Designing Distributed Systems

Design for Performance and Reliability

Goal

We have looked at the services of a distributed operating system. Now we need to understand how to fit those services into a useful architecture!

Because there are so many different ways to build such systems, we will concentrate on large "fan-out" architectures which are typical for portals like Google, Netflix and Co.

Overview

- Design principles
- Caching and replication
- Architecture is key
- Architectural validation
- Large Fan Out-Architecture: caching, replication and asynchronous requests
- Large Fan-Out Architecture: Performance Optimizations
- Large Fan-Out Architecture: Fault-Tolerance

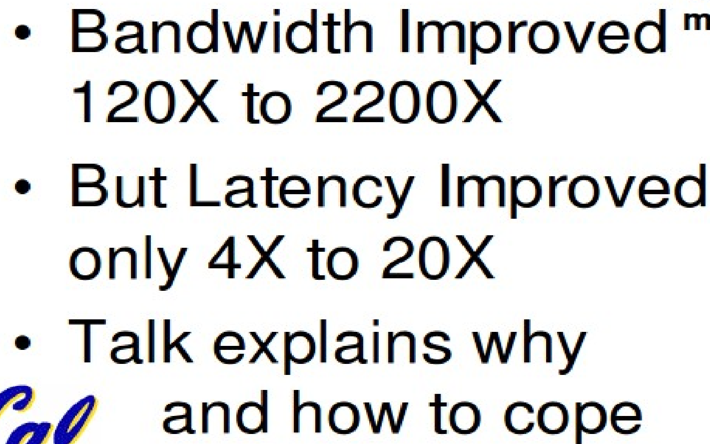
Design Principles of Distributed Systems

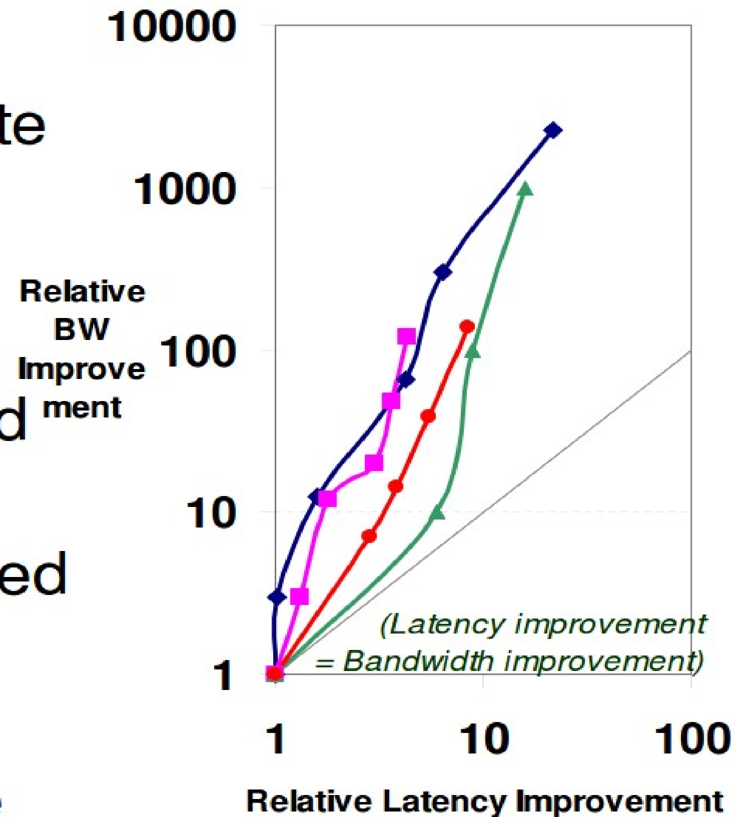
- Beware of Latency: check buffering and RTTs
- Locality: Carefully co-locate components that interact heavily.
- Sharing: Do not perform the same work twice or more times.
- Pooling: re-use expensive resources used in communication (e.g. connection/threadpool)
- Parallelize: Design your system in a way that lets you do things concurrently. Avoid unnecessary serialization.
- Consistency: Carefully evaluate the level of consistency that is needed with respect to caching and replication
- Caching and Replication: Use Prediction and bandwidth to reduce latency
- End-to-end-argument: Avoid heavy guarantees in lower-levels

Know Your No. 1 Enemy: Latency

Over last 20 to 25 years, for 4 disparate technologies, Latency Lags Bandwidth:

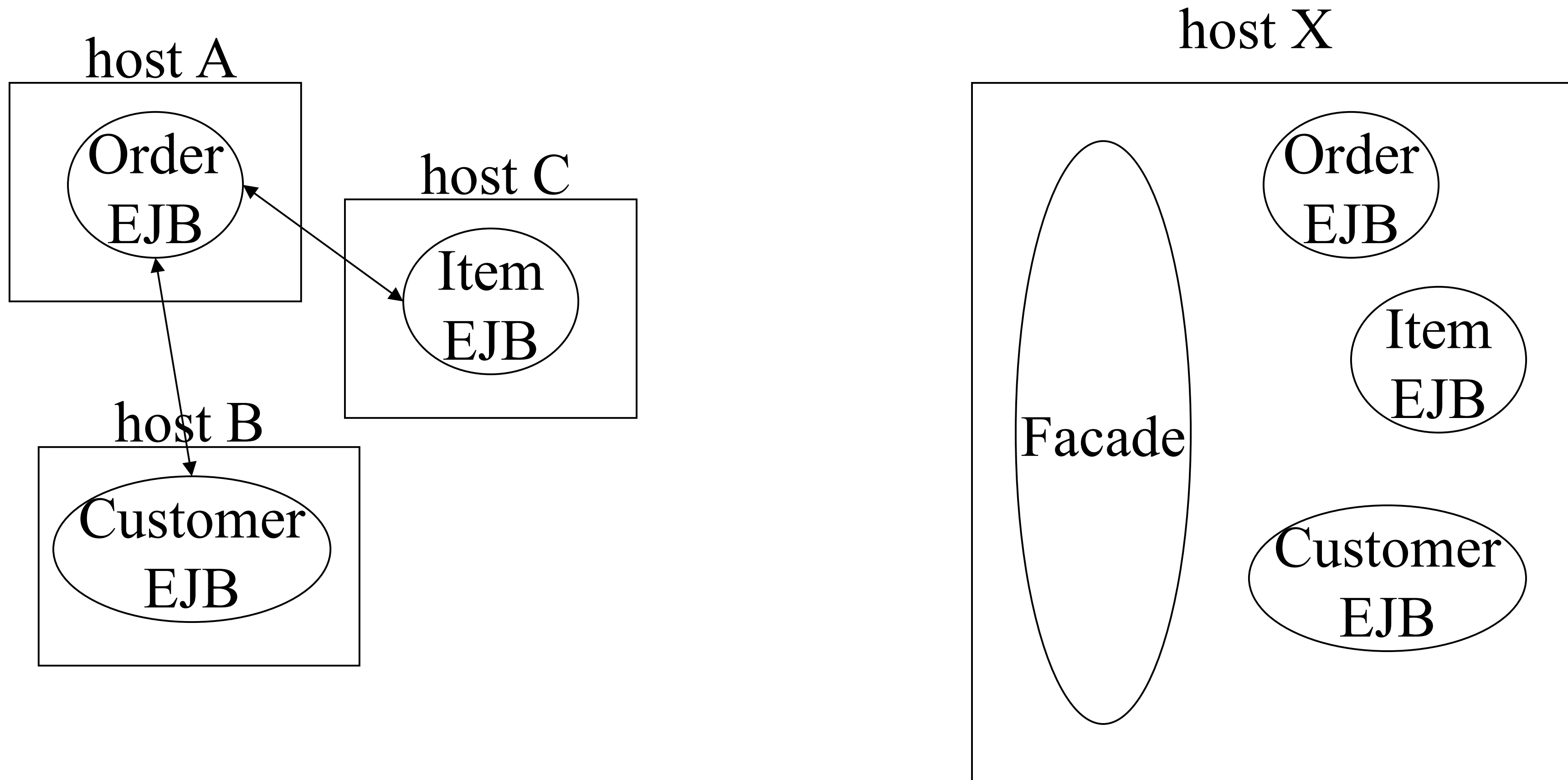
- Bandwidth Improved 120X to 2200X
- But Latency Improved only 4X to 20X
- Talk explains why and how to cope





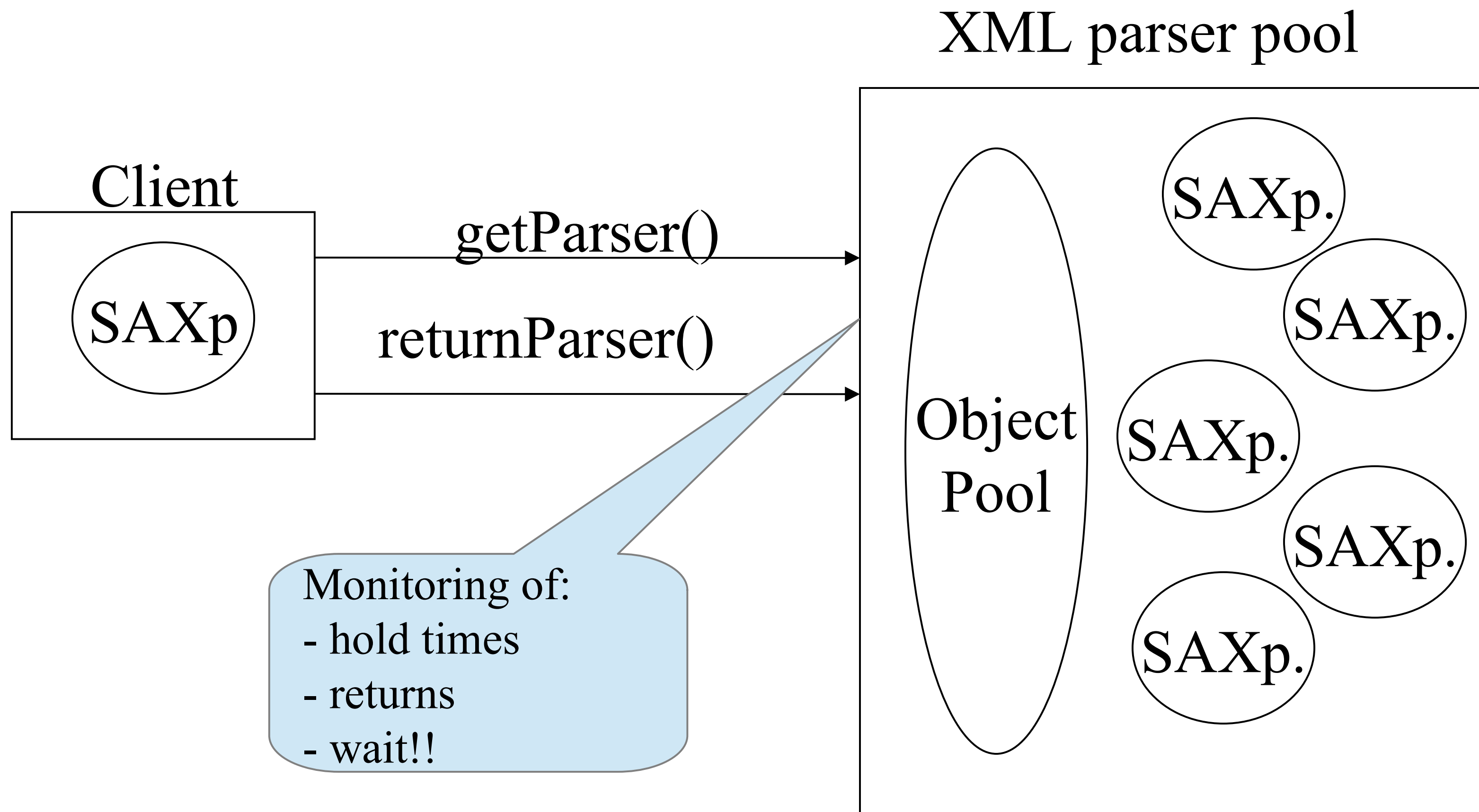
From: D.Patterson, Why Latency Lags Bandwidth, and What it Means to Computing. Latency hides everywhere: nic buffers, OS scheduling etc. Bundle and reduce requests!

Locality Matters!



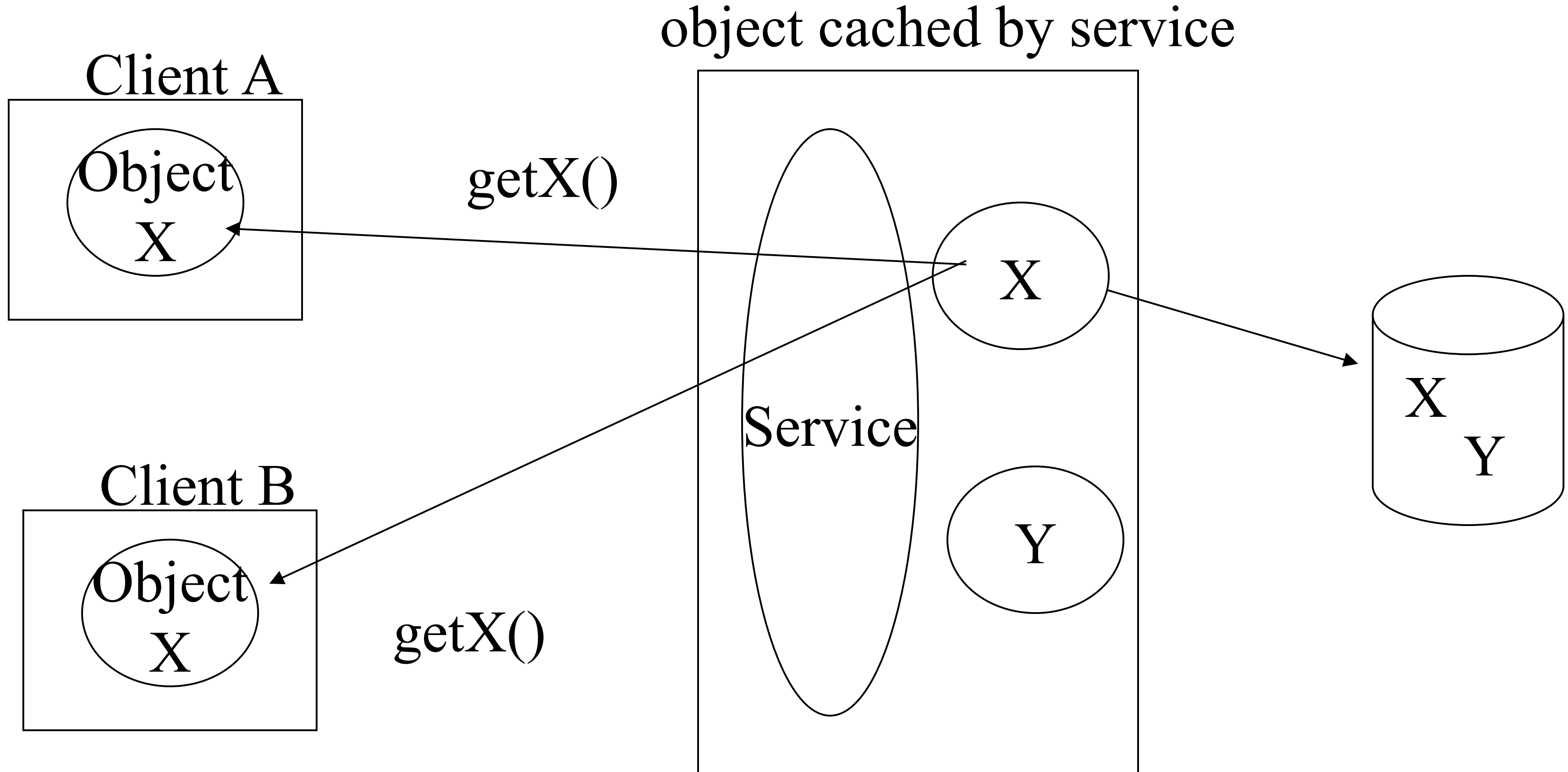
Example: Enterprise Java Beans introduced local interfaces as an addition in Release 2.0 – to respect the principle of locality which suggests to concentrate heavily interacting objects in one place. Even though there is NO functional difference between an item with a remote and a local interface. Do not distribute unnecessarily!

Sharing (1): Resources



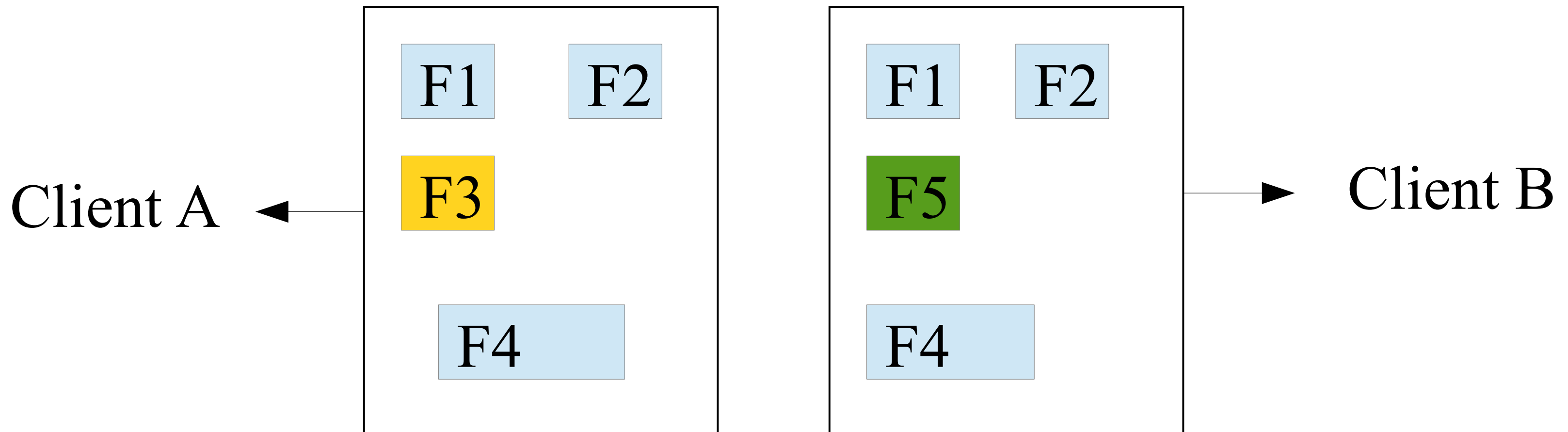
Pooling is useful in almost every case – even locally. But see what happens if you run XML over http and you create a new parser for every request – and there may be MANY requests per second. The profiler is your friend! It will show request counts and memory allocations.

Sharing (2): Data



Distributed applications without caching do not work. Try to minimize backend requests while still keeping application logic sane.

Sharing (3): Fragments



Cut your information in smaller fragments to find re-usable parts!

Connection Pooling

Connections = ((core_count * 2) + effective_spindle_count)
The calculation of pool size in order to avoid deadlock is a fairly simple resource allocation formula:

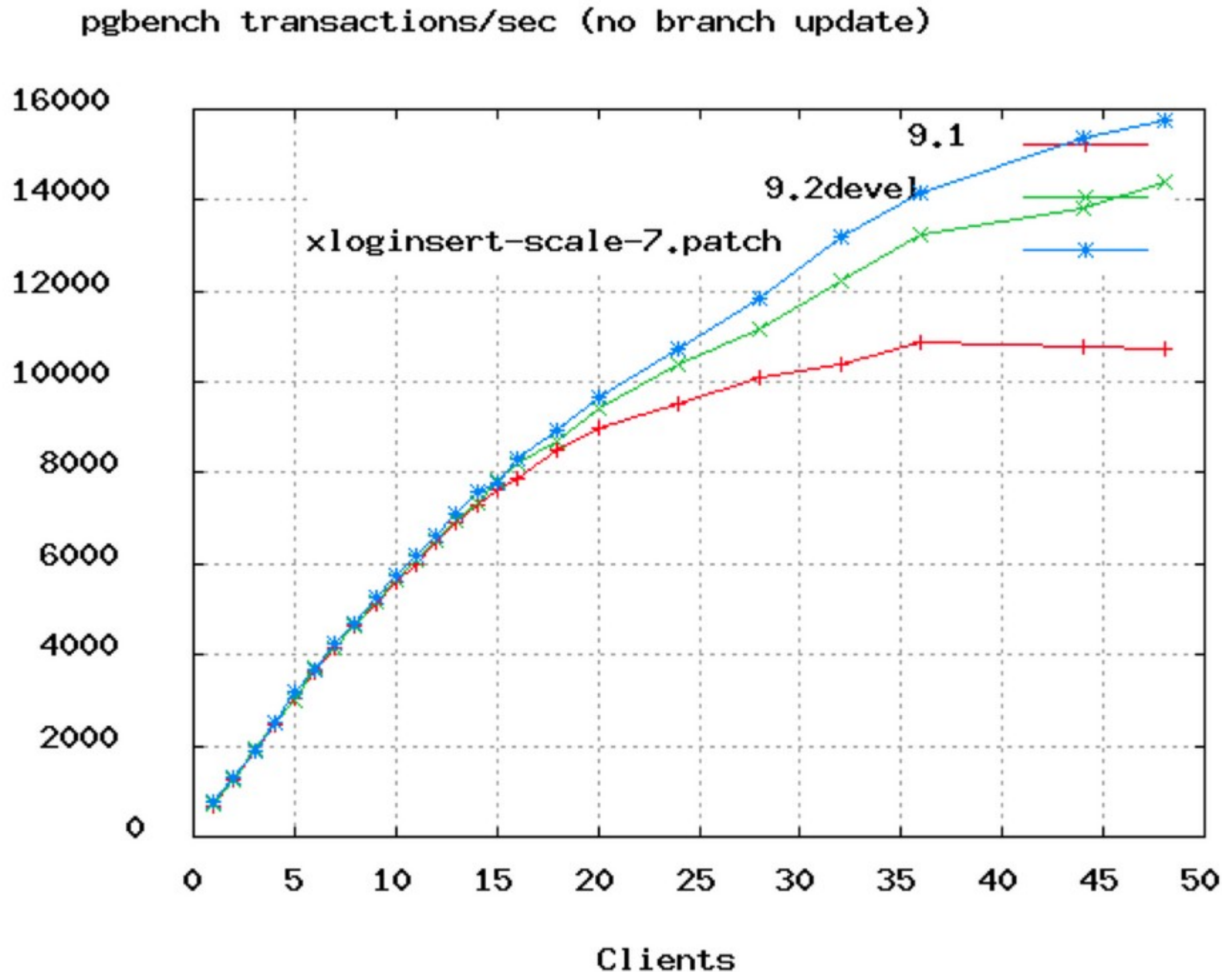
pool size = Tn x (Cm - 1) + 1

Where Tn is the maximum number of threads, and Cm is the maximum number of simultaneous connections held by a single thread.

