Stability Patterns
…and Antipatterns

Michael Nygard
mtnygard@thinkrelevance.com
@mtnygard
Stability Antipatterns
Integration Points

Integrations are the #1 risk to stability.

Every out of process call can and will eventually kill your system.

Yes, even database calls.
Example: Wicked database hang
"In Spec" vs. "Out of Spec"

Example: Request-Reply using XML over HTTP

**"In Spec" failures**
- TCP connection refused
- HTTP response code 500
- Error message in XML response

**"Out of Spec" failures**
- TCP connection accepted, but no data sent
- TCP window full, never cleared
- Server replies with “EHLO”
- Server sends link farm HTML
- Server streams Weird Al mp3s

**Well-Behaved Errors**

**Wicked Errors**
Necessary evil.

Peel back abstractions.

Large systems fail faster than small ones.

Useful patterns: Circuit Breaker, Use Timeouts, Use Decoupling Middleware, Handshaking, Test Harness
Chain Reaction

Failure moves horizontally across tiers
Common in search engines and app servers
Remember This

One server down jeopardizes the rest.
Hunt for Resource Leaks.

Useful pattern: Bulkheads
Cascading Failure

Failure moves vertically across tiers

Common in enterprise services & SOA
“Damage Containment”
Stop cracks from jumping the gap
Scrutinize resource pools

Useful patterns: Use Timeouts, Circuit Breaker
Too many, too clicky
Some malicious users
Buyers
Front-page viewers
Screen scrapers
Handle Traffic Surges Gracefully

- Degrade features automatically
- Shed load.
- Don’t keep sessions for bots.
- Reduce per-user burden:
  - IDs, not object graphs.
  - Query parameters, not result sets.
Blocked Threads

All request threads blocked = “crash”

Impossible to test away

Learn to use java.util.concurrent or System.Threading.

(Ruby & PHP coders, just avoid threads completely.)
Pernicious and Cumulative

Hung request handlers = less capacity.
Hung request handler = frustrated user/caller

Each remaining thread serves 1/(N-1) extra requests
Example: Blocking calls

In a request-processing method

```java
String key = (String) request.getParameter(PARAM_ITEM_SKU);
Availability avl = globalObjectCache.get(key);
```

In GlobalObjectCache.get(String id), a synchronized method:

```java
Object obj = items.get(id);
if (obj == null) {
    obj = strategy.create(id);
}
...
```

In the strategy:

```java
public Object create(Object key) throws Exception {
    return omsClient.getAvailability(key);
}
```
Use proven constructs.
Don’t wait forever.
Scrutinize resource pools.
Beware the code you cannot see.

Useful patterns: Use Timeouts, Circuit Breaker
Attacks of Self-Denial

BestBuy: XBox 360 Preorder

Amazon: XBox 360 Discount

Victoria’s Secret: Online Fashion Show

Anything on FatWallet.com
Defenses

- Avoid deep links
- Static landing pages
- CDN diverts or throttles users
- Shared-nothing architecture
- Session only on 2nd click
- Deal pool
Remember This

Open lines of communication.

Support your marketers.
Unbalanced Capacities

- **Online Store**
  - 20 Hosts
  - 75 Instances
  - 3,000 Threads

- **Order Management**
  - 6 Hosts
  - 6 Instances
  - 450 Threads

- **Scheduling**
  - 1 Host
  - 1 Instance
  - 25 Threads

- SiteScope NYC
- Customers
- SiteScope San Francisco
<table>
<thead>
<tr>
<th></th>
<th>Dev</th>
<th>QA</th>
<th>Prod</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Online Store</strong></td>
<td>1/1/1</td>
<td>2/2/2</td>
<td>20/300/6</td>
</tr>
<tr>
<td><strong>Order Management</strong></td>
<td>1/1/1</td>
<td>2/2/2</td>
<td>4/6/2</td>
</tr>
<tr>
<td><strong>Scheduling</strong></td>
<td>1/1/1</td>
<td>2/2/2</td>
<td>4/2</td>
</tr>
</tbody>
</table>
Unbalanced Capacities

- Scaling effect between systems
- Sensitive to traffic & behavior patterns
- Stress both sides of the interface in QA
- Simulate back end failures during testing
What SLA can Frammitz really guarantee?
Remember This

