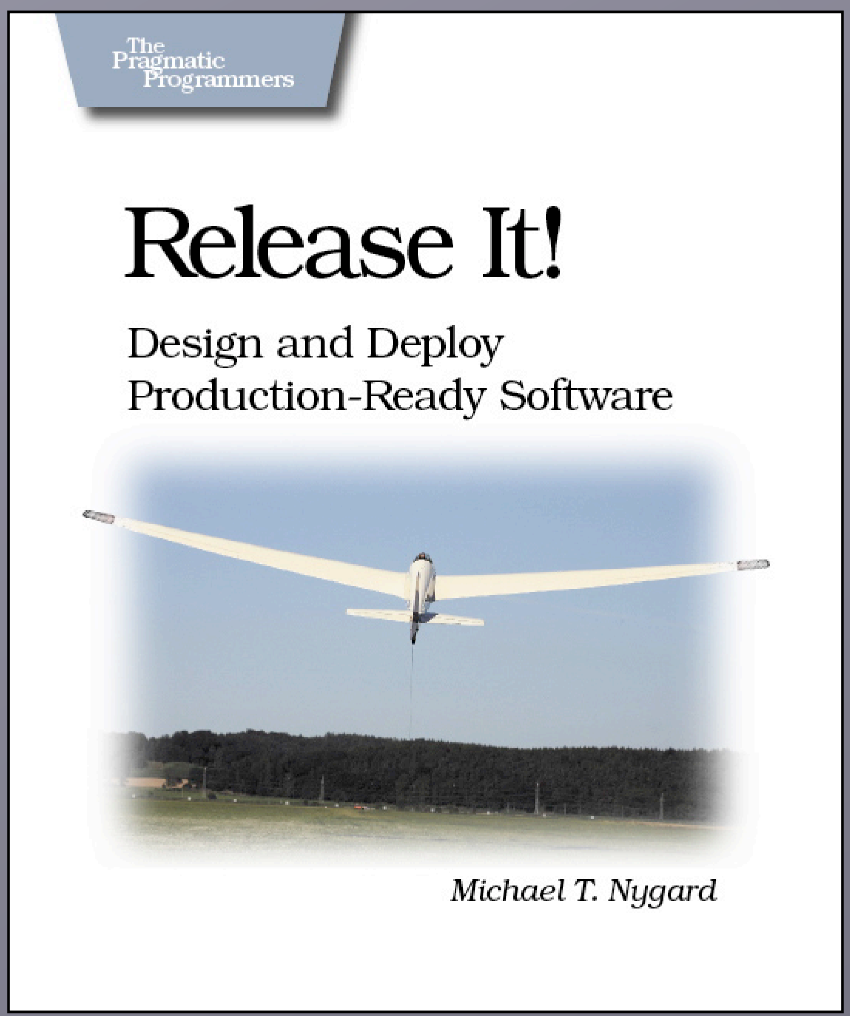
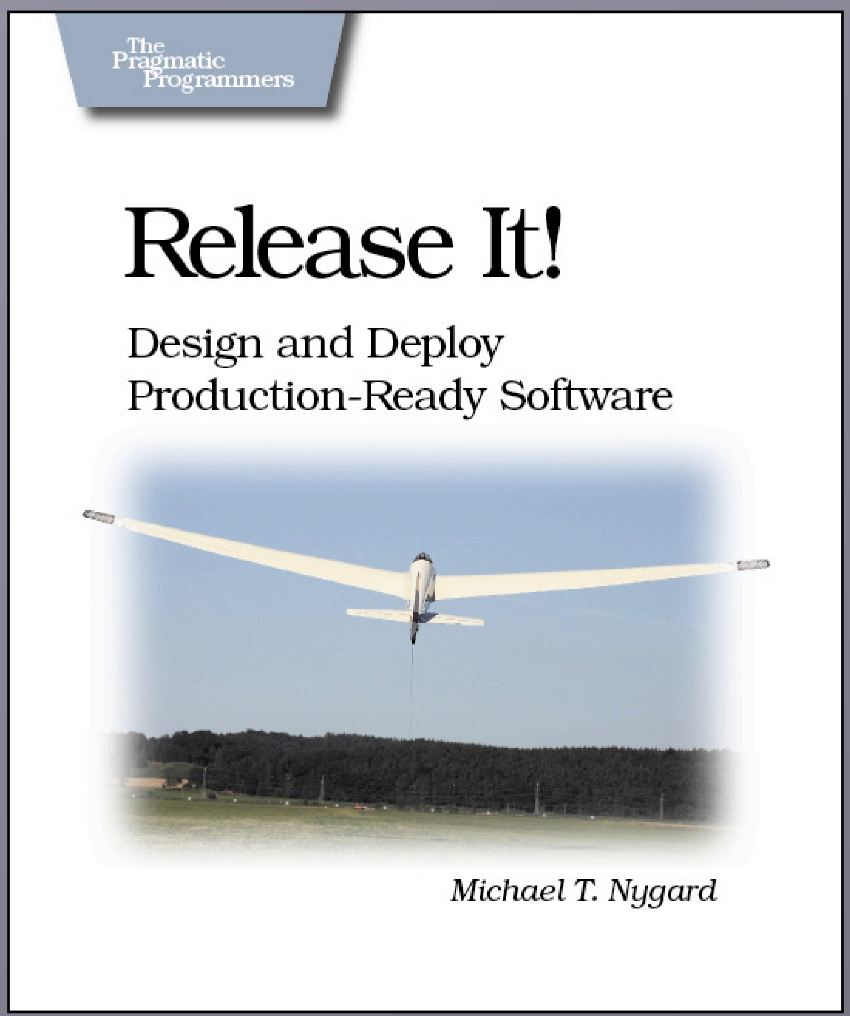
Failure Comes in Flavors
Part II: Patterns

