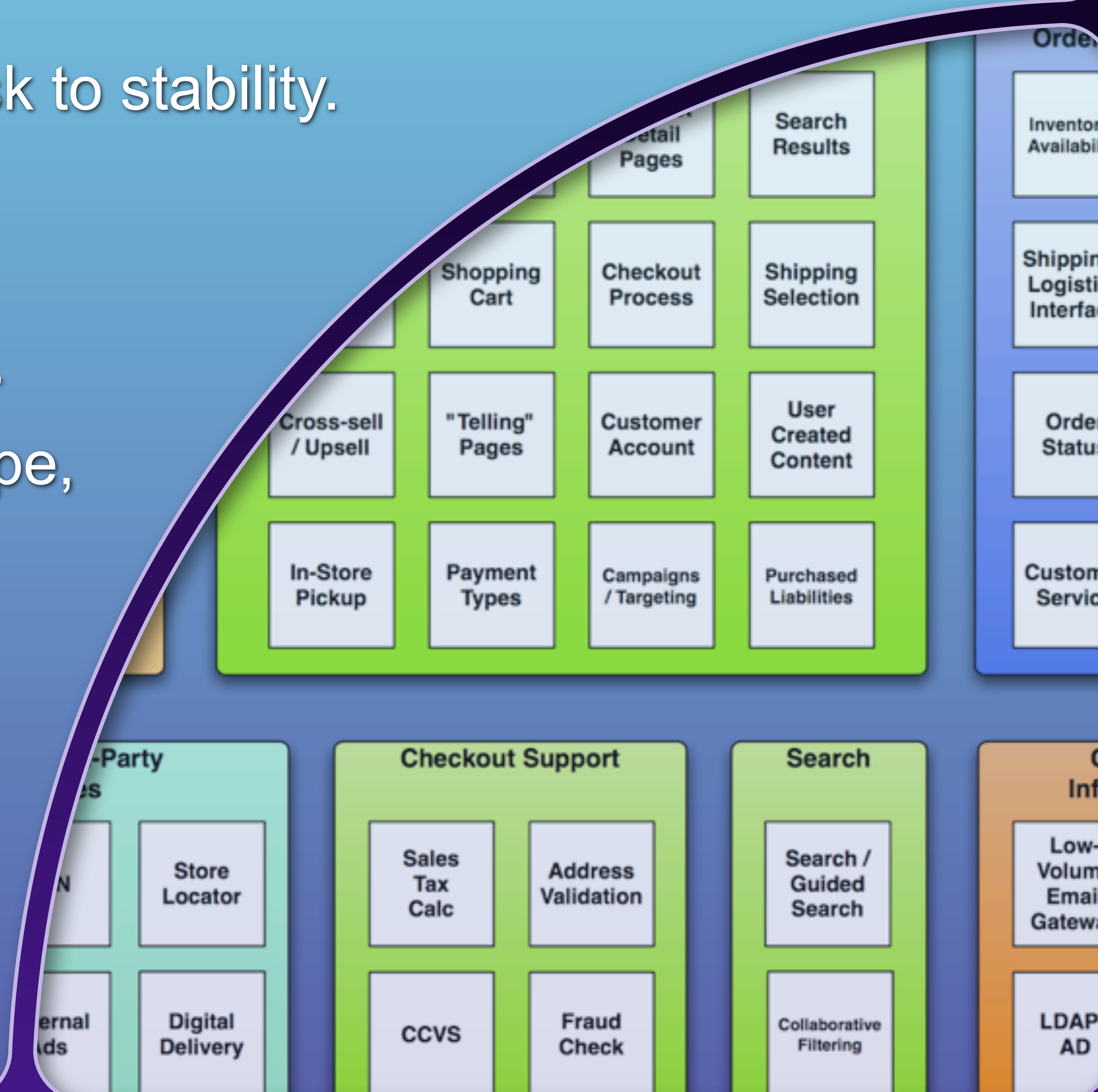
Failure Comes in Flavors
Stability Patterns and Antipatterns

Michael Nygard
mtnygard@thinkrelevance.com
@mtnygard

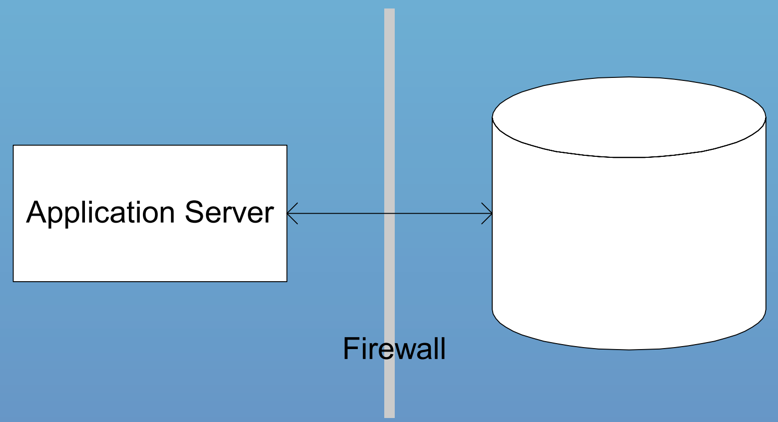
Stability Antipatterns

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



Example: Wicked database hang

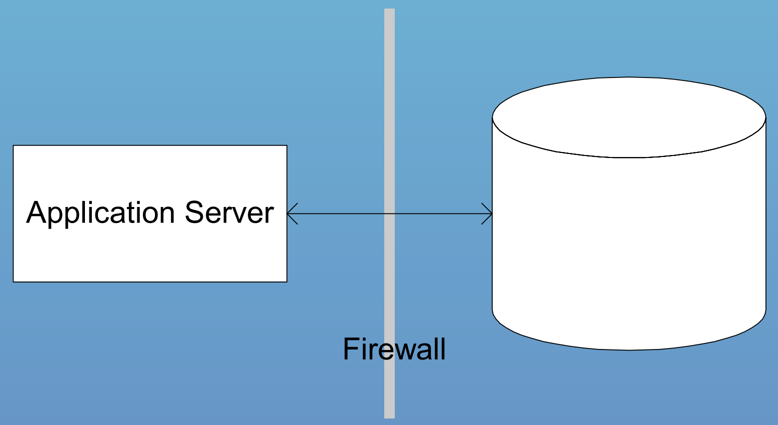
Not at all obvious: Firewall idle connection timeout

“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.

Example: Wicked database hang

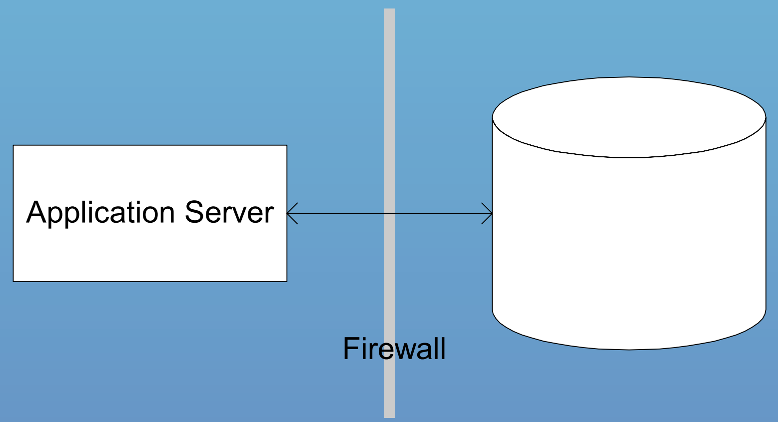
Not at all obvious: Firewall idle connection timeout

“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.

Example: Wicked database hang

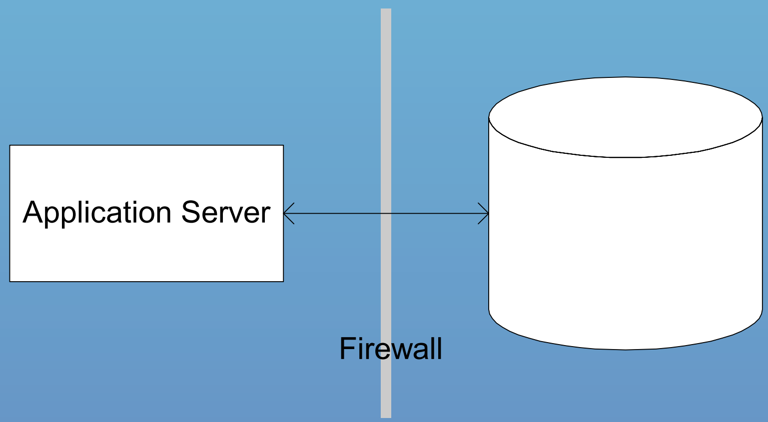
Not at all obvious: Firewall idle connection timeout

"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.

Example: Wicked database hang

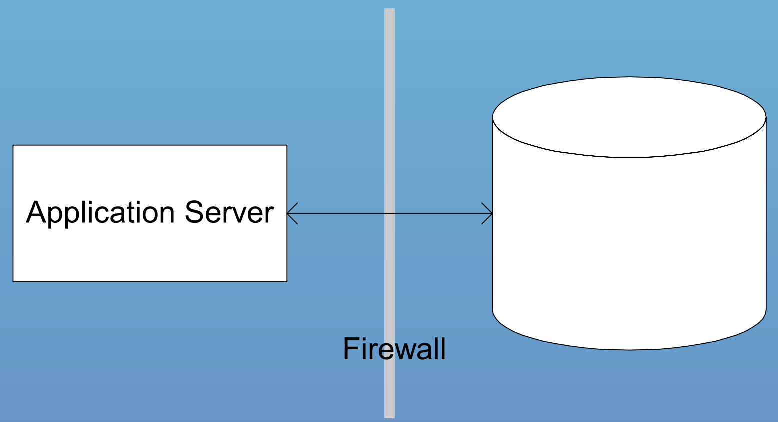
Not at all obvious: Firewall idle connection timeout

"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.

