# Failure Comes in Flavors
## Stability Antipatterns

Michael Nygard
michael@michaelnygard.com

## High-Consequence Environments

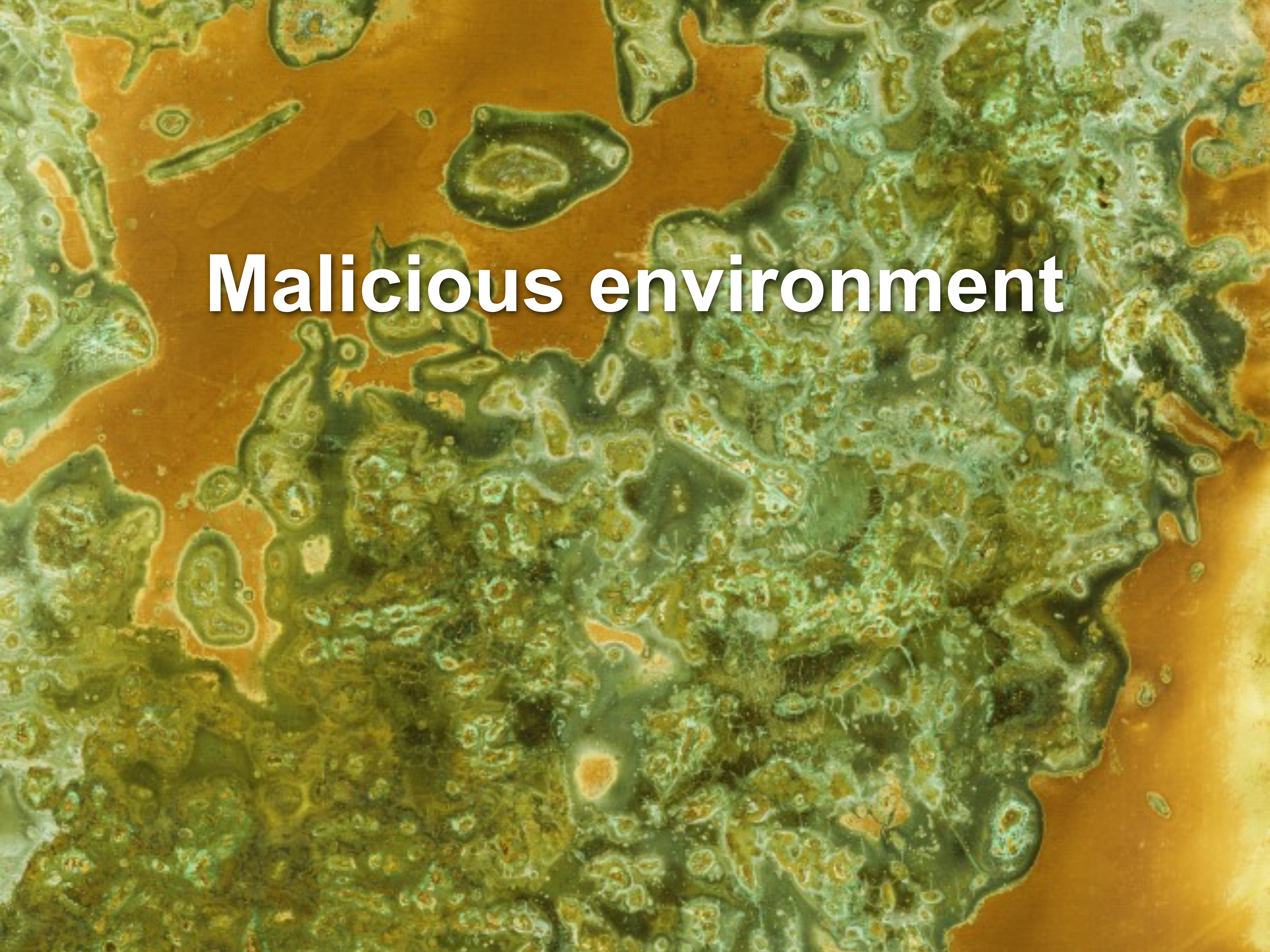
Users in the thousands and tens of thousands

24 hours a day, 365 days a year

Millions in hardware and software

Millions (or billions) in revenue

Highly interdependent systems



## Malicious environment

## Not Just the Network Admins' Problem

Since 2004, most *successful* attacks have been those targeting the application level, *not* the operating system.

