Lecture on

Distributed Services and Algorithms Part One

Without infrastructure there are no stepping stones to greatness. (James Fallows)

Prof. Walter Kriha

Computer Science and Media Faculty

HdM Stuttgart

Overview

- Distributed Services
    - Replication, Availability and Fault-Tolerance
    - Global Server Load Balancing for PoPs
- Typical Cluster Services
    - Fail-over and Load-Balancing
    - Directory Services
    - Cluster Scheduler (neu)
- Distributed Operating Systems
- Example Services and Algorithms
    - Distributed File System with replication and map/reduce
    - Distributed (streaming) Log
    - Distributed Cache with consistent hashing
    - Distributed DB with sharding/partitioning functions
    - Distributed Messaging with event notification and gossip.

What is a Distributed Service?

A function provided to applications by a distributed middleware with:

- high scalability

- high availability

Please NOTE: This is different from application functions implemented as services (e.g. micro-services, SOA, web-services etc.). Here we talk about general services needed for a distributed system to function properly.

Services and Instances

"Distributed systems are composed of services like applications, databases, caches, etc. Services are composed of instances or nodes—individually addressable hosts, either physical or virtual. The key observation is that, conceptually, the unit of interaction lies at the service level, not the instance level. We don't care about which database server we interact with, we just want to talk to a database server (or perhaps multiple). We're concerned with logical groups of nodes."

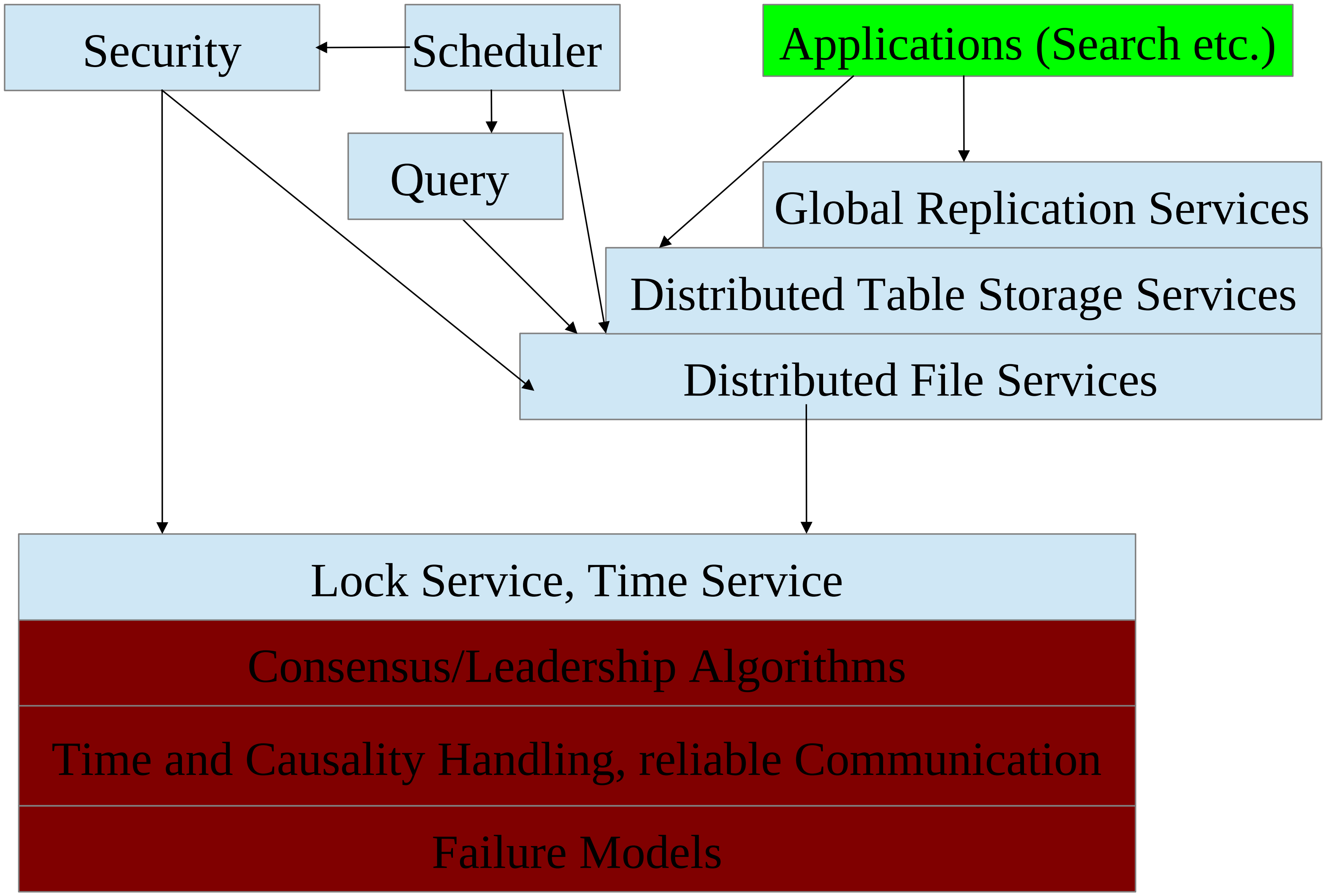
Tylor Treat, Iris Decentralized Cloud Messaging, bravenewgeek.com

Core Distributed Services

- Finding Things (Name Service, Registry, Search)
- Storing Things (all sorts of DBs, data grids, block storage etc.)
- Events Handling and asynchronous processing (Queues)
- Load Balancing and Failover
- Caching Service
- Locking Things and preventing concurrent access (Lock service)
- Request scheduling and control (request multiplexing)
- Time handling
- Providing atomic transactions (consistency and persistence)
- Replicating Things (NoSQL DBs)
- Object handling (Lifecycle services for creation, destruction, relationship service)

Etc.

