Fault Tolerance

CMSC 414

November 29, 2017

What is a Fault?

A **Fault** (also called a **Failure**) is *any undesired behavior*

This may be

- Failure to process input
- Incorrect processing of input
- Additional behavior unrelated to correctly processing input

Not always *adversarial* in nature
$\Rightarrow$ Sometimes things just fail

Often manifests as a violation of *Availability*, but can also be a violation of other security properties

Why Things Fail

Most of the time: *random bugs*

- Program you're using crashes
- Computer becomes unresponsive
- Spontaneous reboots

Often: *hardware failures*

- Drives go bad
- Cables get stretched/frayed
- Insufficient/failing cooling $\Rightarrow$ overheating

Sometimes: *attack*

- Garbage input
- Intrusions
- Denial-of-Service

Processes

We're going to talk about **Processes**

These can be

- Programs/Threads
- Device drivers
- Hosts
- Pretty much any other discrete units of interest

Fault Models

Fail-Stop A process stops running, and issues a notification

- Graceful shutdown
- These can be easy to deal with

Crash-Stop A process stops running, with no notification

- Have to notice that something's wrong
- Might be difficult to distinguish from network issues

Byzantine A process continues running, but *deviates arbitrarily from its protocol*

- Might continue running "normally"
- Might provide output inconsistent with input
- Might perform other actions
- Often a characteristic of *viruses*

Data Backup

Simple way to protect against faults

How much time can you afford to waste?
$\Rightarrow$ Backup data at least that often

Don't keep backups on or next to system being backed up

Off-site better

Multiple copies, at least some in different off-site locations best

Improves Availability by shortening downtimes

Replication

What if any downtime is too much?

Use **Replication** of a process

Different *instances*

- Each capable of running process (eg, a service)
- Ideally in separate physical locations

Instances must be able to *coordinate state*

Replication for Service Backup

We have one server, the **Primary**

Another server, the **Secondary**, sits mostly idle

Periodically, Primary sends state to Secondary

Primary fails $\Rightarrow$ Secondary takes over

Secondary might become new Primary (ie, switch roles), or might cede control back after Primary recovers

Have to know that the Primary has failed!

- Works great for *Fail-stop* faults
- Can use unresponsiveness for *Crash-stop* faults
  $\Rightarrow$ Requires some *additional assumptions*
- Not much help against *Byzantine* faults

Replication for Load Balancing

If we have a Secondary, why not put it to use?

Use both, and handle more input faster!

One fails, that just slows us down instead of stopping us

Challenge: Maintaining consistent state

If Secondary isn't active, it just gets a copy of Primary's data

But if Secondary used as much as Primary

- They have to send data back and forth
- Updates might conflict (out of scope for this class)

Can't really think of them as Primary and Secondary
$\Rightarrow$ *Peers*

If we can have two peers, why not more?

Replication for Censorship Resistance

Similar to load balancing

Replicate processes in separate *jurisdictions* (towns/counties/states/countries)

Use different *providers*

If one instance is shut down, service is still Available

Hostnames/IP addresses vary widely enough $\Rightarrow$ Hard to filter

Group Exercise 1

Use `get-assignment` to clone the `ftol` repository. Task 1 asks you to create a simple primary/secondary replication scheme that detects failure of the primary and starts the secondary.

Dynamic Replication

How much replication you need might depend on

- User demand (Cyber Monday)
- Network status (Congestion $\Rightarrow$ Get instances closer to users)
- Attacks (DoS)

Running replicas can be expensive

- Do you own the hardware?
- Electrical bills
- Maintenance burden

Use something like a *cloud provider* to stand up instances when you need them

More expensive while running, but no cost for an instance that isn't running

What about Byzantine Faults?

Replication great for *Fail-stop* and *Crash-stop* faults

Can we handle *Byzantine* faults?