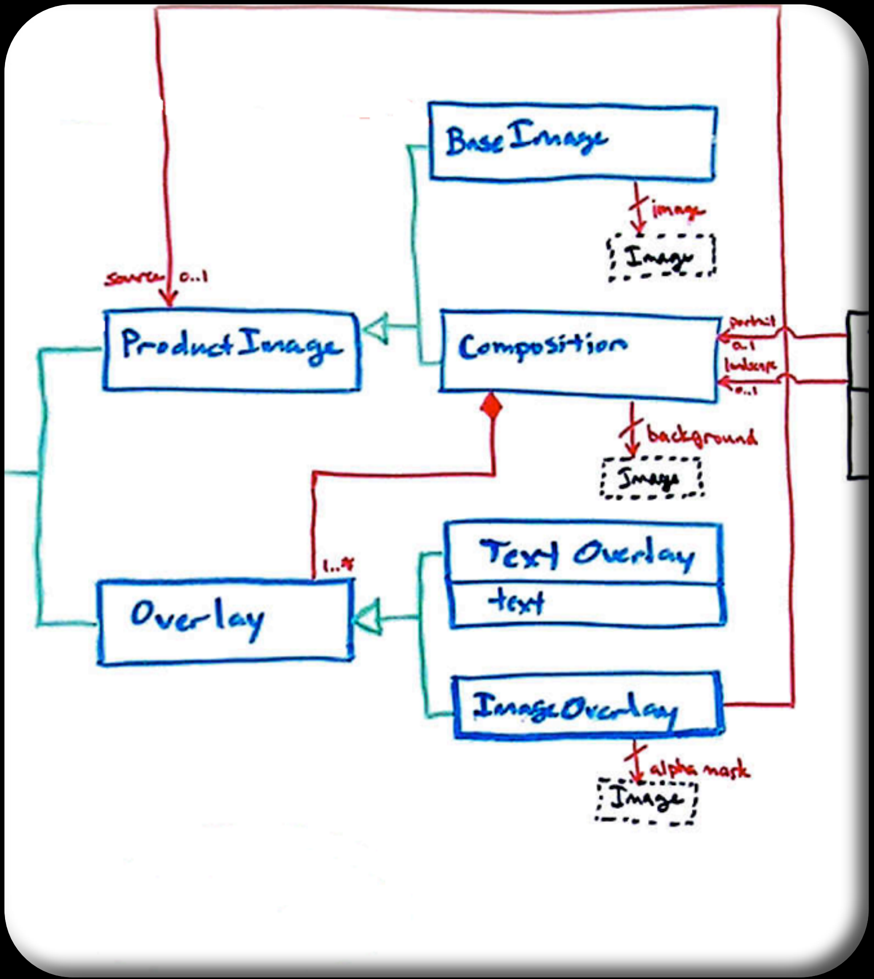


Michael Nygard
mtnygard@gmail.com
www.michaelnygard.com

Failure Comes in Flavors Part II: Patterns



Michael Nygard
mtnygard@gmail.com
www.michaelnygard.com



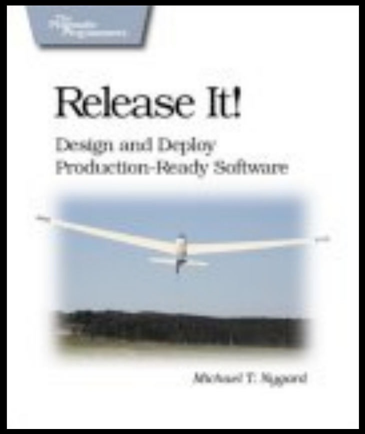
My Rap Sheet

C
C++
Object Pascal
Objective-C
Perl
Java
Smalltalk
Ruby

1989 - 2008: Application Developer
Time served: 18 years

1995: Web Development
Time served: 13 years

2003: IT Operations
Time served: 5 Years



High-Consequence Environments

Users in the thousands and tens of thousands

24 hours a day, 365 days a year

Millions in hardware and software

Millions (or billions) in revenue

Highly interdependent systems

Actively malicious environment

What downtime means for a few of my clients

Manufacturer:
Over 500,000 products and media

Financial services broker:
Average transaction $10,000,000

Top 10 online retailer:
$1,000,000 per hour of downtime

Airline:
Downtime grounds planes and strands travelers

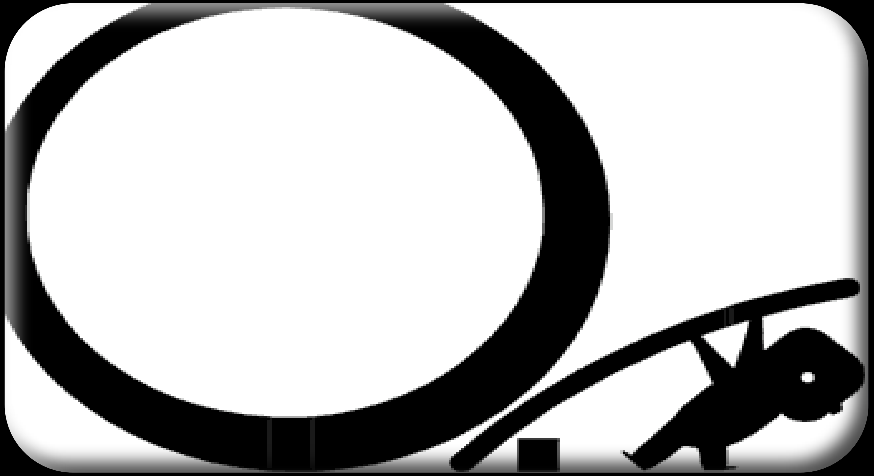
Points of Leverage

Small decisions at every level can have a huge impact:

- Architecture
- Design
- Implementation
- Build & Deployment
- Administration

Good News

Some large improvements are available with little to no added development cost.



Bad News

Leverage points come early. The cost of choosing poorly comes much, much later.

Assumptions

Users care about the things they do (features), not the software or hardware you run.

Severability: Limit functionality instead of crashing completely.

Resilience: Recover from transient effects automatically.

Recoverability: Allow component-level restarts instead of rebooting the world.

Tolerance: Absorb shocks, but do not transmit them.

Together, these qualities produce stability–the consistent, long-term availability of features.