Distributed Services Hierarchy

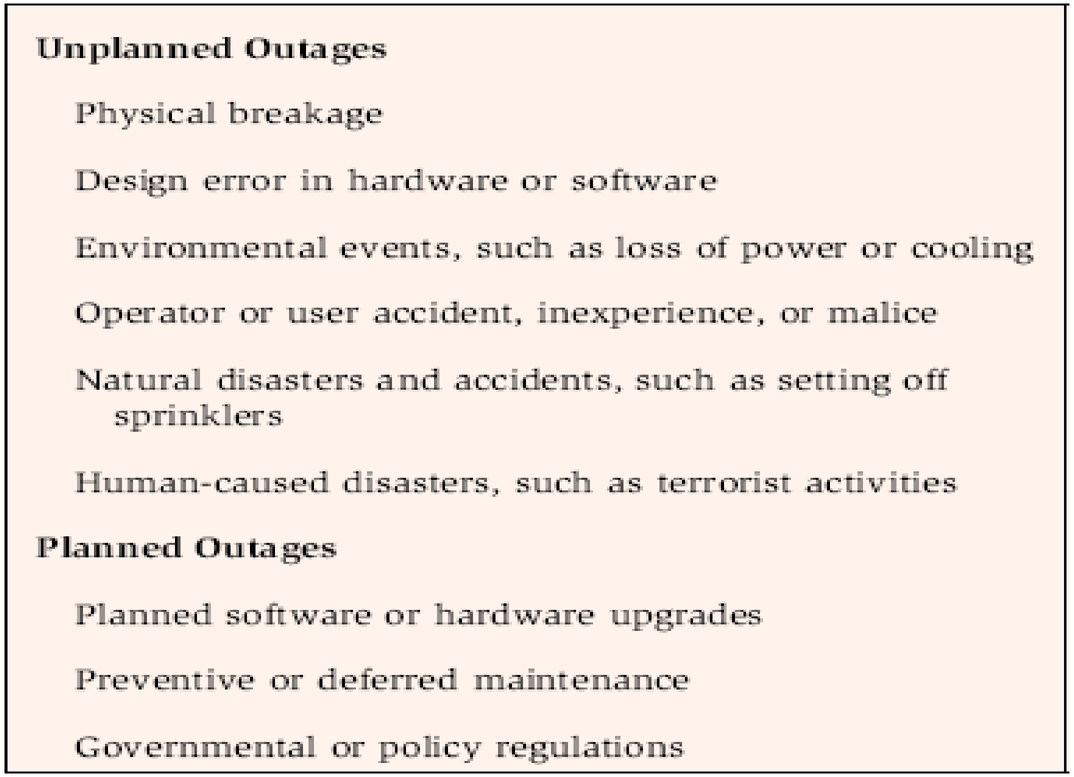

Security
Scheduler
Applications (Search etc.)
Query
Global Replication Services
Distributed Table Storage Services
Distributed File Services
Lock Service, Time Service
Consensus/Leadership Algorithms
Time and Causality Handling, reliable Communication
Failure Models

Availability: 99.999...

$$\textbf{Availability (ratio)} = \frac{\textbf{agreed upon uptime – downtime (planned or unplanned)}}{\textbf{agreed upon uptime}}$$

Continuous availabilty does not allow planned downtime

Outages



Unplanned Outages

- Physical breakage
- Design error in hardware or software
- Environmental events, such as loss of power or cooling
- Operator or user accident, inexperience, or malice
- Natural disasters and accidents, such as setting off sprinklers
- Human-caused disasters, such as terrorist activities

Planned Outages

- Planned software or hardware upgrades
- Preventive or deferred maintenance
- Governmental or policy regulations

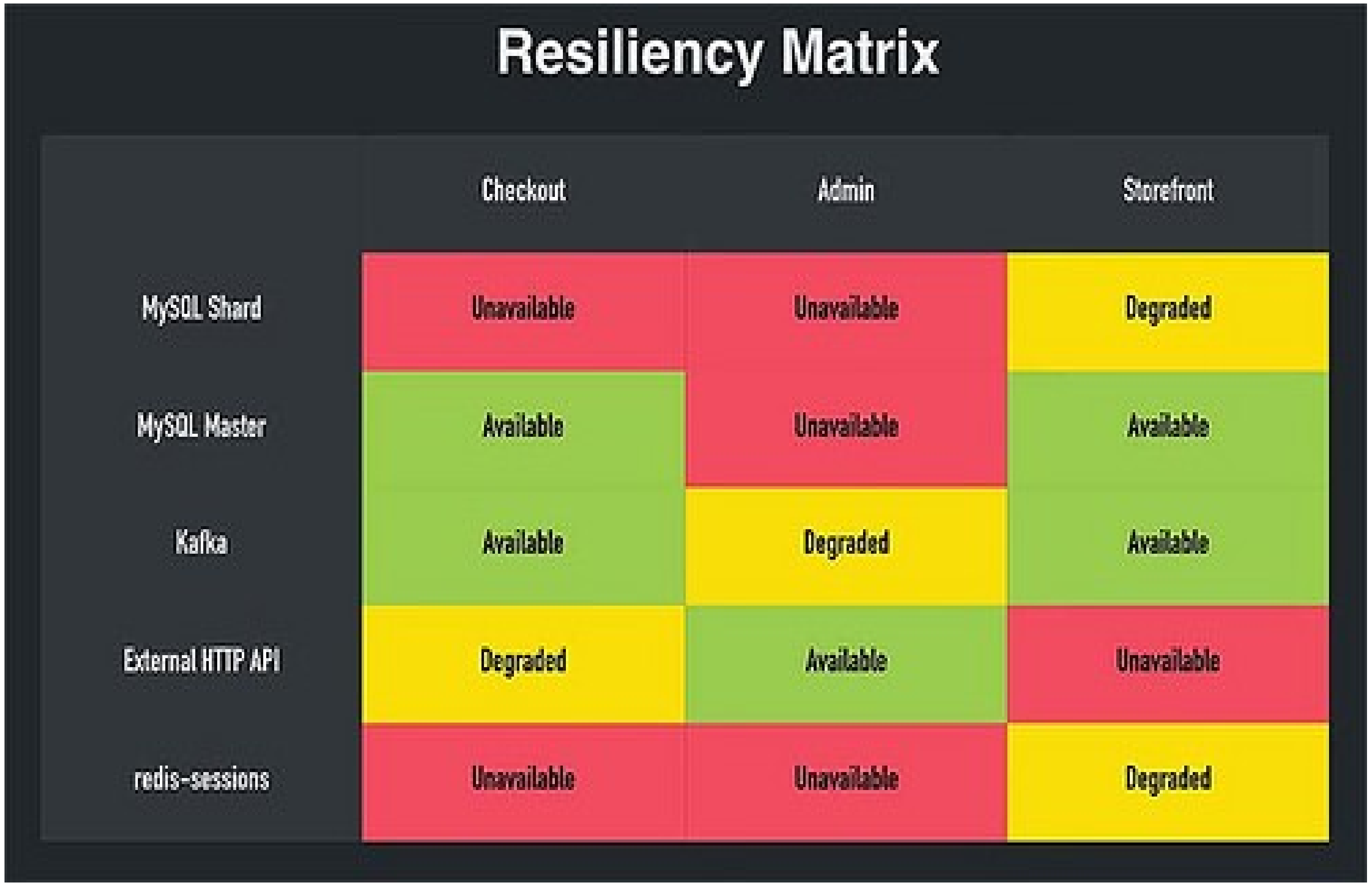
Morrill et.al, Achieving continuous availability of IBM systems infrastructures, IBM Systems Journal Vol. 47, Nr. 4, pg. 496, 2008

Typical Hardware Failures at Google

Typical first year for a new cluster:
~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 network rewiring (rolling ~5% of machines down over 2-day span)
~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
~5 racks go wonky (40-80 machines see 50% packet loss)
~8 network maintenances (4 might cause ~30-minute random connectivity losses)
~12 router reloads (takes out DNS and external vips for a couple minutes)
~3 router failures (have to immediately pull traffic for an hour)
~dozens of minor 30-second blips for dns
~1000 individual machine failures
~thousands of hard drive failuresslow disks, bad memory, misconfigured machines, flaky machines, etc.