## The Failure-Oriented Mindset

## Consider This

10,000,000 page views per day

× 50 assets per page

× 3 years

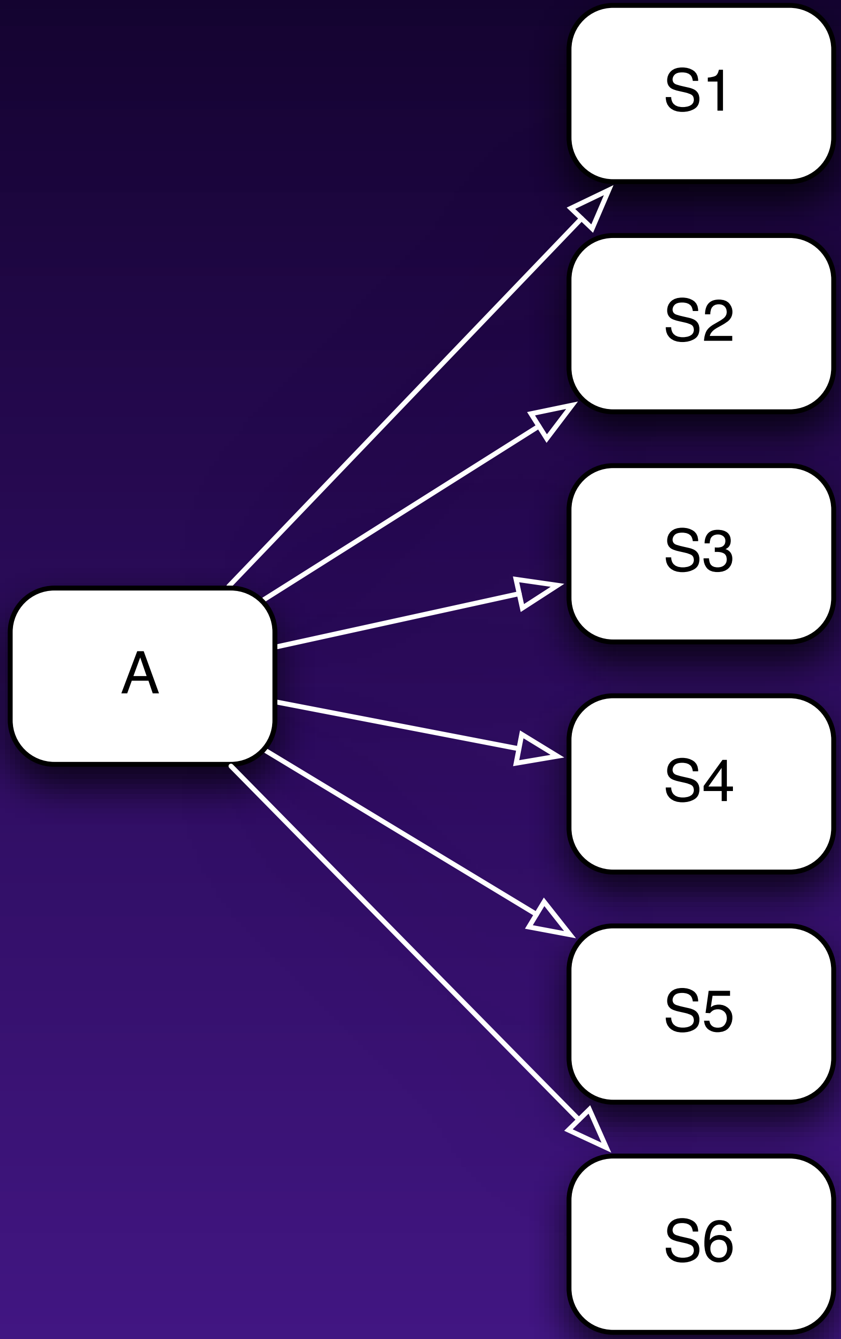
= $5.47 \times 10^{11}$ opportunities for error

“Six Sigma” quality produces 1,861,500 errors.

# Another Way To Look At It

S1 - S6 = 99% available

A ≤ 94%



## Still One More Way

Software Change

- Quarterly releases
- 10 enterprise apps

120 deployments per year

Some of these will be in your dependency set.

Hardware Change

- 200 servers
- 3 year refresh cycle

67 server swaps per year

Deployments **will** happen during a hardware change or failure.

## Failure is an Invariant

No matter what you do, some portion of your application will be malfunctioning some appreciable part of the time.

Stuff happens.

Expect it.

