Mask the failure of the unit so that a system with the faulty unit will continue functioning
Comparison

Detection is easier than masking

How many replicas are required to detect an arbitrary failure?

How many replicas are required to mask an arbitrary failure?

Determinism and Nondeterminism

- A unit is **deterministic** if it always exhibits the same response to an input given in the same state
  - state [input] single unique result
  - (which can be state of unit, or an output, or both)

- Otherwise, it is **nondeterministic**

Handling non-determinism

- Does it exist?
- Is it a problem?
- How do we deal with it?
- Does it matter how we deal with it?

Primary-Backup
Primary-Backup: The Idea

- One replica (primary) executes all non-deterministic events
- Primary broadcasts to other replicas (backups)
  - requests form clients
  - outcome of executing non-deterministic events at the primary

Definitions

- Failover time of a PB service: longest time during which there is no primary
- Server outage at t: Some correct client sends a request at time t to service, but does not receive a response
- \((k, \Delta)\)-bofo server: service in which all server outages can be grouped into at most \(k\) intervals of time, each of at most length \(\Delta\)

Primary-Backup: The Spec

(Budhiraja, Marzullo, Schneider, Toueg)

1. There exists a local predicate \(Prmy_s\) on the state of each server \(s\). At any time, there is at most one server \(s\) whose state satisfies \(Prmy_s\).
2. Each client \(i\) maintains a server identity \(Dest_i\), such that to make a request, client \(i\) sends a message to \(Dest_i\).
3. 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).
4. There exist fixed values \(k\) and \(\Delta\) such that the service behaves like a single \((k, \Delta)\)-bofo server.

A simple protocol...

Assume:
- point-to-point communication
- non-faulty channels
- upper bound on message delivery time
- at most one process crashes
- Primary \(p_1\)
- Backup \(p_2\)

On receipt of a request, process \(p_1\):
- Processes request and updates its state
- Sends info about update to \(p_2\) (state update message)
- Without waiting for an acknowledgment

In addition:
- \(p_2\) sends heartbeat message to \(p_1\) every \(\Delta\) seconds
- Process \(p_2\):
  - updates its state upon receiving state update from \(p_1\)
  - if \(p_1\) doesn't receive heartbeat for \(\Delta\) seconds, \(p_2\) becomes primary
  - informs clients
  - begins processing subsequent requests from clients
Definition of Prmy: 

- $Prmy_p = p_1$ has not crashed 
- $Prmy_p = p_2$ has not received a message from $p_1$, for $\Delta + \Delta$

PB1: $Prmy_p \land \neg Prmy_p_2 = \text{false}$

Failover: Time during which $\neg Prmy_p \land \neg Prmy_p_2$

PB2, PB3: Follow immediately from protocol

PB4: Find $k, D$ to implement $(k, D)$-bofo server

Primary-Backup vs. Active Replication

**Primary Backup**
- does not tolerate arbitrary failures
- if the primary fails, requests may be lost
- service can become unavailable while a leader election algorithm is run to determine the new primary
- consumes less resources

**Active Replication**
- tolerates arbitrary failures
- masks failures
- consumes lots of resources

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:
- primary logs directly to stable storage information needed by backups
- if primary crashes, a newly initialized process is given content of logs—backups are generated “on demand”