From: Jeff Dean, Handling Large Datasets at Google, Current Systems and Future Directions

Resilience Matrix at Shopify

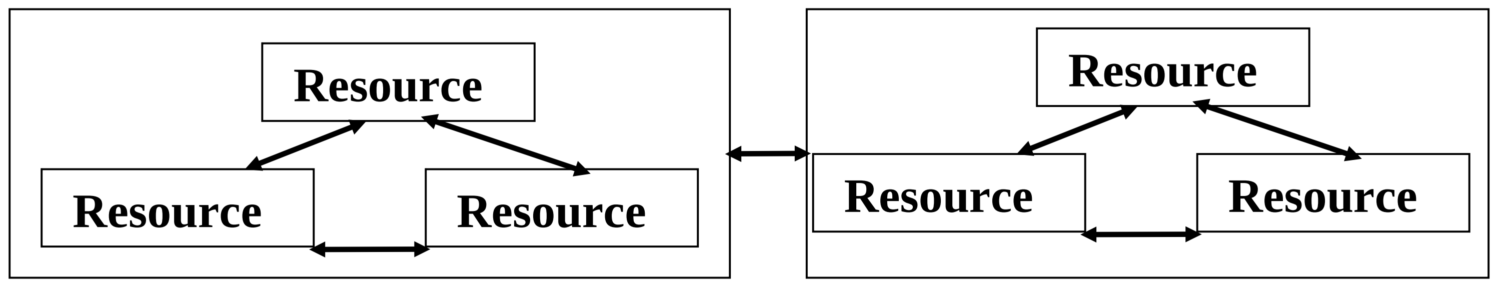


Resiliency Matrix

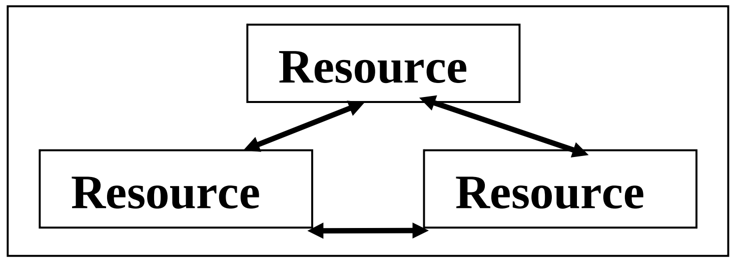
	Checkout	Admin	Storefront
MySQL Shard	Unavailable	Unavailable	Degraded
MySQL Master	Available	Unavailable	Available
Kafka	Available	Degraded	Available
External HTTP API	Degraded	Available	Unavailable
redis-sessions	Unavailable	Unavailable	Degraded

From: DockerCon 2015: Resilient Routing and Discovery by Simon Hørup Eskildsen. Take out SPOFs step by step. Reduce Coupling and failure propagation between business areas.

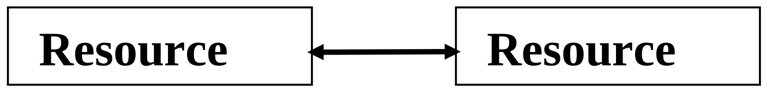
Availability through Redundancy



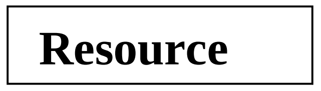
Multi-site data center, Disaster Recovery, Scale



Cluster level, HA, CA, CO, scalability, replication. Quorum algorithms need more machines