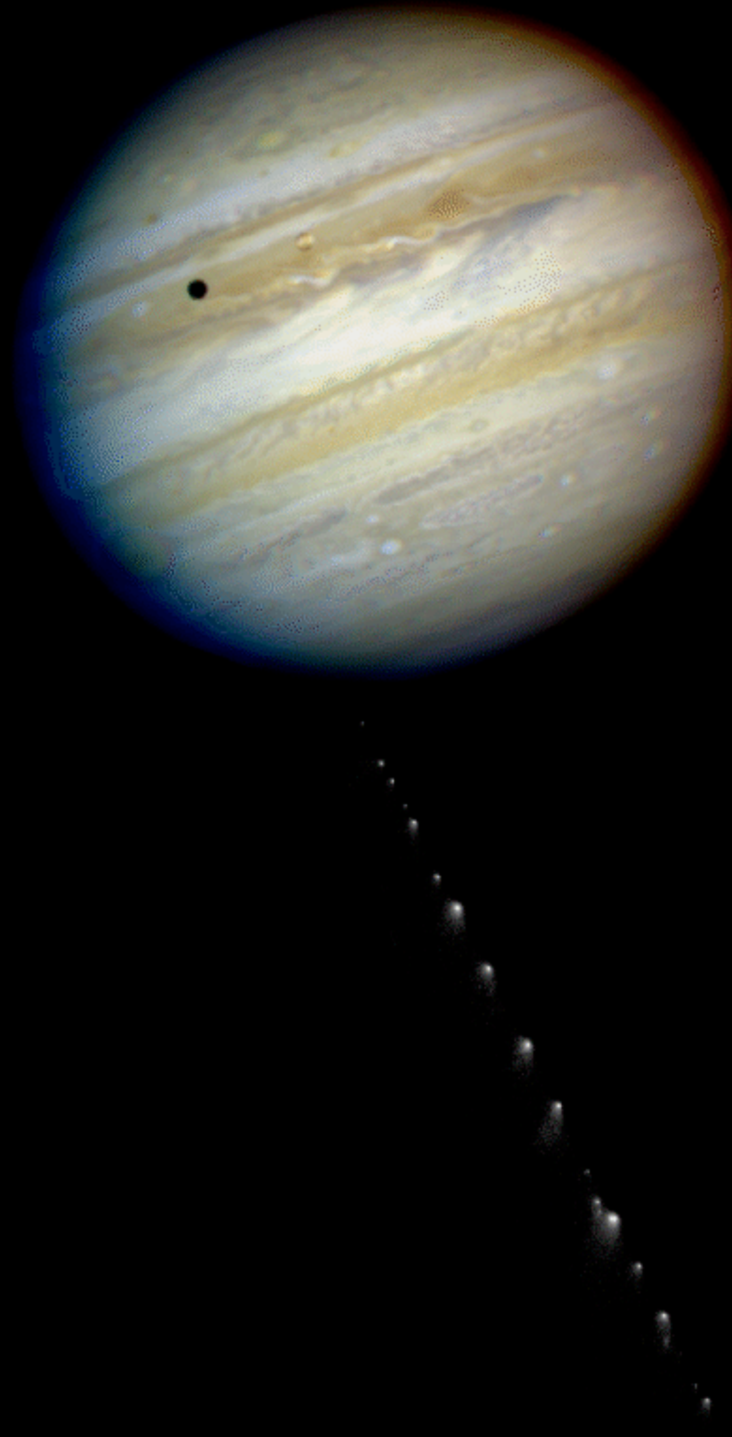
Stability Under Stress

Stability under stress is resilience to transient problems

- User load
- Back-end outages
- Network slowdowns
- Other “exogenous impulses”

There is no such thing as perfect stability; you are buying time

How long is your shortest fuse?

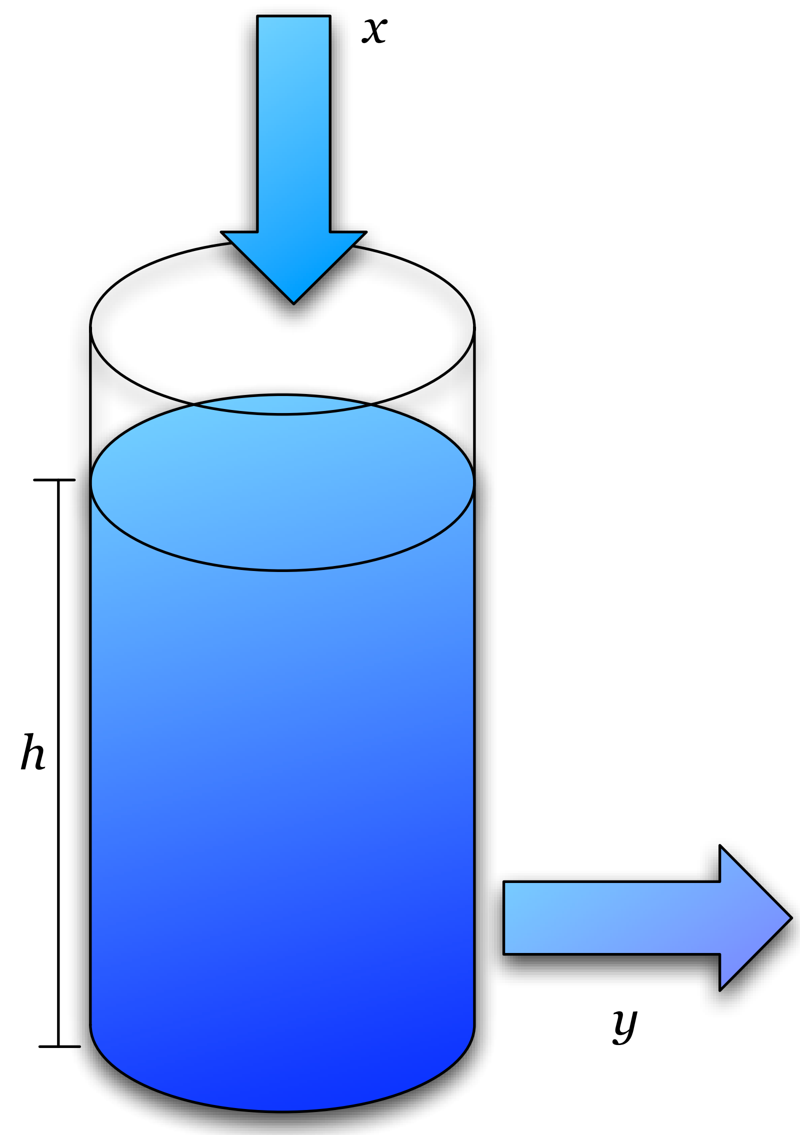


Stability Over Time

How long can a process or server run before it needs to be restarted?

Is data produced and purged at the same rate?

Usually not tested in development or QA. Too many rapid restarts.