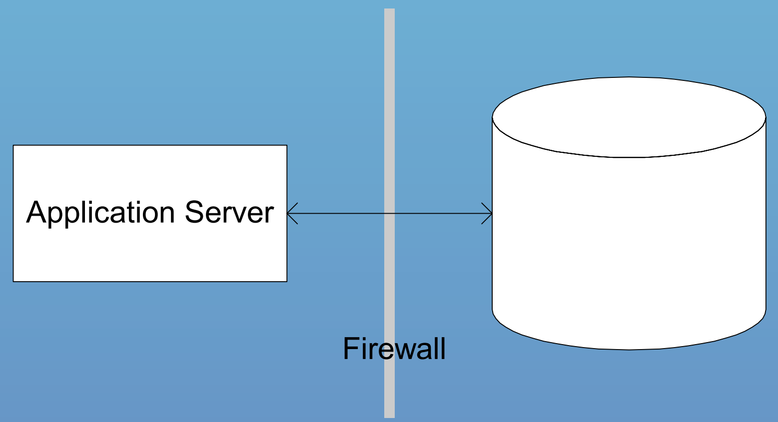
- Causes broken database connections after long idle period, like 2 a.m. to 5 a.m.

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

Alternative solution: timed job to periodically issue trivial query.

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

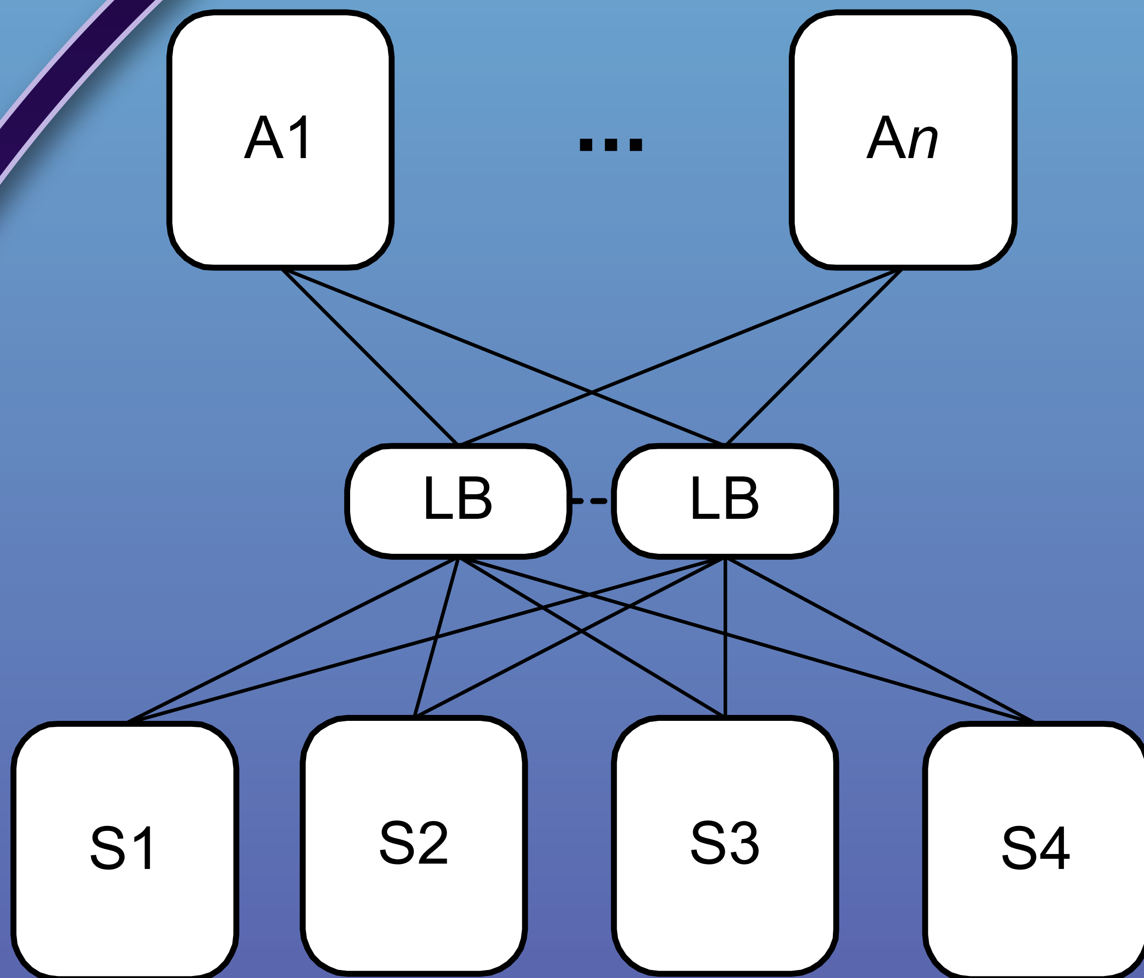
Remember This

- 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

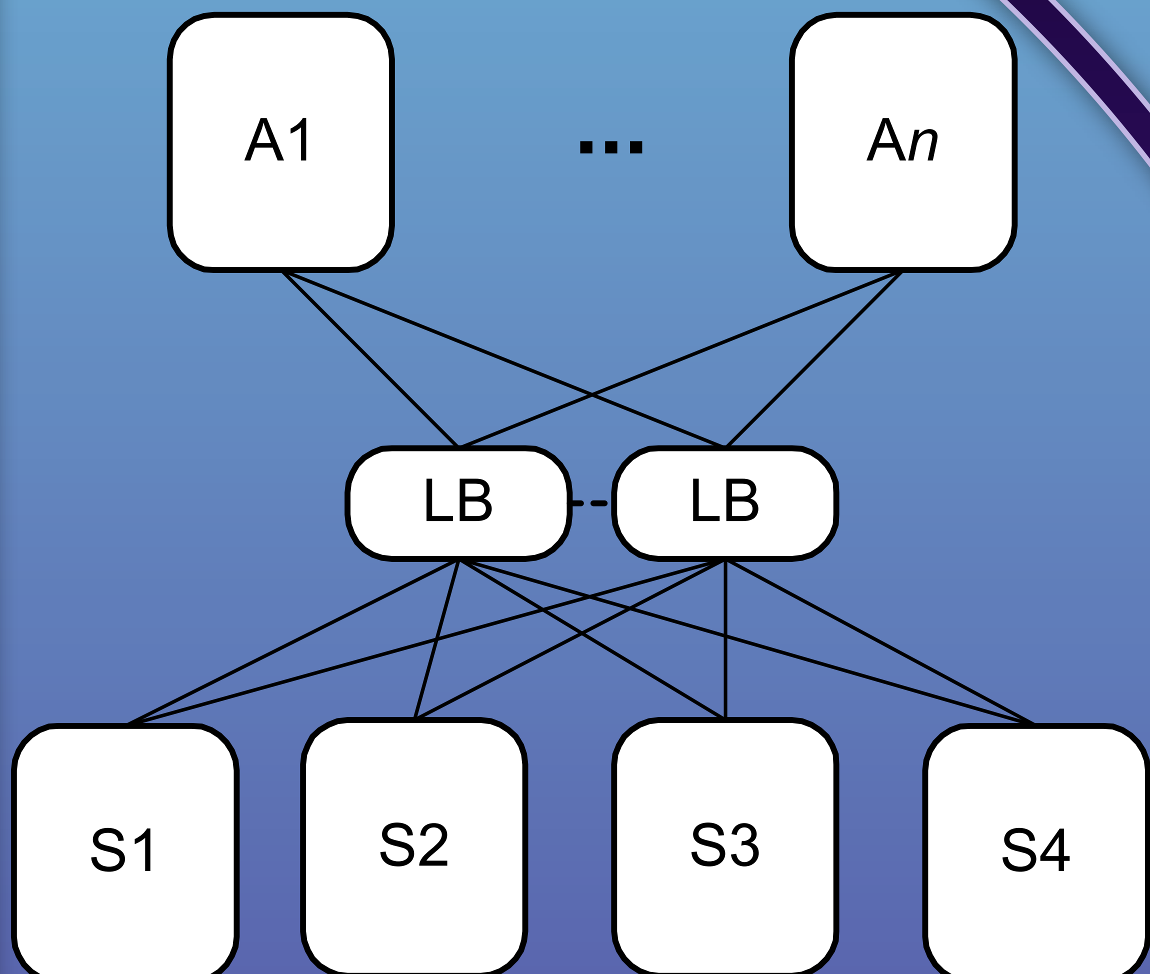


Remember This

- 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”.

Users

Can’t live with them...

- Ways that users cause instability
  - Sheer traffic
  - Flash mobs
  - Click-happy
- Malicious users
  - Screen-scrapers
  - Badly configured proxy servers

Two types of "bad" user

- Buyers
  - Most expensive type of user to service: more pages, more integration points, and SSL
  - High conversion rate is bad for the systems.
- Bargain hunters/Screen scrapers
  - Create useless sessions
  - Divert, throttle, or avoid creating sessions
  - Especially for spiders

Handle Traffic Surges Gracefully

- Turn off expensive features when the system is busy.
- Divert or throttle users. Preserve a good experience for some when you can't serve all.
- Reduce the burden of serving each user. Be especially frugal with memory.
  - Hold IDs, not object graphs.
  - Hold query parameters, not result sets.
- Differentiate people from bots. Don't keep sessions for bots.

Remember This

- Minimize the memory you devote to each user.
- Malicious users are out there.
- But, so are weird random ones.
- Users come in clumps: one, a few, or way too many.