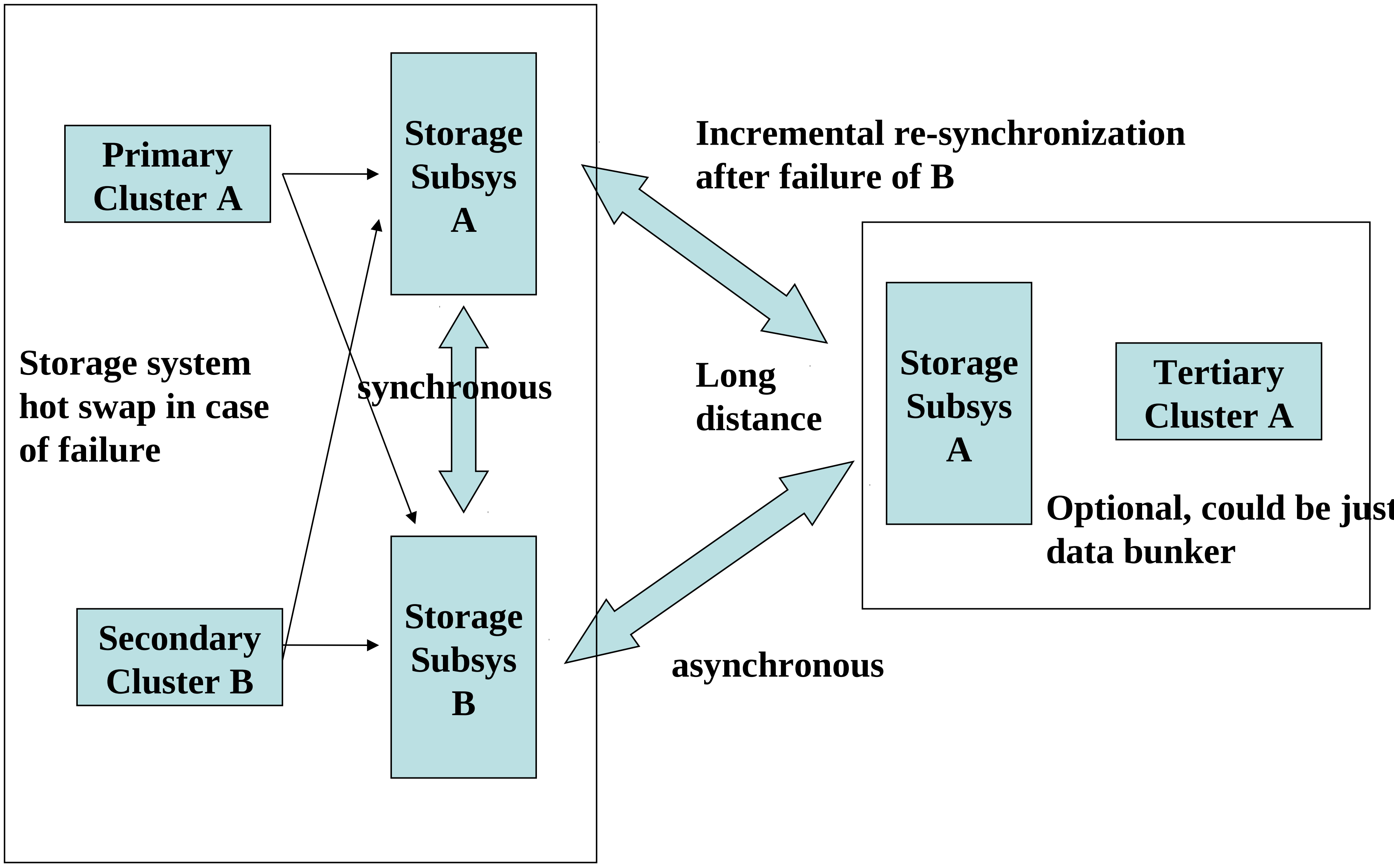


HA, CA, CO possible. Load distribution. overload in case of failure



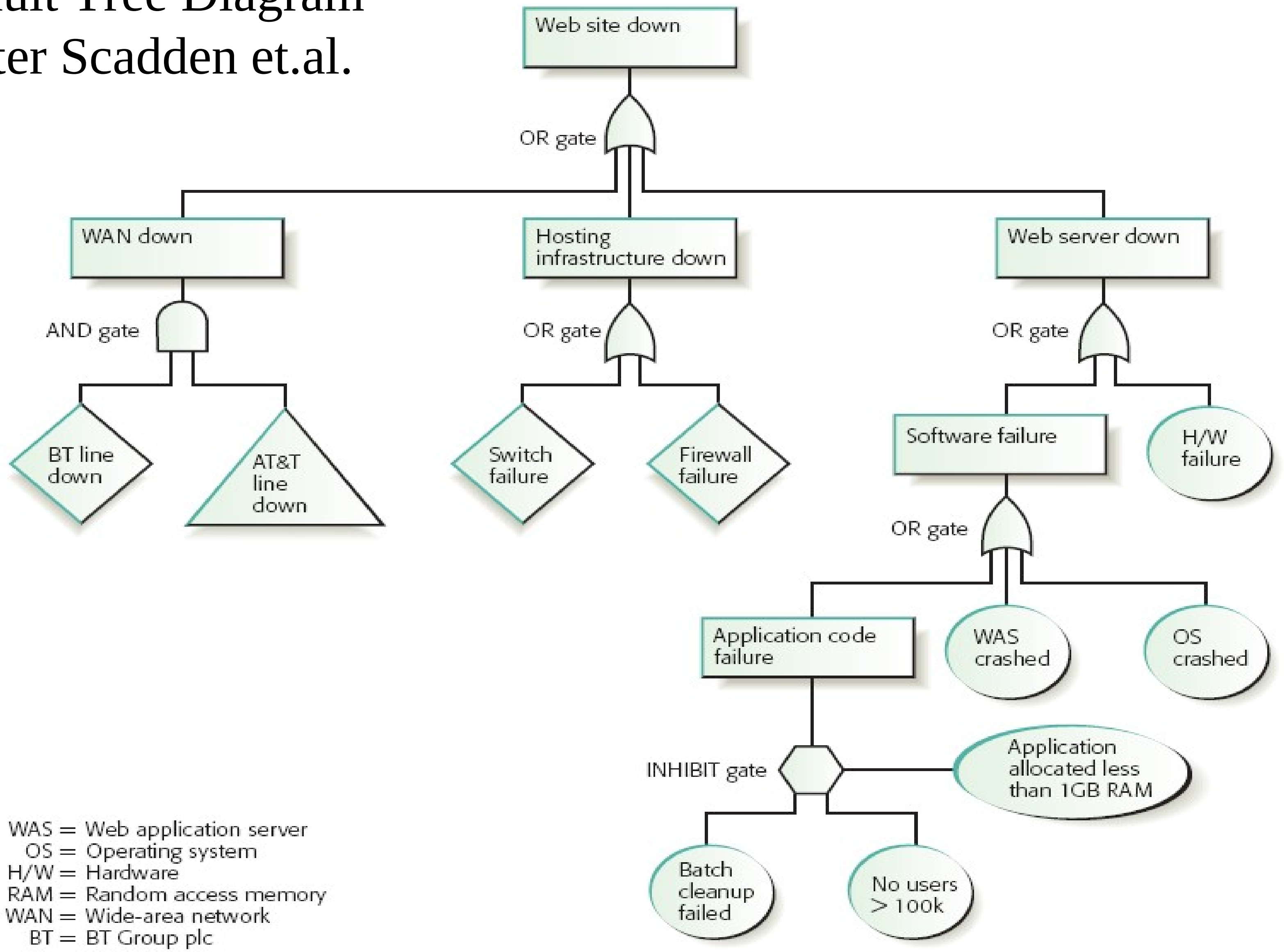
SPOF, easy update, maintenance problems, simple reliability, CO? Vertical scalability

3-copy Disaster Recovery Solution



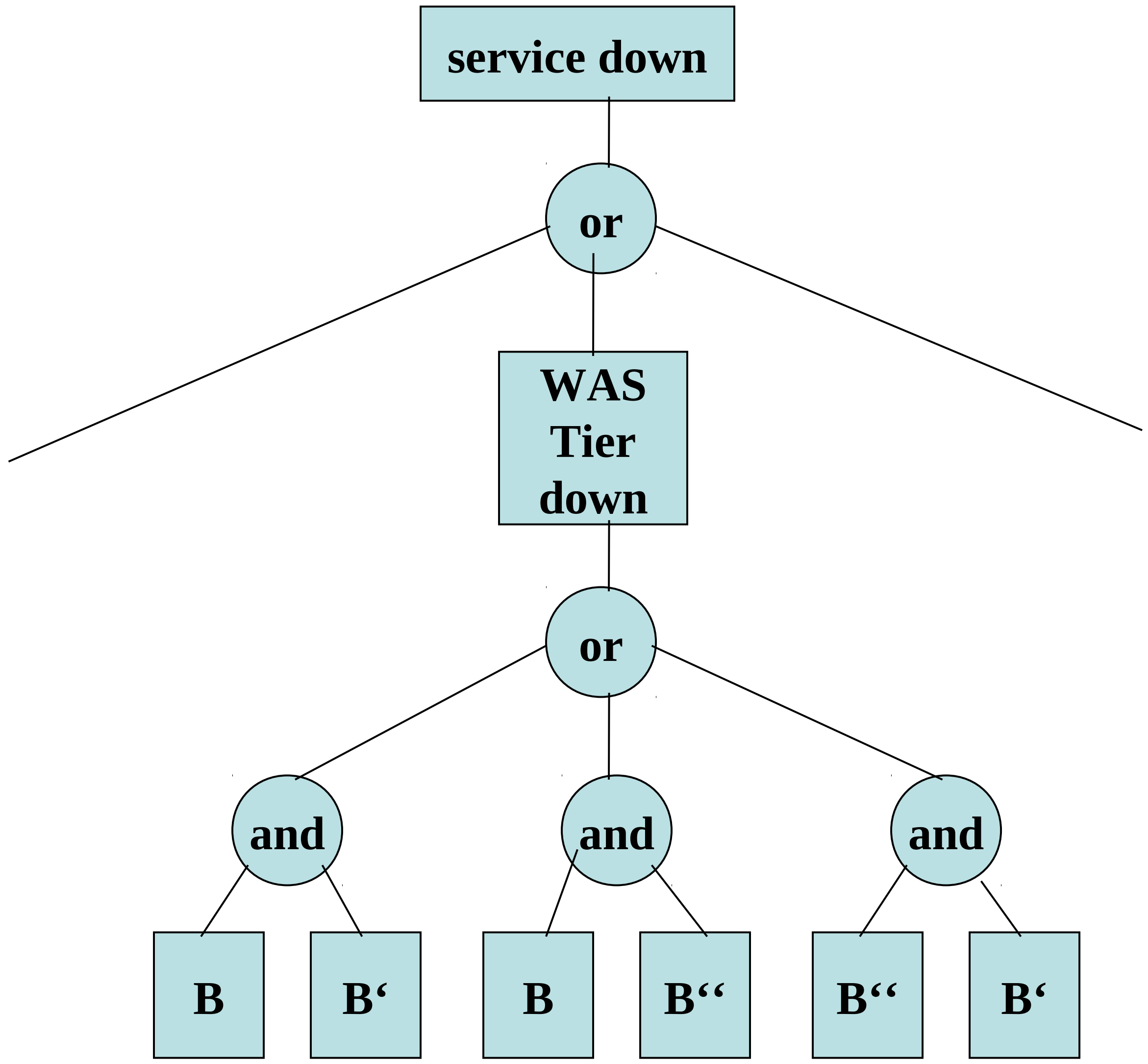
After: Clitherow et.al.