The Sweetness of Success: Stability Patterns

- Use Timeouts
- Circuit Breaker
- Bulkheads
- Steady State
- Fail Fast
- Test Harness
- Decoupling Middleware

Use Timeouts

Don’t hold your breath.

- In any server-based application, request handling threads are your most precious resource
  - When all are busy, you can’t take new requests
  - When they stay busy, your server is down
  - Busy time determines overall capacity
- Protect request handling threads at all costs

Hung Threads

- Each hung thread reduces capacity
- Hung threads provoke users to resubmit work
- Common sources of hangs:
  - Remote calls
  - Resource pool checkouts
- Don’t wait forever... use a timeout

Considerations

- Calling code must be prepared for timeouts.
  - Better error handling is a good thing anyway.
- Beware third-party libraries and vendor APIs.
  - Examples:
    - Veritas's K2 client library does its own connection pooling, without timeouts.
    - Java's standard HTTP user agent does not use read or write timeouts.
- Java programmers:
  - Always use Socket.setSoTimeout(int timeout)

Remember This

Apply to Integration Points, Blocked Threads, and Slow Responses

Apply to recover from unexpected failures.

Consider delayed retries. (See Circuit Breaker.)

Circuit Breaker

Defend yourself.

Have you ever seen a remote call wrapped with a retry loop?