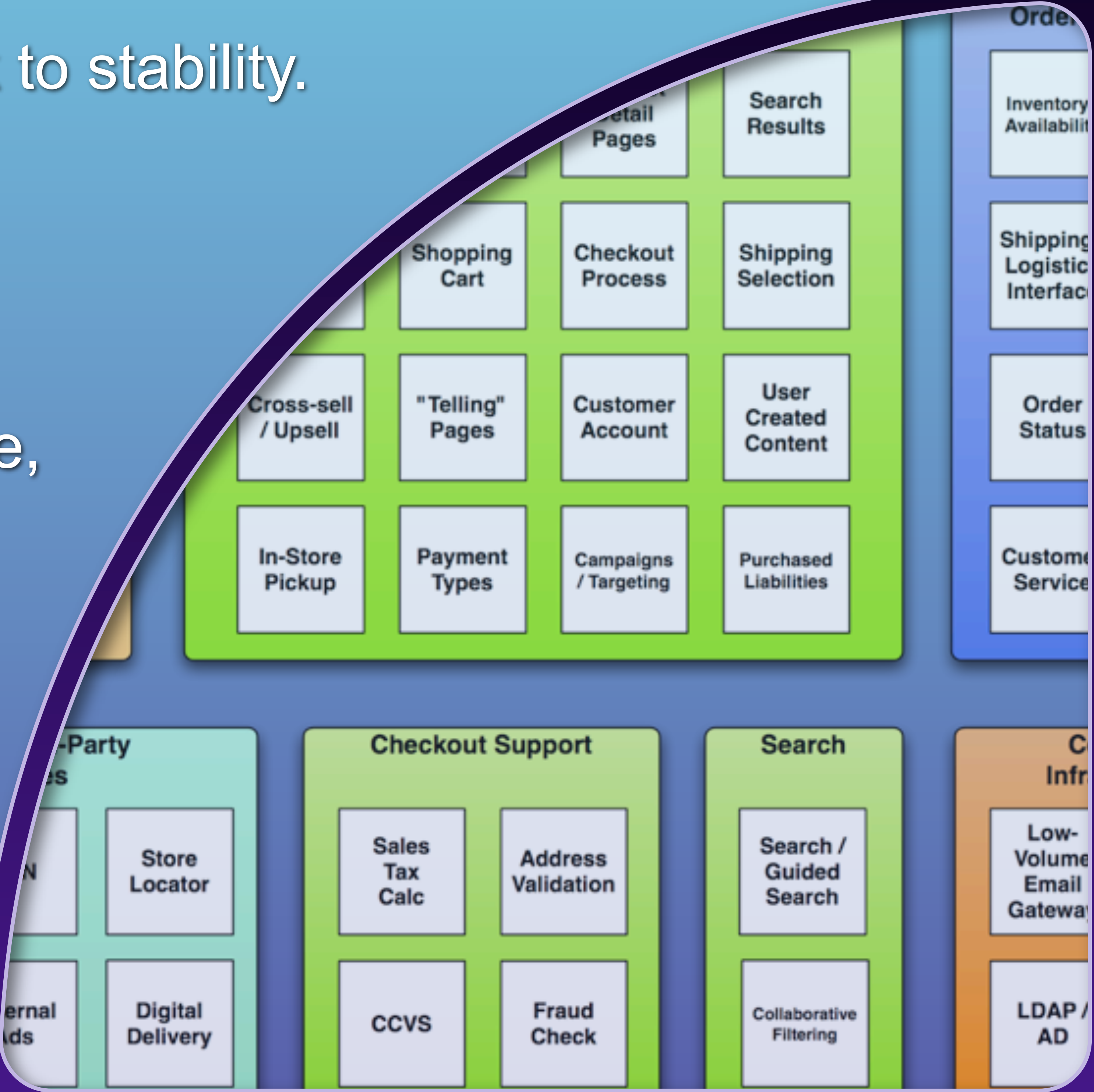
Deal with it.

## Stability Antipatterns

# Integration Points

- Integrations are the #1 risk to stability.

- Your first job is to protect against integration points.
- Every socket, process, pipe, or remote procedure call can and will eventually kill your system.
- Even database calls can hang, in obvious and not-so-obvious ways.



## Example: Wicked database hang

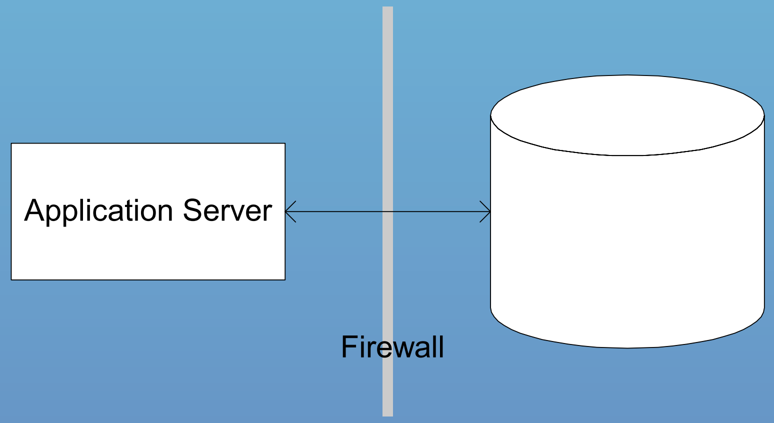
**Not at all obvious:** Firewall idle connection timeout

"Connection" is an abstraction.

The firewall only sees packets.

It keeps a table of "live" connections.

When the firewall sees a TCP teardown sequence, it removes that connection from the table.



To avoid resource leaks, it will drop entries from table after idle period timeout.

- Causes broken database connections after long idle period, like 2 a.m. to 5 a.m.