Fault Tree Diagram after Scadden et.al.



WAS =	Web application server
OS =	Operating system
H/W =	Hardware
RAM =	Random access memory
WAN =	Wide-area network
BT =	BT Group plc

Figure 2
Fault tree for Web-based system


service down
or
WAS Tier down
or
and
and
and
B
B‘
B
B‘‘
B‘‘
B‘

Reliability Block Diagram

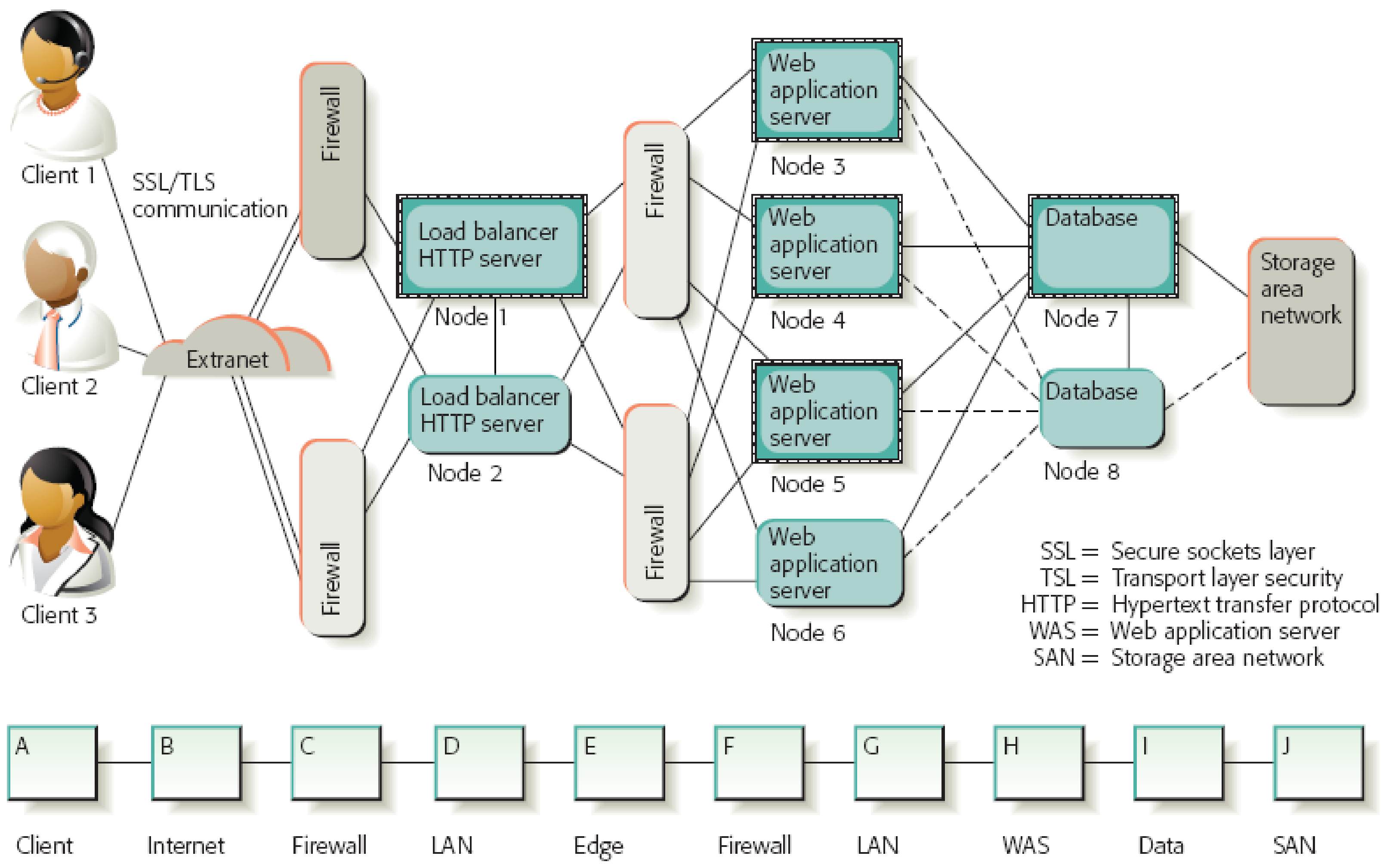
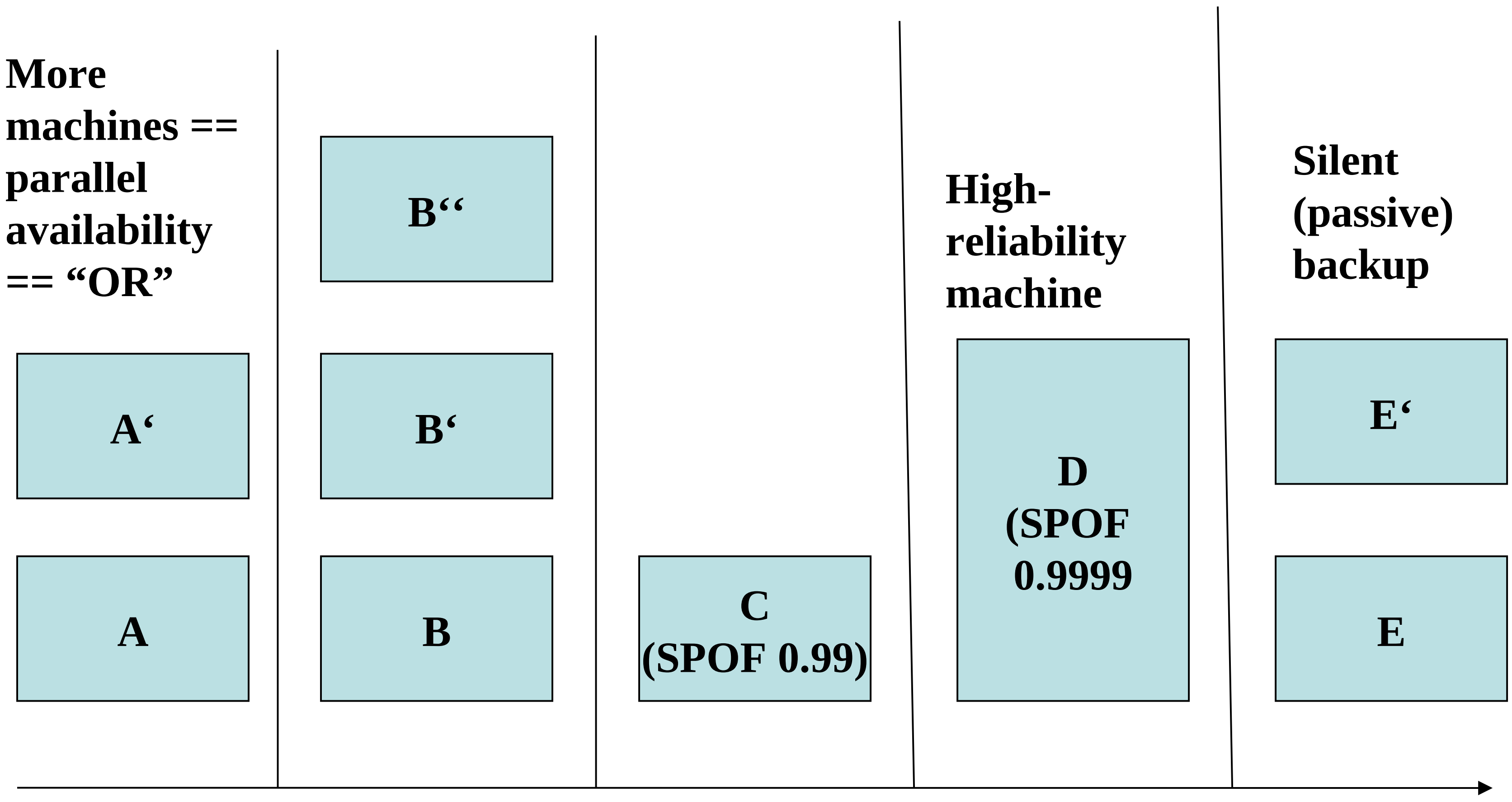