There are a number of caveats behind those heuristics: match server CPU with database CPU, no unnecessary blocking anywhere, app threads not holding onto connections, measure wait time in pool carefully, check I/O rates with new hardware, understand what a “connection” to your storage really IS, watch core/thread ratio, etc.

https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing#connections--core_count--2--effective_spindle_count
(with good Oracle video)

More Connections Better?



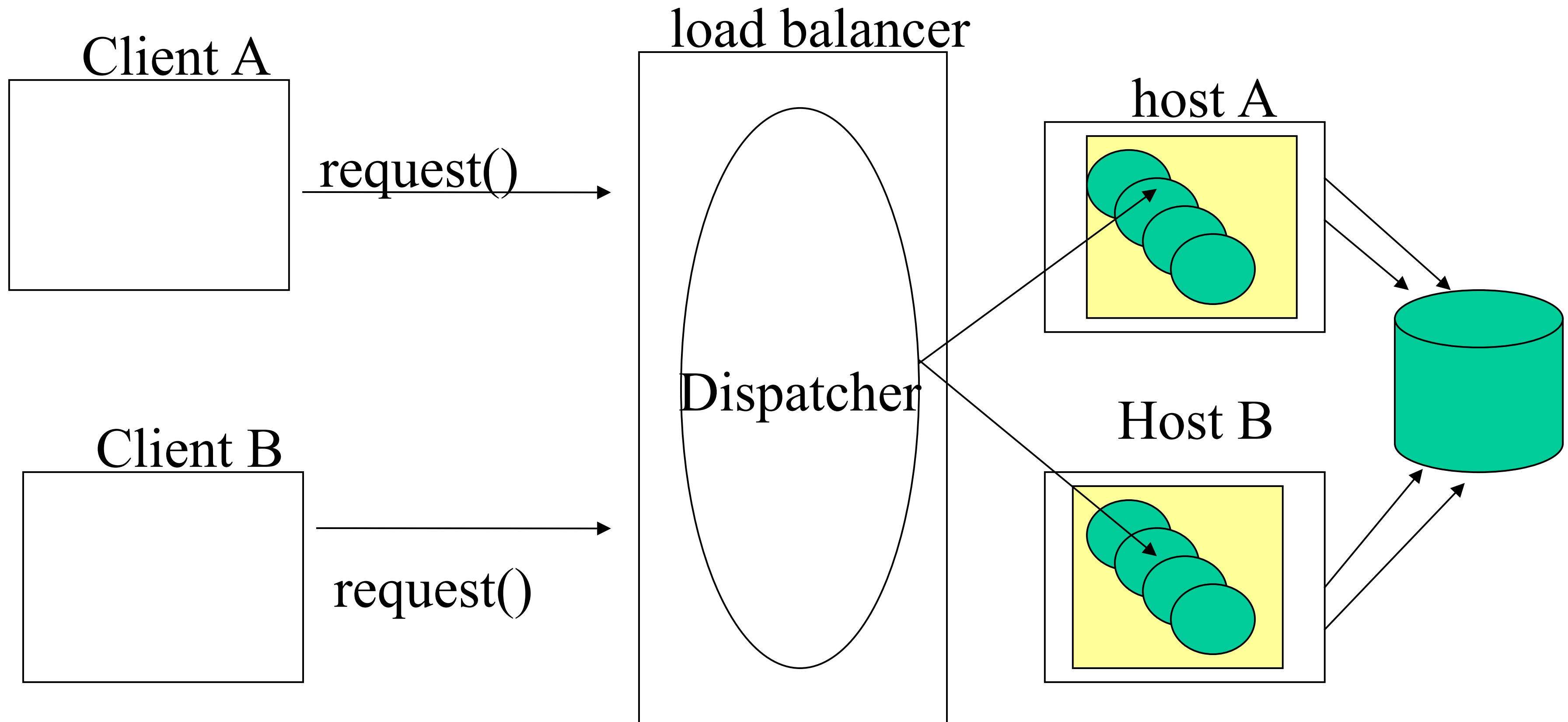
OLTP Performance - Concurrent Mid-Tier Connections,
https://www.dailymotion.com/video/x2s8uec

A Correct Analysis?

Number of web dynos	Max connection pool / dyno	Average RPS	Max RPS
1	200	292	350
2	200	482	595
4	100	565	1074
4	100	837	1297
8	50	1145	1403
8	50	1302	1908
16	30	1413	1841
16	30	1843	2559
16	30	2562	3717
20	25	2094	3160
24	20	2192	2895
24	20	2889	3533
30	16	2279	2924
36	14	2008	3070
36	14	3296	4014

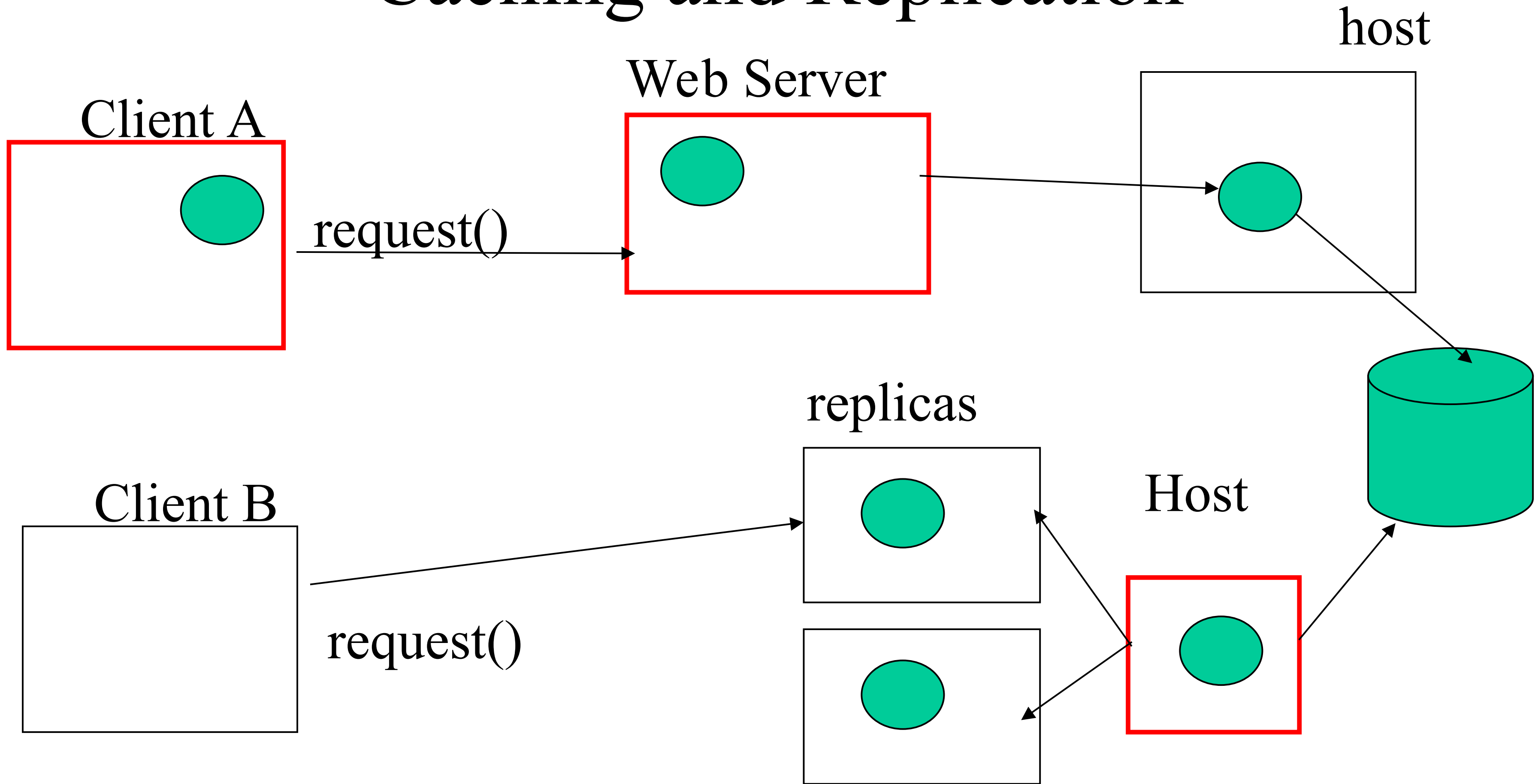
Serving Millions of Users in Real-Time with Node.js & Microservices [Case Study]
https://blog.risingstack.com/nodejs-microservices-scaling-case-study/

Parallelize: Scale Horizontally



A design that respects parallel processing scales much better: here every request can get handled by any thread running on any host. Avoid synchronization (wait) points e.g. in servlet engines or database connections

Caching and Replication



With caching the caching components bear the responsibility for data validity. In case of replication the data source is responsible to keep the replicas consistent and up-to-date. Make sure you reduce back-end requests!

End-to-End Argument

“The function in question can completely and correctly be implemented only with theknowledge and help of the application standing at the end points of the communicationsystem. Therefore, providing that questioned function as a feature of the communicationsystem itself is not possible. (Sometimes an incomplete version of the function providedby the communication system may be useful as a performance enhancement.)”

	Compensating Behavior
User/Developer	
Applic. Layer	Special Commands (e.g. Select for Update, Synchronized, Begin Transaction)
Intermediate Layer	Compiler/Languages: STM, Memory Models
Base Layer	CPU cache coherence, DB isolation levels, Realtime-Streaming etc.

END-TO-END ARGUMENTS IN SYSTEM DESIGN, J.H. Saltzer, D.P. Reed and D.D. Clark*M.I.T. Laboratory for Computer Science, http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf. There are two forms of the EtEA: When first invented, it focussed on guarantees, that could only provided at the endpoints of communications. Later it became clear, that in many cases it is easier, to have less guarantees on lower levels and to provide more powerful ones at higher levels. This leads to extremely efficient and flexible lower layers. But there are exceptions!

Design-Methodology

- „back-of-the-envelope calculations“ (J.Dean, Google)
- Decide geographical distribution/replication
- Data segregation. Single- or multi-tenancy model, possible partitioning along functions, data or customers
- Divide business requirements into REST-like services
- Define SLAs for services. Availability, latency, throughput, consistency, durability must be defined.
- Define Security context with IAAA (identity, authentication, authorization, and audit) and perform risk analysis.
- Define complete monitoring and logging
- Define deployment and release changes, testing approaches. You can use fault-tolerant features for maintenance.

Partially after: M.Cavage, There's Just No Getting around It: You're Building a Distributed System, ACM Queue, April. 2013

102 Uncomfortable Questions...

Server sizing
1. How many application servers are needed to support the customer base?
2. What is the optimal ratio of users to web servers?
4. What is the maximum number of users per server?
5. What is the maximum number of transactions per server?

Server tuning and optimization
6. Which specific hardware configurations provide the best performance?

Capacity planning
9. What is the current production server capability?

Browser/user profile issues
21. What do the users do? (These are business process definitions.)
22. How fast do the users do it? What are the transaction rates of each business process?
23. When do they do it? What time of day are most users using it?
24. What major geographic locations are they doing it from?

Web server issues
69. How many connections can the server handle?
70. How many open file descriptors or handles is the server configured to handle?
71. How many processes or threads is the server configured to handle?
72. Does it release and renew threads and connections correctly?
73. How large is the server's listen queue?
74. What is the server's "page push" capacity?
75. What type of caching is done?

…................

Partially after: Todd DeCapua, 102 performance engineering questions every software development team should ask, http://techbeacon.com/102-performance-engineering-questions-every-software-development-team-should-ask

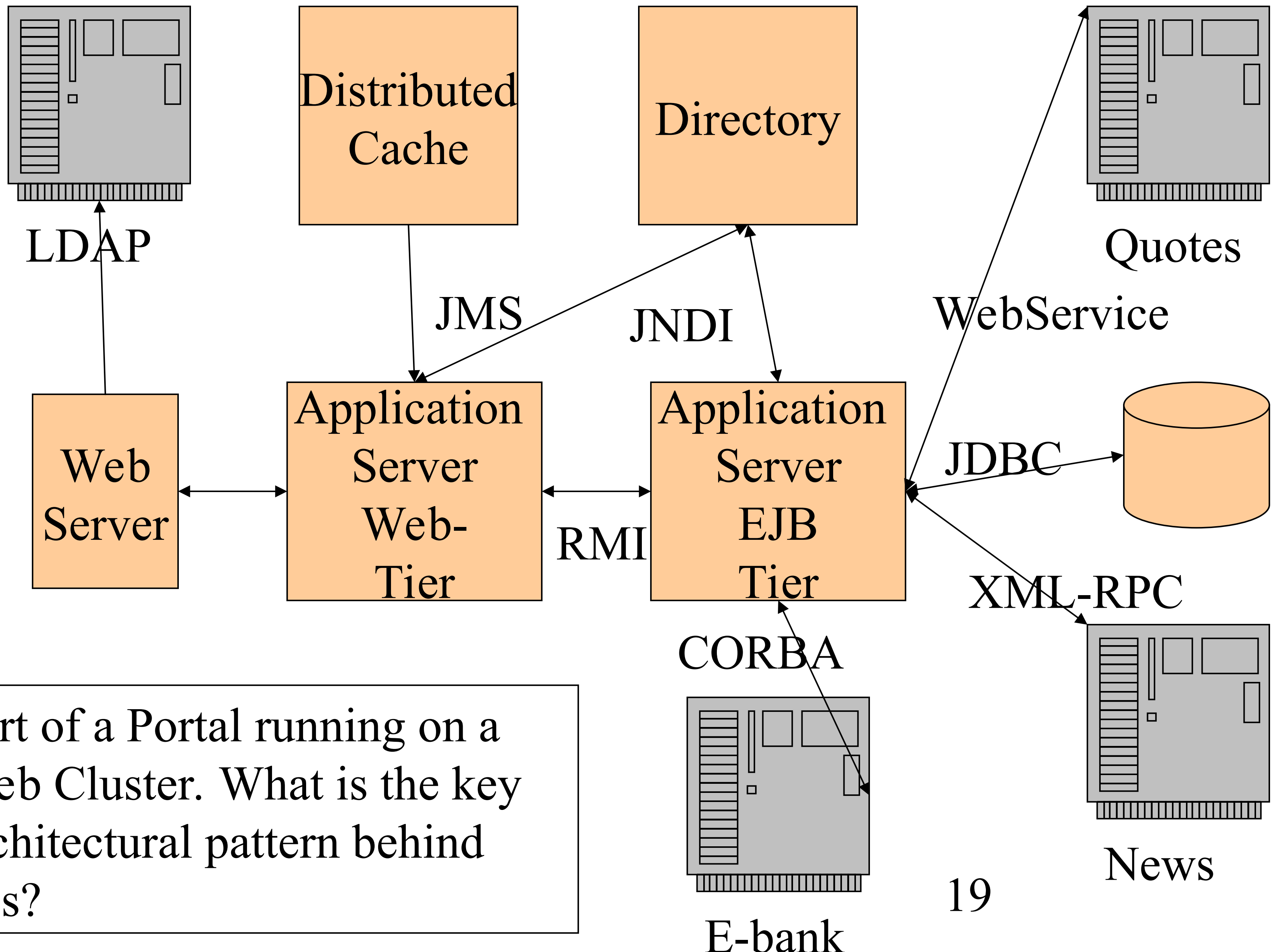
Architecture is key

Sooner or later a distributed computing project will need to define the following artefacts:

- Information Architecture
- Distribution Architecture
- System Architecture
- Physical Architecture
- Architectural Validation

This is a question of “pay me now or pay me later” as you will see in the following sections

Example: Distributed Technologies in a Portal



Part of a Portal running on a Web Cluster. What is the key architectural pattern behind this?

Example: Inpol

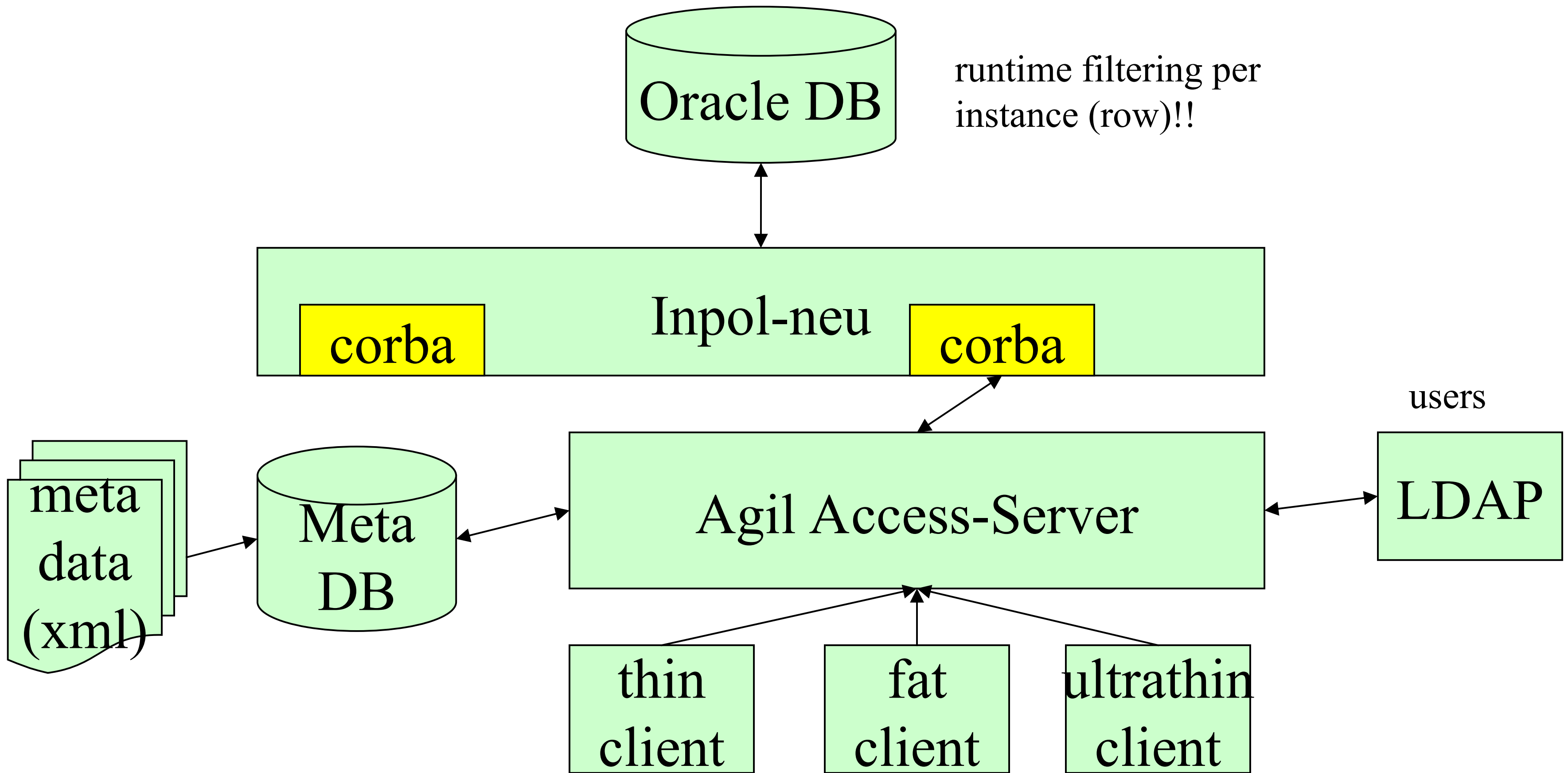
The German state police wanted a new IT-System „Inpol-neu“.

The system should store all kinds of crime-related data and allow fancy queries in realtime. The owners are the states of the BRD.

The federal structure of the German police played a major role in this project...

Large projects have their own rules and problems, besides the fact that most of them are somehow distributed designs. If you want to survive those projects, read „Death march projects“ by. E.Yourdon.

The large distributed project gone foul



A fairly standard design. The bottleneck was supposedly the handling of access rights very late in the DB using stored SQL. Every user used the same table-schema but runtime DB access control filtered out things that a user should not see. User rights were additionally complicated by the multi-party character of the project: Every state in the BRD defined rules and variables in a special way.