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: Blocking calls

- Example:

  In a request-processing method:

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

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);
}
…
```

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”

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.

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

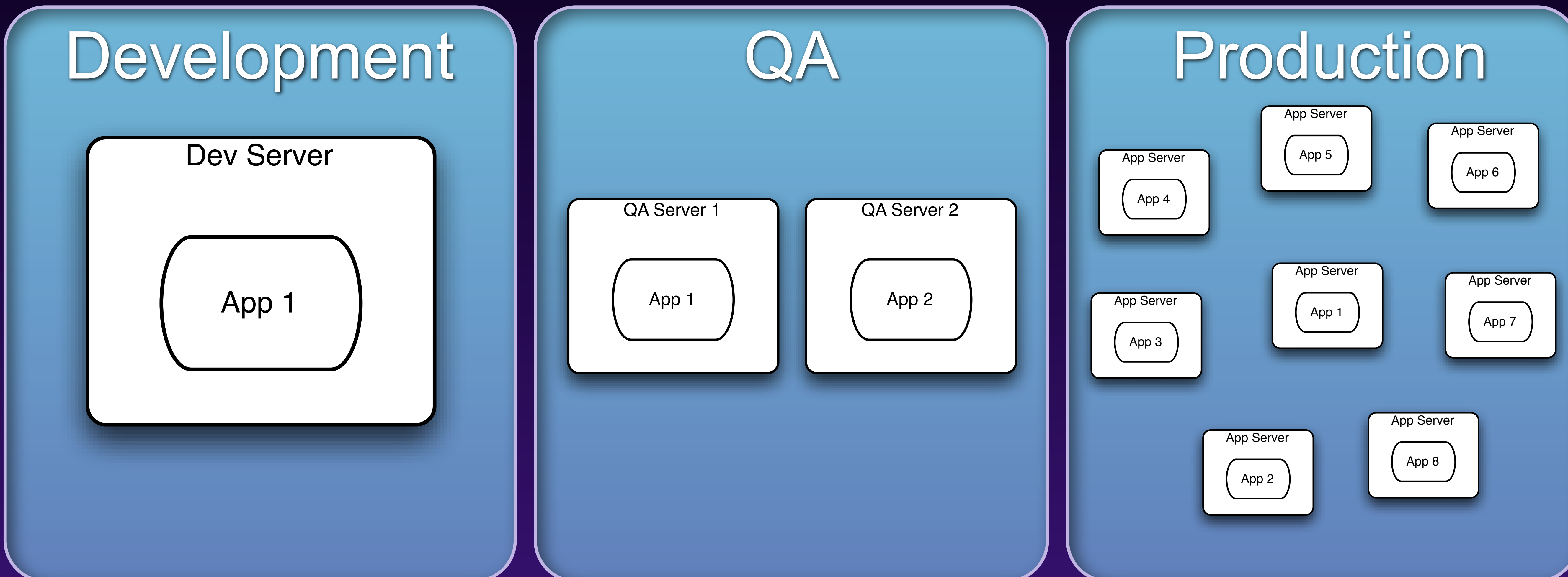
- Keep lines of communication open
  - Support the marketers. If you don't, they'll invent their way around you, and might jeopardize the systems.
- Protect shared resources
- Expect instantaneous distribution of exploits

Scaling Effects

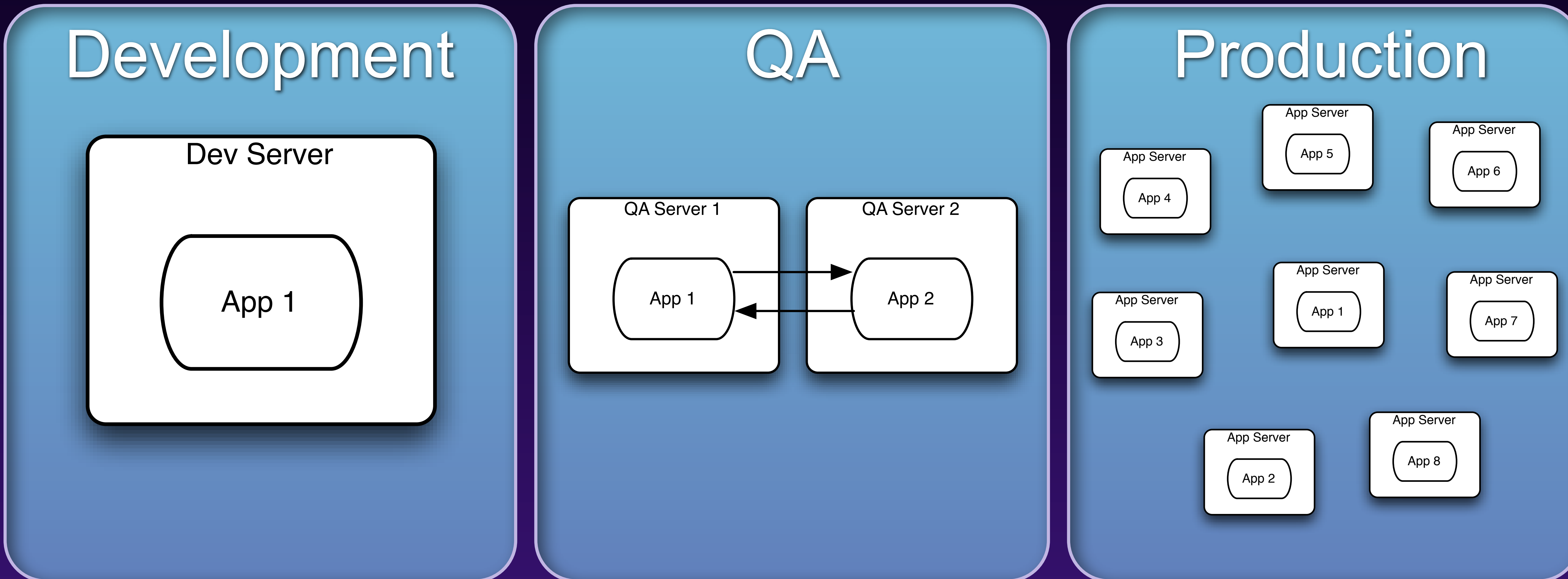
Understand which end of the lever you are sitting on.

- 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

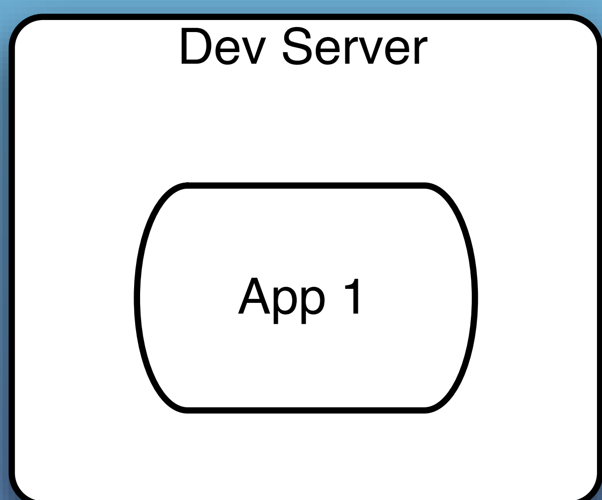


Example: Point to Point Cache Invalidation

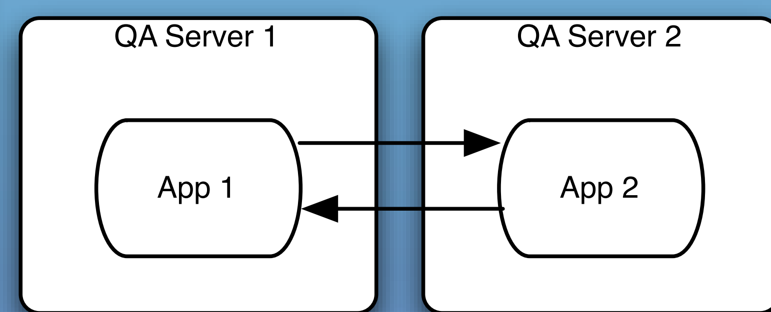