Figure 3
A three-tier server configuration for a Web-based application and its RBD

Source: D. Bailey et.al.

(B & B‘) || (B & B‘‘) || ….
(OR mixed with AND)



Serial Availability: ALL tiers must be available („AND“)

Serial chain of ALL needed components: multiplying availabilities gives less overall availability or: the more chain members the higher individual availability needs to be

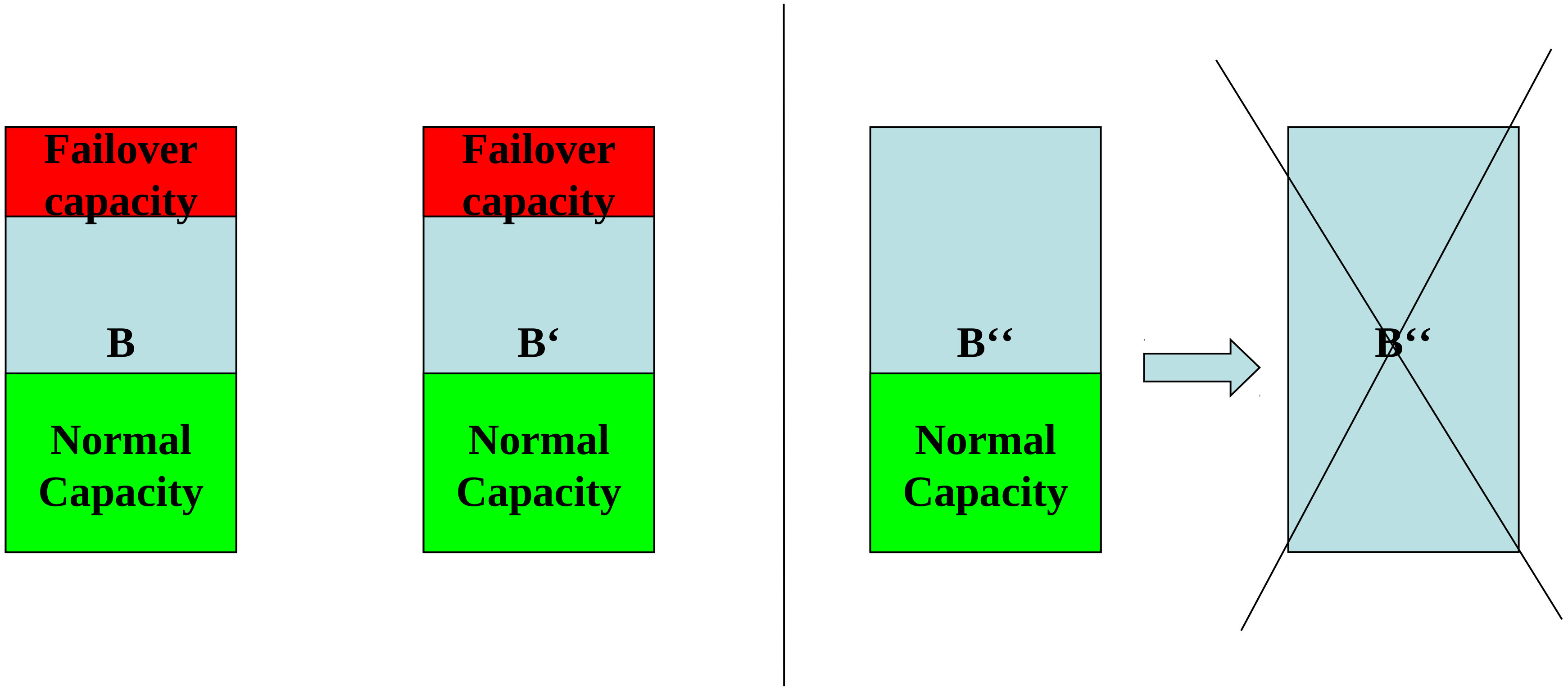
$$SerialAvailability = \prod_{i=1}^{N} ComponentAvailability_{(i)},$$

Redundant, parallel components where only ONE needs to be up: multiply unavailabilities and subtract from 1.

$$ParallelAvailability = 1 - \left[\prod_{i=1}^{N} \left(1 - ComponentAvailability_{(i)}\right) \right].$$

From: Scadden et.al., pg. 537

Pairwise Replication Problems

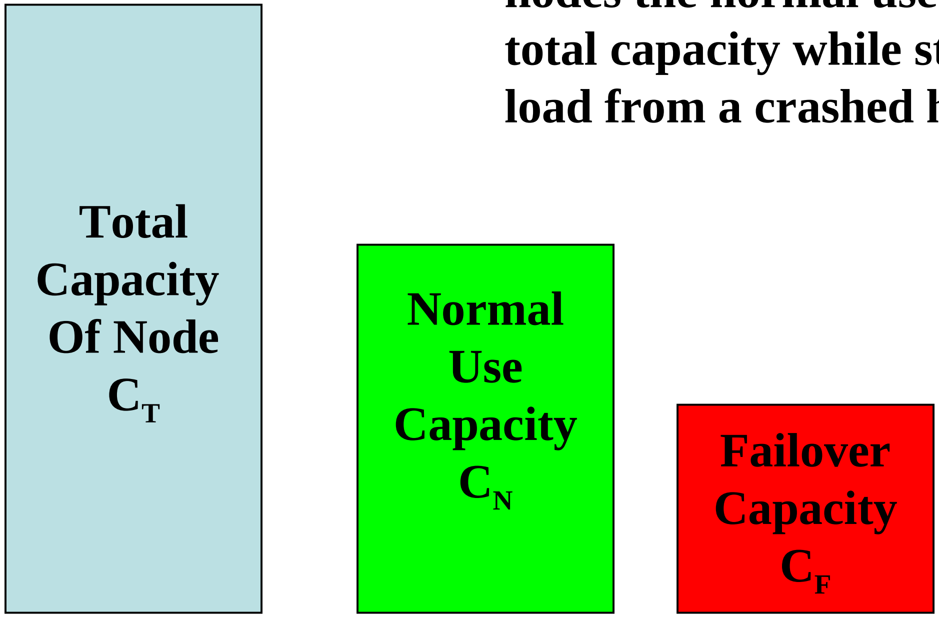


Effect of number of nodes on wasted capacity (assuming homogeneous hardware and no sticky sessions bound to special hosts aka session pairing)