Key ingredients for desaster

- planned costs >100 Mio. (attracts many „experts", highly political)
- high daily burn-rate (forces rash decisions)
- overloaded with features
- New technology driven at the limit (XML meta-data approach to cover serious differences within customers)
- High scalability requirements: lots of data to be handled, high performance, security etc. Especially bad when coupled with latest (immature) technology
- Multi-party support: means too many different requirements and tedious project handling.
- Lots of paper pushing while the core assumptions of the whole system stay unchallenged (in this case the very late and complex access control in the DB)

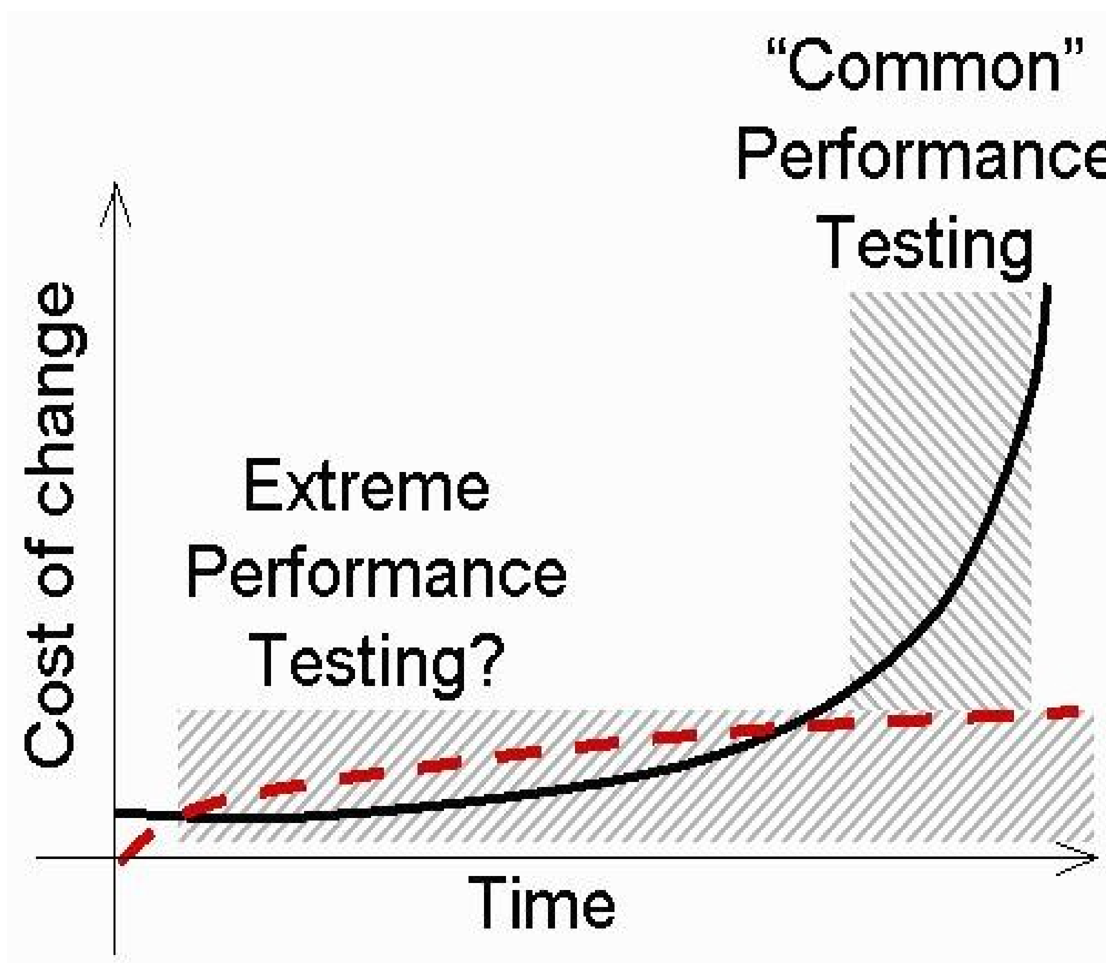
Some of these cannot be prevented easily. At least an Extreme Testing Approch could be followed to discover architectural flaws early. But it seems that it is the structure of those large-scale projects that makes these simple and effective countermeasures impossible. Common sense and sanity seem to be missing in those projects and you can feel it. Use early architectural validation to avoid those problems.

Architectural Validation

- how does the architecture handle change? In the inpol-case deleting data required changes in base-algorithms on all levels because a re-arrangement/split of data was necessary.
- Where are the main bottlenecks in the system?
- Is horizontal scaling possible (if not, specify approach for failover). Remember the facade (component) pattern along whose lines machines can be split.
- Is only vertical scalability possible? How far?

architectural validation is the phase in a project where these questions are answered. Don‘t start a big project before you have those answers. (Of course, if already a big team has been assembled you may not get the time to do the basic validation – the results are well known

Use of extreme testing to reduce architectural risk



Source: Ted Osborne from Empirix

Large-Scale Fan-out Architectures

Early fan-out Architectures: Portal project

An enterprise portal combines different applications and data sources within an enterprise or across enterprises into a consistent and convenient view to clients.

Portals use many different kinds of infrastructure, protocols and services and are therefore distributed applications.

Portal projects are also notorious for their performance and reliability problems.

The examples shown are from a real portal project at a large bank. They show an early version of what is today called “fan-out architecture”. The project was using massively parallel sub-request processing – at that time not allowed for applications in a JEE container...

Common: customize, filter, contact etc.

Dynamic and personalized homepage

Welcome Mrs. X,
[...] would like to point you to our
New Instrument X that fits nicely
To your current investment strategy.

Portfolio: Siemen[...]
Swisskom, Esso,

Messages: 3 new
From foo: hi Mrs. Rich

Common: Banner

Quotes: UBS 500,
ARBA 200

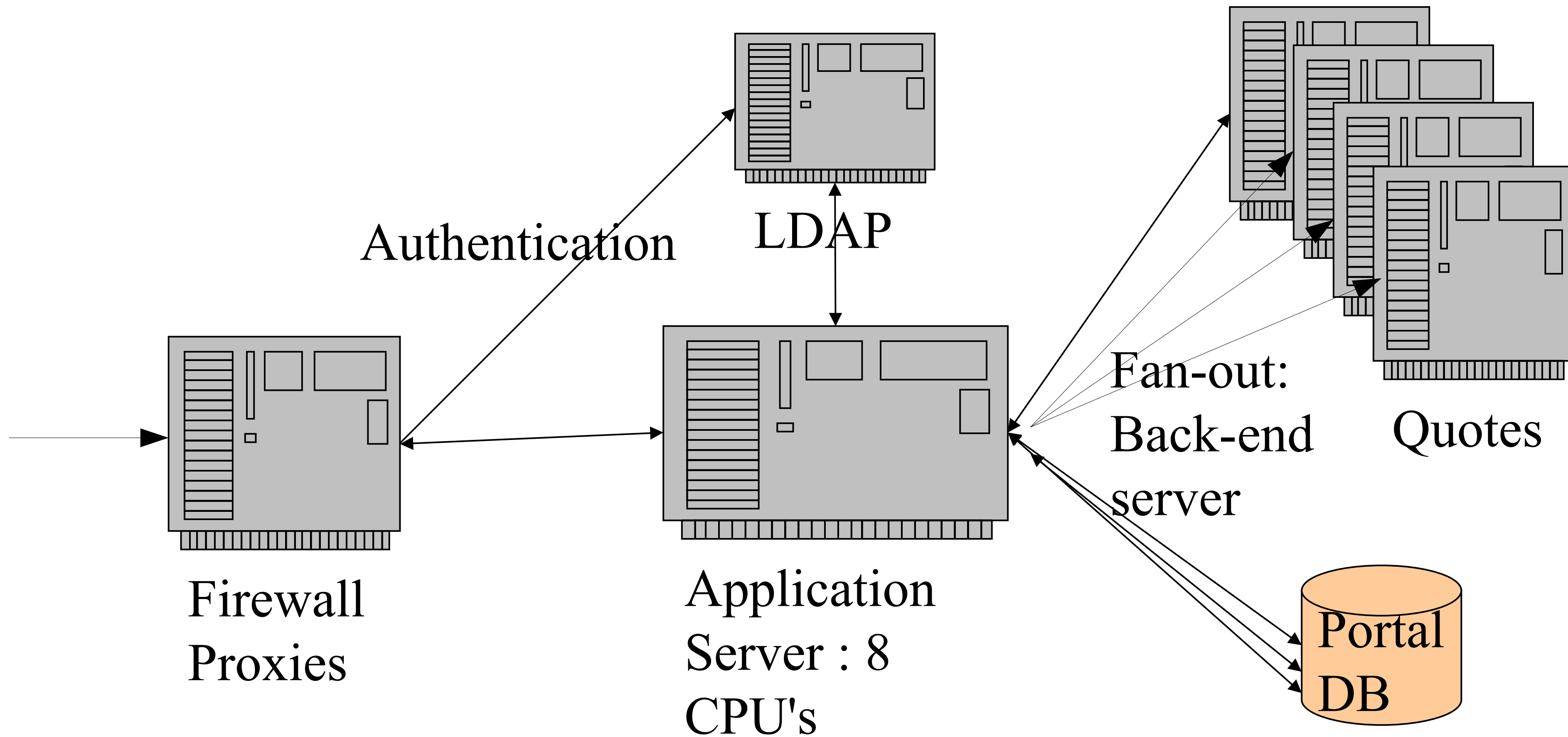
News: IBM invests in company Y

Links: myweather.com,
bank glossary etc.

Charts: Sony

Research: asian equity update

Physical Portal Architecture



The application process is running on a JVM on the Application Server. The AS aggregated and integrated various information for homepage construction. Initial load-time for homepage: 3 minutes! Frequent stalls and crashes!

Exercise: Explain Slow/Fragile Homepage!

Performance Problems:
-
-
-
-

Reliability/Availability Problems:
-
-
-
-

Give some possible reasons for the slow and erratic behavior of the portal software!

Portal Problem Analysis

- GUI design: Long time with empty page
- Design: No System Architecture Diagram!
- Performance: Very slow construction of home-page
- Reliability: Frequent stalls and crashes of the application
- Throughput: 10 users max. with top-notch hardware!
- Team: Little understanding of performance or architecture

GUI-Design: General Performance Hints

- Show something quickly!
- Compress files
- Order downloads for optimal rendering speed
- Use sub-domains or Http2.0 for high-speed download
- Bundle css/js content
- Optimize images
- Exploit http caching options and infrastructure!
- Use expires header
- Much more

In many cases, client side optimizations are “low hanging fruits”, because they have little impact on server side code. You can find lots of information from Jakob Schröter or Steve Souder on this topic.

GUI-Design: Special Performance Hints

- Design your GUI for balanced and constant load requests
- Split expensive functions across sub-menues
- Allow for asynchronous processing
- Use kill-switches in case of overload and tell users
- Offer different qualities of information depending on system load
- Split business-requests into several requests to improve constant load
- Use “Hejunka” at the client side too..
- Investigate streaming with http2 and http3
- Measure the cost of powerful frontend frameworks carefully...

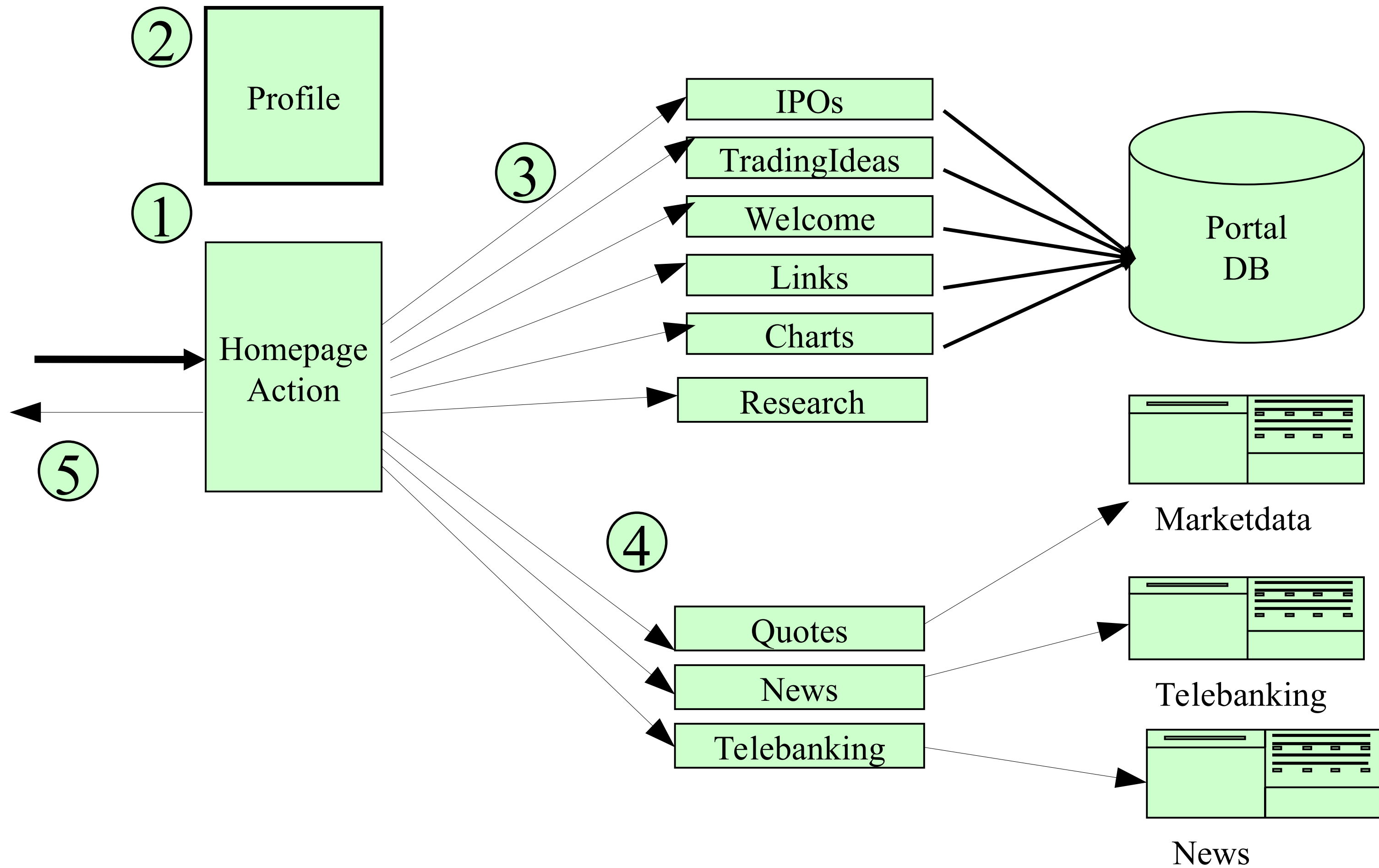
These measures depend on your application. Ultra-large scale sites design GUI functionality around the infrastructure and measure request percentiles carefully.

System Architecture Diagram

The system architecture captures the main objects and their interactions. It describes the processing that happens within the system.

Important: Try to capture the essence of the system architecture in one diagram. It serves as a communication tool between developers and to other groups (management) as well. It also makes system inherent problems more visible.

PortalPage Request Flow and Assembly



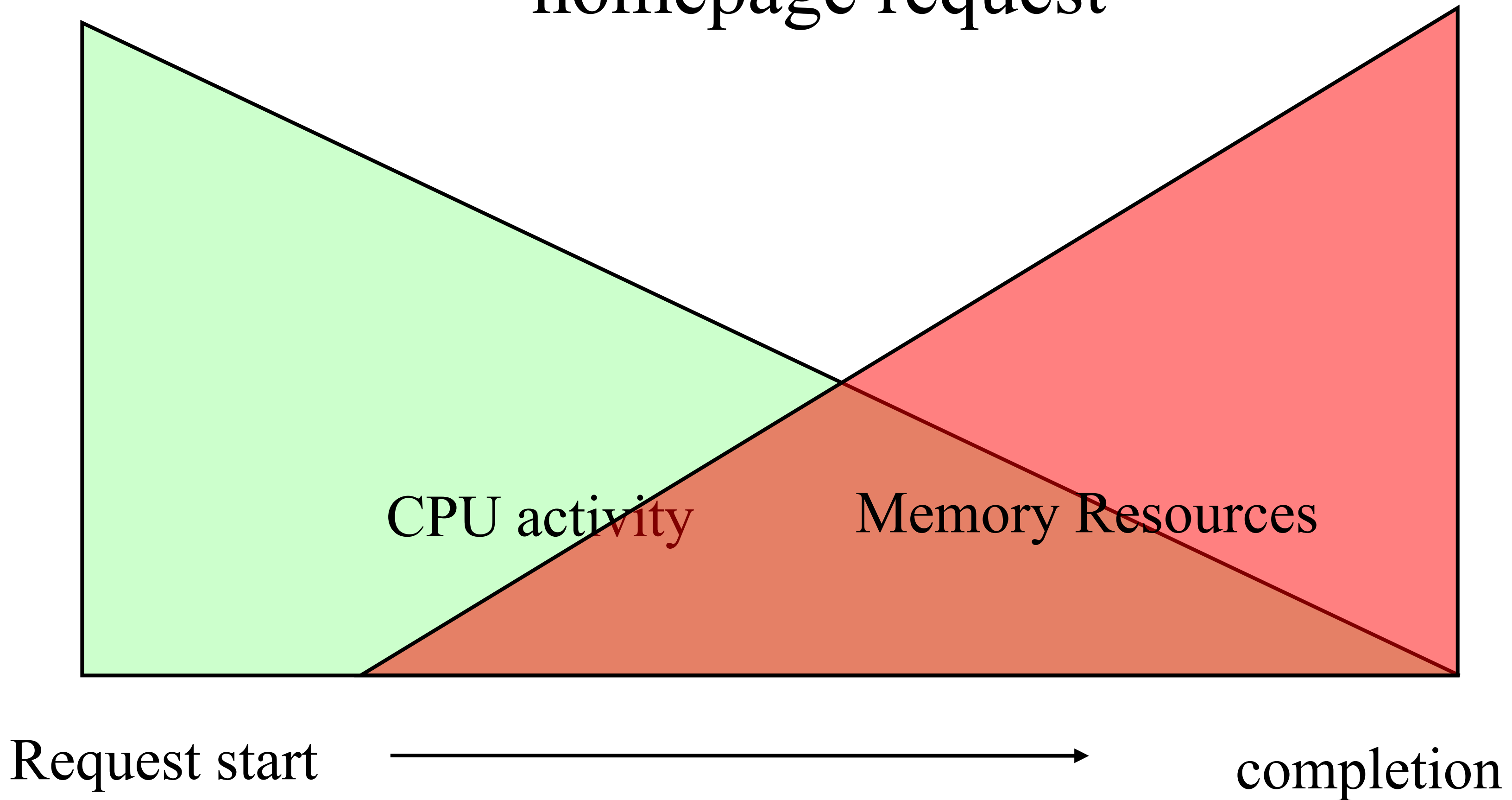
1. receive request (thread per connection model) 2. lookup user profile, 3. sequentially contact DB-tables and back-end server, 4. aggregate results and forward to template engine for html construction

Lessons Learned from SAD

- The overall request time is the sum of all individual calls
- Each delay in an individual call adds directly to the runtime
- Possible back-end server problems (overload etc.) have a disastrous effect on request time
- Long timeout settings would kill response times
- With many concurrent home-page requests, even small improvements to sub-requests count!

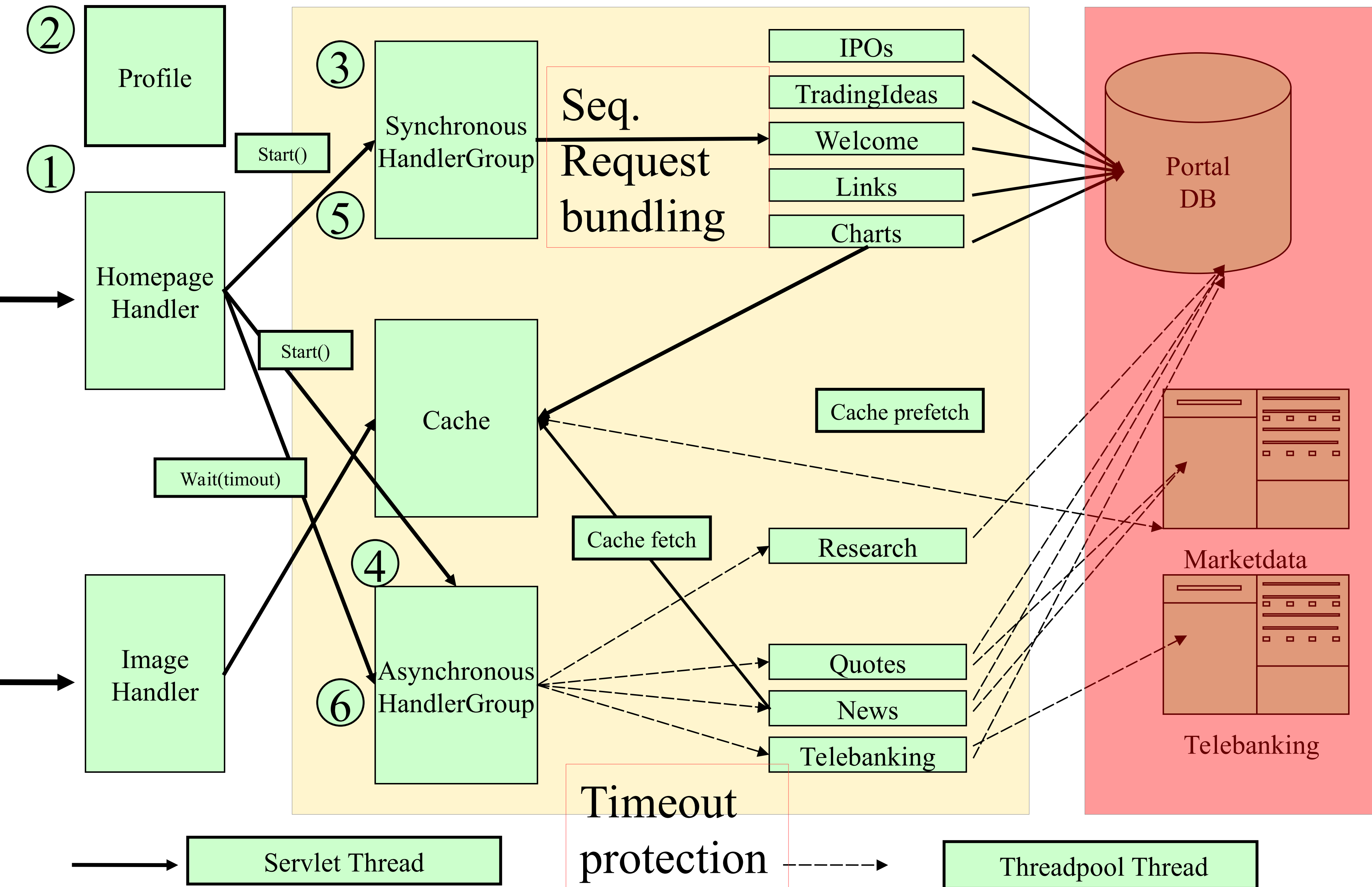
Without a central architecture diagram, developers cannot understand their individual impact on overall performance and throughput! JEE at that time did not allow applications to create their own threads!