Example: Point to Point Cache Invalidation

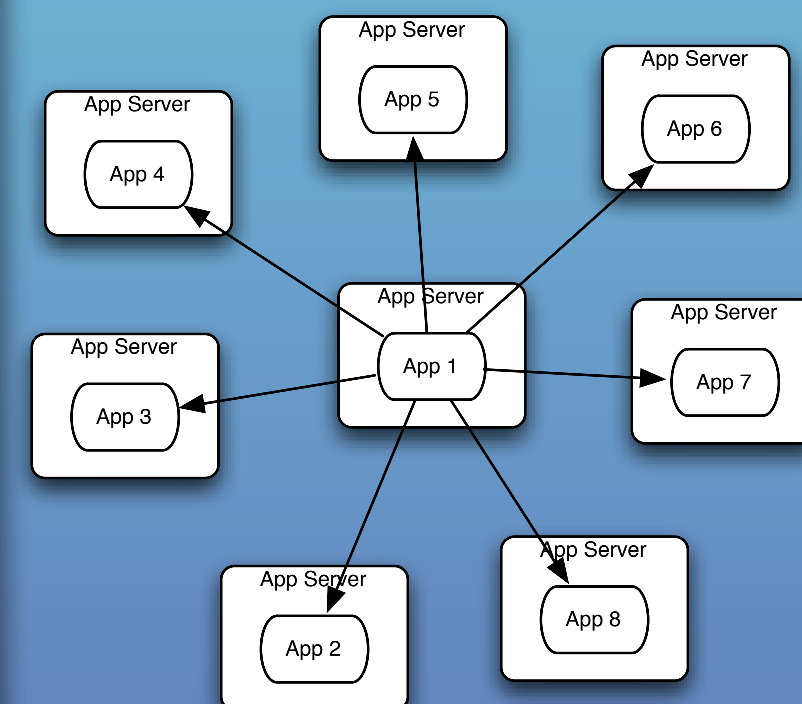
Development



QA

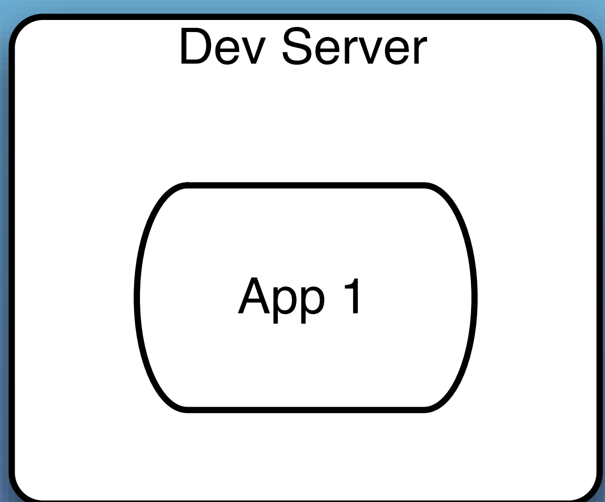


Production

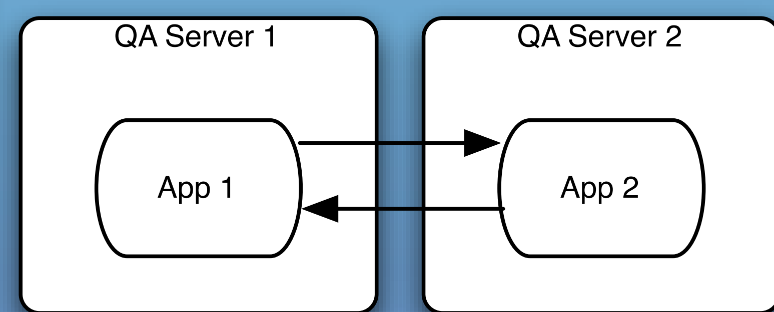


Example: Point to Point Cache Invalidation

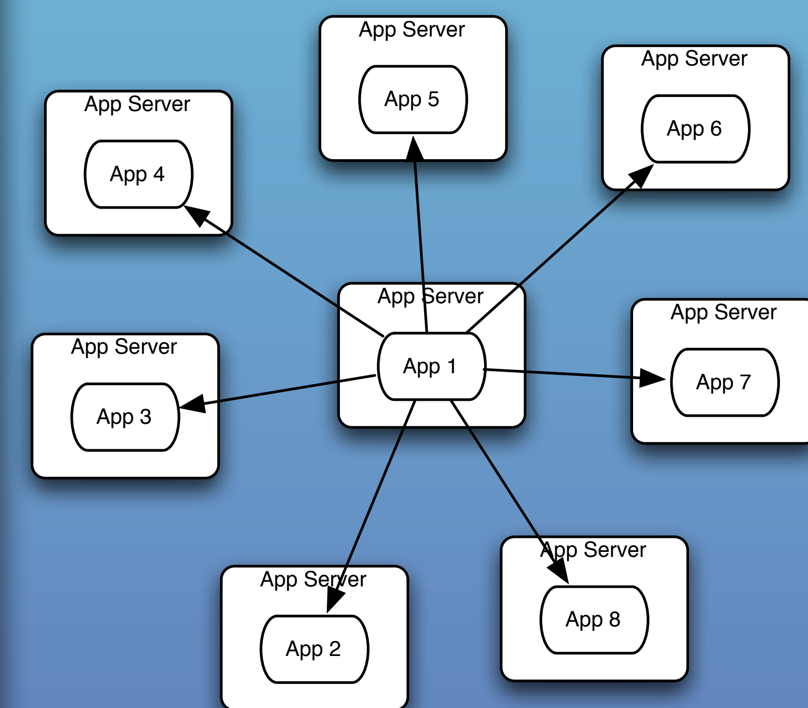
Development



QA



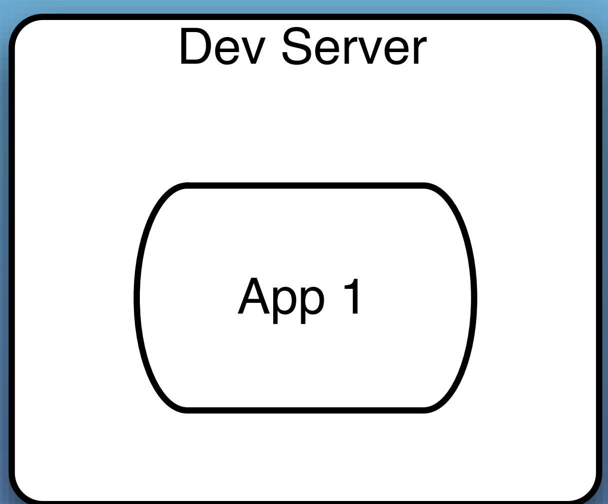
Production



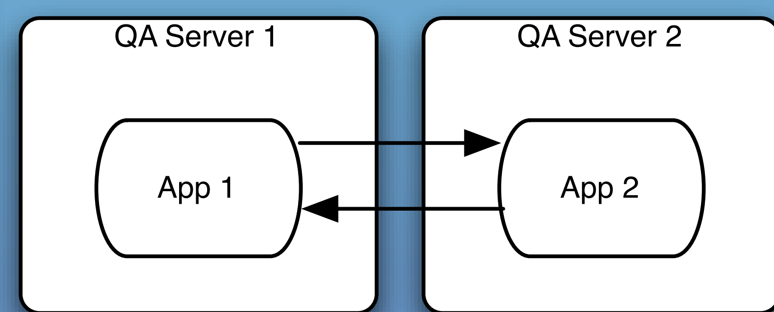
1 server
1 local call
No TCP connections

Example: Point to Point Cache Invalidation

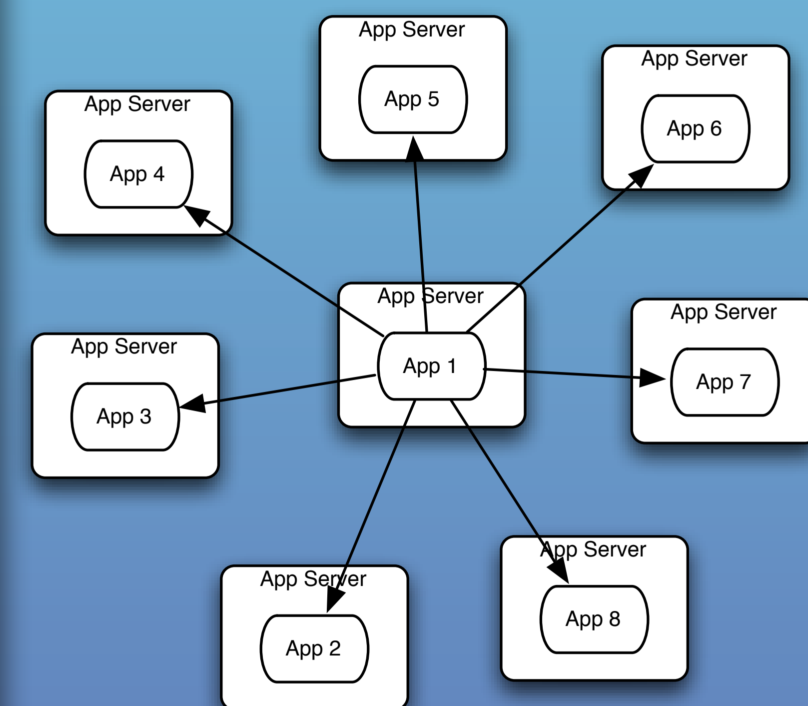
Development



QA



Production

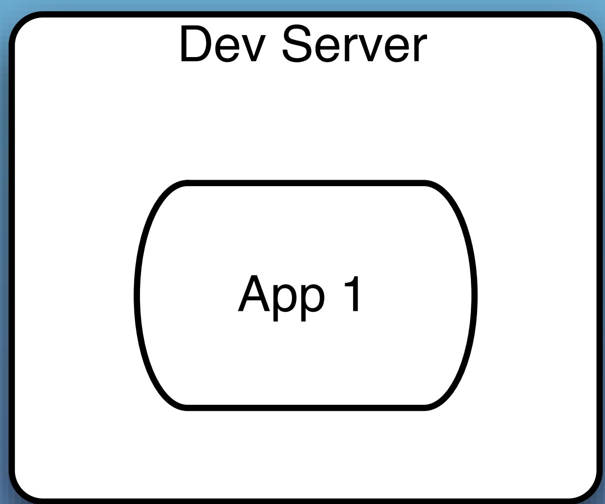


1 server
1 local call
No TCP connections

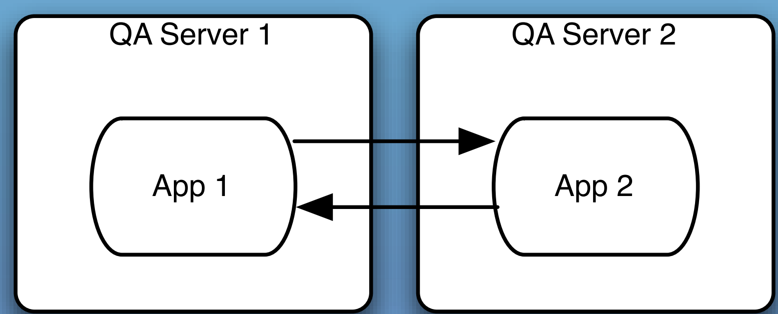
2 servers
1 local call
1 TCP connection

Example: Point to Point Cache Invalidation

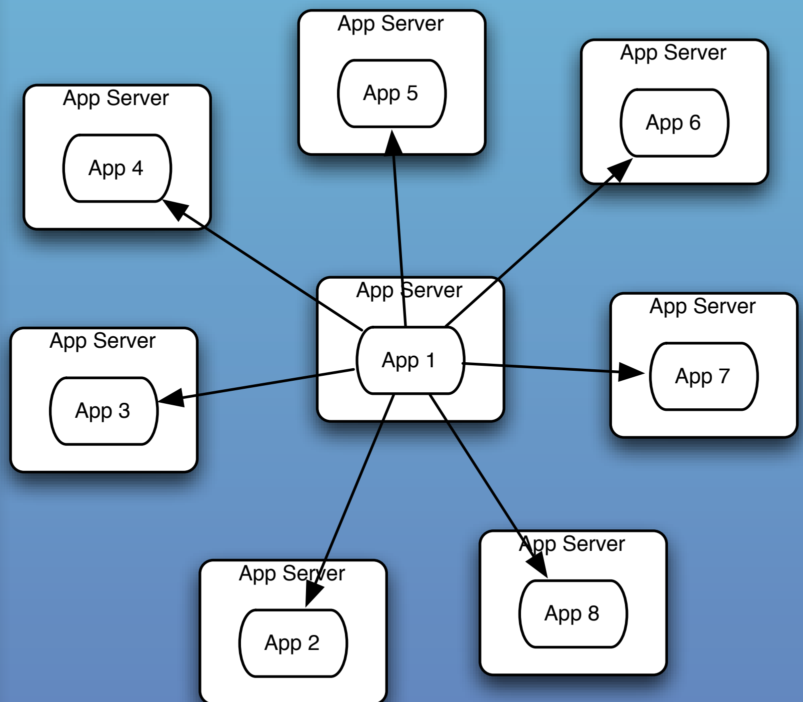
Development



QA



Production

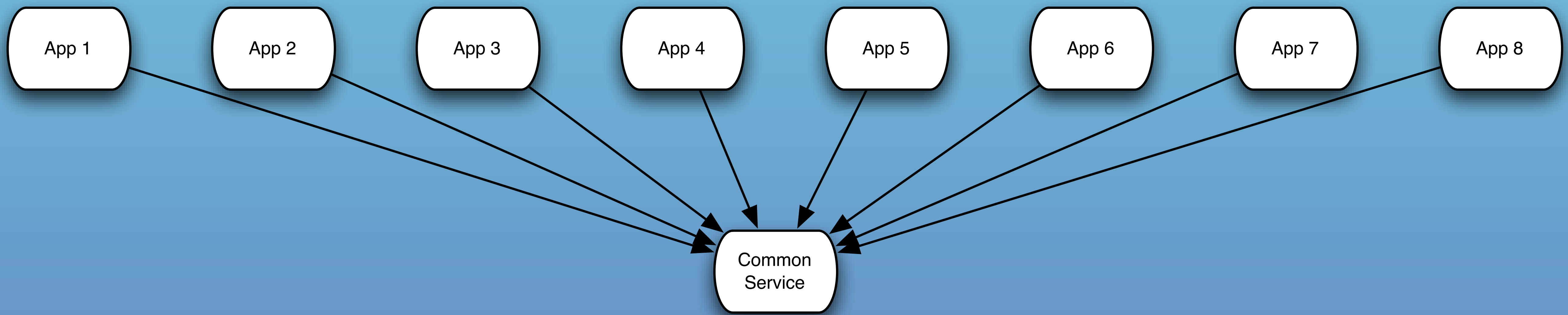


1 server
1 local call
No TCP connections

2 servers
1 local call