**Simple solution:** Enable "dead connection detection" (Oracle) or similar feature to keep connection alive.

**Alternative solution:** timed job to periodically issue trivial query.

**What about prevention?**

# "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

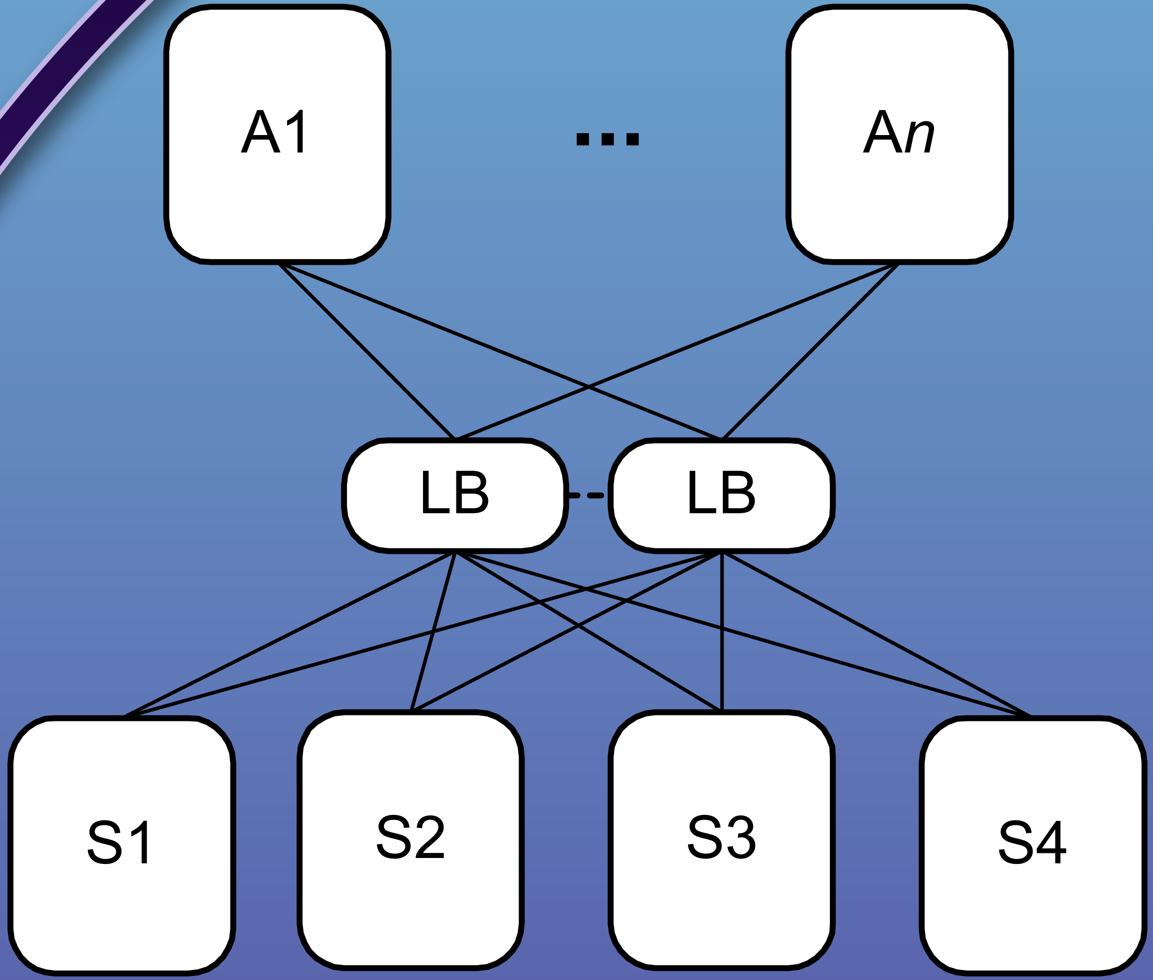
## Remember This

- Beware this necessary evil.
- Prepare for the many forms of failure.
- Know when to open up abstractions.
- Failures propagate quickly.
- Large systems fail faster than small ones.
- Apply “Circuit Breaker”, “Use Timeouts”, “Use Decoupling Middleware”, and “Handshaking” to contain and isolate failures.
- Use “Test Harness” to find problems in development.

# Chain Reaction

Failure in one component raises probability of failure in its peers

- Example:
  - Suppose S4 goes down
  - S1 - S3 go from 25% of total to 33% of total
  - That’s 33% more load
- Each one dies faster
- Failure moves horizontally across tier
- Common in search engines and application servers