Java VM memory consumption during complex homepage request



At that time, Java GC was having problems with extremely large heaps caused by processing stalls in the application (e.g. due to back-end server problems). GC freqently could not recover and crashed.

Re-Design (parallelized): PortalPage Request Flow and Assembly



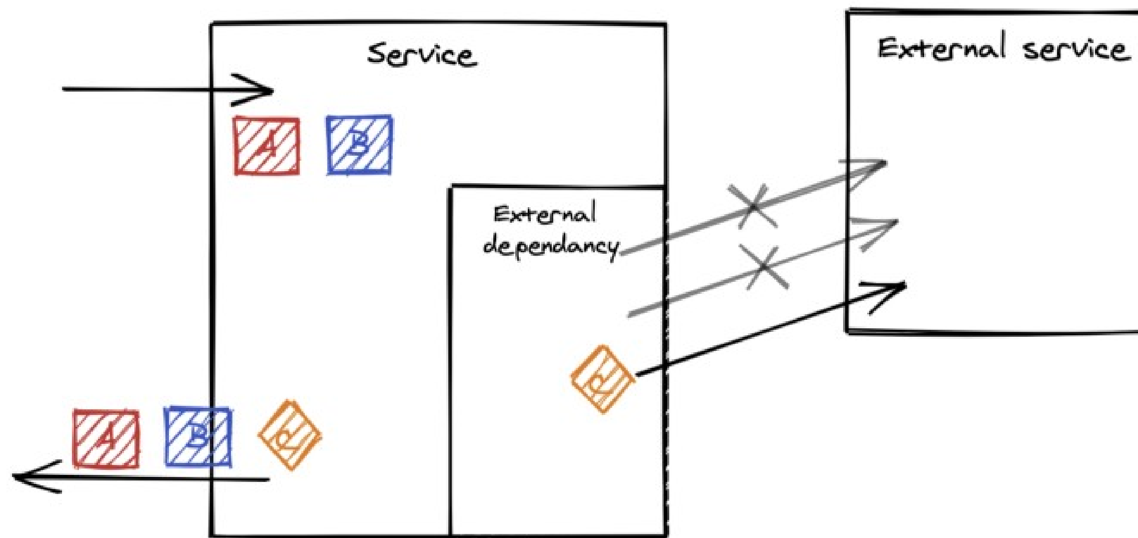
Simplified SAD

Frontend (Rendering Problems)	Fan-out (Performance and throughput Problems)	Dependencies (Reliability Problems)

Parallel Calls (fan-out)

- The overall request time is the time of the slowest sub-request
- Each delay in an individual call adds directly to the runtime
- Long timeout settings would kill response times
- Add short timeouts to back-end server calls: prevents piling up of threads on unavailable servers and keeps memory heap small.
- Running extremely short requests, e.g. to the portal DB, within their own thread, can be counter-productive: No use, if back-end server calls take much longer anyway. Solution: Bundling of short sequential requests to save on thread overhead! (batching)
- In case of problems with one sub-call: Return an error but do not block the whole request waiting (fail fast, use outdated data, have a fallback)
- Prevent all threads getting stuck on one dysfunctional sub-call (bulkhead)
- Temporarily close dead connections (circuit-breaker)

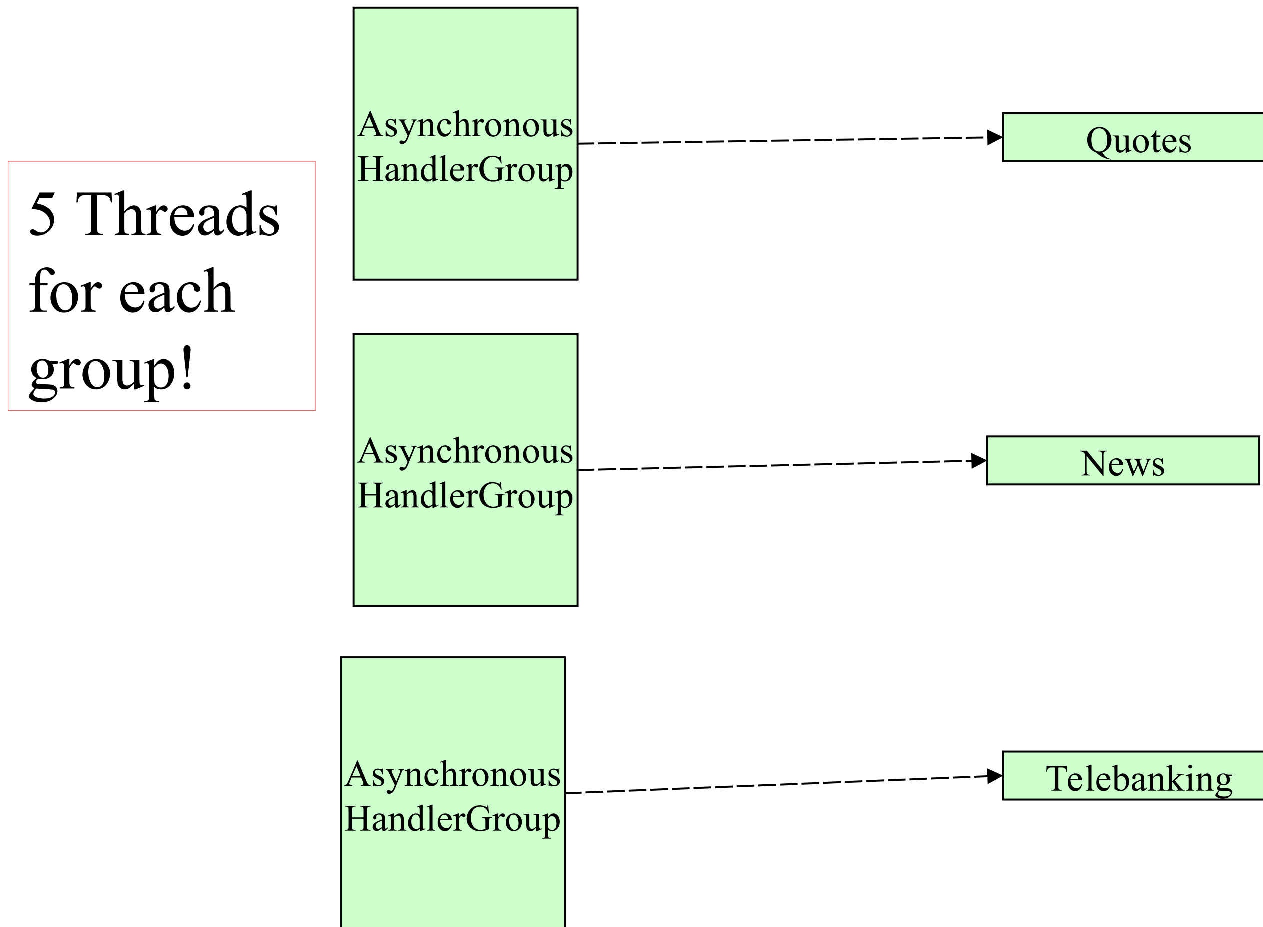
Engineering For Failure: retries



From: Boris Cherkasky, https://medium.com/riskified-technology/engineering-for-failure-f73bc8bc2e87

BUT: if services decide on too many retries we do a DOS attack on our backends!!

Bulkhead Pattern



A back-end server that is unavailable will only cause 5 threads to wait for time-out! Go and find more patterns in the Hystrix documentation or “Site Reliability Engineering” by Google.

Reliability Problems from Dependencies

- Hanging requests keep allocated resources busy and cause severe GC which makes system load worse
- Even short timeouts allowed threads to pile-up on dead servers (I did not think about splitting async. Thread groups into what is known today as "bulkheads"
- The portal suffered from failing back-end servers a lot
- Do not let the hompage action handler wait long for outstandung sub-requests: Today: "Fail-fast" pattern

- Request time was down to 17 seconds but that was still too much...

The solution found was a little different to what Netflix does today. It is based more on the concept of a group-communication or failure-detection layer below the application.

Missing: Distribution Architecture

Data Type	Source	Protocol	Port	Avg. Resp.	Worst Resp.	Down-times	Max Conn.	Load-bal.	Security	Contact/SLA
News	hostX	http/xml	3000	100ms	6 sec.	17.00-17.20	100	client	plain	Mrs.X/News-SLA
Research	hostY	RMI	80	50ms	500ms .	0.00-1.00	50	server	SSL	Mr.Y/res-SLA
Quotes	hostZ	Corba/IDL	8080	40ms	25 sec.	Ev.Monday 1 hour	30	Client	plain	Mr.Z/quotes-SLA
Personal	hostW	JDBC	7000	30ms	70ms	2 times Per week	2000	server	Oracle JDBC dr.	Mrs.W/pers-SLA

Not shown but also important: bandwidth available, number of requests planned, distance to device, availability numbers

Results from the Distribution Architecture

- several back-end servers show either huge latencies and/or huge variations in response times
- these delays determine the slow home page construction times
- getting to those servers for every request ist nearly impossible
- the variation in response time from dependencies causes instabilities in the portal application

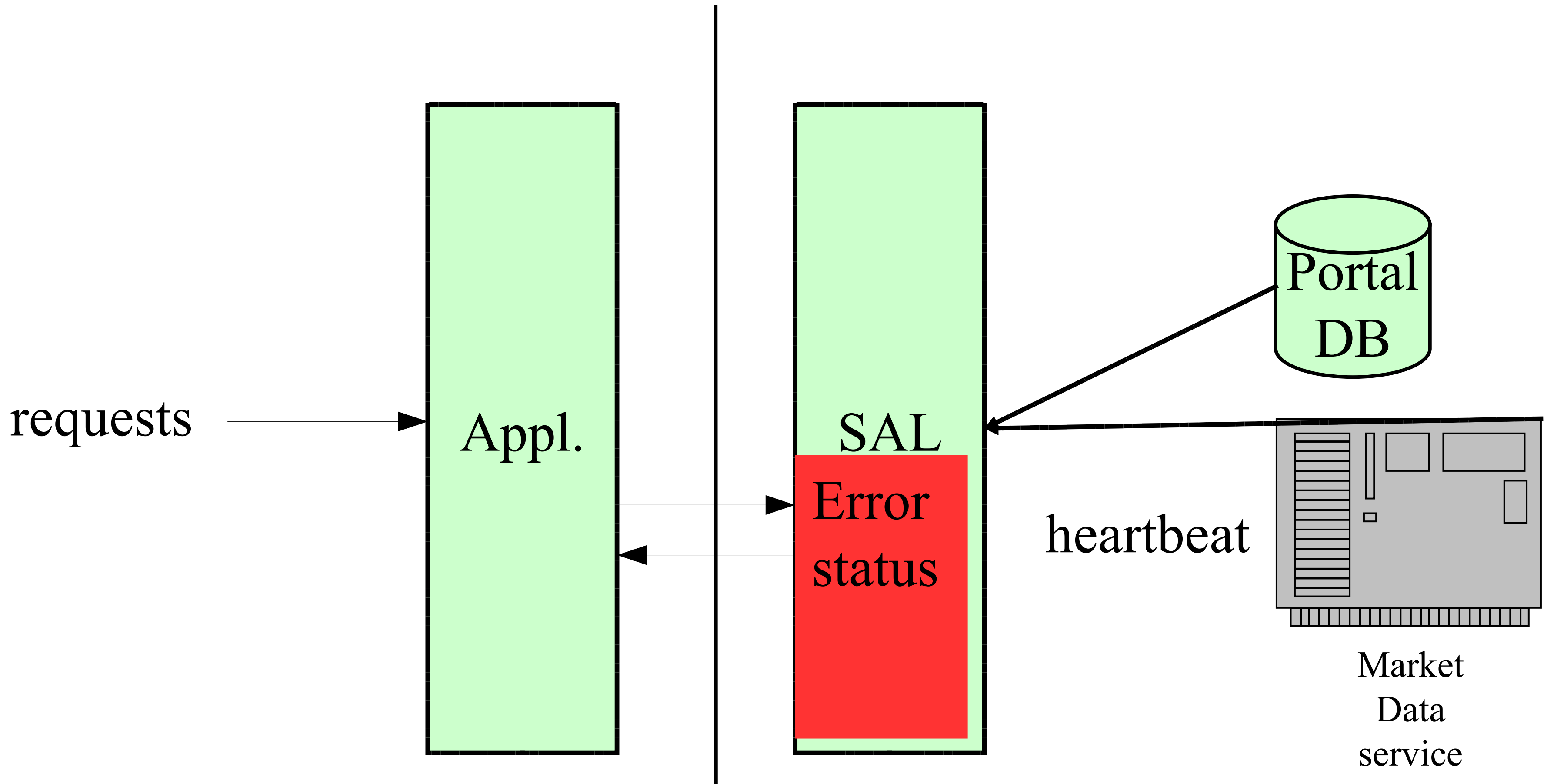
Possible cures for those problems could be client-side load-balancing as well as more and faster back-end servers. But MOST IMPORTANT: the fan-out layer needs to use reliability patterns! A famous framework ist e.g. Hystrix by Netflix

Data Aggregation: What, Where and How?

- Sources, Protocols, Schemata
- Data rates
- Response times (average, over day, downtimes)
- QOS (e.g. Realtime quotes)
- Push/Pull
- Security (encryption etc.)

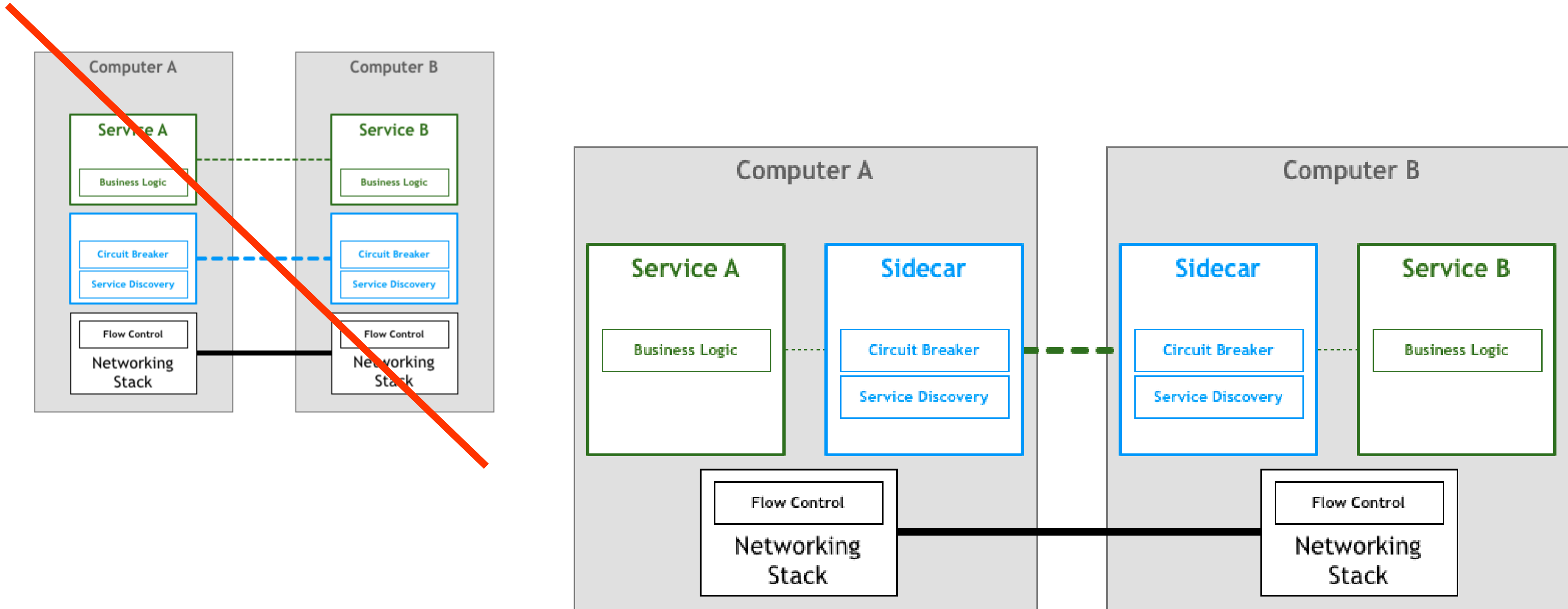
- Handle interface changes
- Disable broken connections
- Add new sources
- Poll and re-enable sources
- Keep statistics on sources

A Service Access Layer (SAL)



The service access layer tracks backend system connections and prevents request from blocking on dead connections. It also provides "fail-fast" capabilities, thereby saving on system resources!

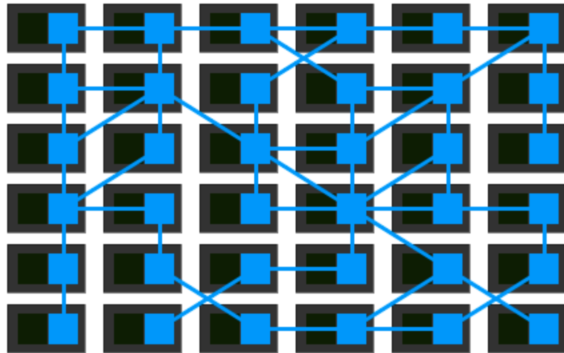
From SAL to Sidecar

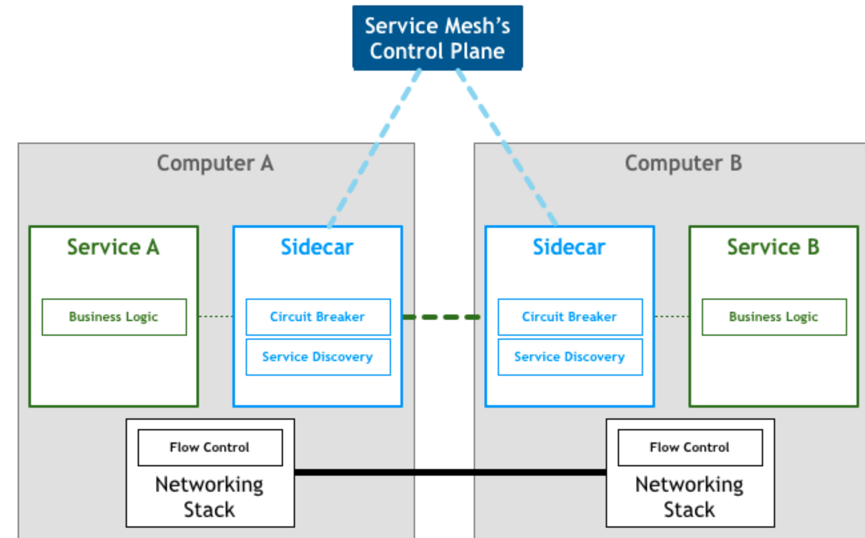


From: http://philcalcado.com/2017/08/03/pattern_service_mesh.html

We could build the functionality into the application (bad) or into the network stack (bad as well, think end-to-end argument). The proxy pattern is a rather unintrusive and maintainable solution: A true DS middleware

From Sidecar to Service Mesh: Control vs. data plane





From: http://philcalcado.com/2017/08/03/pattern_service_mesh.html

The proxies, together with the control plane or mesh master form a separate middleware to control routing etc.

Still Performance Problems...

- Delivering a complete homepage still took too long!
- Everything had been parallelized already!
- lots of CPU and I/O used for dynamic page construction!

Clearly, the effort for a complete HP request was considerable. How could we reduce latency and effort?

Possible Causes for Performance Problems

- No Caching (we did not know what ???)
- No Pooling (we started pooling heavy objects like connections, threads, parsers)
- No Threading (we did parallelize everything. This does not reduce latency across machines)
- Persistent session state large or slow (oops, 25k/req. Into DB...)
- High-level Synchronization (we checked our source for "synchronized" statements at important bottleneck functions
- Synchronous instead of asynchronous requests (Java did not support async. I/O at that time)

No caching is possible without an information architecture, which specifies - from a business point of view - how current information has to be! If your infrastructure cannot deliver at that rate: " Business, we have to talk..."

On Performance, Caching and Architecture

- No Information Architecture existed: Information not qualified with respect to aging and QOS.
- Caching possibilities not used (http) or underestimated (20 secs. Are static!)
- No compression or web accelerators used.
- Architecture not fit to support caching (where and what analysis missing)
- Large scale portal needs fragment architecture
- Tactical mistakes: no automatic service time control, no automatic DB connection hold control, internal threading introduced too early…

Caching: Why, What, Where and how much?