1 TCP connection

8 servers
1 local call
7 TCP connection

Example: Shared Resources



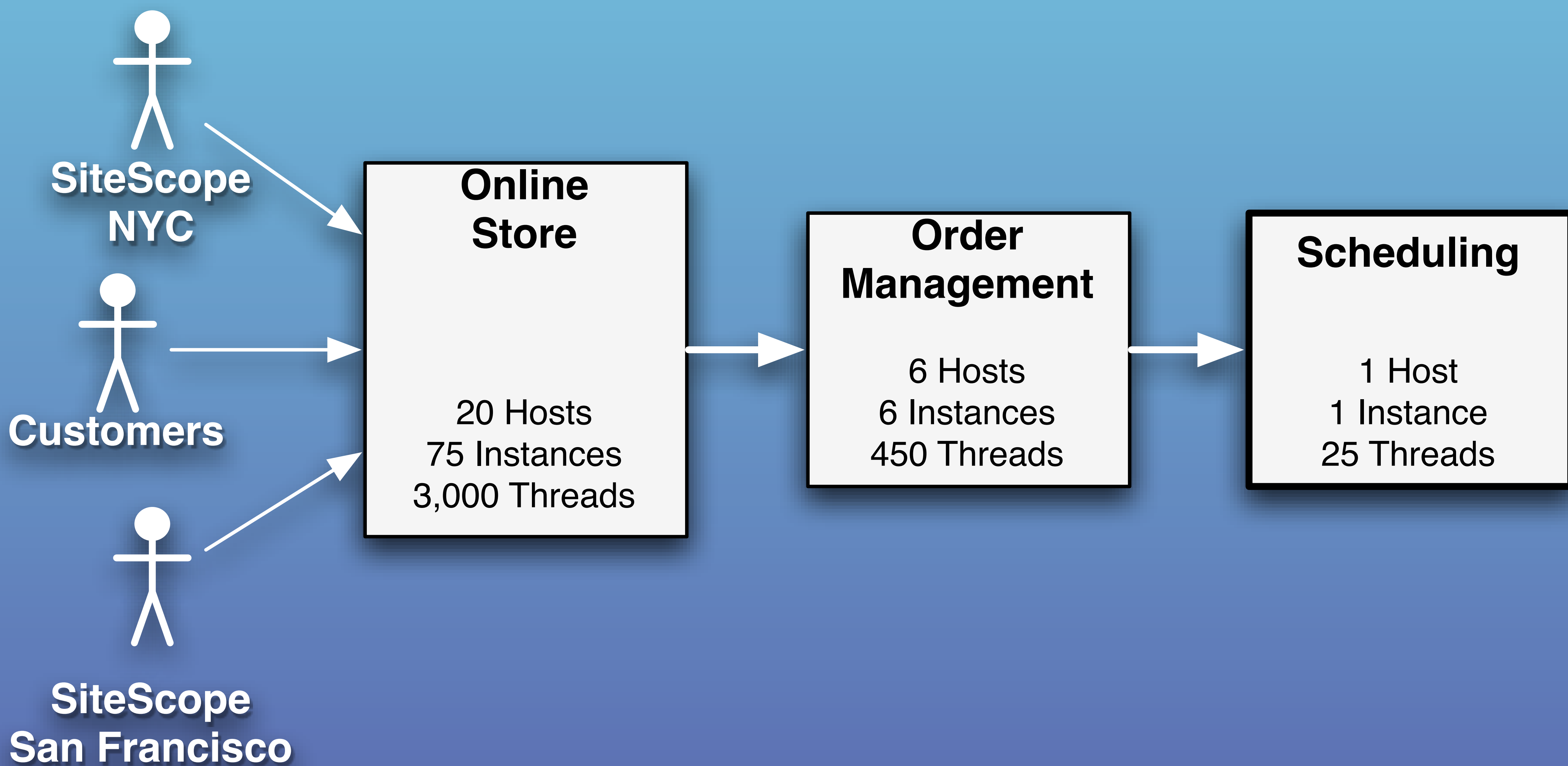
Shared resources commonly appear as lock managers, load managers, query distributors, cluster managers, and message gateways. They're all vulnerable to scaling effects.

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

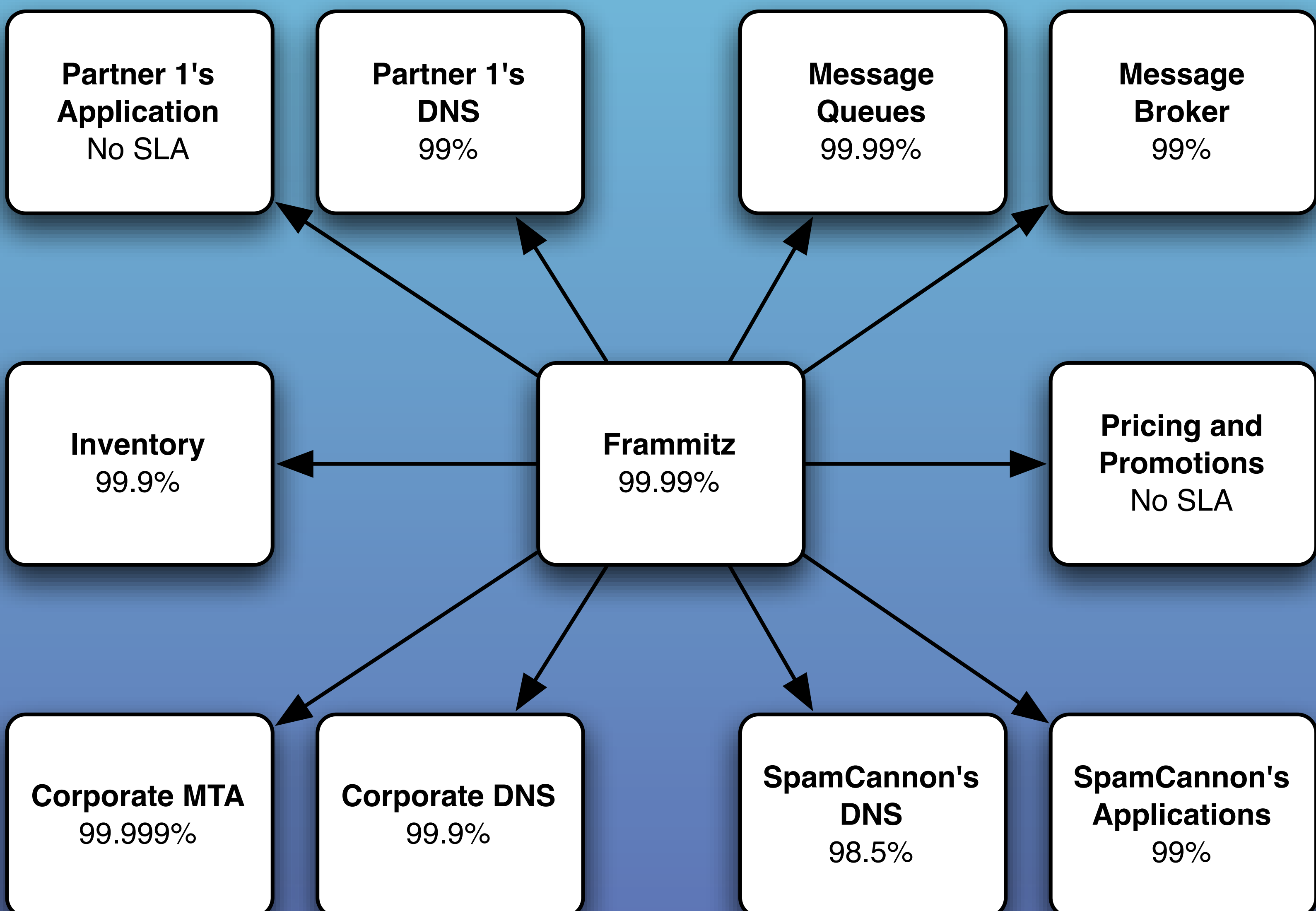
- Look at the latency:
  - TCP connection refused comes back in ~10 ms
  - TCP packets not acknowledged, sender retransmits for 1 – 10 min
- Causes of slow responses:
  - Too much load on system
  - Transient network saturation
  - Firewall overloaded
  - Protocol with retries built in (NFS, DNS)
  -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

Stability Patterns

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

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?

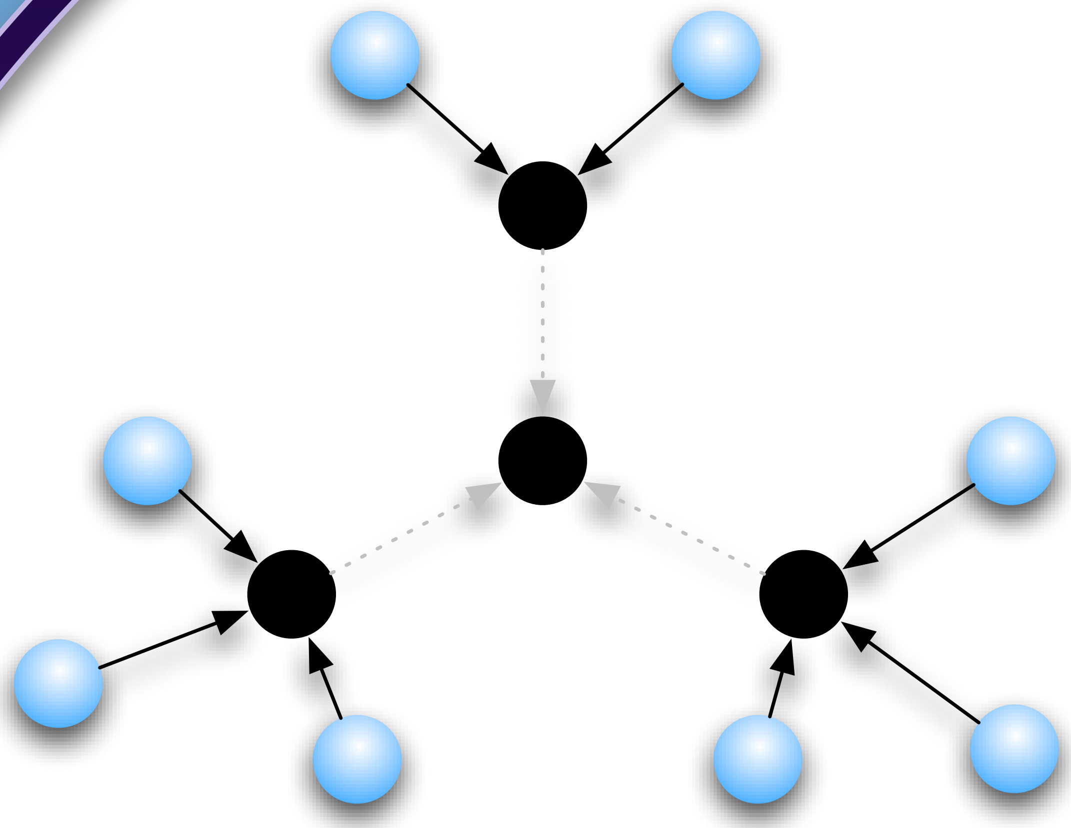
```
int remainingAttempts = MAX_RETRIES;