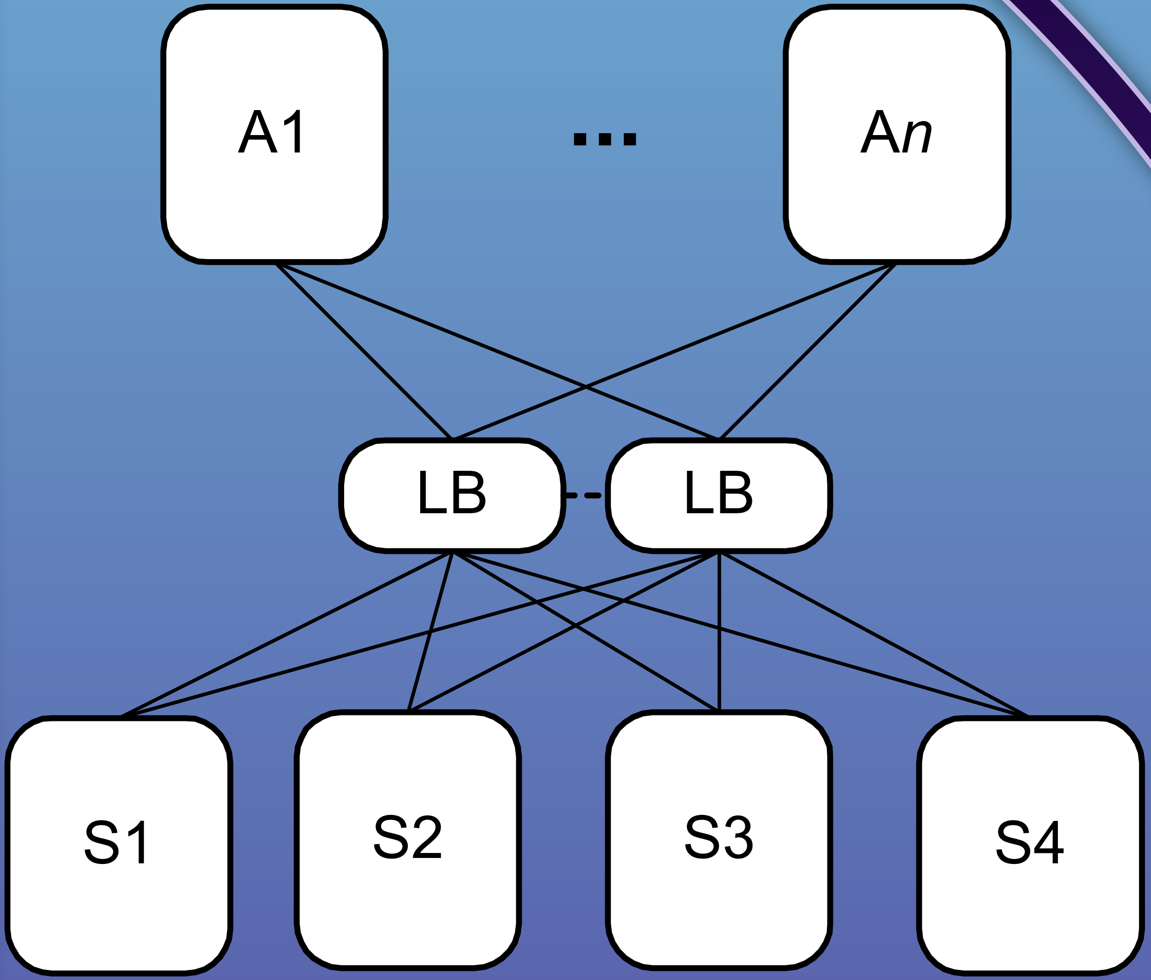


## Remember This

- One server down jeopardizes the rest.
- Hunt for Resource Leaks.
- Defend with “Bulkheads”.

# Cascading Failure

Failure in one system causes calling systems to be jeopardized



Example:

System S goes down, causing calling system A to get slow or go down.

- Failure moves vertically across tiers
- Common in enterprise services and SOAs

## Remember This

- Prevent Cascading Failure to stop cracks from jumping the gap.
- Think “Damage Containment”
- Scrutinize resource pools, they get exhausted when the lower layer fails.
- Defend with “Use Timeouts” and “Circuit Breaker”.

## Blocked Threads

Request handling threads are precious.  Protect them.

- Most common form of “crash”: all request threads blocked
- Very difficult to test for
  - Combinatoric permutation of code pathways.
  - Safe code can be extended in unsafe ways.
  - Errors are sensitive to timing and difficult to reproduce
  - Dev & QA servers never get hit with 10,000 concurrent requests.
- Best bet: keep threads isolated.  Use well-tested, high-level constructs for cross-thread communication.
  - Learn to use java.util.concurrent or System.Threading

## Pernicious and Cumulative

- Hung request handlers reduce the server’s capacity.
- Eventually, a restart will be required.
- Each hung request handler indicates a frustrated user or waiting caller
- The effect is non-linear and accelerating
  - Each remaining thread serves 1/N-1 extra requests

# Example: Blocking calls

- Example:

  In a request-processing method:

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

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

```
Object obj = items.get(id);
if(obj == null) {
  obj = remoteSystem.lookup(id);
}
…
```

- Remote system stopped responding due to “Unbalanced Capacities”
- Threads piled up like cars on a foggy freeway.

## Remember This

- Scrutinize resource pools.  Don’t wait forever.
- Use proven constructs.
- Beware the code you cannot see.
- Defend with “Use Timeouts”.

# Attacks of Self-Denial

Good marketing can kill your system at any time.

- Ever heard this one?
  - A retailer offered a great promotion to a “select group of customers”.
  - Approximately a bazillion times the expected customers show up for the offer.
  - The retailer gets crushed, disappointing the avaricious and legitimate.
- It’s a self-induced Slashdot effect.

Victoria’s Secret: Online Fashion Show

BestBuy: XBox 360 Preorder

Amazon: XBox 360 Discount

*Anything* on FatWallet.com

# Defending the Ramparts

- Avoid deep links
- Set up static landing pages
- Only allow the user's second click to reach application servers
- Allow throttling of incoming users
- Set up lightweight versions of dynamic pages.
- Use your CDN to divert users
- Use shared-nothing architecture

One email I saw went out with a deep link that bypassed Akamai. Worse, it encoded a specific server and included a session ID.

Another time, an email went out with a promo code. It could be used an unlimited number of times.

Once a vulnerability is found, it will be flooded within seconds.

# Remember This

- Keep lines of communication open
  - Support the marketers.  If you don’t, they’ll invent their way around you, and might jeopardize the systems.
- Protect shared resources
- Expect instantaneous distribution of exploits

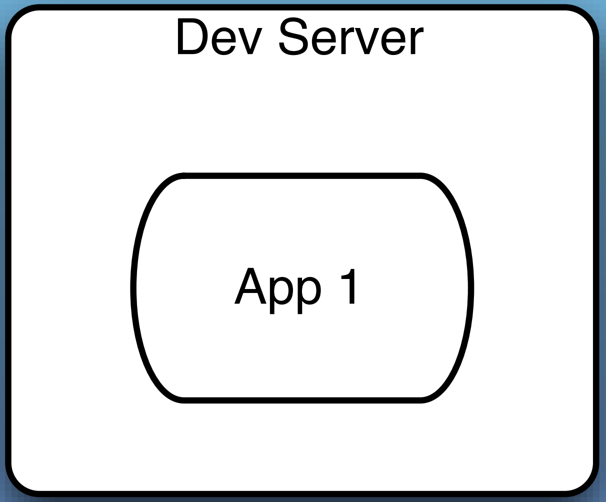
## Scaling Effects

Understand which end of the lever you are sitting on.