Information Architecture

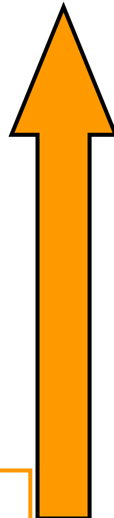
Fragment Architecture

determine

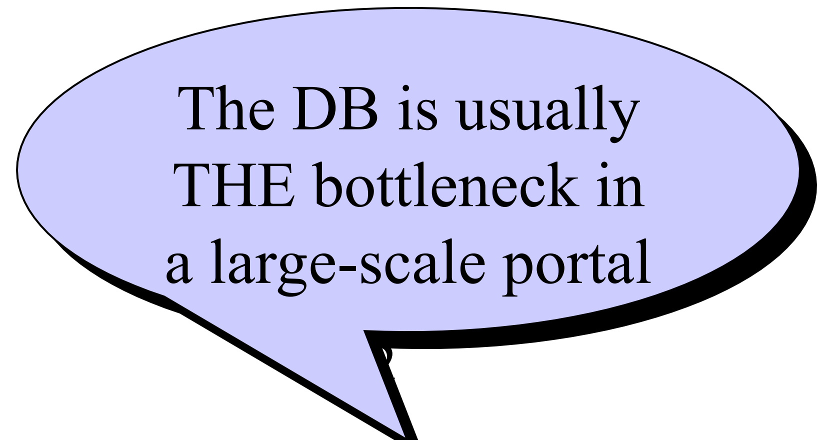
- Lifecycle
- Fragmentation
- QOS (e.g. Realtime quotes)

- Result Objects/Value Objects
- Invalidation mechanism
- Addressing of fragments
- Cache Subsystem QOS (e.g. automatic re-load)

Caching possibilities



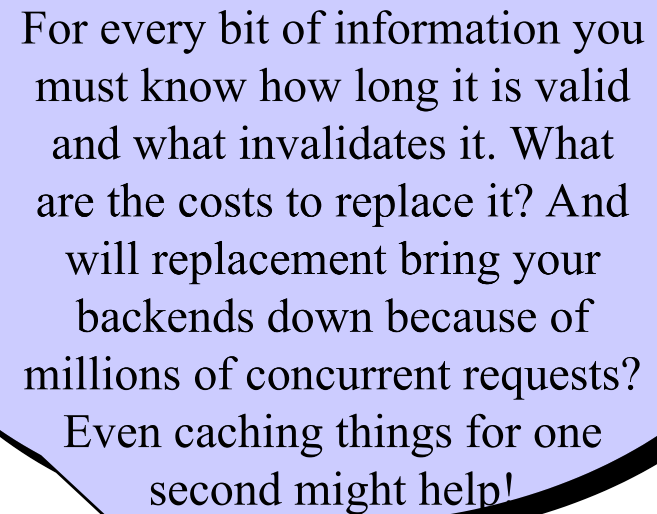
Throughput/ Performance



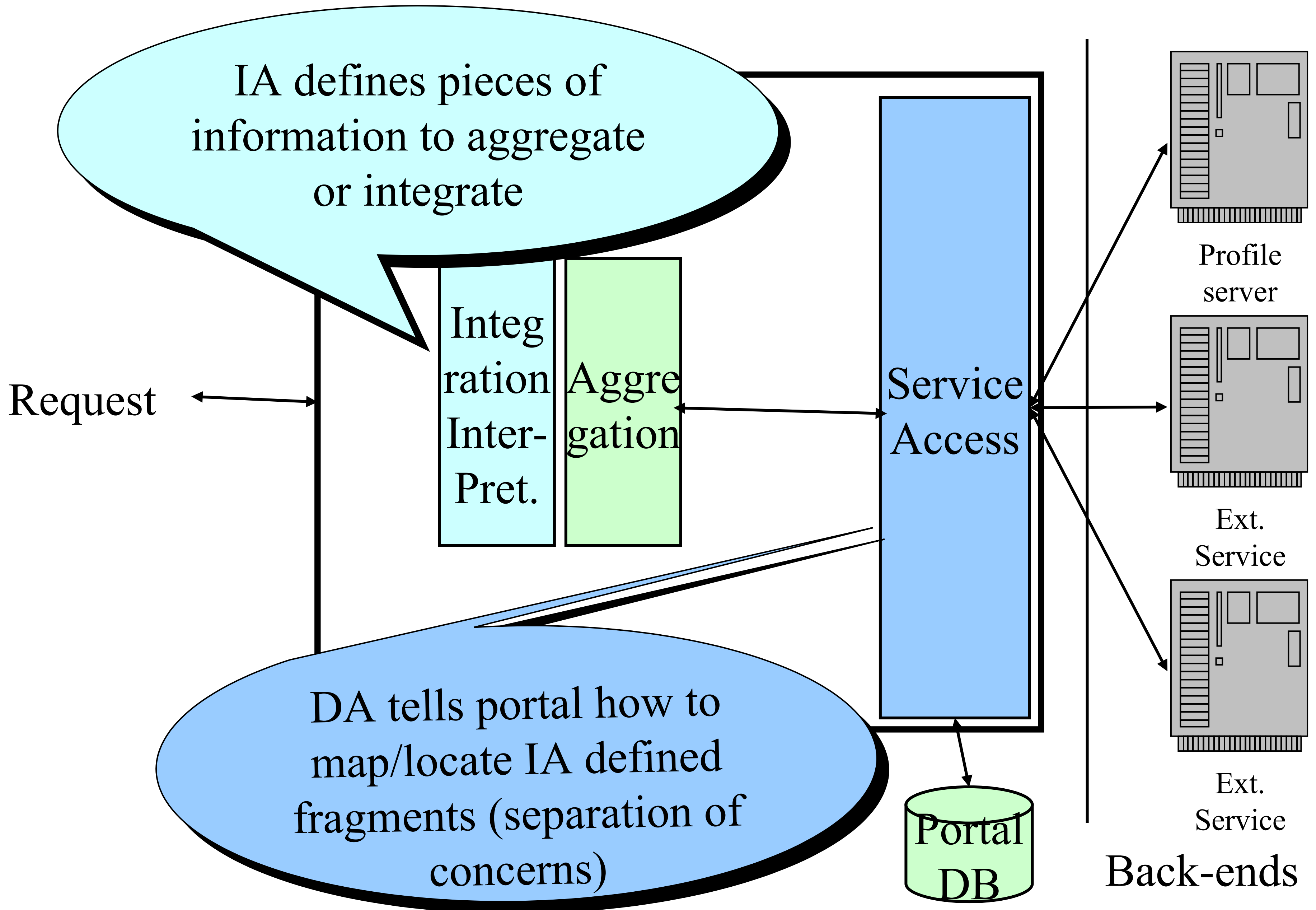
Problem analysis

Information Architecture – Lifecycle Aspects

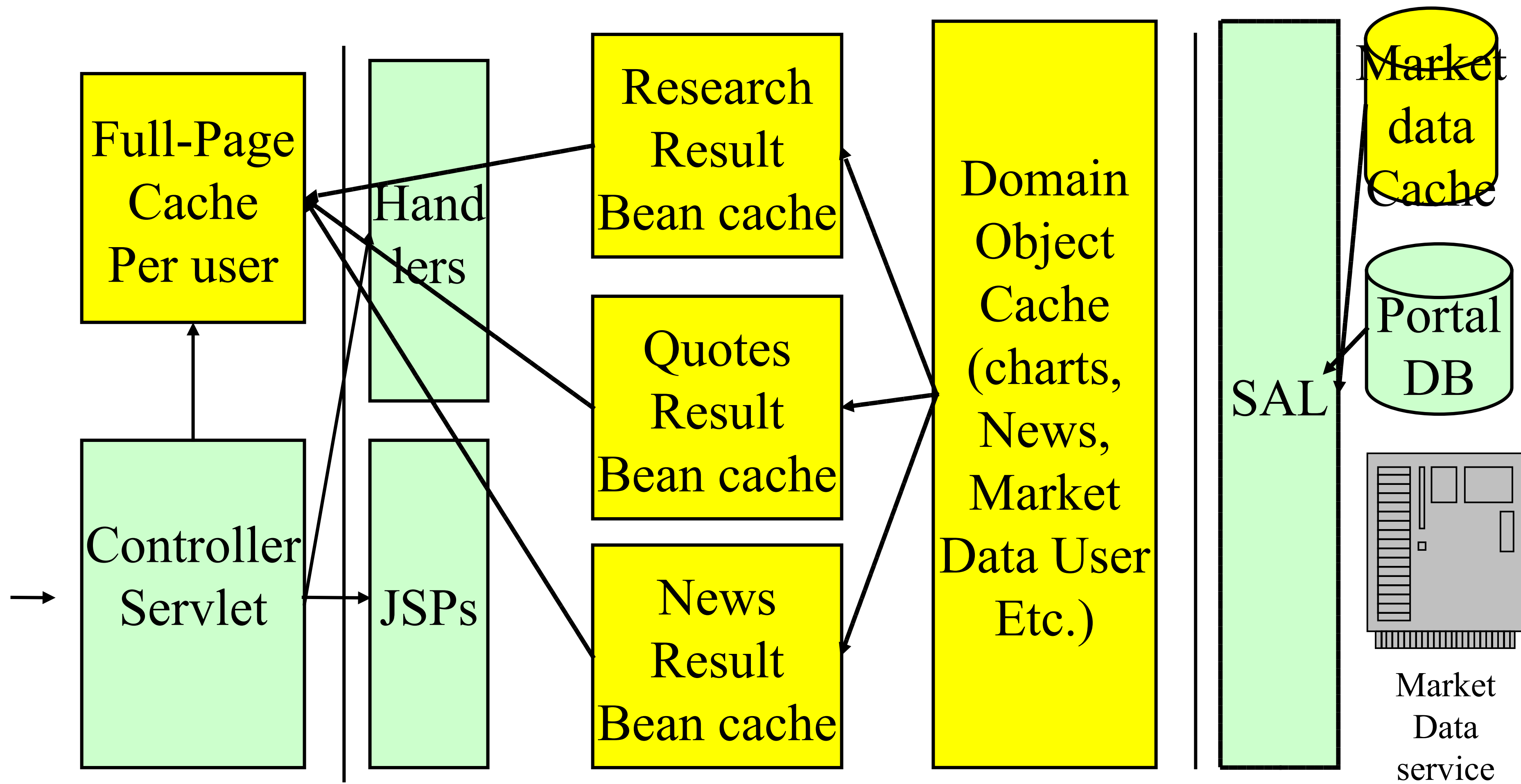
Data / changed by	Time	Personalization
Country Codes	No (not often, reference data)	No
News	Yes (aging only)	No, but personal selections
Greeting	No	Yes
Message	Yes (slowly aging)	Yes
Stock quotes	Yes (close to real-time)	No, but personal selections
Homepage	Yes (message numbers, quotes) Question: how often?	Yes (greeting etc.)



Information- and Distribution Architecture



Cache fragments, locations and dependencies (without client and proxy side caches)



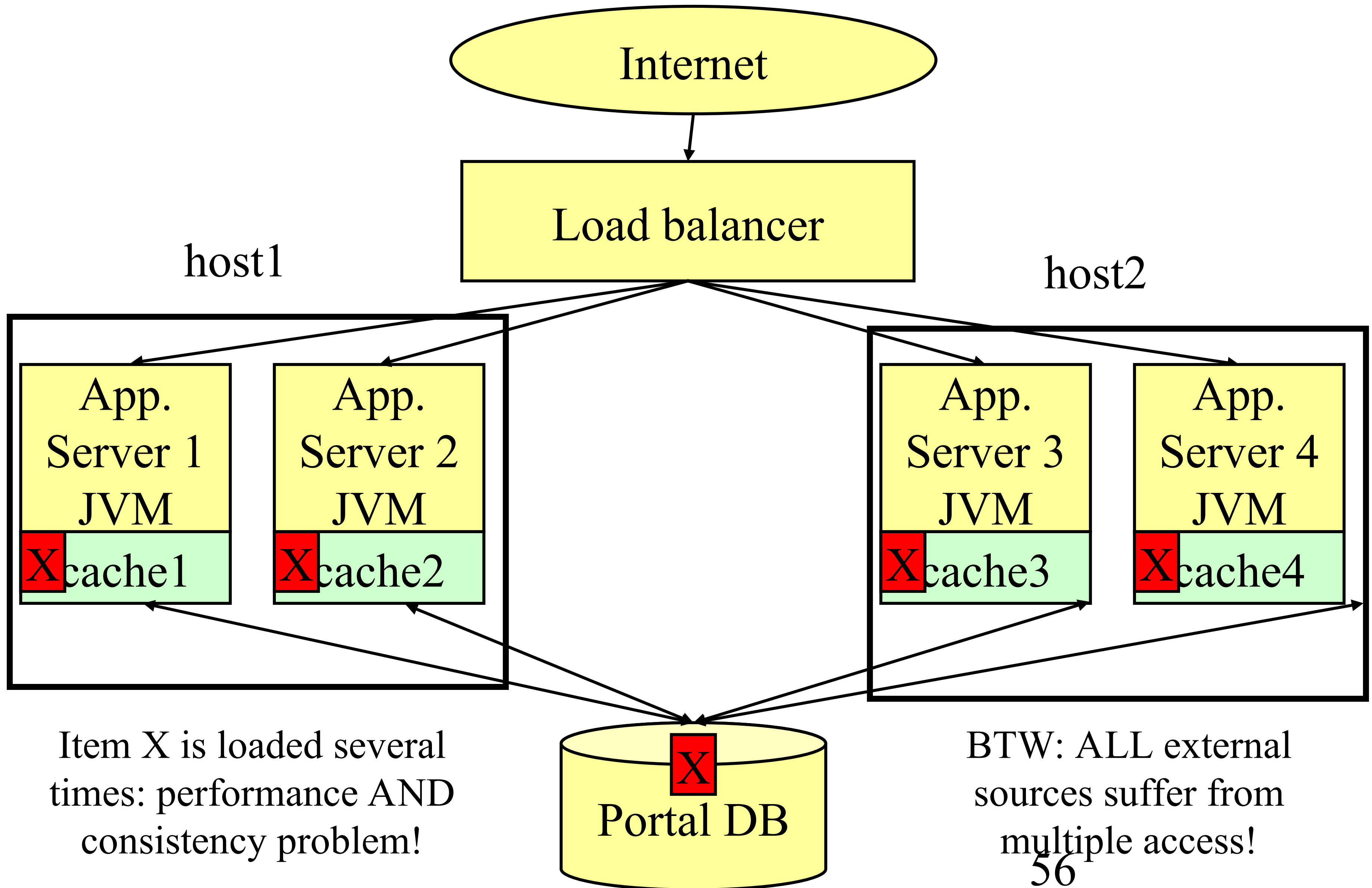
Fully processed Page

Page parts, processed

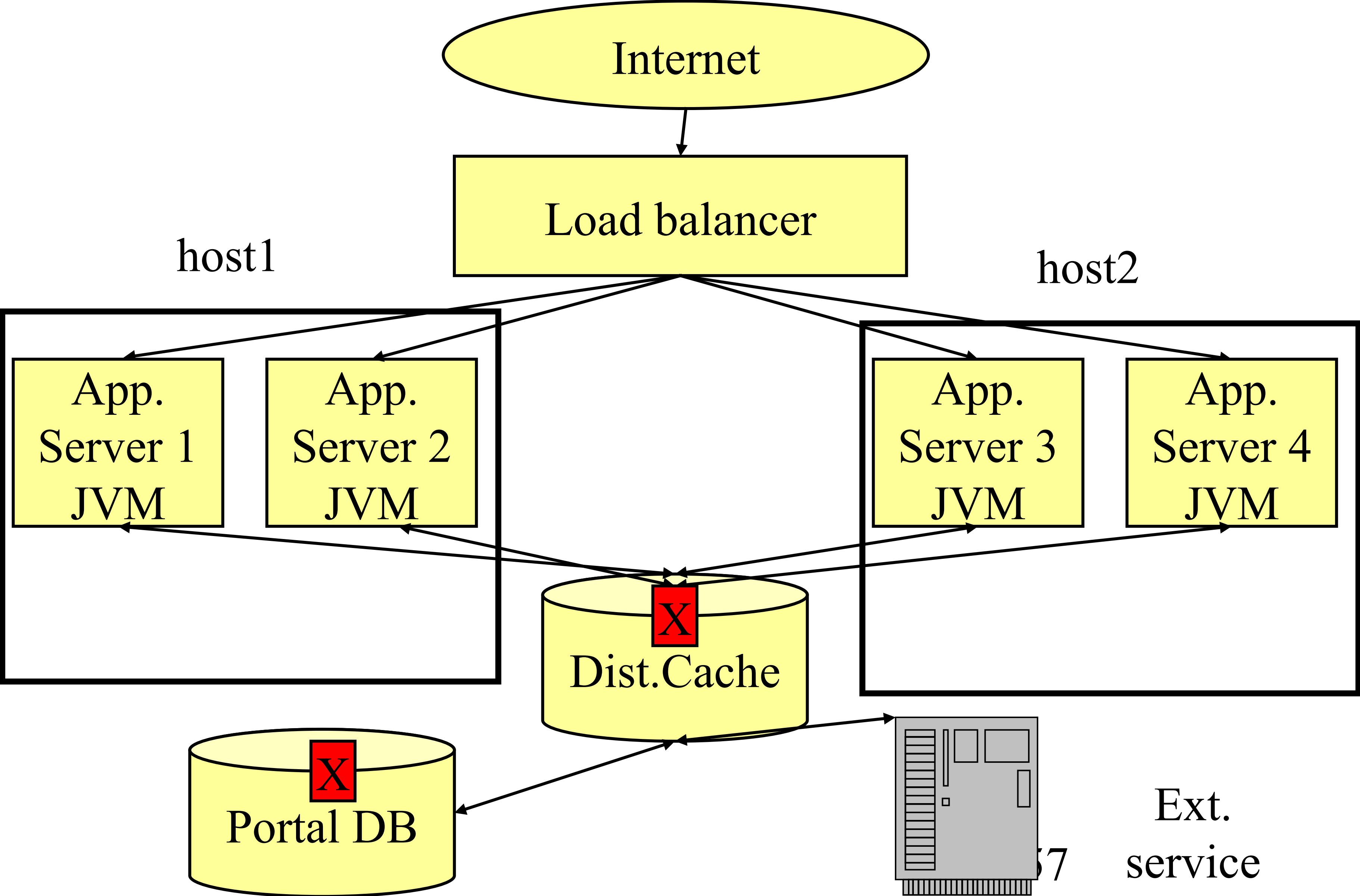
Distributed cache, raw data

Service Access Layer

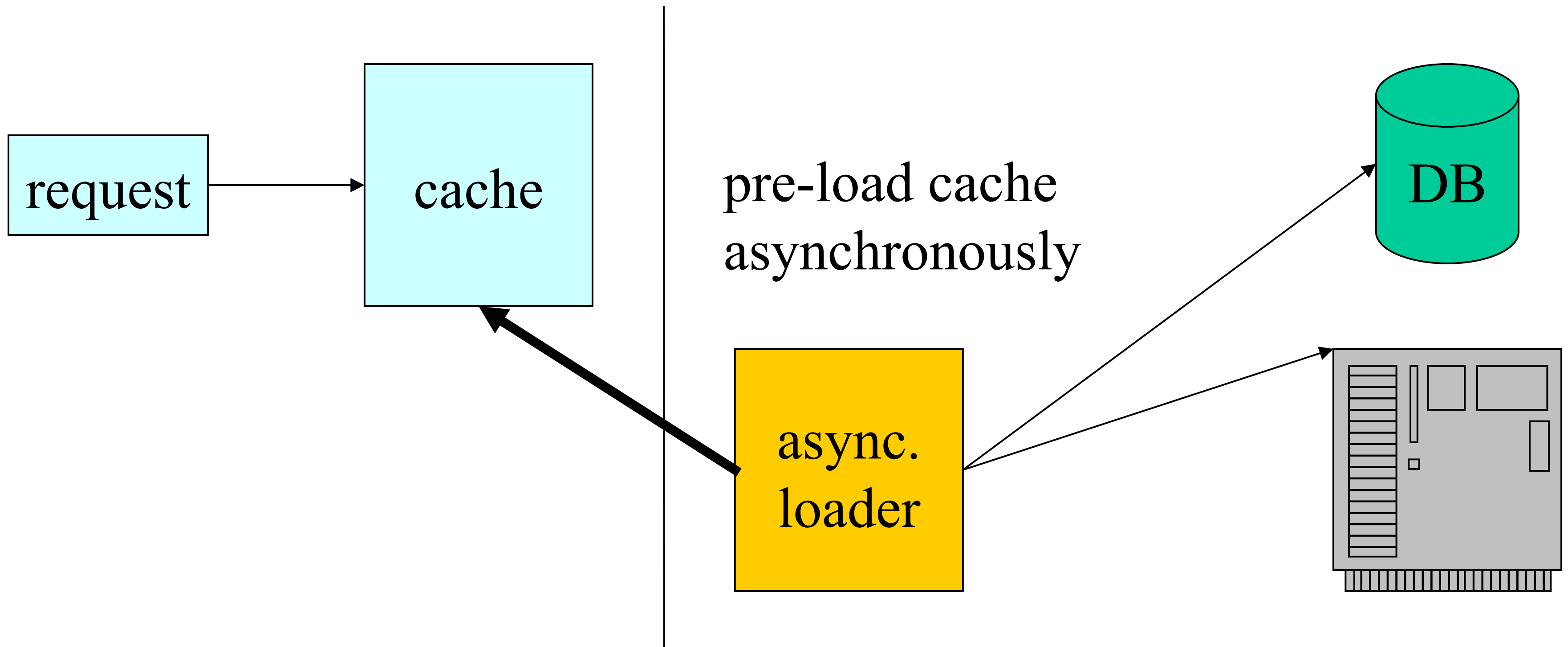
Several Hosts Without Distributed Cache



Several Hosts With Distributed Cache



The need for an asynchronous loader



The async. loader decouples synchronous request time from asynchronous retrieve time. The is a tight limit on what can be done in a distributed system while a user is waiting.