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?

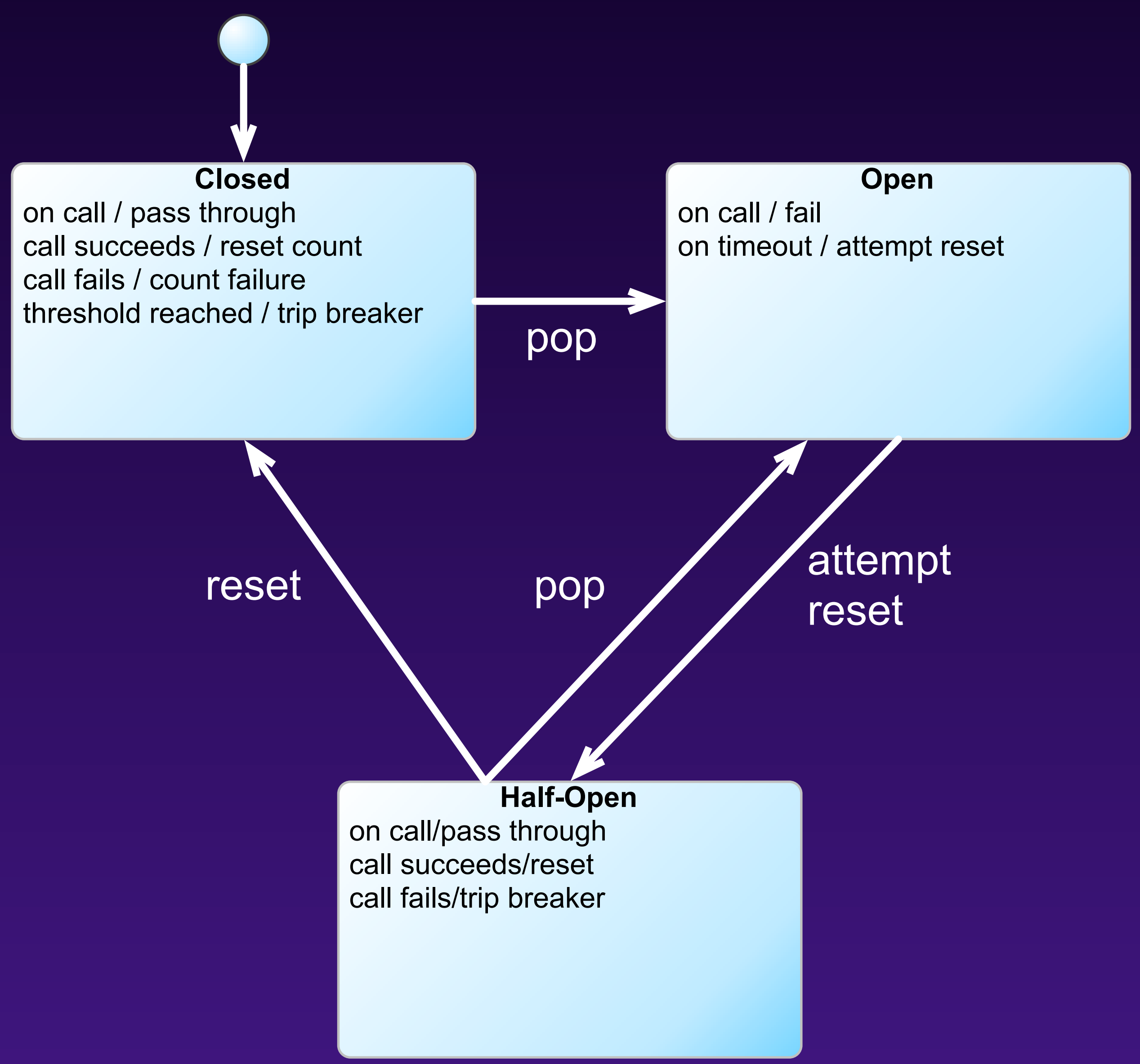
Systems:

- 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

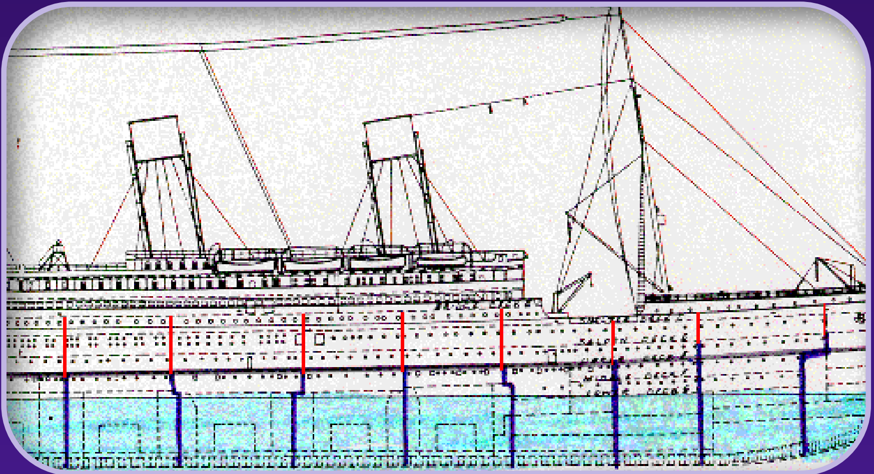
Bulkheads

Save part of the ship, at least.