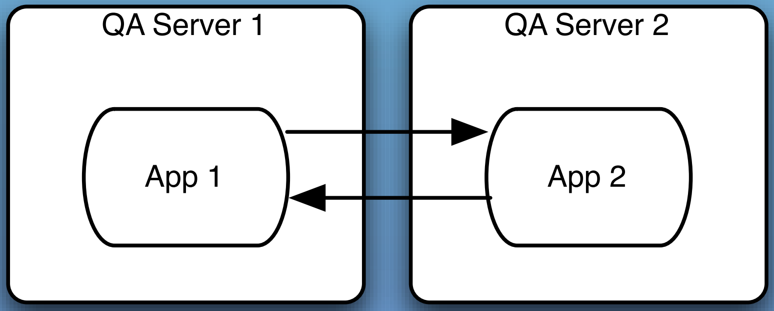
- Ratios in dev and QA tend to be 1:1
  - Web server to app server
  - Front end to back end
- They differ wildly in production, so designs and architectures may not be appropriate

# Example: Point to Point Cache Invalidation

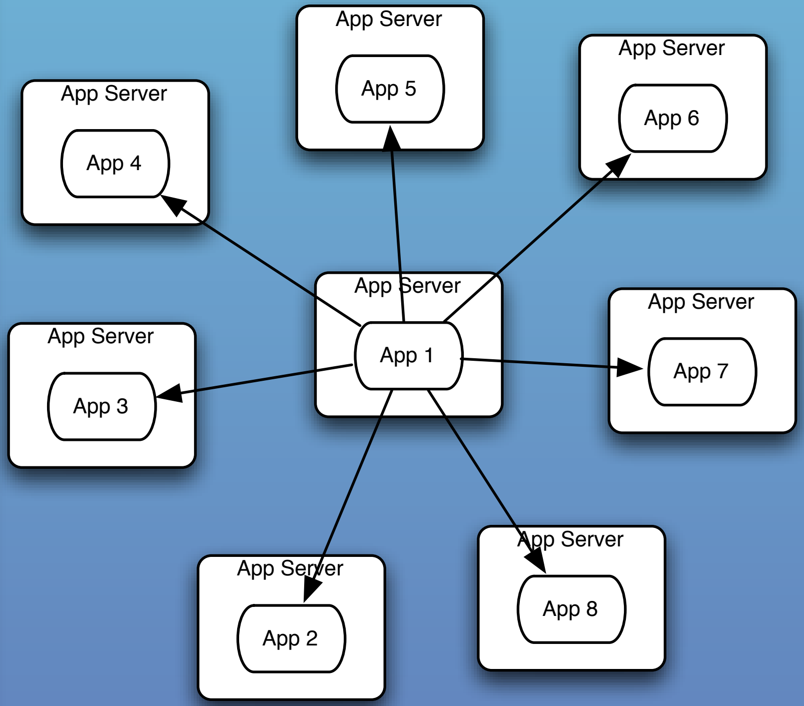
## Development



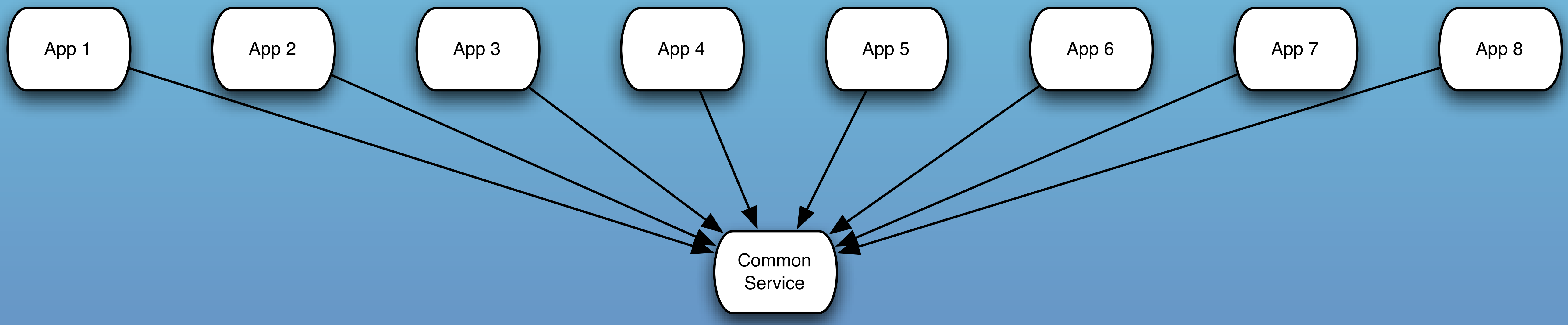
## QA



## Production



# Example: Shared Resources



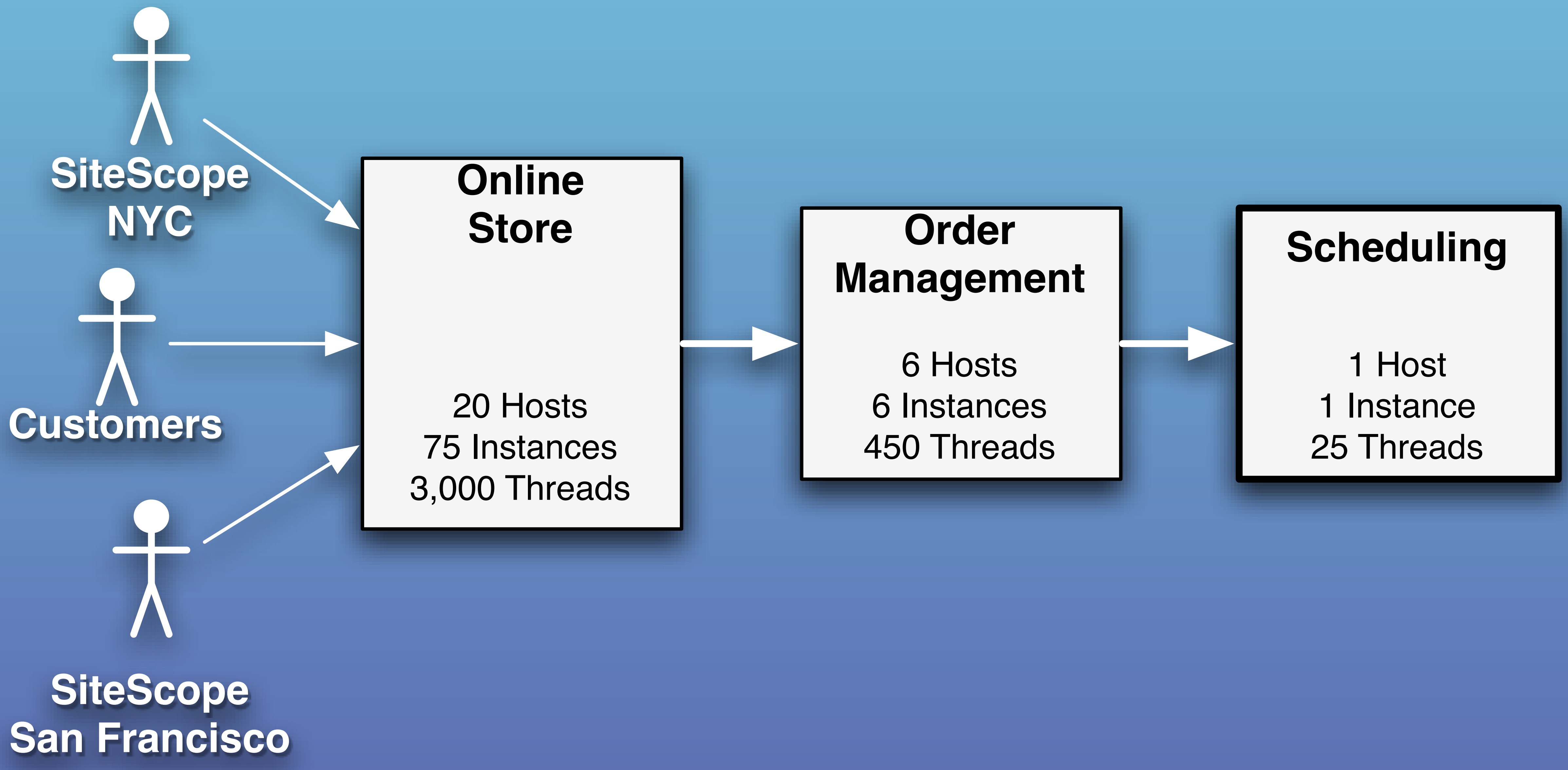
Shared resources commonly appear as lock managers, load managers, query distributors, cluster managers, and message gateways.  They're all vulnerable to scaling effects.

## Remember This

- Examine production versus QA environments to spot scaling effects.
- Watch out for point-to-point communications. It rarely belongs in production.
- Watch out for shared resources.

# Unbalanced Capacities

Traffic floods sometimes start inside the data center walls.



## Unbalanced Capacities

- Unbalanced capacities is a type of scaling effect that occurs between systems in an enterprise.
- It happens because
  - All dev systems are one server
  - Almost all QA environments are two servers
  - Production environments may be 10:1 or 100:1
- May be induced by changes in traffic or behavior patterns

## Remember This

- Examine server and thread counts
- Watch out for changes in traffic patterns
- Stress both sides of the interface in QA
- Simulate back end failures during testing

# Slow Responses

Slow response is worse than no response

What does your server do when it’s overloaded?

- “Connection refused” is a fast failure, the caller’s thread is released right away
- A slow response ties up the caller’s thread, makes the user wait
- It uses capacity on caller and receiver
- If the caller times out, then the work was wasted