Fragments

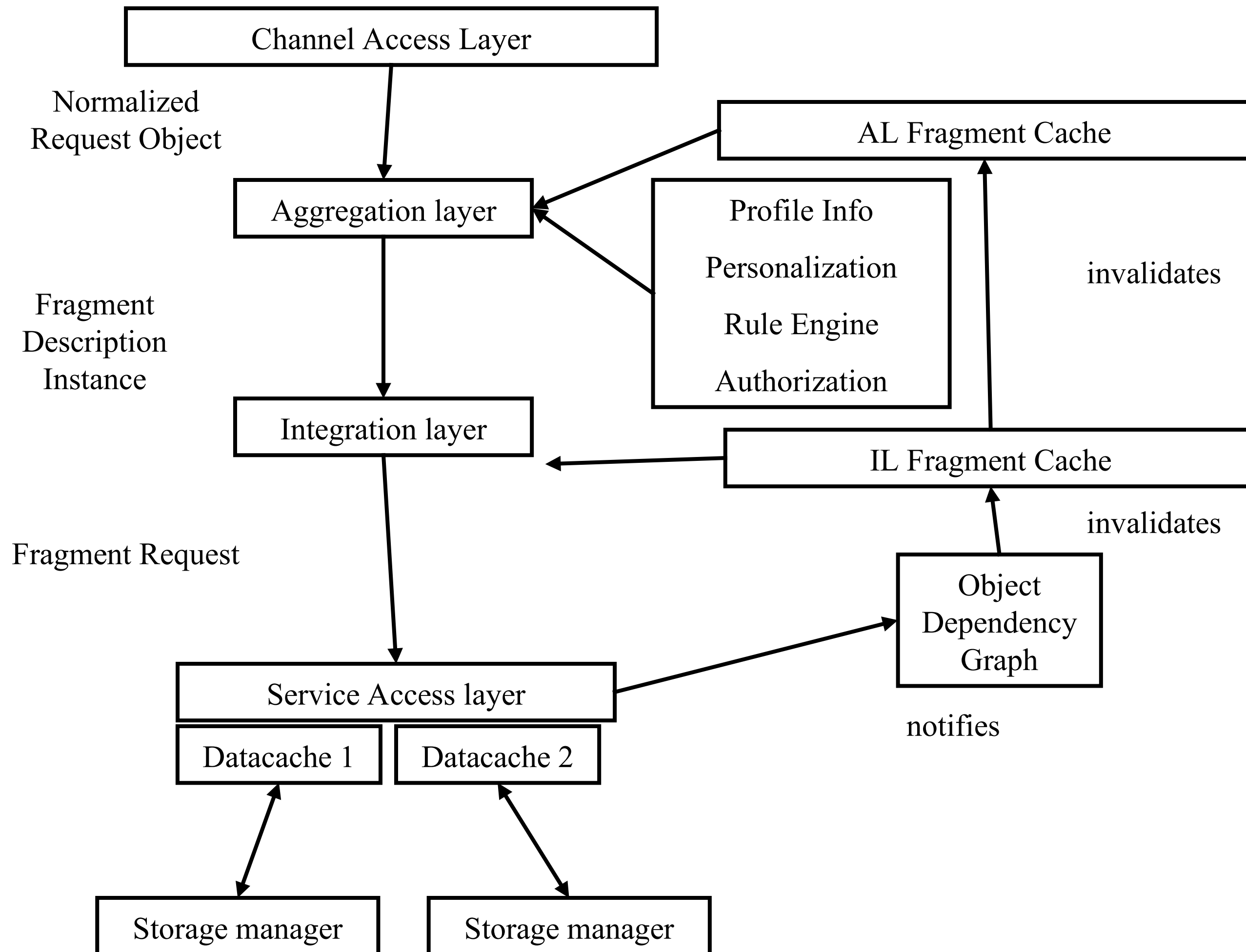
- Complete pages are frequently uniqe to customers. They cannot be re-used for others

- Page fragments can be shared and re-used heavily in most cases.

- Keeping fragments separately allows a huge reduction in back-end requests.

- The downside: If fragments change, you need a mechanism to invalidate dependend pages

Fragment Based Information Architecture



Goal: minimize backend access through fragment assembly (extension of IBM Watson research)

Physical and Process Architecture

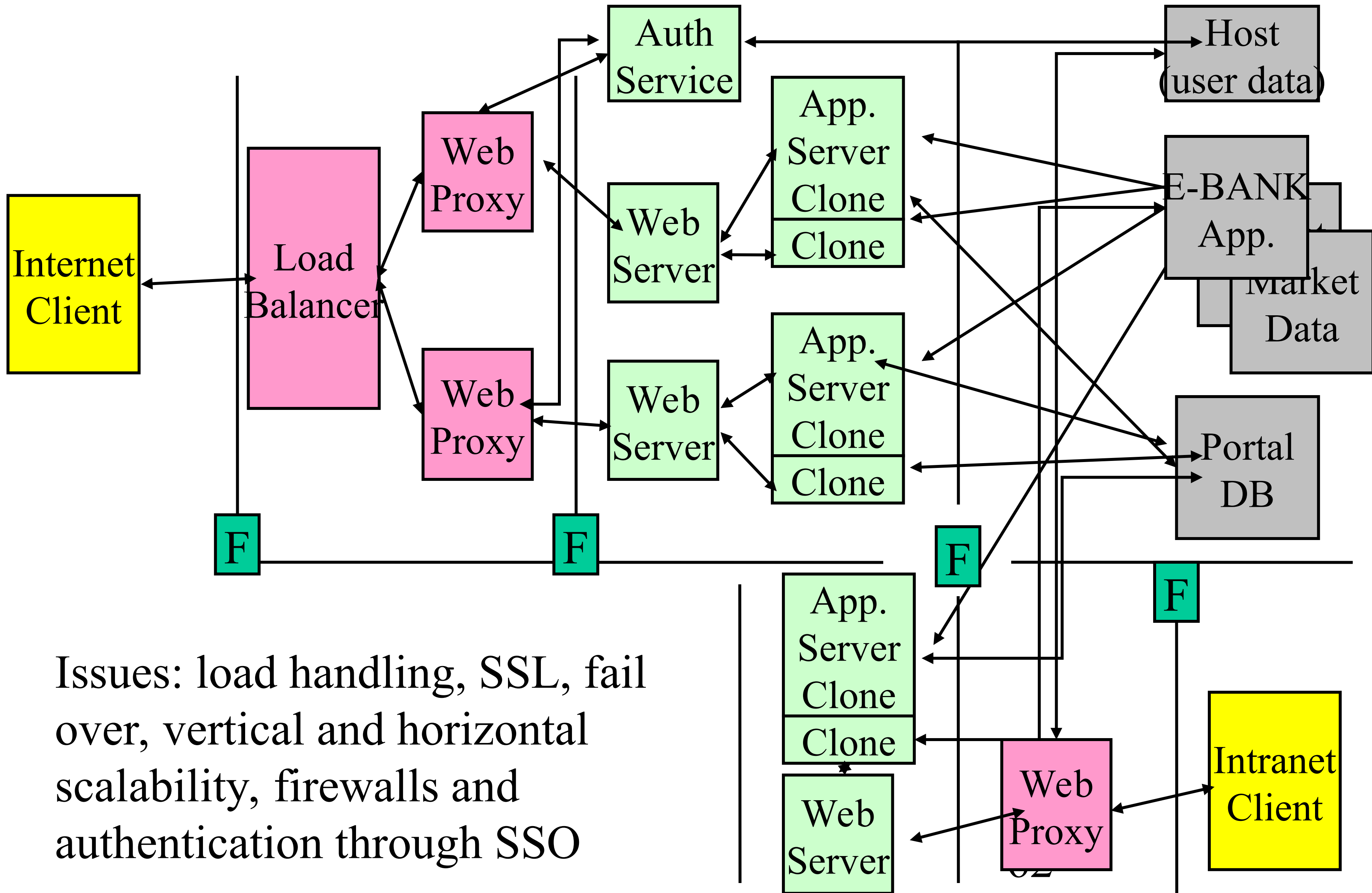
The physical architecture deals with reliability issues (replication, high-availability etc.) and horizontal and/or vertical scalability.

A projects physical architecture needs to define the scalability methods FROM THE BEGINNING because of their influence on the overall system architecture (e.g. distributed caching)

A horizontally scalable application can be replicated on more hosts. It avoids a single point of failure.

If an application scales only vertically this means that one can only install more CPUs or RAM on the single instance of the applications host. This type of application has limited scalability and availability (a so called HA-application)

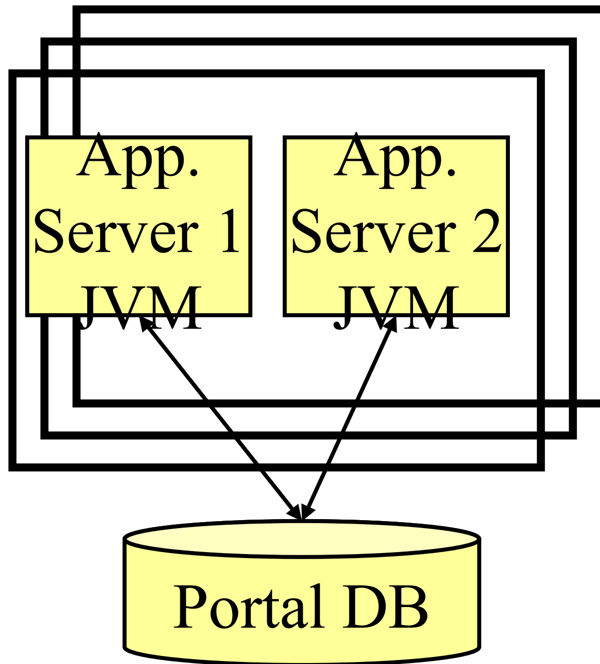
Physical Portal Architecture: Web Cluster



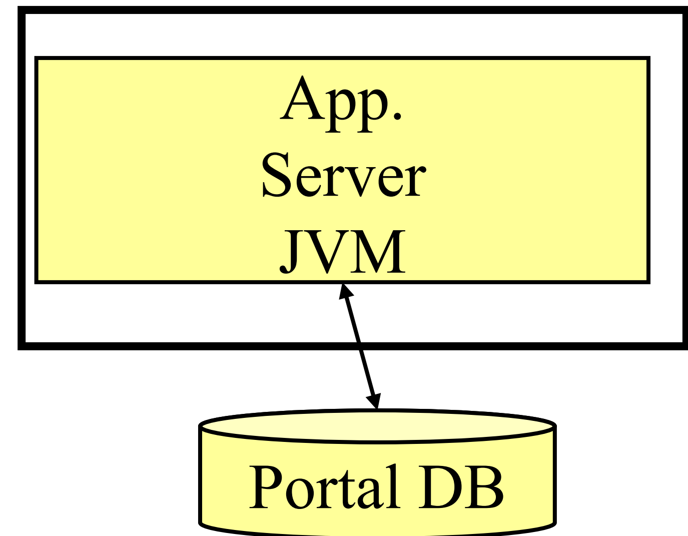
Issues: load handling, SSL, fail over, vertical and horizontal scalability, firewalls and authentication through SSO

Physical Architecture Alternatives

several copies of the application on one or more (smaller) hosts



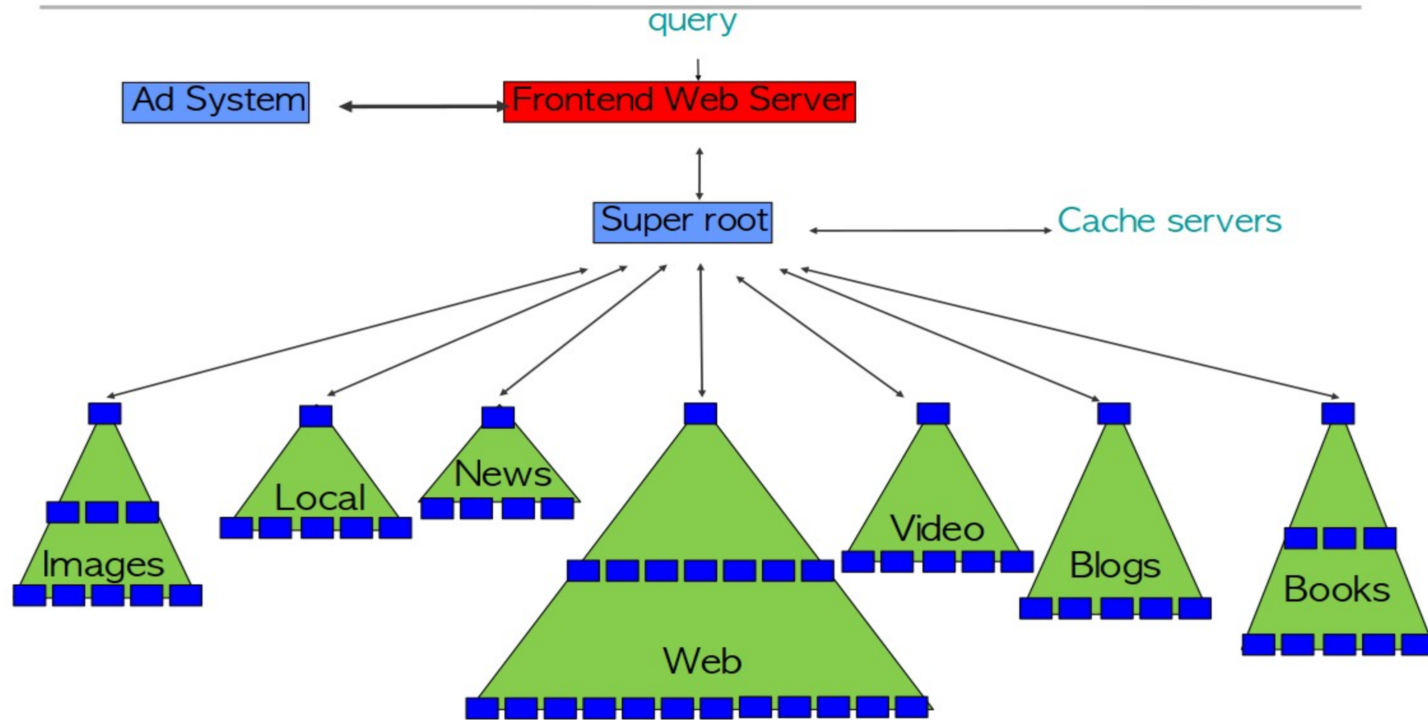
one (big) application instance only



Current Fan-Out Architecture

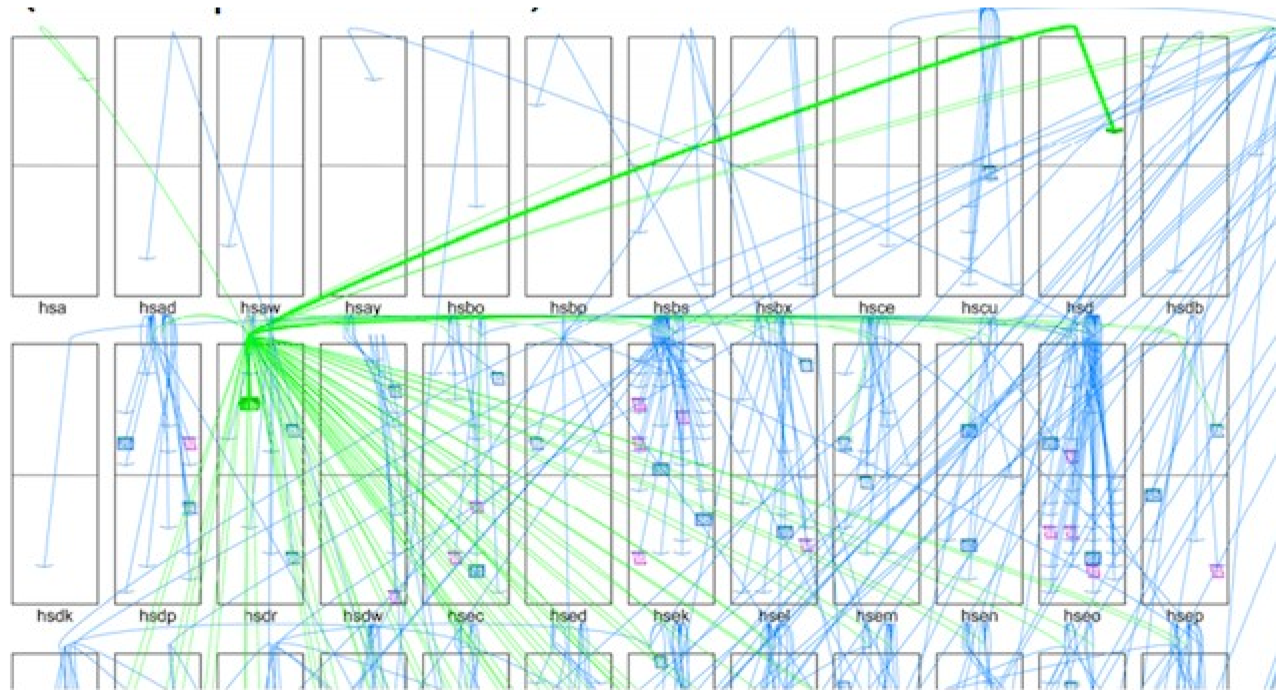
Performance and Throughput

Large fan-out Architectures at Google



A portal is a typcial "large-fan-out architecture" with long-tail problems. See how google handles this: Talk by Jeff Dean, http://static.googleusercontent.com/media/research.google.com/en//people/jeff/Berkeley-Latency-Mar2012.pdf

Google Search Query Trace



From:

"the Nyquist theorem and limitations of sampling profilers today with glimpses of tracing tools from the future", http://danluu.com/perf-tracing/

Reminder: The Costs of Delays

100 sub-calls, 1% delayed, how many calls will experience a delay?

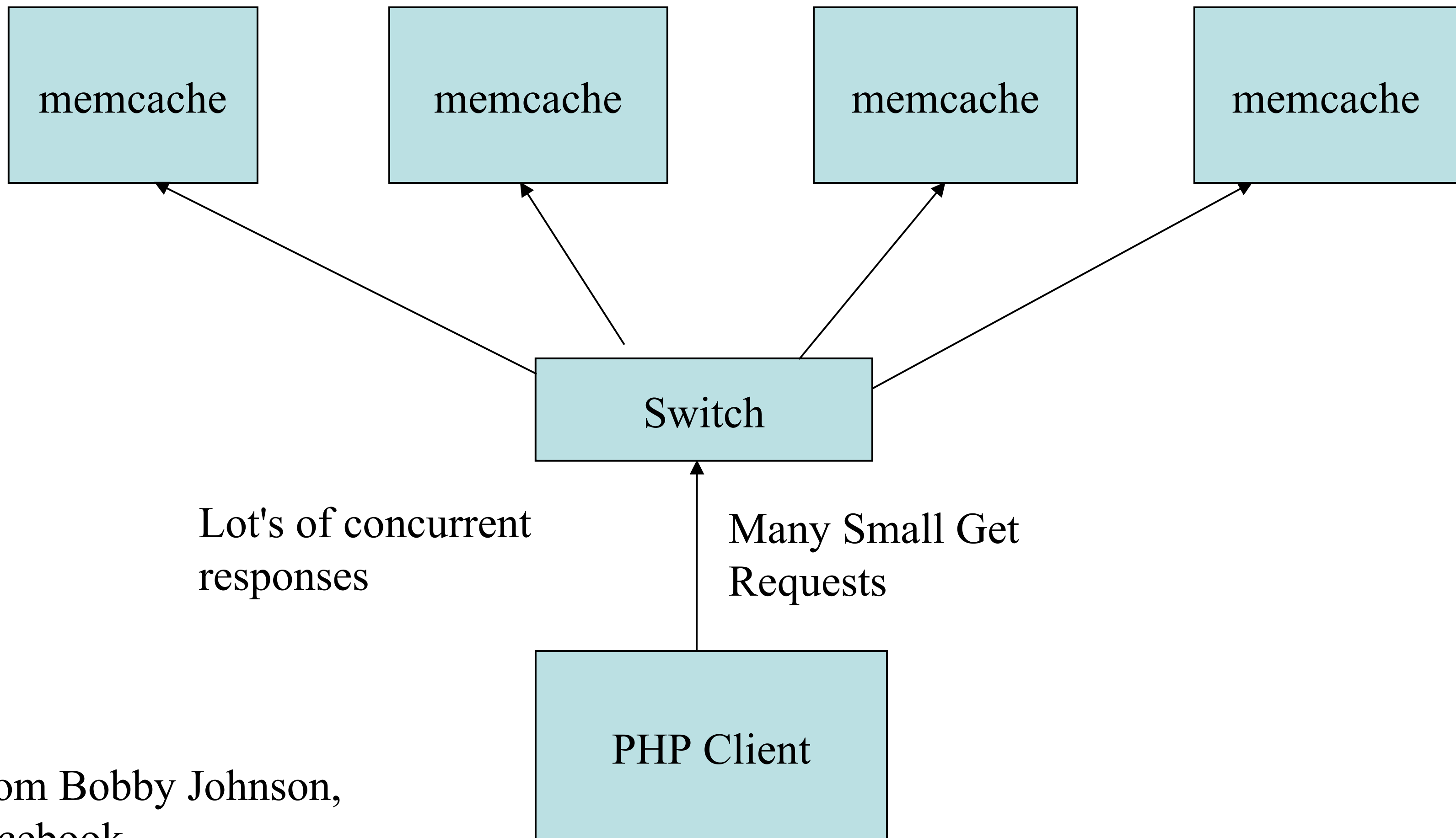
"Stalls" of any kind are critical in this architecture. What can we do against hiccups and stalls? Watch out for requests beyond the 99%ile! (Gile Tene, Azul: How NOT to measure latency)

Lessons learned on Latency Reduction/Tolerance in FO-Arc.