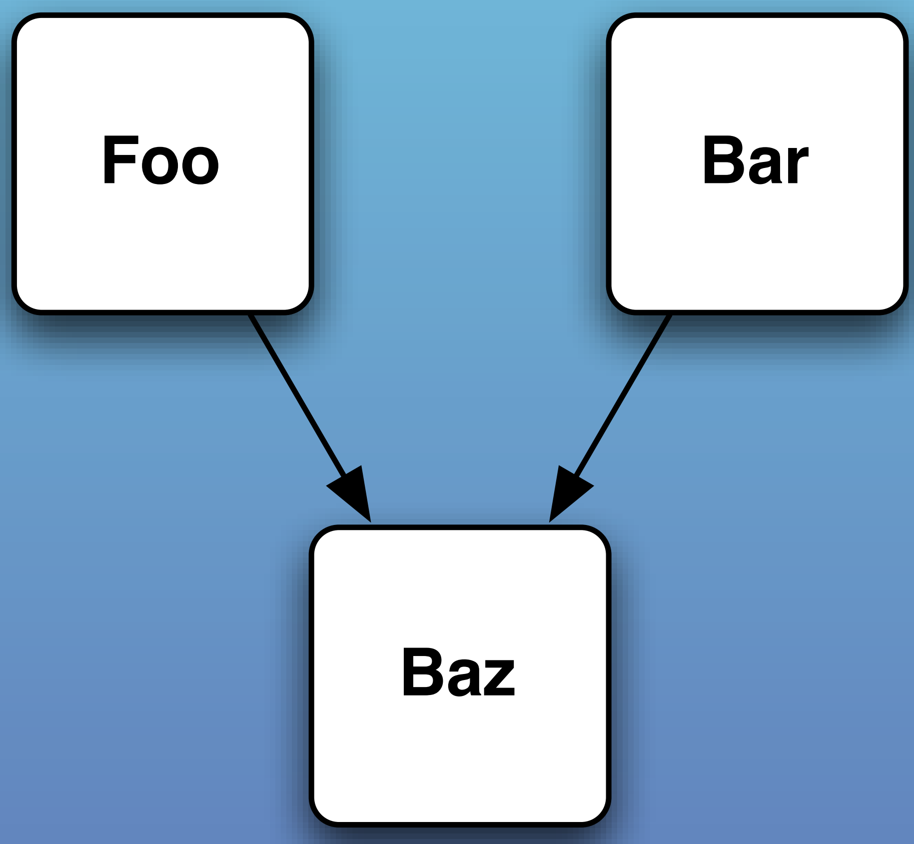
- Increase resilience by partitioning (compartmentalizing) the system
  - One part can go dark without losing service entirely
- Apply at several levels
  - Thread pools within a process
  - CPUs in a server (CPU binding)
  - Server pools for priority clients

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.



Common Mode Dependency: Service-Oriented Architecture



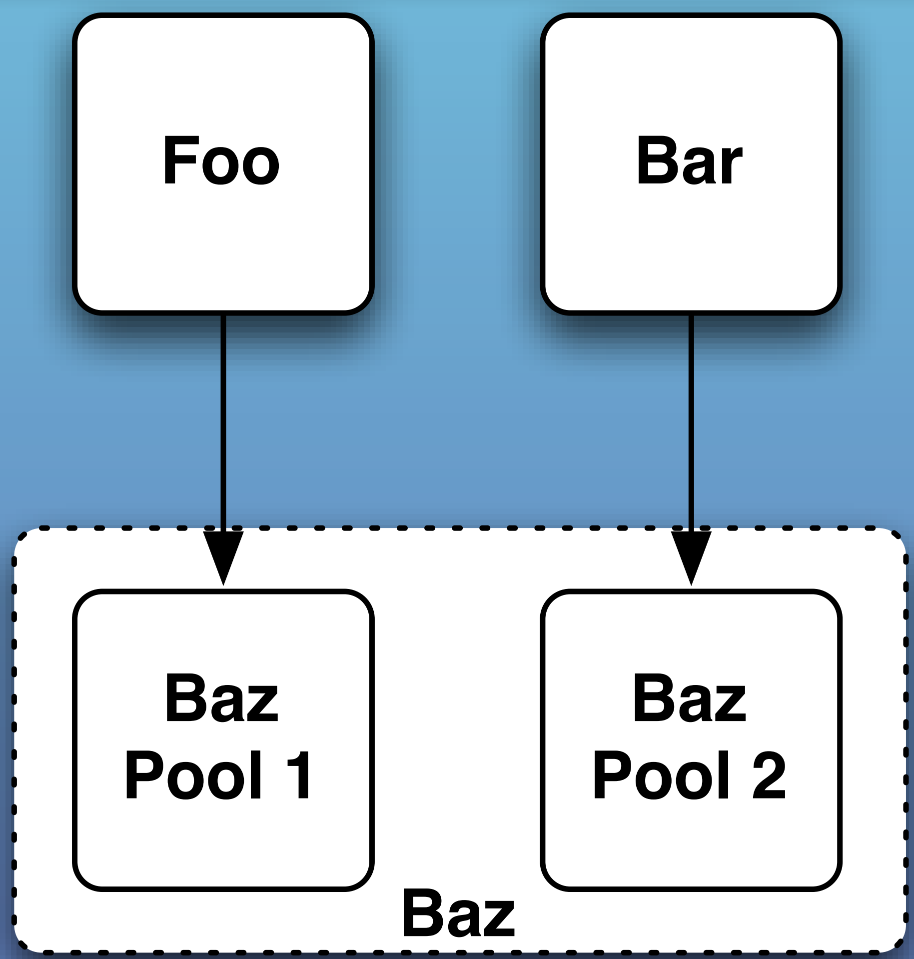
Foo and Bar are coupled by their shared use of Baz

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

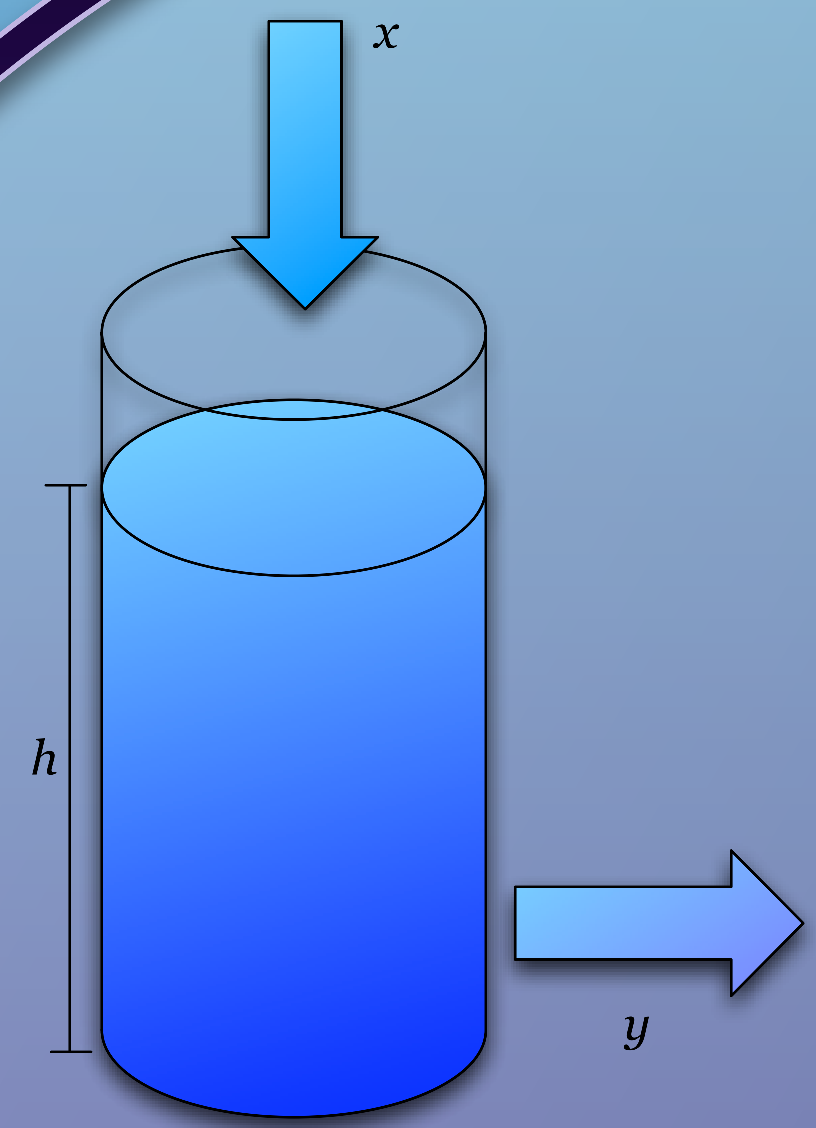
Steady State

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

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

How long is your shortest fuse?