CN + CF = CT

CF = CT/(n – 1) (n = number of nodes)

CN = CT – (CT/(n-1)) with growing number of nodes the normal use capacity gets closer to the total capacity while still allowing failover of the load from a crashed host



Multi-Homed-Sites

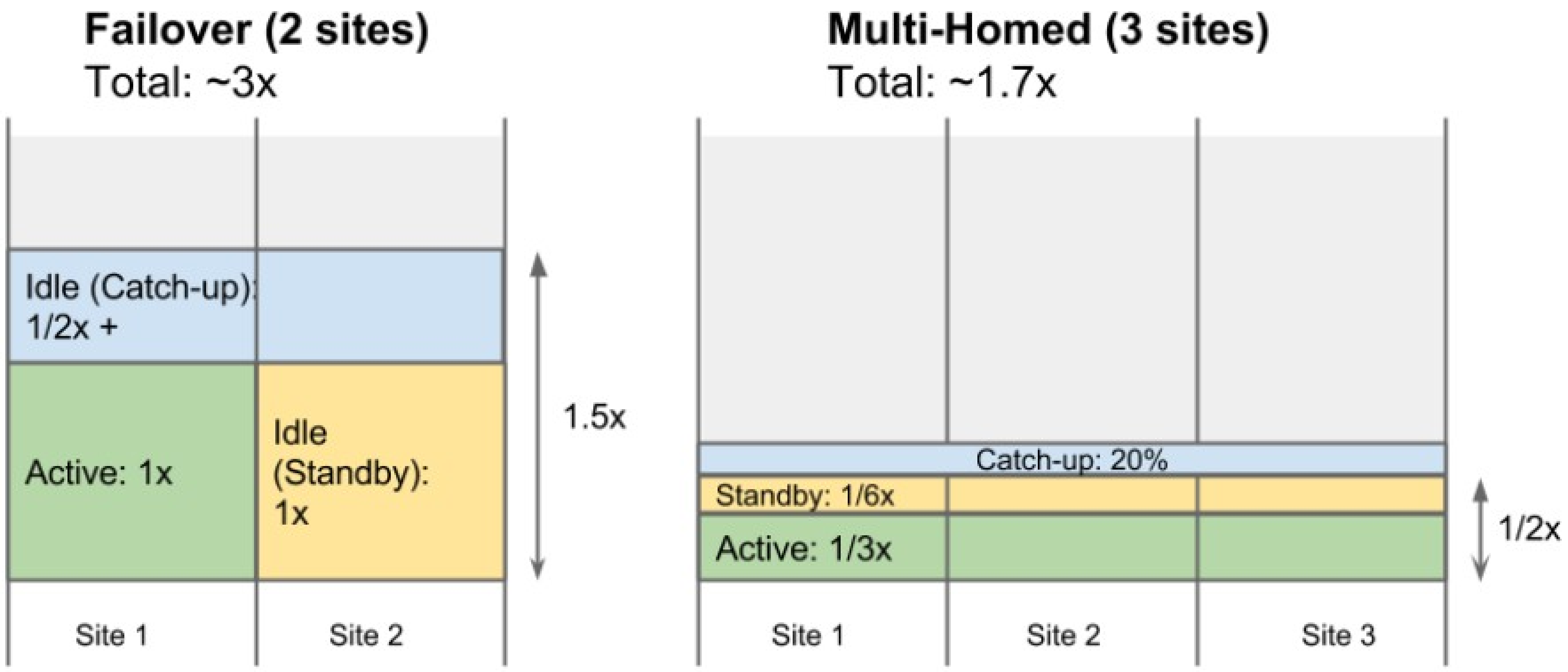


Fig. 1. Resource Cost for failover-based and multi-homed systems

From: High-Availability at Massive Scale: Building Google's Data Infrastructure for Ads
Ashish Gupta and Je Shute, Google
https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44686.pdf

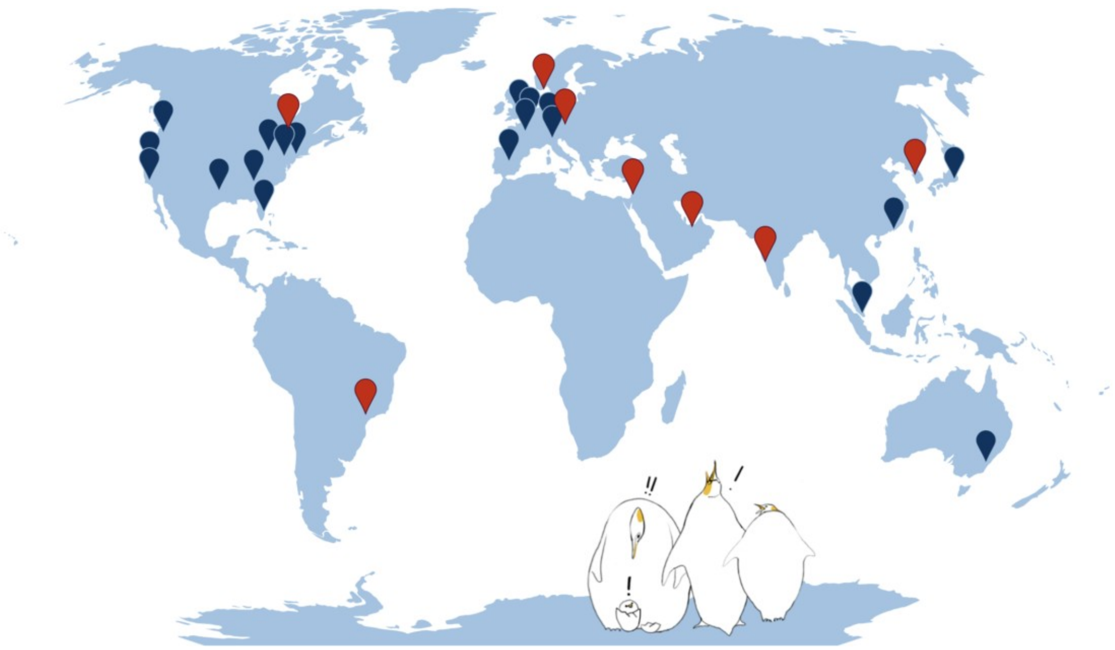
Global Server Load Balancing (GSLB)