## Slow Responses

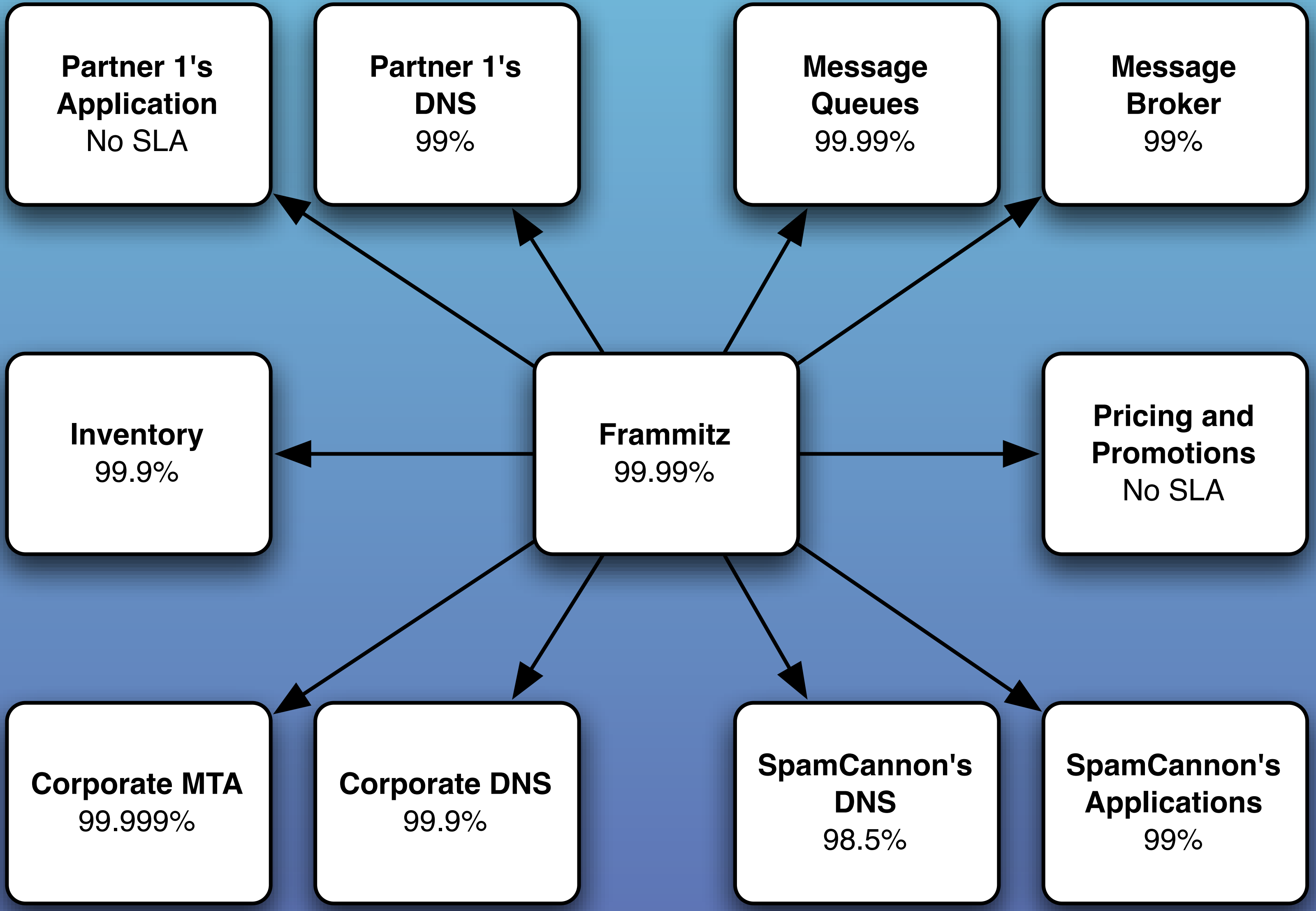
- Look at the latency:
  - TCP connection refused comes back in ~10 ms
  - TCP packets not acknowledged, sender retransmits for 1 – 10 min
- Causes of slow responses:
  - Too much load on system
  - Transient network saturation
  - Firewall overloaded
  - Protocol with retries built in (NFS, DNS)
  - Chatty remote protocols

## Remember This

- Slow responses trigger cascading failures
- For websites, slow responses invite more traffic as the users pound “reload”
- Don’t send a slow response; fail fast
- Hunt for memory leaks or resource contention

# SLA Inversion

Surviving by luck alone.



What SLA can Frammitz *really* guarantee?

Absent other protections, the best SLA you can offer is the *worst* SLA provided by your dependencies.

The dreaded SPOF is a special case of SLA Inversion.

Do your web servers have to ask DNS to find the application server's IP address?

## Remember This

- Don’t make empty promises.  Be sure you can deliver the SLA you commit to.
- Examine every dependency.  Verify that *they* can deliver on their promises.
- Decouple your SLAs from your dependencies’.
- Measure availability by feature, not by server.
- Be wary of “enterprise” services such as DNS, SMTP, and LDAP.

# Unbounded Result Sets

Limited resources, unlimited data volume

- Development and testing is done with small data sets
- Test databases get reloaded frequently
- Queries that perform acceptably in development and test bonk badly with production data volume.
  - Bad access patterns can make them very slow
  - Too many results can use up all your server's RAM or take too long to process
  - You never know when somebody else will mess with your data

## Unbounded Result Sets: Databases

- SQL queries have no inherent limits
- ORM tools are bad about this
- It starts as a degenerating performance problem, but can tip the system over.
- For example:
  - Application server using database table to pass message between servers.
  - Normal volume 10 – 20 events at a time.
  - Time-based trigger on every user generated 10,000,000+ events at midnight.
  - Each server trying to receive **all** events at startup.
  - Out of memory errors at startup.

## Unbounded Result Sets: SOA

- Often found in chatty remote protocols, together with the N+1 query problem
- Causes problems on the client and the server
  - On server: constructing results, marshalling XML
  - On client: parsing XML, iterating over results.
- This is a breakdown in handshaking.  The client knows how much it can handle, not the server.

# Remember This

- Test with realistic data volumes
  - Scrubbed production data is the best.
  - Generated data also works.
- Don’t rely on the data producers.  Their behavior can change overnight.
- Put limits in your application-level protocols:
  - WS, RMI, DCOM, XML-RPC, etc.

# Questions?

Michael Nygard
michael@michaelnygard.com