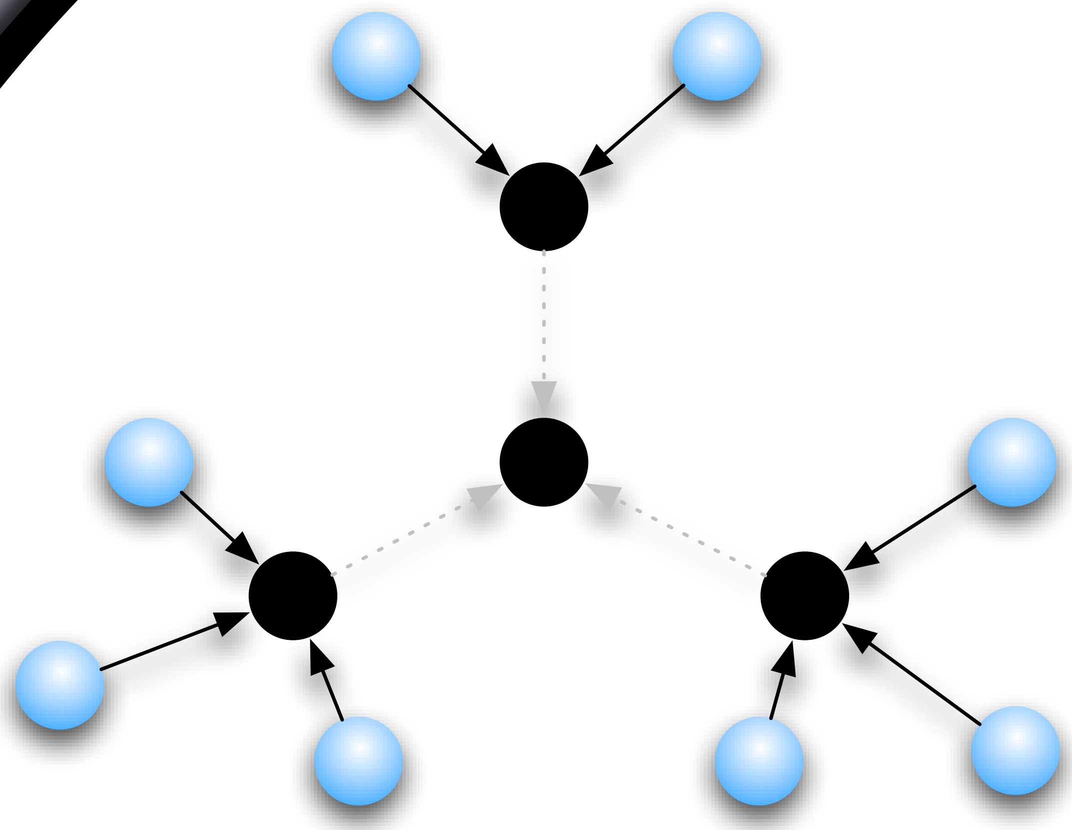
```
int remainingAttempts = MAX_RETRIES;

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

Why?

Faults Cluster

Problems with the remote host, application or the intervening network are likely to persist for an extended period of time... minutes or maybe even hours



Faults Cluster

Fast retries only help for dropped packets, and TCP already handles that for you.

Most of the time, the retry loop will come around again while the fault still persists.

Thus, immediate retries are overwhelmingly likely to also fail.

Retries Hurt Users and Systems

Users:

Retries make the user wait even longer to get an error response.

After the final retry, what happens to the users' work?

The target service may be non-critical, so why damage critical features for it?

Systems:

Ties up caller's resources, reducing overall capacity.

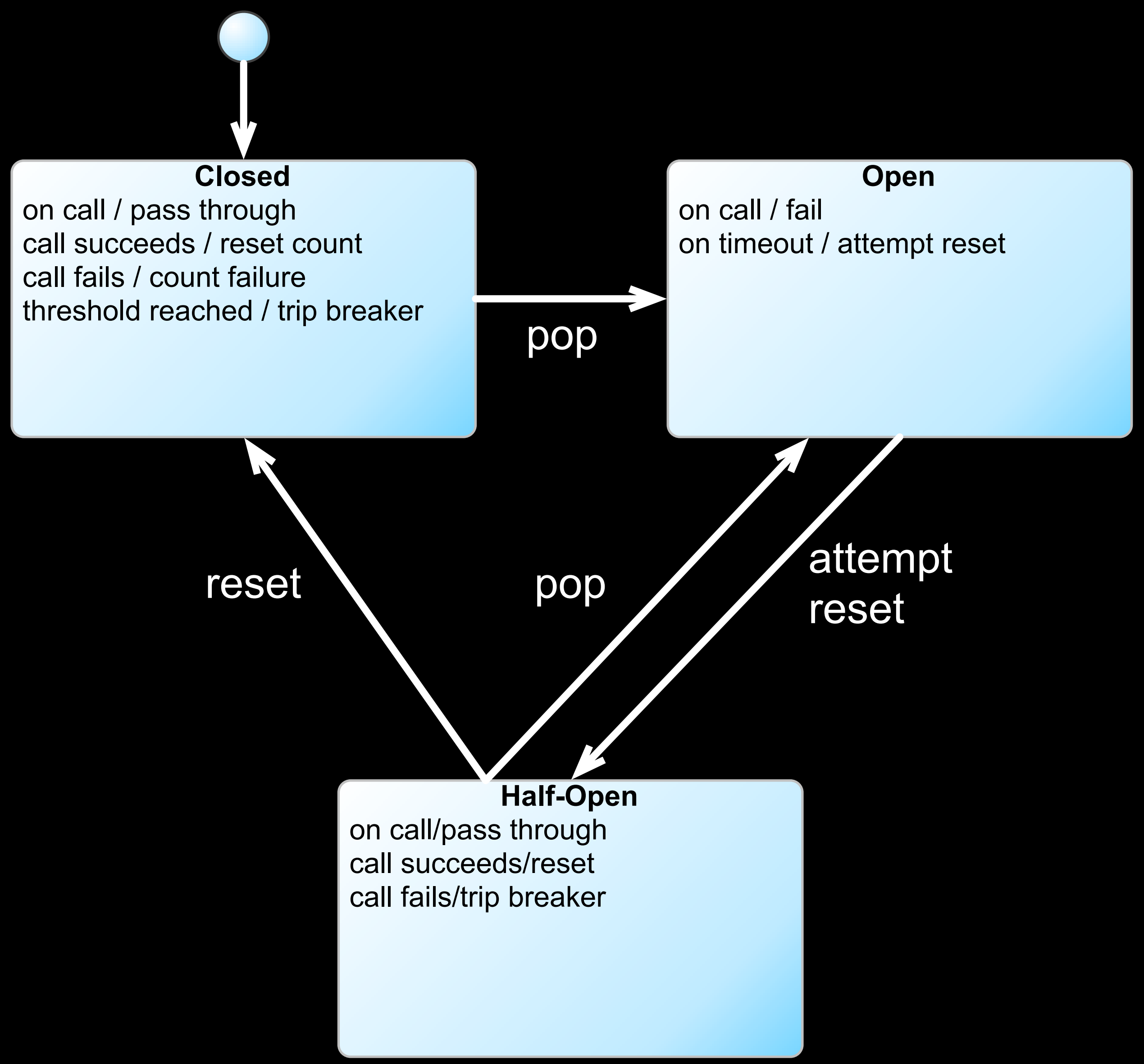
If target service is busy, retries increase its load at the worst time.

Every single request will go through the same retry loop, letting a back-end problem cause a front-end brownout.

Stop Banging Your Head

Circuit Breaker:

- Wraps a “dangerous” call
- Counts failures
- After too many failures, stop passing calls through
- After a “cooling off” period, try the next call
- If it fails, wait for another cooling off time before calling again



Considerations

Circuit Breaker exists to sever malfunctioning features.

Calling code must be prepared to degrade gracefully.

Critical work must be queued for later processing

- Might motivate changes in business rules. Conversation needed!

Threading is very tricky... get it right once, then reuse the component.

- Avoid serializing all calls through the CB
- Deal with state transitions during a long call

Can be used locally, too. Around connection pool checkouts, for example.

Remember This

Don’t do it if it hurts.

Use Circuit Breakers together with Timeouts

Expose, track, and report state changes

Circuit Breakers prevent Cascading Failures

They protect against Slow Responses

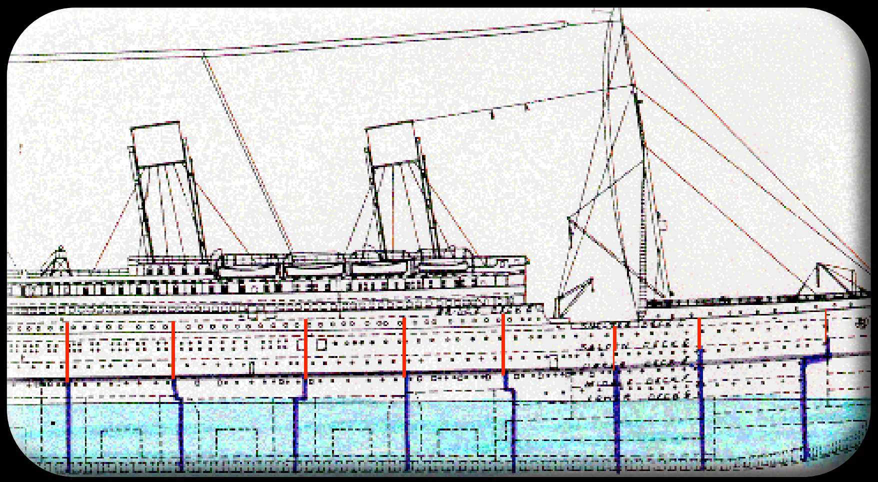
Bulkheads

Save part of the ship, at least.

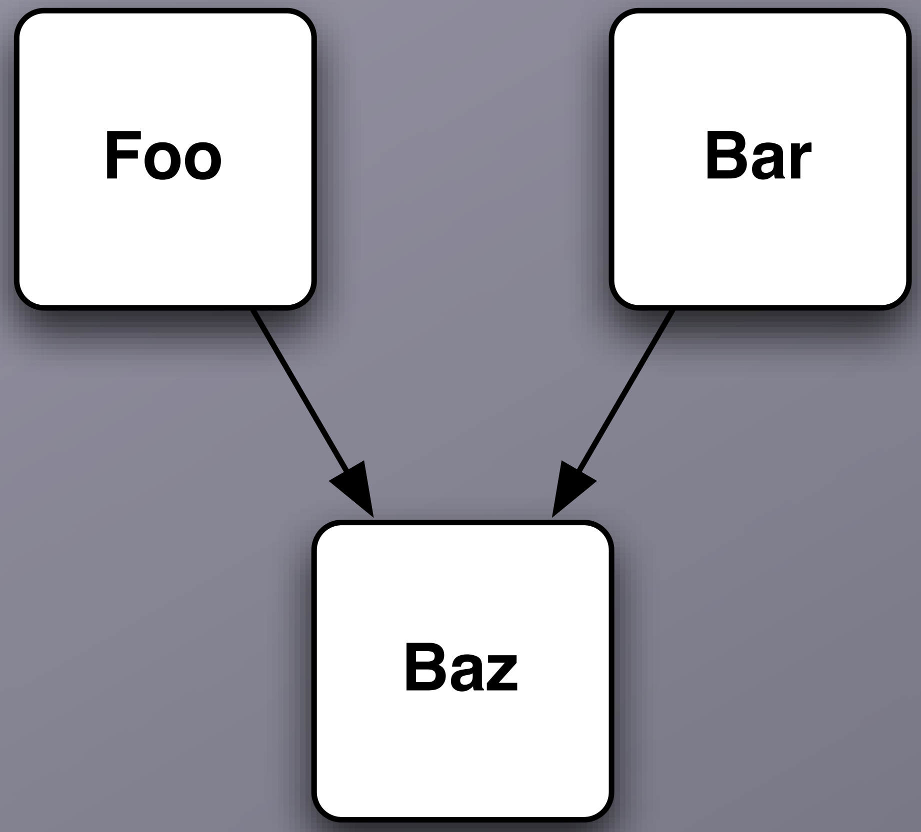
- Increase resilience by partitioning (compartmentalizing) the system
  - One part can go dark without losing service entirely
- Apply at several levels
  - Thread pools within a process
