3PC: THE WEAKNESS REVEALED

How communication failures can compromise safety…
The election protocol

- Processes agree on linear ordering (e.g. by pid)
- Each process $p$ maintains a set $UP_p$ of all processes that it believes to be operational
- When $p$ detects failure of $c$, it removes $c$ from $UP_p$ and chooses smallest $q$ in $UP_p$ to be the new coordinator
- If $p = q$, then $p$ is the new coordinator
- Otherwise, $p$ sends UR-ELECTED to $q$

A network partition could split the processes into two sets, each believing the other set to contain faulty processes
3PC: THE ALTERNATIVE

There exists a variation of the 3PC protocol that is safe despite communication failures.

However, it is no longer live (non-blocking)!

(as we will see later in the semester, we can’t have both)
STATE MACHINE REPLICATION
MODELING FAULTS

- Mean Time To Failure/Mean Time To Recover
  - used mostly for disks
  - of questionable value in expressing reliability

- Threshold: $f$ out of $n$
  - makes condition for correct operation explicit
  - measures fault-tolerance of the architecture, not of individual components

- Enumerate failure scenarios
A HIERARCHY OF FAILURE MODELS

- Fail-stop
- Crash
- Send omission
- Receive omission
- General omission
- Arbitrary (Byzantine) failures

○ = benign failures
A HIERARCHY OF FAILURE MODELS
FAULT TOLERANCE: THE PROBLEM

Clients

Server

Solution: replicate the server
REPLICATION IN TIME

- When a server fails, restart it or replace it
- Failures are detected, not masked
- Lower maintenance, lower availability
- Tolerates only benign failures
Replication in space

- Run multiple copies of a server (replicas)
- Vote on replica output
- Failures are masked
- High availability and can tolerate arbitrary failures
  - but at high cost
THE ENEMY: non-determinism

An event is non-deterministic if its output is not uniquely determined by its input

The problem with non-determinism:

- Replication in time: must reproduce the original outcome of all non-deterministic events
- Replication in space: each replica must handle non-deterministic events identically
The solution: state machines

Design the server as a deterministic state machine
THE SOLUTION: STATE MACHINES

State machine example: a switch
STATE MACHINE REPLICATION

Ingredients: a server

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure that all replicas go through the same sequence of state transitions
4. Vote on replica outputs
STATE MACHINE REPLICAION

Ingredients: a server

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure that all replicas go through the same sequence of state transitions
4. Vote on replica outputs
**State Machine Replication**

**Ingredients: a server**

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure that all replicas go through the same sequence of state transitions
4. Vote on replica outputs
All state machines receive all commands in the same order.

**State Machine Replication**

Ingredients: a server

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure that all replicas go through the same sequence of state transitions
4. Vote on replica outputs
Ingredients: a server

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure that all replicas go through the same sequence of state transitions
4. Vote on replica outputs
1. Evil Lorenzo Speaks French
2. And was born in Corsica
3. Went to Dartmouth instead of Cornell
4. Rides a Ducati instead of a Moto Guzzi
5. Still listens opera, but he doesn't care for Puccini

When in trouble, cheat!
Voter and client share fate!

4. Vote on replica outputs
PRIMARY-BACKUP
Failure model: \textbf{crash}

Network model: \textbf{synchrony}

- Reliable, FIFO channels
- All messages are delivered within $\delta$ time
- Tolerates $f$ crash failures
The idea

- Clients communicate with a single replica (primary)

- Primary:
  - sequences and processes clients’ requests
  - updates other replicas (backups)

- Backups use timeouts to detect failure of primary

- On primary failure, a backup becomes the new primary
A SIMPLE PRIMARY-BACKUP PROTOCOL

\((f = 1)\)

Active replication: \(\text{sync} = \text{client request(s)}\)
Passive replication: \(\text{sync} = \text{state update}\)
WEAKENING THE MODEL

Failure model: **crash**

Network model: **synchrony**

- Unreliable, FIFO channels
- Channels may drop messages
- All messages are delivered within $\delta$ time
  - (looks paradoxical)

Tolerates $f$ crash failures
A SLIGHTLY DIFFERENT PRIMARY-BACKUP PROTOCOL

\( (f = 1) \)
GENERALIZING TO MORE BACKUPS

Primary

\[ f \text{ backups} \]
GENERALIZING TO MORE BACKUPS

update
Primary

\( f \) backups
GENERALIZING TO MORE BACKUPS

update

Primary

\( f \) backups
GENERALIZING TO MORE BACKUPS

(active updates)

$\text{Primary}$

$f$ backups
GENERALIZING TO MORE BACKUPS

(passive updates)

Primary

\( f \) backups
Generalizing to more backups

(passive updates)
Generalizing to more backups

$f \text{ backups}$

Primary

ack ack ack ack ack
GENERALIZING TO MORE BACKUPS

reply

Primary

\( f \) backups
HANDLING QUERIES

query

Primary

f backups
Handling queries

Primary

f backups
HANDLING QUERIES

However…

$f$ backups
Handling queries

Primary

$\mathcal{f}$ backups
The primary cannot respond until it has received all acks for prior updates.
Chain replication

Head

$f + 1$ replicas

Tail

update

query

reply
Chain replication

- Head
- \( f + 1 \) replicas
- Tail

update
Chain replication

Head \[ f + 1 \] replicas Tail

update
Chain replication

Head \quad f + 1 \text{ replicas} \quad \text{Tail}

update

reply
Chain replication

Tail can respond immediately, without waiting for the new update

Head \[ f + 1 \] replicas Tail
**SOME LIKE IT HOT**

- **Hot** backups process information from the primary as soon as they receive it.

- **Cold** backups log information received from the primary and process it only if the primary fails.