- Keep response times in a tight percentile but be aware of stragglers
- fight stragglers with backup requests and cross server cancellation.
- Watch for overload at sender when responses come back
- Do NOT distribute load evenly: synchronize background load across machines e.g. every 5 minutes.
- Reduce head-of-line blocking (partition large requests)
- Partition data across machines
- Cheat: come back with partial data
- Cross request adaptation
- Increase replication count
- Beware of the incast problem

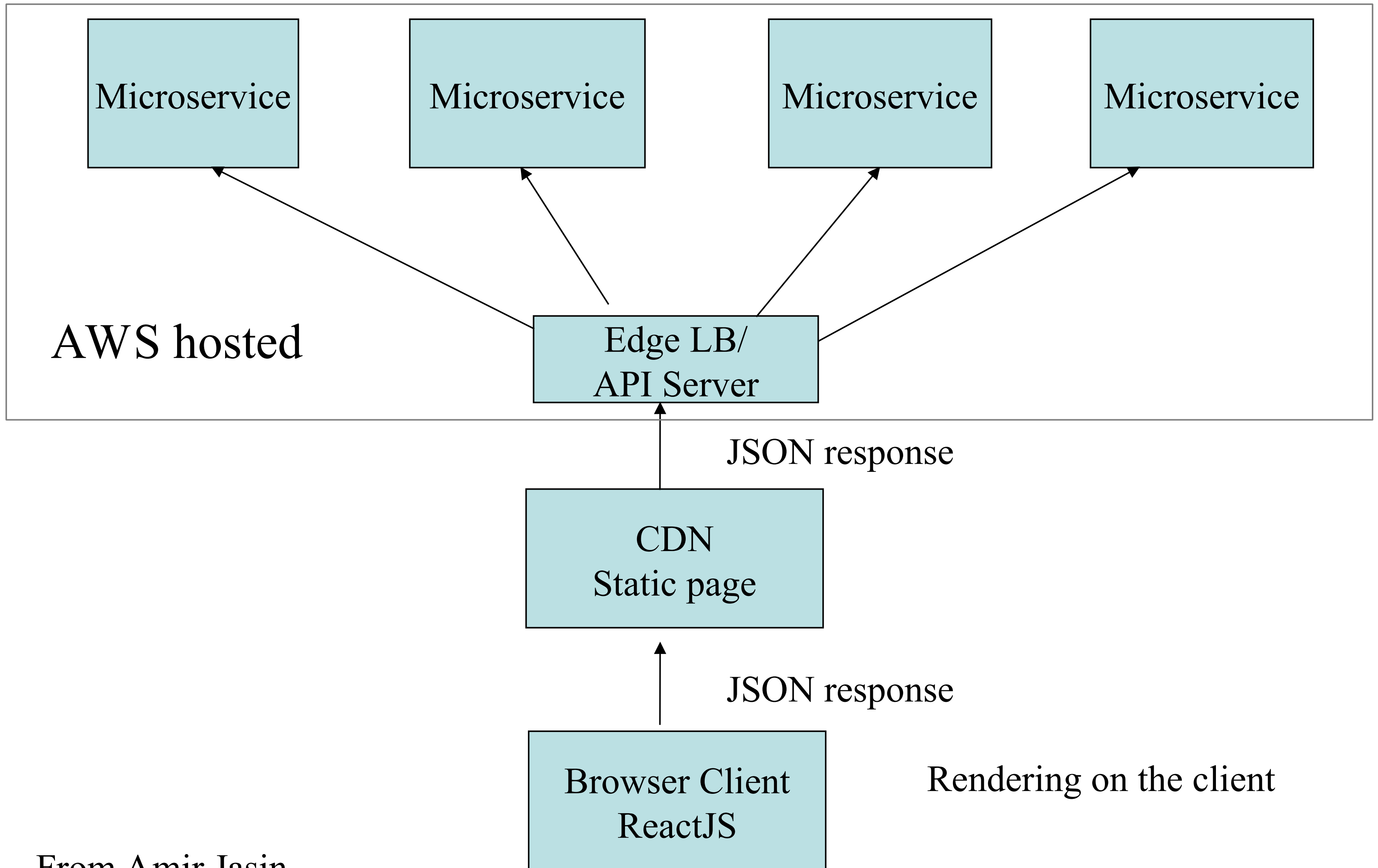
from: Jeff Dean, " Achieving Rapid Response Times in Large Online Services"

Network Incast



From Bobby Johnson, Facebook

Rendering



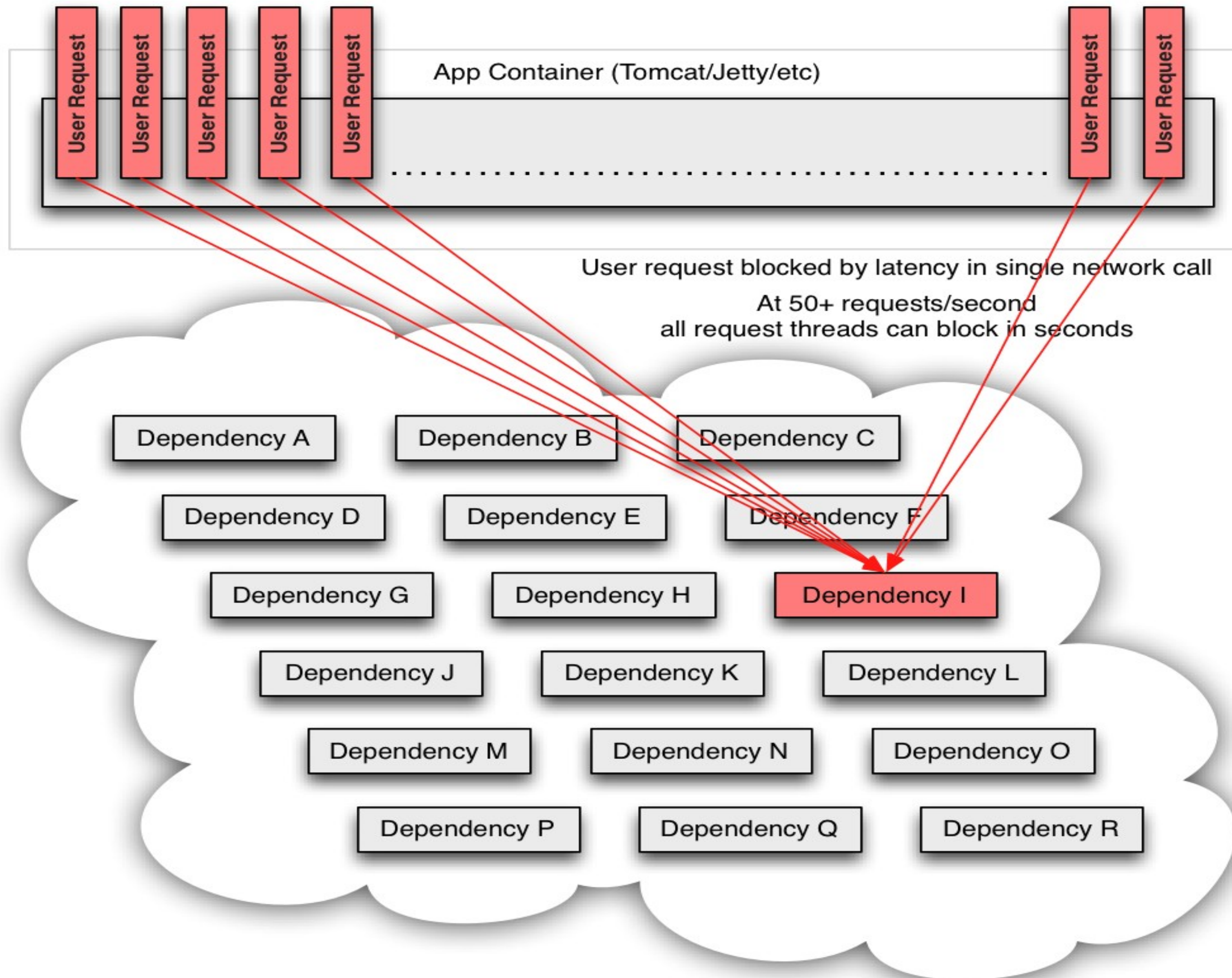
From Amir Jasin,
https://medium.com/swlh/scaling-on-the-cheap-933e46944886#.5e4khjmjm

Availability Aspects of Fan-Out Architectures

The following diagrams are from:

Fault Tolerance in a High Volume, Distributed System, by Ben Christensen (Netflix), http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html

Failure of a Service



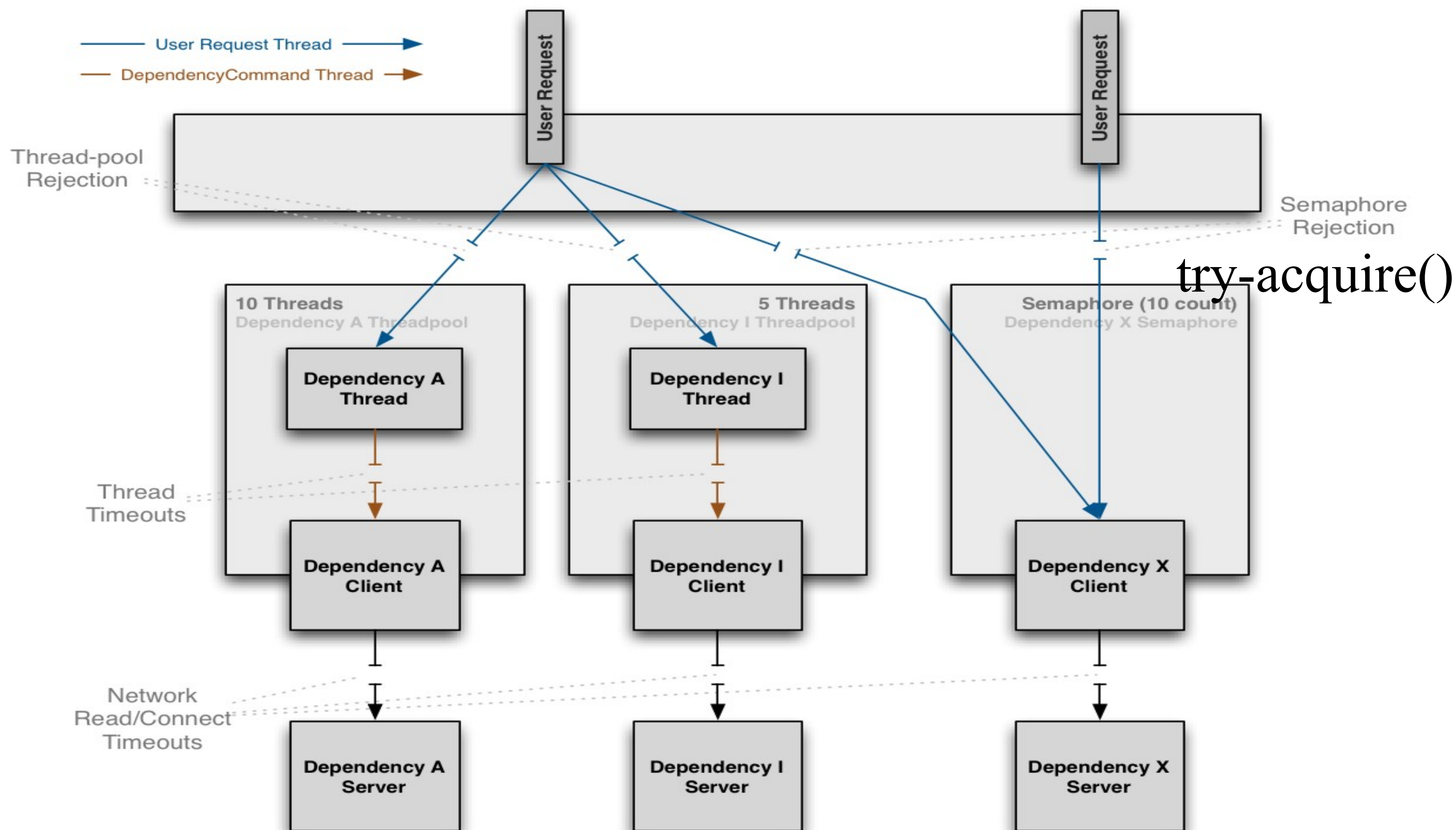
From: Ben Christensen (Netflix), http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html

Avoid Getting Stuck!

- Fail Fast: do not wait for resources which have problems.
- Always use timeouts when accessing a service
- Use exponentially decreasing re-tries if needed
- Use alternatives when a service does not work (fallback), e.g. serve stale data
- Cache: Retrieve data from local or remote caches if the realtime dependency is unavailable, even if the data ends up being stale
- Eventual Consistency: Queue writes (such as in SQS) to be persisted once the dependency is available again
- Stubbed Data: Revert to default values when personalized options can't be retrieved
- Empty Response ("Fail Silent"): Return a null or empty list which UIs can then ignore

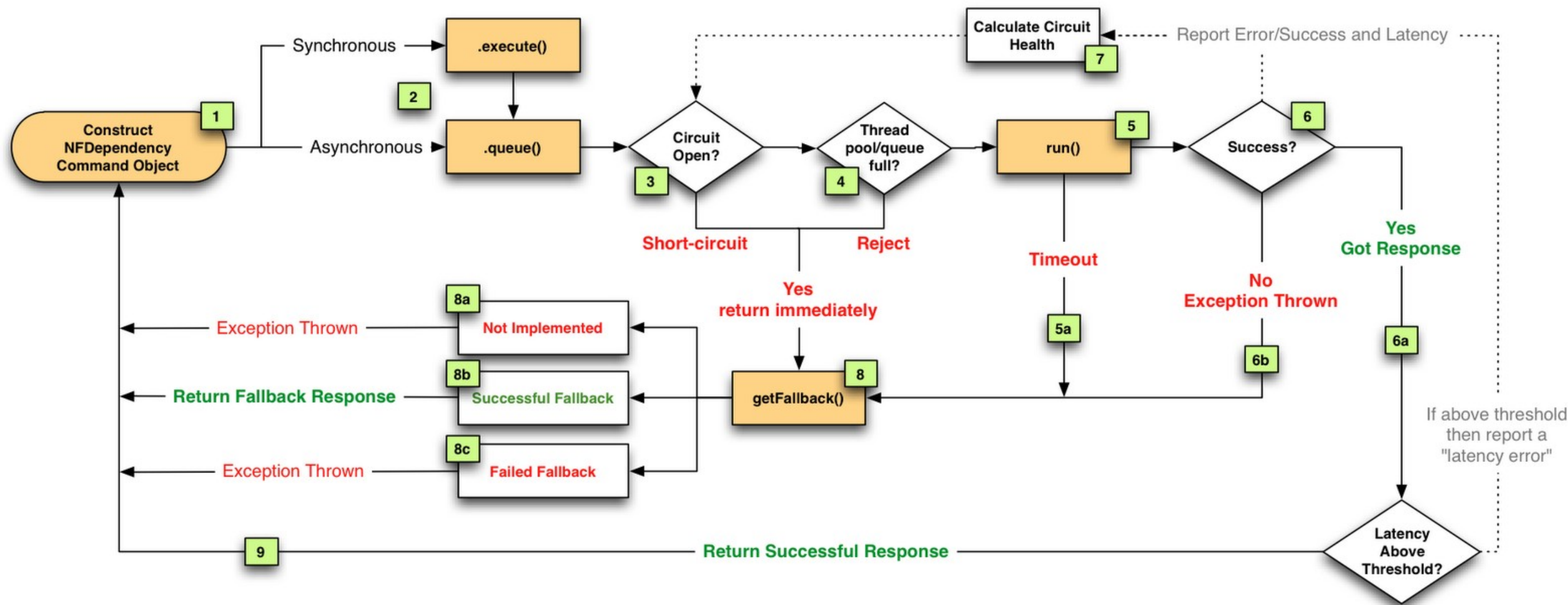
After B.Christensen

Prevent stuck Services: “Bulkheads”



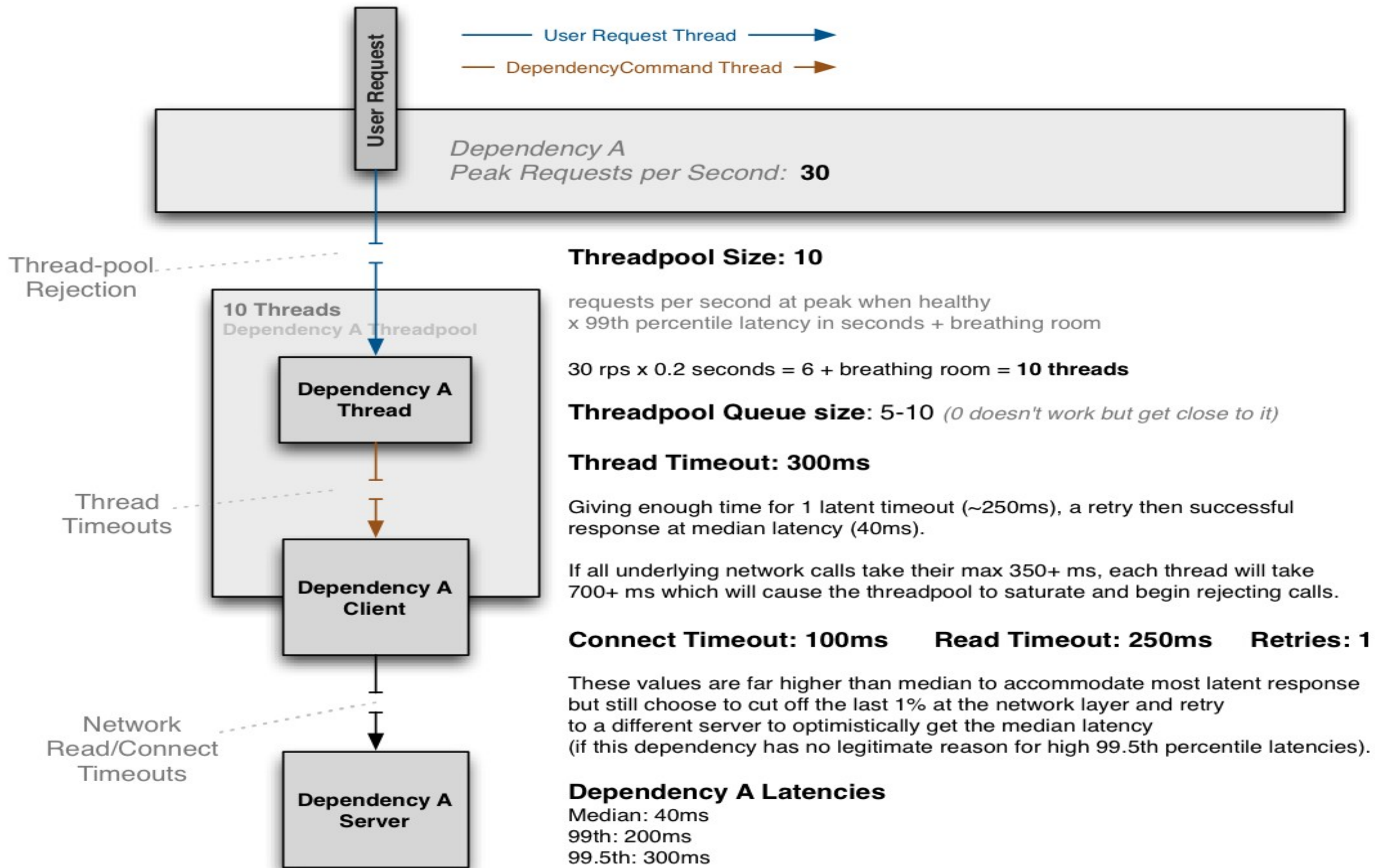
Diag: B.Christensen. Semaphores and Threadpools implement the “bulkhead” pattern. Otherwise all available threads will quickly block at an overloaded/crashed service

Change Processing: Circuit-Breaker



Diag: B.Christensen. Feedback-control is becoming an important feature of data-center design!

Example



Diag: B.Christensen. Timeouts, pools and retries combine to avoid stuck requests.

Black-Swans

Some subspecies of black swan

Hitting limits

Spreading slowness

Thundering herds

Automation interactions

Cyberattacks

Dependency problems

Laura Nolan, what breaks our systems?

https://www.usenix.org/conference/lisa18/presentation/nolan

Cell-Architectures

- Blast Radius Reduction

- Shuffle Sharding

Amazon Kinesis Incident

November, 25th 2020

We wanted to provide you with some additional information about the service disruption that occurred in the Northern Virginia (US-EAST-1) Region on November 25th, 2020.

Amazon Kinesis enables real-time processing of streaming data. In addition to its direct use by customers, Kinesis is used by several other AWS services. These services also saw impact during the event. The trigger, though not root cause, for the event was a relatively small addition of capacity that began to be added to the service at 2:44 AM PST, finishing at 3:47 AM PST.

There were a number of services that use Kinesis that were impacted as well. Amazon Cognito uses Kinesis Data Streams to collect and analyze API access patterns. While this information is extremely useful for operating the Cognito service, this information streaming is designed to be best effort. Data is buffered locally, allowing the service to cope with latency or short periods of unavailability of the Kinesis Data Stream service. Unfortunately, the prolonged issue with Kinesis Data Streams triggered a latent bug in this buffering code that caused the Cognito webservers to begin to block on the backlogged Kinesis Data Stream buffers.

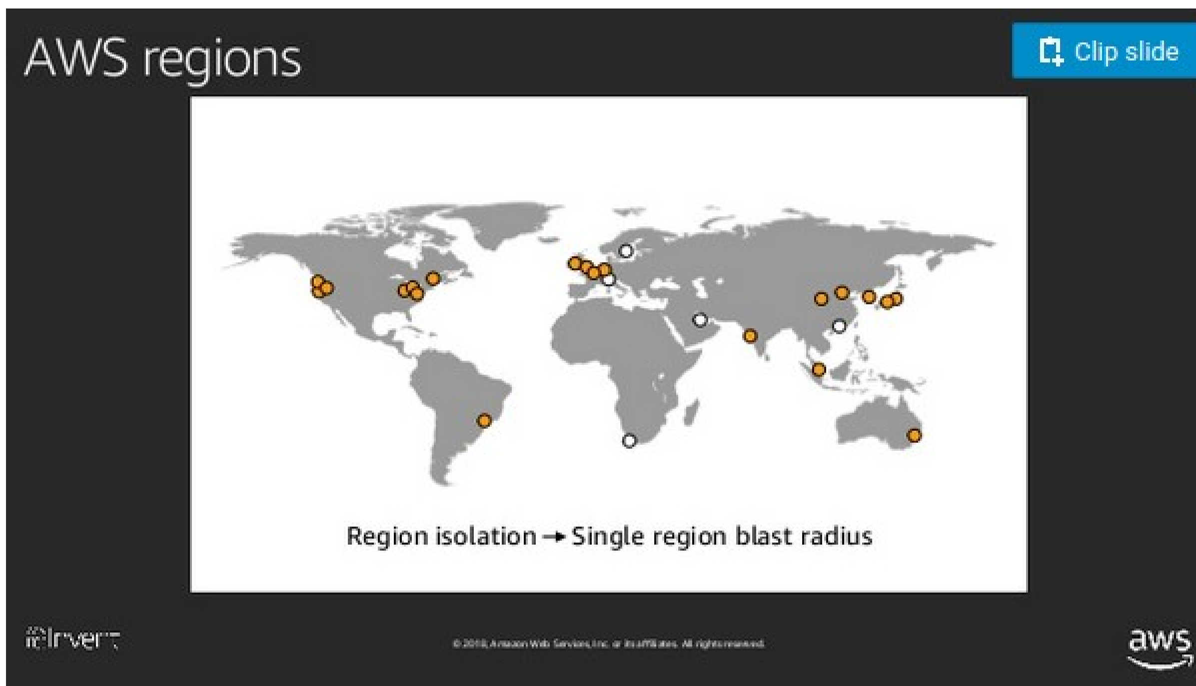
CloudWatch uses Kinesis Data Streams for the processing of metric and log data.