  - CPUs in a server (CPU binding)
  - Server pools for priority clients

Wikipedia says:

Compartmentalization is the general technique of separating two or more parts of a system in order to prevent malfunctions from spreading between or among them.



Example: Service-Oriented Architecture



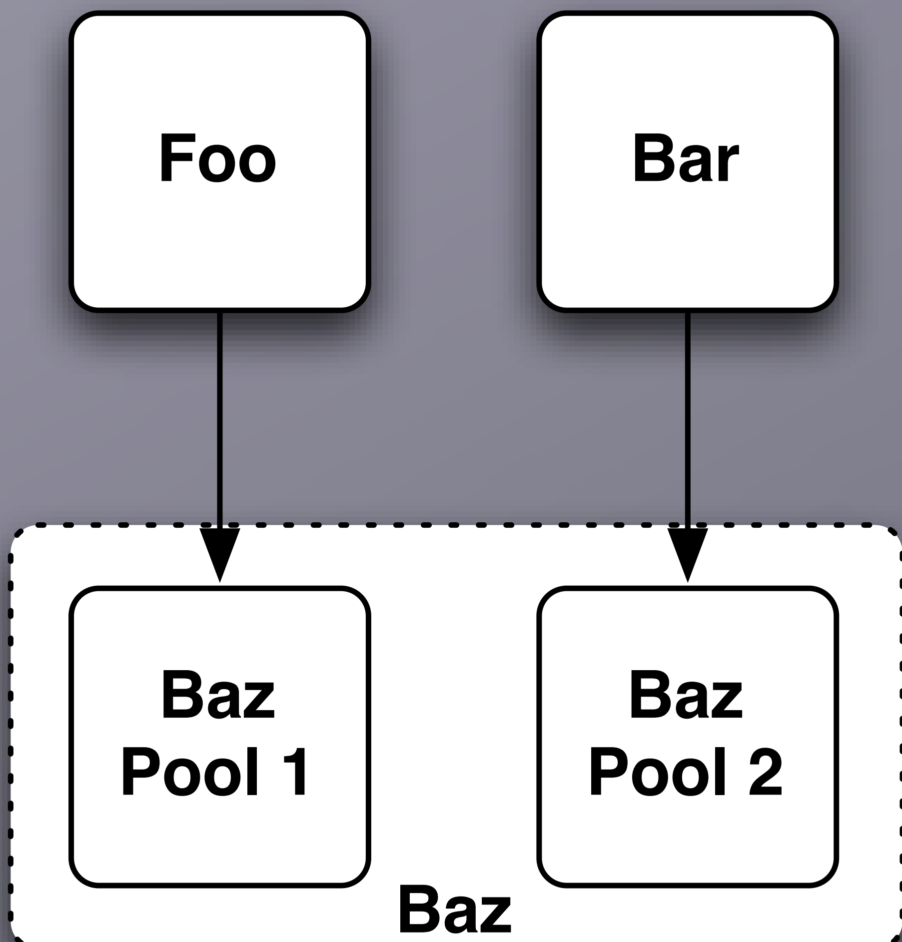
Foo and Bar are coupled by their shared use of Baz

An single outage in Baz will take eliminate service to both Foo and Bar.

(Cascading Failure)

Surging demand–or bad code–in Foo can deny service to Bar.

SOA with Bulkheads



Foo and Bar each have dedicated resources from Baz.

Each pool can be rebooted, or upgraded, independently.

Surging demand–or bad code–in Foo only harms Foo.

Considerations

- Partitioning is both an engineering and an economic decision. It depends on SLAs the service requires and the value of individual consumers.
  - Consider creating a single “non-priority” partition.
  - Governance needed to define priorities across organizational boundaries.
- Capacity tradeoff: less resource sharing across pools.
  - Exception: virtualized environments allow partitioning *and* capacity balancing.

Remember This

Save part of the ship

Decide whether to accept less efficient use of resources

Pick a useful granularity

Very important with shared-service models

Monitor each partitions performance to SLA

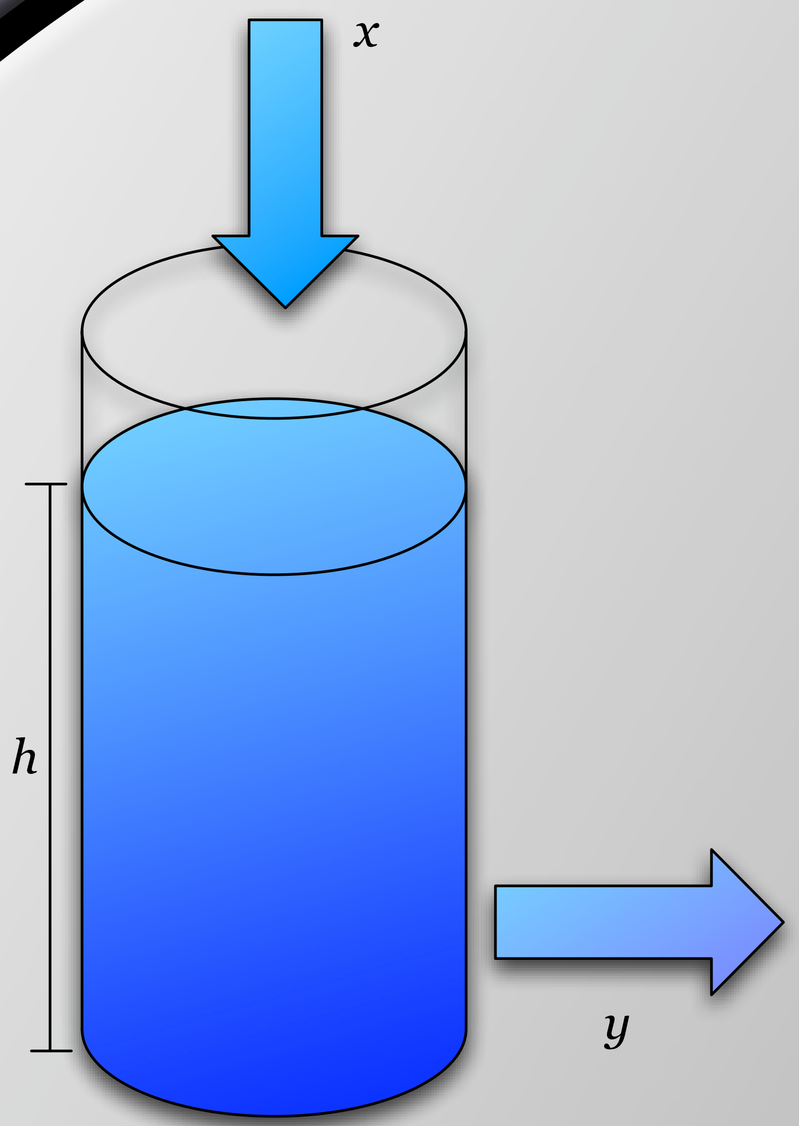
Steady State

Run indefinitely without fiddling.

- Run without crank-turning and hand-holding
- Human error is a leading cause of downtime
  - Therefore, minimize opportunities for error
  - Avoid the “ohnosecond”: eschew fiddling
- If regular intervention is needed, then missing the schedule will cause downtime
  - Therefore, avoid the need for intervention

Routinely Recycle Resources

- All computing resources are finite
- For every mechanism that accumulates resources, there must be some mechanism to reclaim those resources
  - In-memory caching
  - Database storage
  - Log files



Three Common Violations of Steady State

Runaway Caching

Meant to speed up response time

When memory low, can cause more GC

∴ Limit cache size, make "elastic"

Database Sludge

Rising I/O rates

Increasing latency

DBA action ⇒

application errors

Gaps in collections

Unresolved references

∴ Build purging into app

Log File Filling

Most common ticket in Ops