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

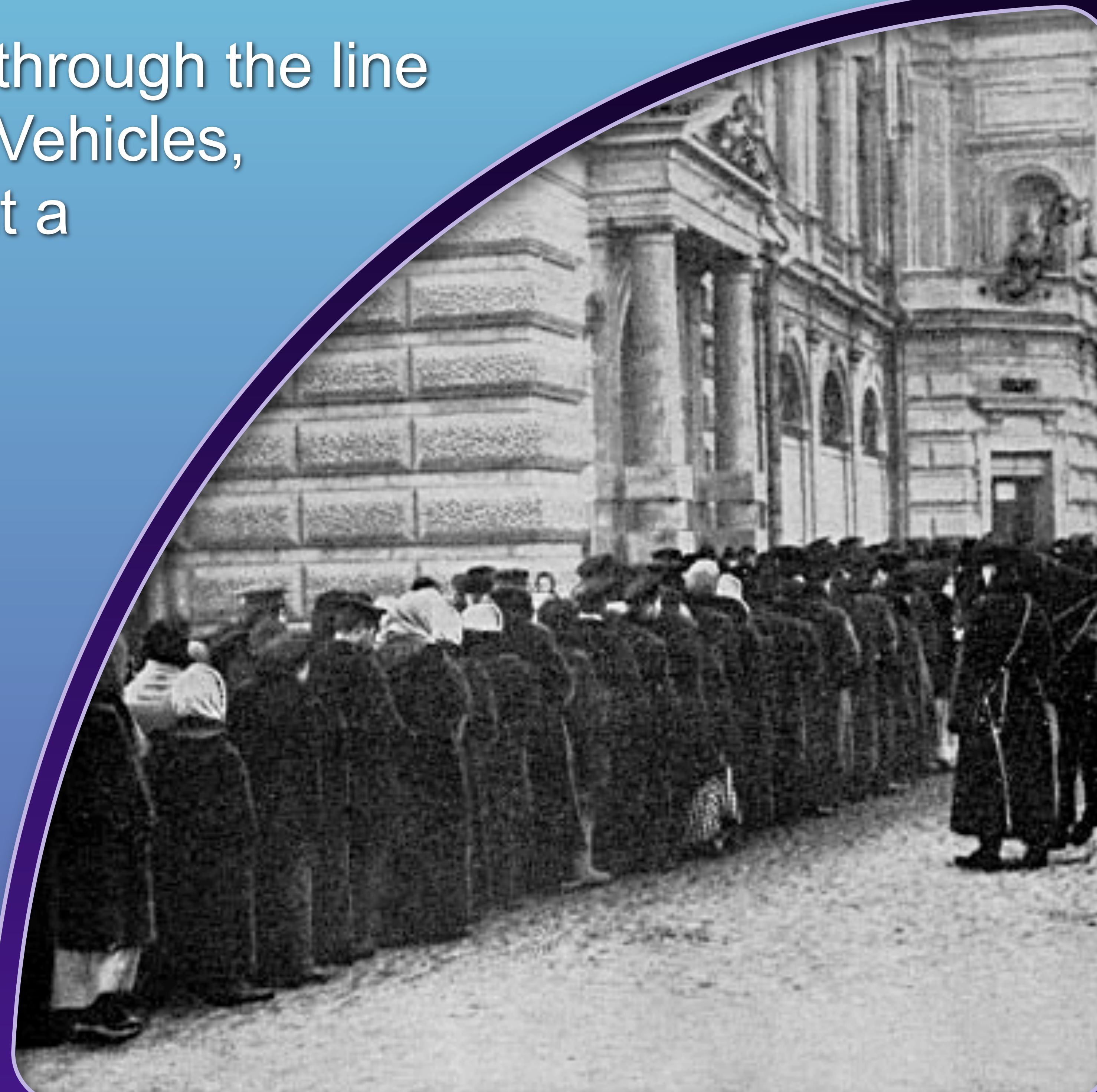
- Avoid fiddling
- Purge data with application logic
- Limit caching
- Roll the logs

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
- Helps preserve capacity when parts of system have already failed.

Remember This

- 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.

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?

"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.

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

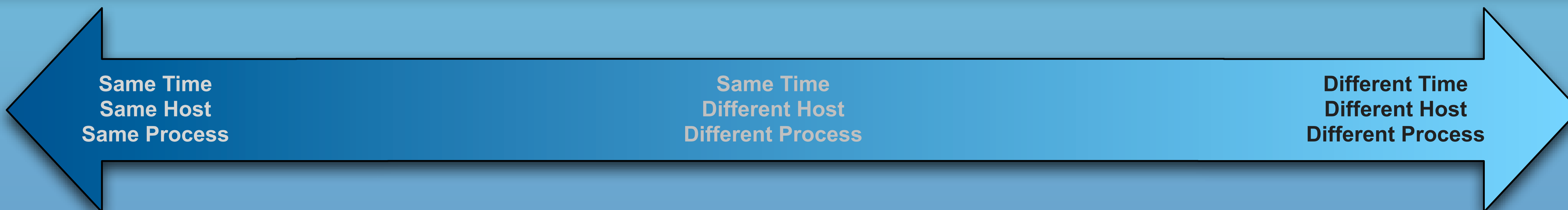
- Produce out-of-spec failures to ensure robustness of the caller
- Stress the caller
- Leverage shared harnesses across interfaces and projects, for common network-level errors
- Supplement, don't replace, other testing methods

Decoupling Middleware

Fire and forget.

- Most stability problems arise due to excessively tight coupling.
- Synchronous request-reply calls are particularly risky.
  - Ties up request-processing threads.
  - May not ever come back.

Spectrum of Coupling



In-Process Method Calls	Interprocess Communication	Remote Procedure Calls	Message-Oriented Middleware	Tuple Spaces
C Functions	Shared Memory	DCE RPC	MQ	JavaSpaces
Java Calls	Pipes	DCOM	Pub-Sub	TSpaces
Dynamic Libs	Semaphores	RMI	SMTP	GigaSpaces
	Windows Events	XML-RPC	SMS	
		HTTP		

Request-reply: logical simplicity, operational complexity

Message passing: logical complexity, operational simplicity

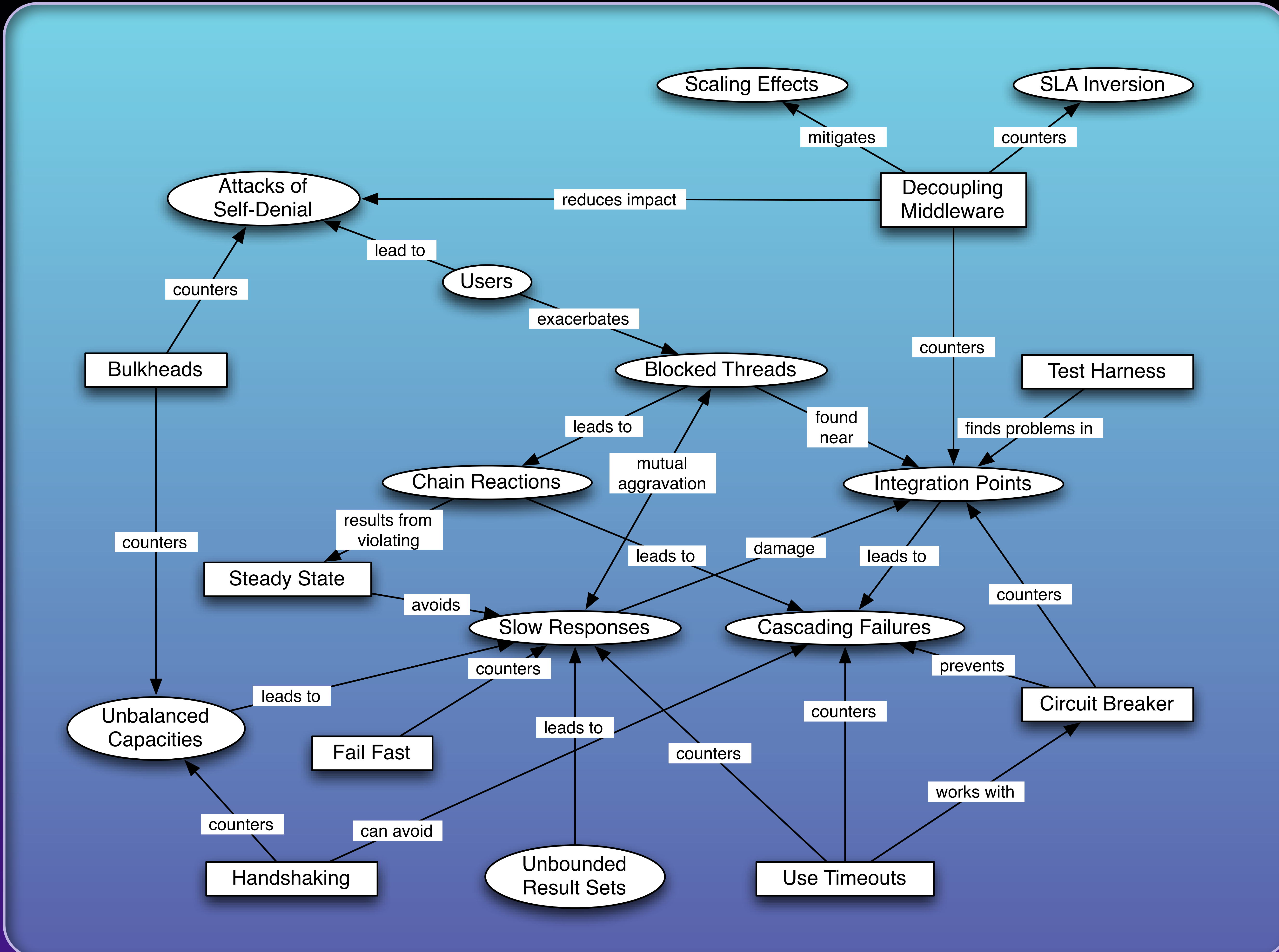
Tuple Spaces: logical complexity, operational complexity

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
damage
leads to
leads to
Steady State
counters
avoids
Slow Responses
Cascading Failures
counters
prevents
leads to
Circuit Breaker
Unbalanced Capacities
Fail Fast
leads to
counters
counters
works with
counters
can avoid
Handshaking
Unbounded Result Sets
Use Timeouts

Michael Nygard
mtnygard@thinkrelevance.com
@mtnygard