Failures & Fault Tolerance

Failures are common!

- You should find this statement absurd.
  - How many of you have had laptops fail on you?
  - How long between failures?
  - What was the failure?

And yet...

[Journal]: MTBF for server ~ 30 years
Typical year at a datacenter: 1-5% of disks fail. [2009]

~1 PDU will fail each year ~ 500 machines down for 1-6 hours

~20 rack failures
~1000 server failures

Anecdotal "personal observation"

- Booting 15 EC2 instances almost always succeeds

- Booting 100 EC2 instances never succeeds (for me)
  - At least 1-2 instances stuck somewhere

- Reliability of 1 server ≠ Cluster reliability

How to address this problem?

- Design fault tolerant systems
  - But how? Many ways to skin this cat...
- Rep. state machine
- 2 phase commit
- Primary backup replication
- Stateless computation w/ external store

→ Each approach
  → Makes different assumptions
  → Has different performance & efficiency tradeoffs

→ Application fault tolerance
  → Inherits all assumptions & tradeoffs

→ Hard to reason about & test correctness under failures.

→ Changes to applications/services can change assumptions & tradeoffs

Change the assumed failure model

All in all makes application administration HEAVY!
Netflix's thoughts on doing better

**Mantra:**

- **Engineer Services for Failures**
  
  - Services must remain available despite some class of failures

**Netflix o Don't Wait for Things to Fail; Proactively Inject Failures!!**

Why? Easy to undo injected failures. Find fault tolerance bugs sooner

**Chaos Monkey**

**Q:** What to fail?
**A:** Choose at random.

**Benefits:**

- Simple
- Assumes little or no information
- Given enough time, explores many/most failures
Problems: - Inefficient!

Today's paper: Does more information help?

Lineage & Provenance
- What inputs & code paths contributed to an output.
  [Oldish idea from databases]

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Price</th>
<th>Fried</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fabiol</td>
<td>4.00</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wings</td>
<td>20.00</td>
<td>T</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>480</td>
<td>Bao</td>
<td>10.00</td>
<td>F</td>
</tr>
<tr>
<td>700</td>
<td>Idli</td>
<td>12.00</td>
<td>F</td>
</tr>
</tbody>
</table>

Useful to track what data contributes to a result.

Why?
(a) Incremental computation / View maintenance
(b) Reason about data privacy
(c) Debug program: why did we compute this value.
Not restricted to databases/directly
Roughly similar to PFG.

But tracking is expensive.

What does this have to do with fault tolerance?

- Think of services/microservices as sources & sinks of data.

```
let subinfo = subscriber.get_session();
if (subinfo.allows(request)) {
  let d = streamdb.get_stream(request);
  telemetry.record_content(d);
  return make_stream(d, videos.get_content(d));
} else {
  telemetry.record_unauth(subinfo);
  return not_authorized(subinfo);
}
```

Don't forget, we want to answer two questions:
During testing, aim to avoid failures:

- What service failures are more likely to impact availability? Prioritize testing for those.

```javascript
let subinfo = subscriber.get_session();
if (subinfo.allows(request)) {
    let d = streamdb.get_stream(request);
    telemetry.record.content(d);
    return make_stream(d, video.get.content(d));
} else {
    telemetry.record.warn(subinfo);
    return not.authorized(subinfo);
}
```

Telemetry failures might slow down response, but unlikely to impact quality.

Think back to DQBARGE, etc.

- When to emulate failures?
So might give us enough information to limit how many tests to run.

1) No information + full coverage
2) Fail all services at all possible program points
3) Lineage
   4) On average, fewer C?

Note: no better in worse case.

Aside: How to target failures?
Elephant in the Room. Analysis above required code analysis

- Static analysis
  - Very powerful
  - Fun to think about
  - Hard to apply/use in the wild
    - Need to make assumptions about
      - language, execution model, ...
    - Esp. hard in polyglot

- Can we use lineage style fault injection without a system that records lineage.

Also kind of? Use traces

First things first: **Traces Do Not Capture Lineage**

- No information about how results are used

```plaintext
let subinfo = subscriber.get_session();
if (subinfo.allows(request)) {
  telemetry.record_get();
```

let \( d = \text{streamdb} \cdot \text{get-stream}(\text{request}) \);

\[ \text{telemetry} \cdot \text{record-content}(d) ; \]

\[ \text{return make-stream}(d, \text{videos} \cdot \text{get-content}(d)) ; \]

else

\[ \text{telemetry} \cdot \text{record-urauth}(\text{subinfo}) ; \]

\[ \text{return not-authorized}(\text{subinfo}) ; \]

\[ \rightarrow \text{limited information about fallbacks} \]

let \( s_i = \text{subscriber} \cdot \text{get-session}(\) ;

\[ \text{if } (s_i \cdot \text{is-error}()) \rightarrow \]

\[ s_i = \text{subpolicy} \cdot \text{default-sub}(\text{request}) ; \]

\[ \ldots \]

\[ \rightarrow \text{Indistinguishable} \]

\[ \ldots \]

let \( s_i = \text{subscriber} \cdot \text{get-session}(\) ;

\[ \text{if } (\text{date} \cdot \text{today}()) = \text{date}(04,01,2023) \rightarrow \]

\[ \text{let } o\text{s}_i = \text{subpolicy} \cdot \text{default-sub}(\text{request}) ; \]

\[ \ldots \]

\[ \text{let } u\text{es}_i = \text{pick-stronger}(s_i, o\text{s}_i) ; \]

\[ \ldots \]

\[ \text{not enough information to} \]
But still, have enough information to

- Identify dependencies for a particular execution

  ➔ We already track RPC calls & responses

  ✔ When calls are made.

  ➔ Can add a little more to find fallbacks

  ➔ See how trace changes when services are failed

  ➔ Again, aside on how failures are injected matters here.

What is missing?

- Cannot distinguish benign vs. non-benign failures
- Cannot always identify failure dependencies

```javascript
let si = subscriber.get_session(s);
if (si.is_error()) {
    si = subpolicy.default_sub(request);
}```
if (!si.allowed(request))
    si = subpolicy.maybe_upgrade(request);

...

- Generally: some opportunities to improve
efficiency but not as good as real
lineage?

SLFI: How to actually use this information

- In testing rather than production
  → Why?

- Broadly, this is a common approach for testing any
  scheduling like algorithm.

- Many ways to think about this: I like trees & forests
- Core questions: Order of exploration

When to prune

\[ \text{Want to at least do it when executions are equivalent} \]

But how to check?
When to stop?

Explore individual services on all of them?