Best case: lose logs

Worst case: errors

∴ Compress, rotate, purge

∴ Limit by size, not time

In crunch mode, it’s hard to make time for housekeeping functions.

Features always take priority over data purging.

This is a false trade: one-time development cost for ongoing operational costs.

Remember This

Avoid fiddling

Purge data with application logic

Limit caching

Roll the logs

Fail Fast

Don't make me wait to receive an error.

Imagine waiting all the way through the line at the Department of Motor Vehicles, just to be sent back to fill out a different form.

Don't burn cycles, occupy threads and keep callers waiting, just to slap them in the face.

Predicting Failure

Several ways to determine if a request will fail, before actually processing it:

- Good old parameter-checking
- Acquire critical resources early
- Check on internal state:
  - Circuit Breakers
  - Connection Pools
  - Average latency vs. committed SLAs

Being a Good Citizen by Failing Fast

In a multi-tier application or SOA, Fail Fast avoids common antipatterns:

- Slow Responses
- Blocked Threads
- Cascading Failure

Preserve capacity when parts of system have already failed.

Remember This

Be a good citizen.

Avoid slow responses; fail fast

Reserve resources

Verify integration points early

Validate input; fail fast if not possible to process request

Test Harness

Violate every protocol in every way possible.

Many failure modes are hard to create in unit or functional tests

Integration tests can verify response to “in-spec” behavior, 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

Well-Behaved Errors

“Out of Spec” failures

- TCP connection accepted, but no data sent
- TCP window full, never cleared
- Server never ACKs TCP, causing very long delays as client retransmits
- Connection made, server replies with SMTP hello string
- Server sends HTML “link-farm” page
- Server sends one byte per second
- Server sends Weird Al catalog in MP3

Wicked Errors

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

They never happen during testing...

unless you force them to.

Provoking Failure Modes

The caller can always feed bad parameters to the service and verify expected errors.

Switches and test modes in the integration test environments can force other errors, at the cost of test modes in the code base.

But what about really weird, “out of specification” errors?