CloudWatch Events and EventBridge experienced increased API errors and delays in event processing starting at 5:15 AM PST. As Kinesis availability improved, EventBridge began to deliver new events and slowly process the backlog of older events. Elastic Container Service (ECS) and Elastic Kubernetes Service (EKS) both make use of EventBridge to drive internal workflows used to manage customer clusters and tasks. This impacted provisioning of new clusters, delayed scaling of existing clusters, and impacted task de-provisioning. By 4:15 PM PST, the majority of these issues had been resolved.

At 9:39 AM PST, we were able to confirm a root cause, and it turned out this wasn't driven by memory pressure. Rather, the new capacity had caused all of the servers in the fleet to exceed the maximum number of threads allowed by an operating system configuration. As this limit was being exceeded, cache construction was failing to complete and front-end servers were ending up with useless shard-maps that left them unable to route requests to back-end clusters.

https://aws.amazon.com/message/11201/

Blast Reduction I: Regions



Peter Voss, AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338) https://www.youtube.com/watch?v=swQbA4zub20&feature=youtu.be

Blast Reduction II: Planes

Control planes and data planes

Clip slide

- Control plane: administration of resources
- Data plane: usage of resources

Service	Control plane	Data plane
Amazon DynamoDB	DescribeTable API	Query API
Amazon EC2	RunInstances API	A running EC2 instance
AWS Lambda	CreateFunction API	Invoke API

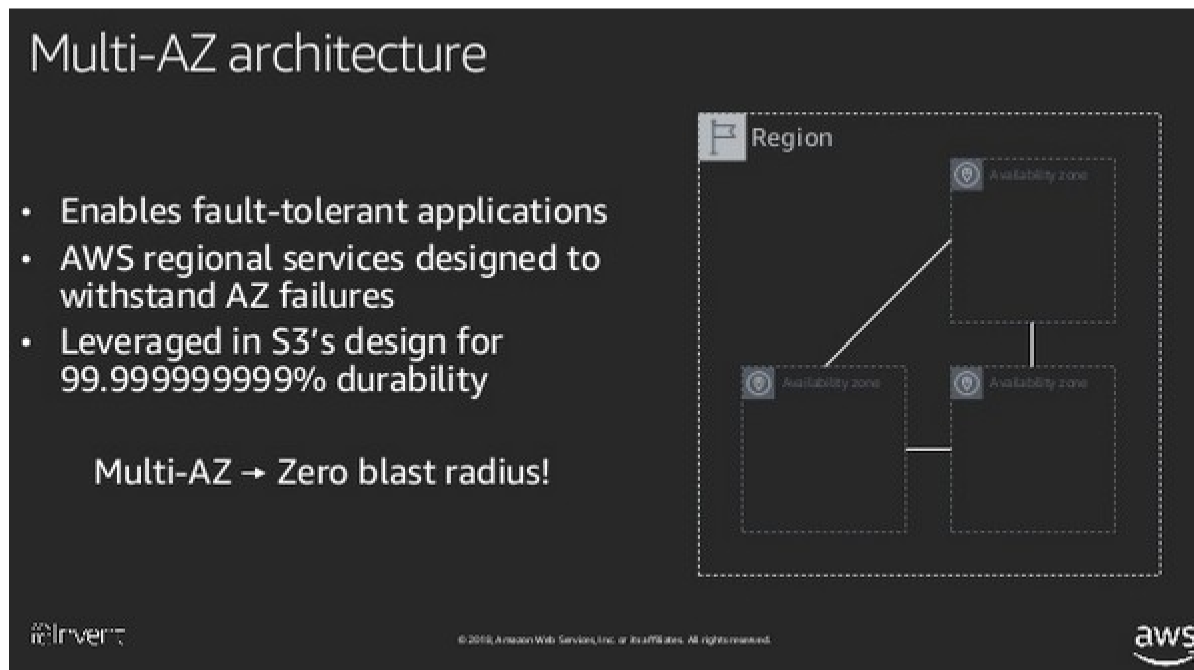
- Separating control plane from data plane reduces blast radius

re:Invent

aws

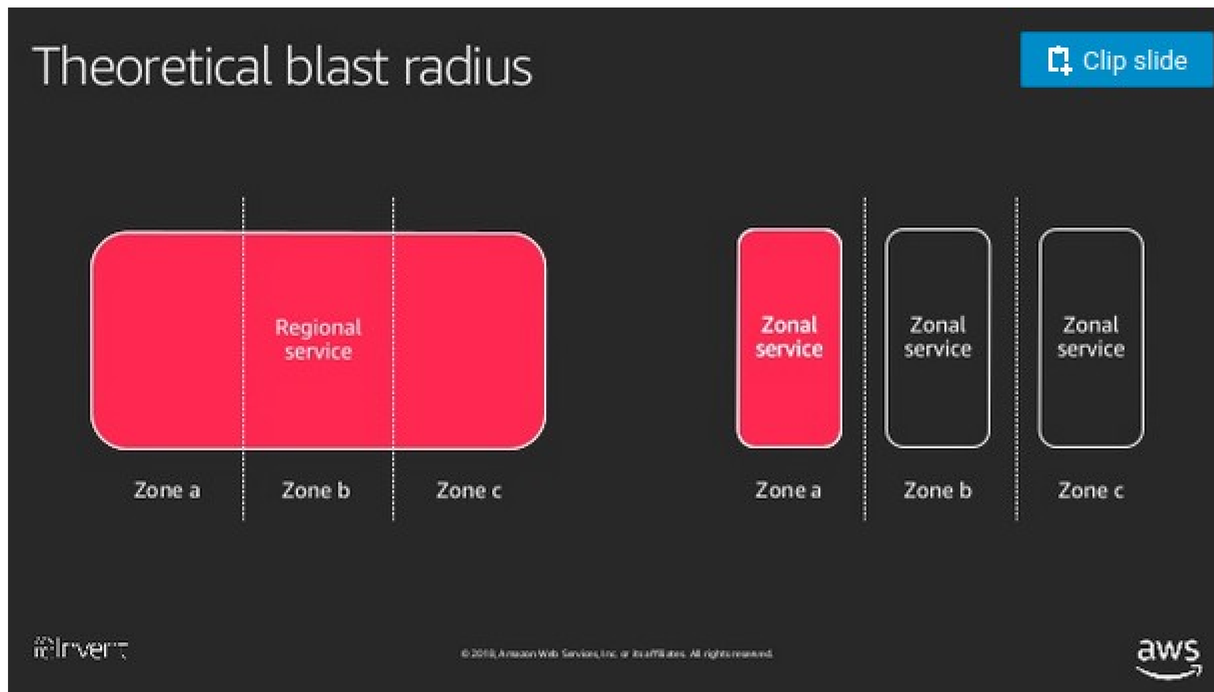
Peter Voss, AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338) https://www.youtube.com/watch?v=swQbA4zub20&feature=youtu.be

Blast Reduction III: Availability Zones



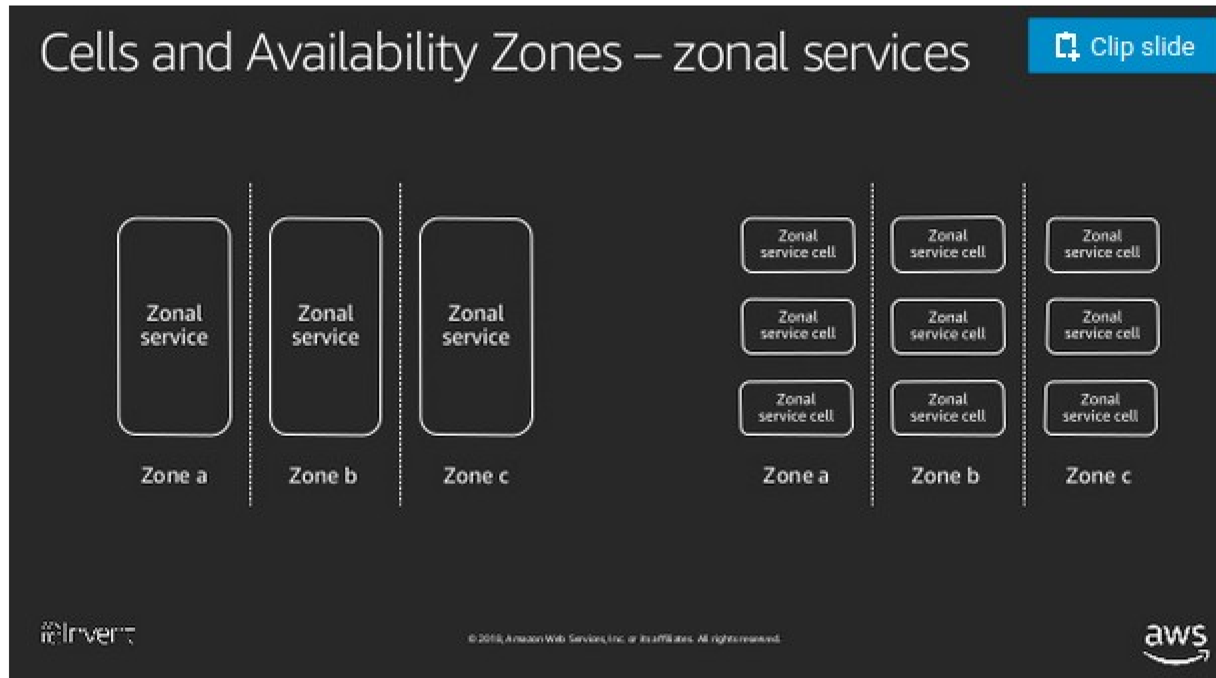
Peter Voss, AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338) https://www.youtube.com/watch?v=swQbA4zub20&feature=youtu.be

Blast Reduction III: Availability Zones



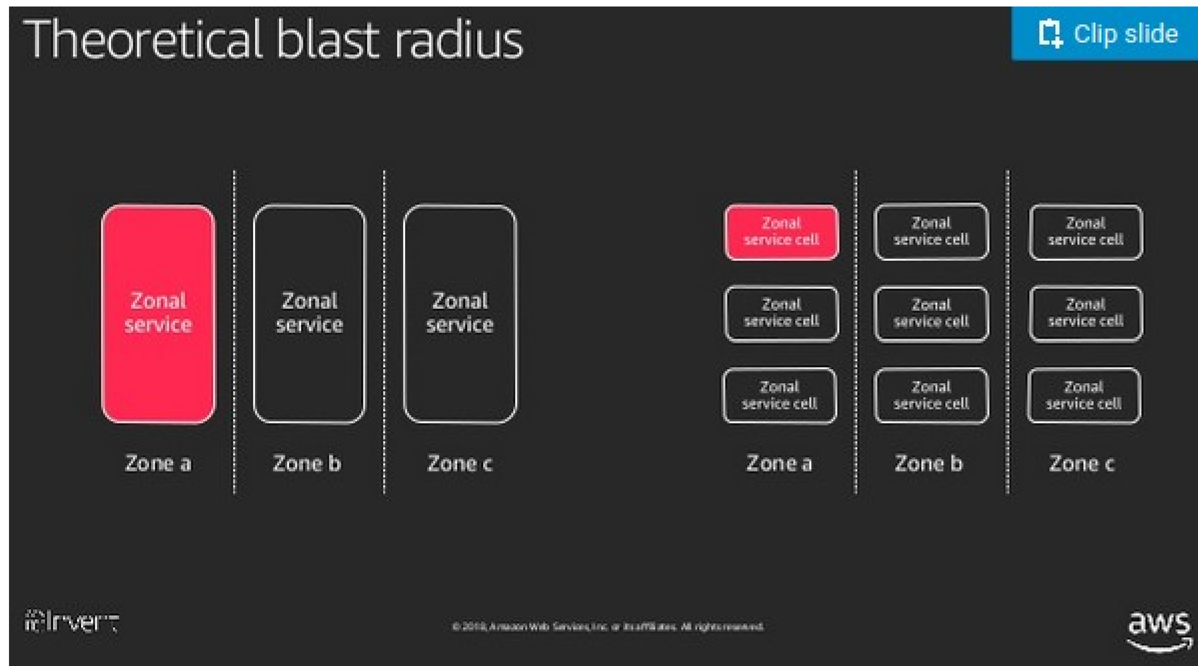
Peter Voss, AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338) https://www.youtube.com/watch?v=swQbA4zub20&feature=youtu.be

Blast Reduction IV: Cells and Zones



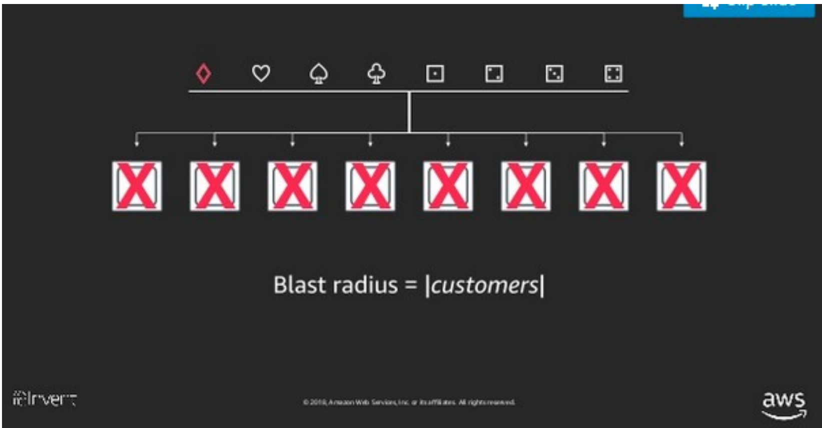
Peter Voss, AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338) https://www.youtube.com/watch?v=swQbA4zub20&feature=youtu.be

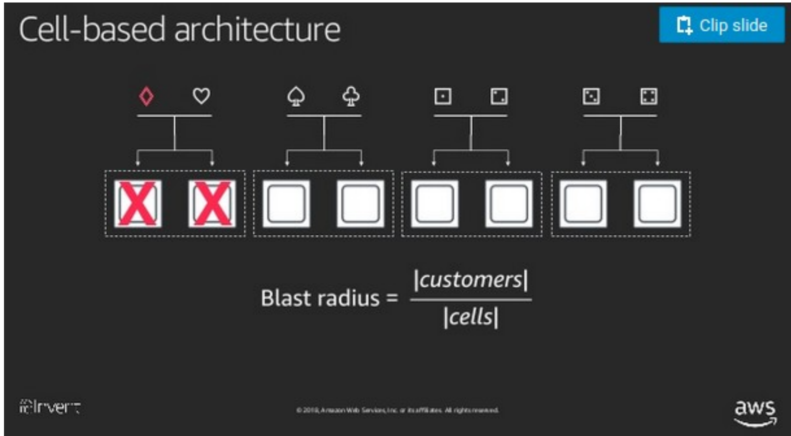
Blast Reduction IV: Cells

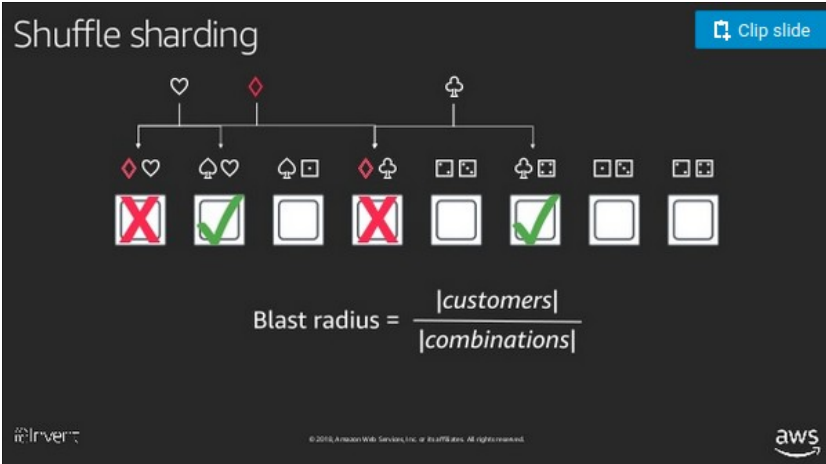


Peter Voss, AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338) https://www.youtube.com/watch?v=swQbA4zub20&feature=youtu.be

Blast Reduction V: Shuffle Sharding

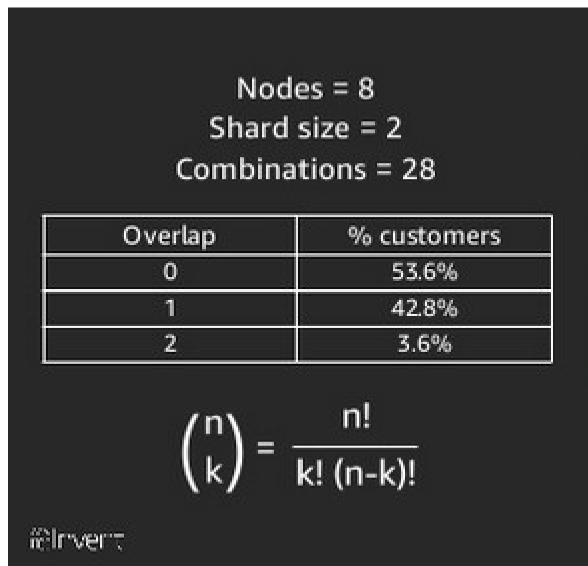


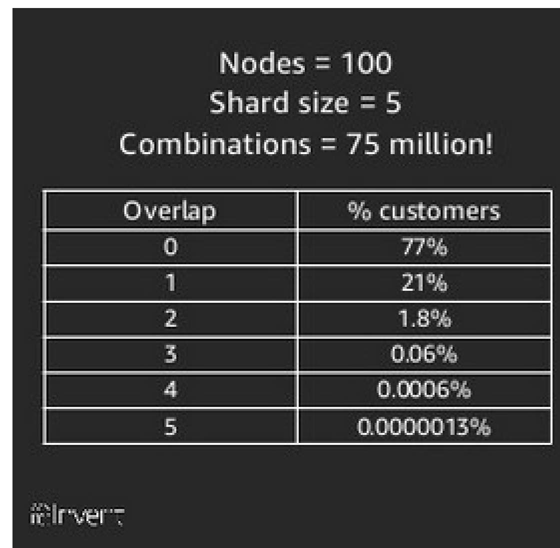




Peter Voss, AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338) https://www.youtube.com/watch?v=swQbA4zub20&feature=youtu.be

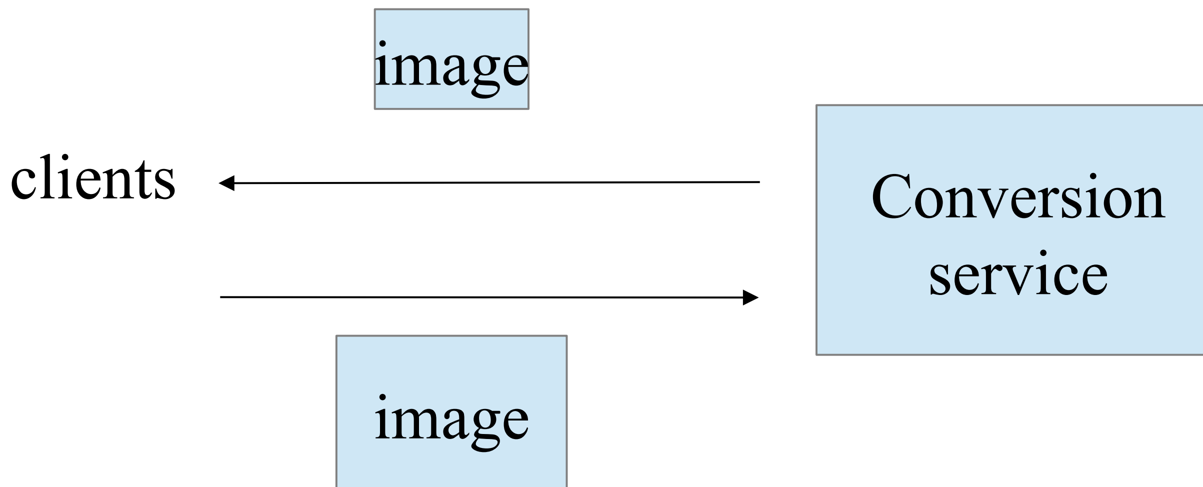
Blast Reduction V: Shuffle Sharding





Peter Voss, AWS re:Invent 2018: How AWS Minimizes the Blast Radius of Failures (ARC338) https://www.youtube.com/watch?v=swQbA4zub20&feature=youtu.be

Exercise: Image Conversion Service



Users: 100,000, requests: 10000/sec max. Availability: 99.9, Latency: 99th percentile <500ms with images <1MB (example from M.Cavage, ACM Queue) Authentication needed. Do some back-of-the-envelope calculations and draw a system architecture!

Resources (1)

- Desaster Inpol-neu, Christiane Schulzki-Haddouti, C‘t 24/2001 pg. 108ff. A typical case of large-scale, top-notch IT-project gone foul.
- Darrel Ince, Developing Distributed and e-commerce Applications. A very good introduction to all the topics necessary for building real-world apps. Still rather thin. The content and style comes close to what is covered in this lecture.
- David Purcell, Moving to a cluster... www.sys-con.com/story/print.cfm?storyid=47354
- IBM Websphere clustering redpaper on www.redbooks.ibm.com
- Luiz Andre Barroso et.al, Web Search for a planet: the google clustering architecture. Describes an architecture optimized for read/search access and not the typical transaction processes. Compare the machine types and numbers with a large web shop.
- Sing Li, High-impact Web tier clustering, Part 1 and 2: Scaling Web services using Java Groups etc. (www.ibm.com/developerworks)
- Thomas Smits, Unbreakable Java – A java server that never goes down. Describes SAPs approach for creating reliable Java VM environments by separating session state from VM using shared memory technology. Also processes are separated from VMs

- Don’t Settle for Eventual: Scalable Causal Consistency for Wide-Area

Resources (2)

- L.Reith, Concept of data distribution for worldwide distributed services in service -oriented architectures, HDM/DaimlerChrysler 2007.
- http://jroller.com/page/ rolsen?entry=building_a_dsl_in_ruby1
- James Hamilton, http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/
- Steve Souder at Velocity:High Performance Web Sites: 14 Rules for Faster Loading Pages http://stevesouders.com/docs/velocity-20090622.ppt
-