No empty promises.
Monitor your dependencies.
Decouple from your dependencies.
Measure availability by feature, not by server.
Beware infrastructure services: DNS, SMTP, LDAP.
Development and testing is done with small data sets
Test databases get reloaded frequently
Queries often bonk badly with production data volume
Unbounded Result Sets: Databases

SQL queries have no inherent limits
ORM tools are bad about this
Appears as slow performance degradation
Unbounded Result Sets: SOA

Chatty remote protocols, N+1 query problem
Hurts caller and provider
Caller is naive, trusts server not to hurt it.
Remember This

Test with realistic data volumes
Don’t trust data producers.
Put limits in your APIs.
Stability Patterns
Ever seen a remote call wrapped with a retry loop?

```java
int remainingAttempts = MAX_RETRIES;
while(--remainingAttempts >= 0) {
  try {
    doSomethingDangerous();
    return true;
  } catch(RemoteCallFailedException e) {
    log(e);
  }
}
return false;
```
Faults Cluster

Fast retries good for dropped packets (but let TCP do that)

Most other faults require minutes to hours to correct

Immediate retries very likely to fail again
Problems with the remote host, application or the network will probably persist for an long time... minutes or hours
Bad for Users and Systems

Users:
- Wait longer to get an error response.
- What happens after final retry?

Systems:
- Ties up threads, reducing overall capacity.
- Multiplies load on server, at the worst times.
- Induces a Cascading Failure
Stop Banging Your Head

Wrap a “dangerous” call
Count failures
After too many failures, stop passing calls
After a “cooling off” period, try the next call
If it fails, wait some more before calling again

Closed
- on call / pass through
- call succeeds / reset count
- call fails / count failure
- threshold reached / trip breaker

Open
- on call / fail
- on timeout / attempt reset

Half-Open
- on call/pass through
- call succeeds/reset
- call fails/trip breaker

Reset
- pop
- attempt reset
- pop

Saturday, June 23, 12
Considerations

Sever malfunctioning features

Degrade gracefully on caller

Critical work must be queued for later
Stop doing it if it hurts.

Expose, monitor, track, and report state changes

Good against: Cascading Failures, Slow Responses

Works with: Use Timeouts
Bulkheads

- Partition the system
- Allow partial failure without losing service
- Applies at different granularity levels
Common Mode Dependency

Foo and Bar are coupled via Baz
 Foo and Bar have dedicated resources from Baz.
Remember This

Save part of the ship
Decide if less efficient use of resources is OK
Pick a useful granularity
Very important with shared-service models
Monitor each partition’s performance to SLA
Real-world failures are hard to create in QA
Integration tests work for “in-spec” errors, but not “out-of-spec” errors.
"In Spec" vs. "Out of Spec"

Example: Request-Reply using XML over HTTP

"In Spec" failures
- TCP connection refused
- HTTP response code 500
- Error message in XML response

"Out of Spec" failures
- TCP connection accepted, but no data sent
- TCP window full, never cleared
- Server replies with "EHLO"
- Server sends link farm HTML
- Server streams Weird Al mp3s

Well-Behaved Errors

Wicked Errors
“Out-of-spec” errors happen all the time in the real world.

They never happen during testing...

unless you force them to.
Killer Test Harness

Daemon listening on network
Substitutes for the remote end of an interface
Can run locally (dev) or remotely (dev or QA)
Is totally evil
# Just a Few Evil Ideas

<table>
<thead>
<tr>
<th>Port</th>
<th>Nastiness</th>
</tr>
</thead>
<tbody>
<tr>
<td>19720</td>
<td>Allows connections requests into the queue, but never accepts them.</td>
</tr>
<tr>
<td>19721</td>
<td>Refuses all connections</td>
</tr>
<tr>
<td>19722</td>
<td>Reads requests at 1 byte / second</td>
</tr>
<tr>
<td>19723</td>
<td>Reads HTTP requests, sends back random binary</td>
</tr>
<tr>
<td>19724</td>
<td>Accepts requests, sends responses at 1 byte / sec.</td>
</tr>
<tr>
<td>19725</td>
<td>Accepts requests, sends back the entire OS kernel image.</td>
</tr>
<tr>
<td>19726</td>
<td>Send endless stream of data from /dev/random</td>
</tr>
</tbody>
</table>

Now *those* are some out-of-spec errors.
Remember This

- Force out-of-spec failures
- Stress the caller
- Build reusable harnesses for L1-L6 errors
- Supplement, don’t replace, other testing methods
Michael Nygard
mtnygard@thinkrelevance.com
@mtnygard