Killer Test Harness

A killer test harness:

- Runs in its own process
- 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

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

Now those are some out-of-spec errors.

Remember This

Produce out-of-spec failures to ensure robustness of the caller

Stress the caller

Leverage shared harnesses across interfaces and projects, for common network-level errors

Supplement, don’t replace, other testing methods

Decoupling Middleware

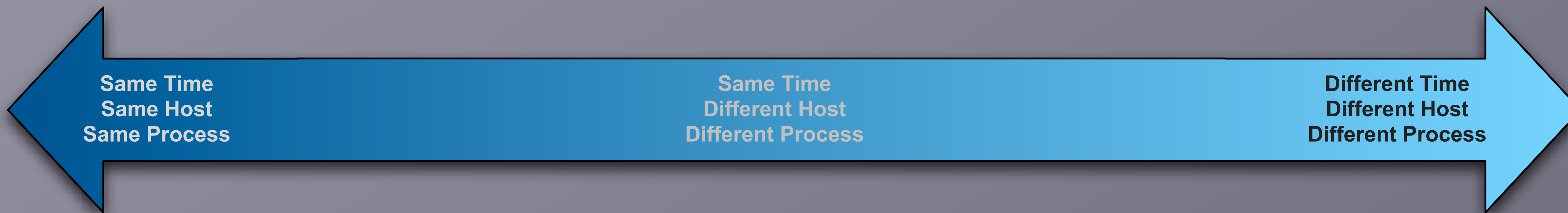
Fire and forget.

Synchronous coupling causes stability problems.

Synchronous RPC is inherently risky.

- Ties up request-processing threads.
- May not ever come back.
- Trusts the remote system!

Spectrum of Coupling



In-Process Method Calls
C Functions
Java Calls
Dynamic Libs

Interprocess Communication
Shared Memory
Pipes
Semaphores
Windows Events

Remote Procedure Calls
DCE RPC
DCOM
RMI
XML-RPC
HTTP

Message-Oriented Middleware
MQ
Pub-Sub
SMTP
SMS

Tuple Spaces
JavaSpaces
TSpaces
GigaSpaces

Request-reply: logical simplicity, operational complexity

Message passing: logical complexity, operational simplicity

Tuple Spaces: logical complexity, operational complexity

Consideration

Changing middleware usually implies a rewrite.

Changing from synchronous to asynchronous semantics implies business rule discussions.

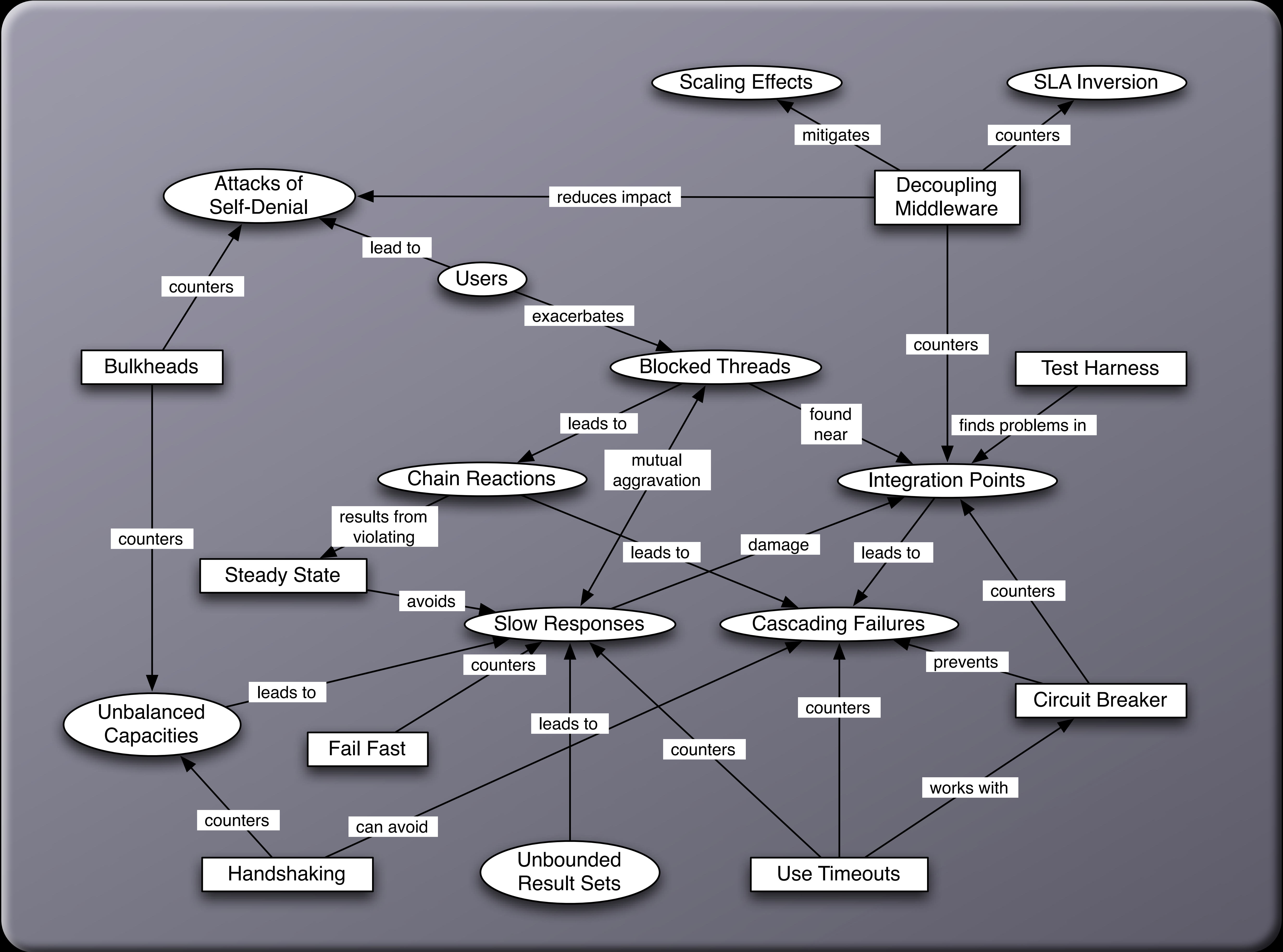
Middleware decisions are often handed down from the ivory tower.

Remember This

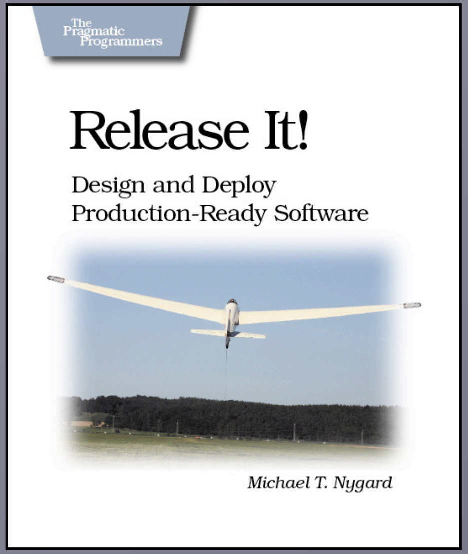
Decide at the last *responsible* moment.

Avoid many failure modes at once by total decoupling.

Learn many architecture styles, choose among them as appropriate.


Scaling Effects
SLA Inversion
mitigates
counters
Attacks of Self-Denial
reduces impact
Decoupling Middleware
lead to
Users
counters
exacerbates
counters
Bulkheads
Blocked Threads
Test Harness
leads to
found near
finds problems in
mutual aggravation
Chain Reactions
Integration Points
results from violating
counters
leads to
damage
leads to
Steady State
counters
avoids
Slow Responses
Cascading Failures
counters
prevents
leads to
counters
Unbalanced Capacities
leads to
Circuit Breaker
Fail Fast
counters
counters
works with
counters
can avoid
Handshaking
Unbounded Result Sets
Use Timeouts

Thank You



Michael Nygard
mtnygard@gmail.com
www.michaelnygard.com