Malware, intrusions can *persist* for a while

- Corrupt files
- Exfiltrate data

Can't trust any single instance

Quorum Systems

Quorum A subset of a group comprising the minimum number of members required to perform some action
e.g., 3/4 of an organization might need to be present to hold a vote

Quorum System A Distributed System in which some subset of processes must coordinate to perform a system-wide operation

Consider a distributed data store's operations:

- `Write` k v should reliably store value v in key k
- `Read` k should reliably return the last successful `Write` to k

We can do this by writing and reading using *Quorums*

We might have *different-sized* **Read** and **Write Quorums**

Important part is that they obey certain *intersection properties*

Voting Quorums (Warning: Math)

System with N processes

Quorum size Q (same for reading and writing) determined by:

- Q processes participate in a `Write` operation
- Q processes participate in a subsequent `Read` operation
- The *majority* of processes in the read quorum must *agree* on the last-written value

That means $\lfloor Q/2 \rfloor + 1$ of read quorum must have been in write quorum

$\Rightarrow Q - (\lfloor Q/2 \rfloor + 1)$ *might not* have been
$\Rightarrow N = Q + Q - (\lfloor Q/2 \rfloor + 1) = 3Q/2 - 1$
$\Rightarrow Q = 2(N+1)/3$

With version-numbered values, only 1 member has to overlap

$\Rightarrow N = 2Q - 1 \Rightarrow Q = (N+1)/2$

Efficient Intersection Quorums

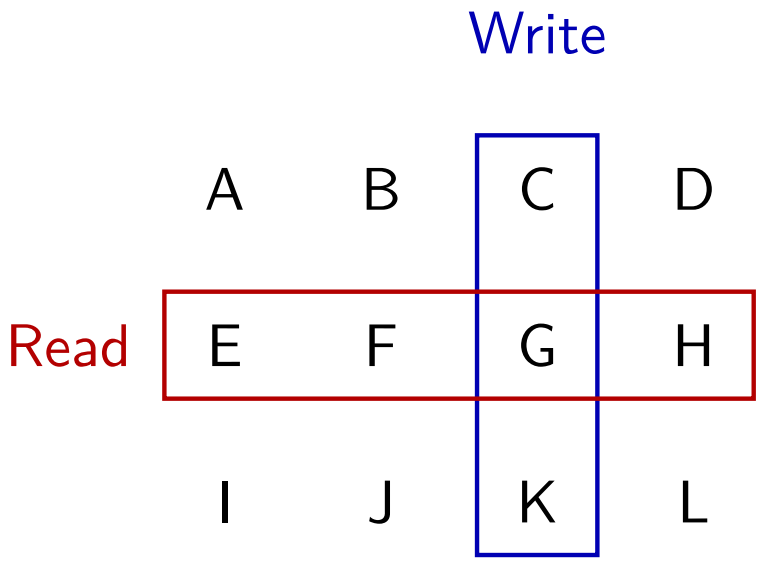
Previous system had *any* subset of size Q for a read or write

We can do better!

Processes A–L

Writes are expensive

Reads are relatively cheap



Handling Different Fault Models

Assume at most T faulty processes

Fail-stop and Crash-stop are easy

$\Rightarrow$ Faulty processes just don't participate, so $N - Q \leq T$

Byzantine is harder, because faulty processes might *lie*

$\Rightarrow$ Quorums must contain more *correct* processes than T

$\Rightarrow |Q_W \cap Q_R| \geq T + 1$

Handling Byzantine Faults with Voting

Just saw that intersection $|Q_W \cap Q_R| \geq T + 1$

$Q - (T + 1)$ might not be in a quorum

$N = Q + |\text{processes not in quorum}| = 2Q - (T + 1)$

Faulty processes might also just not participate
$\Rightarrow Q - (T + 1) = T \Rightarrow Q = 2T + 1$

Put this all together $\Rightarrow N = 2(2T + 1) - (T + 1) = 3T + 1$

Group Exercise 2

Task 2 asks you to investigate some simple properties of quorum systems.