- DNS Round Robin
- BGP Anycast
- Geo-DNS
- Real User Measurements (RUM)

https://blogs.dropbox.com/tech/2018/10/dropbox-traffic-infrastructure-edge-network/

https://opensource.com/article/18/10/internet-scale-load-balancing

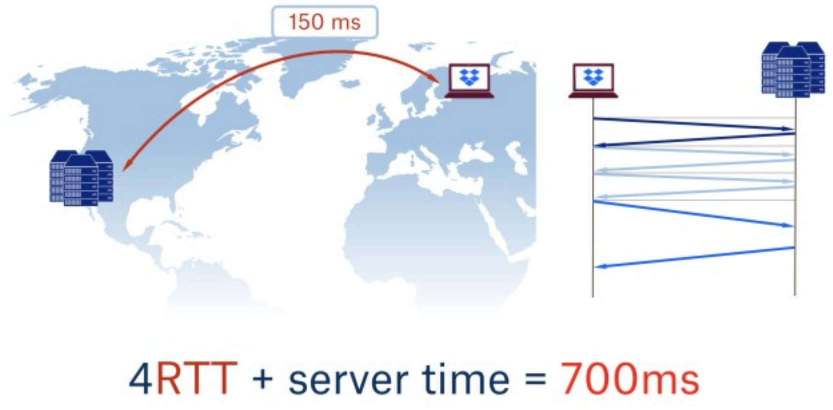
PoP (Edge) Selection at Dropbox

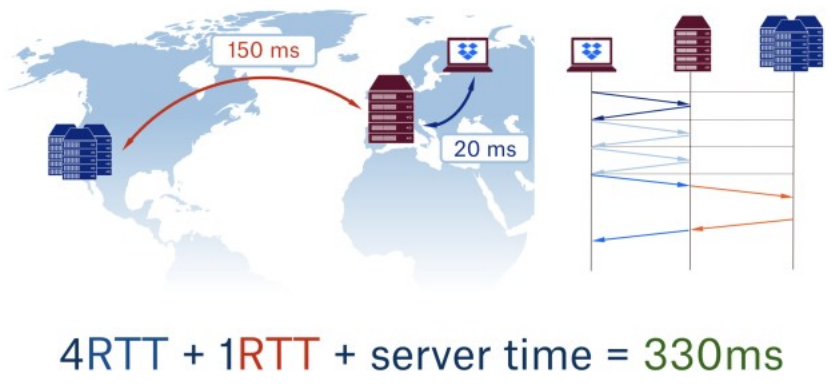


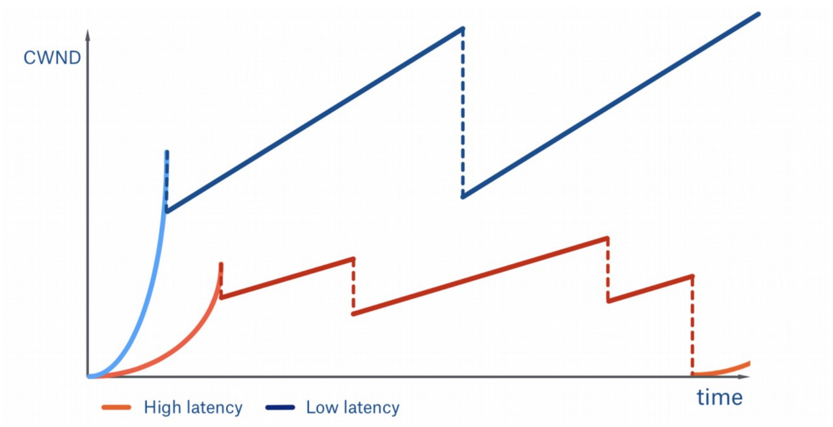
- “nearest to user” (public vs. internal net)
- datacenter load/fail/maintenance
- backbone capacity, peering connectivity, submarine cables,
- location w.resp. to all the other PoPs.
- population size, internet quirks

https://blogs.dropbox.com/tech/2018/10/dropbox-traffic-infrastructure-edge-network/

Living at the Edge..

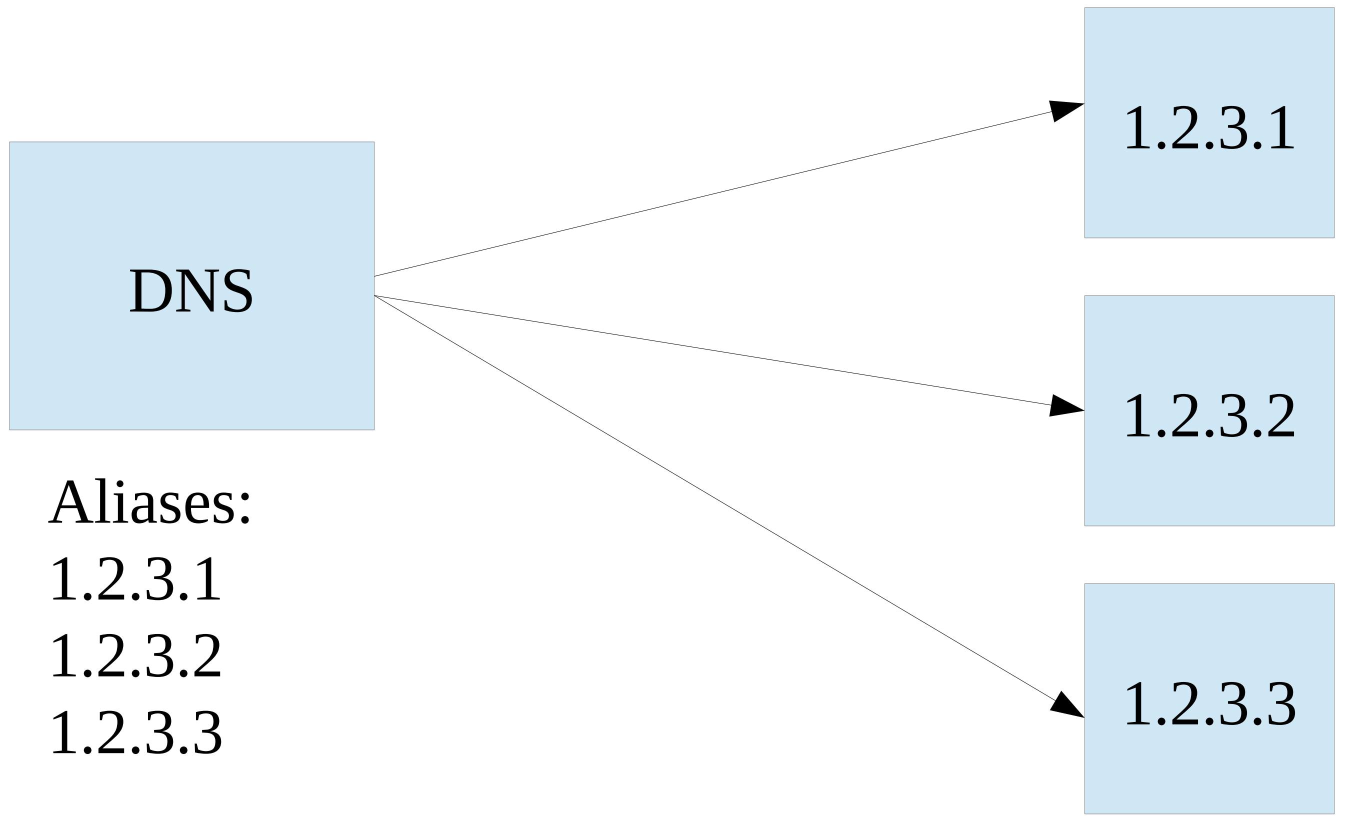






https://blogs.dropbox.com/tech/2018/10/dropbox-traffic-infrastructure-edge-network/

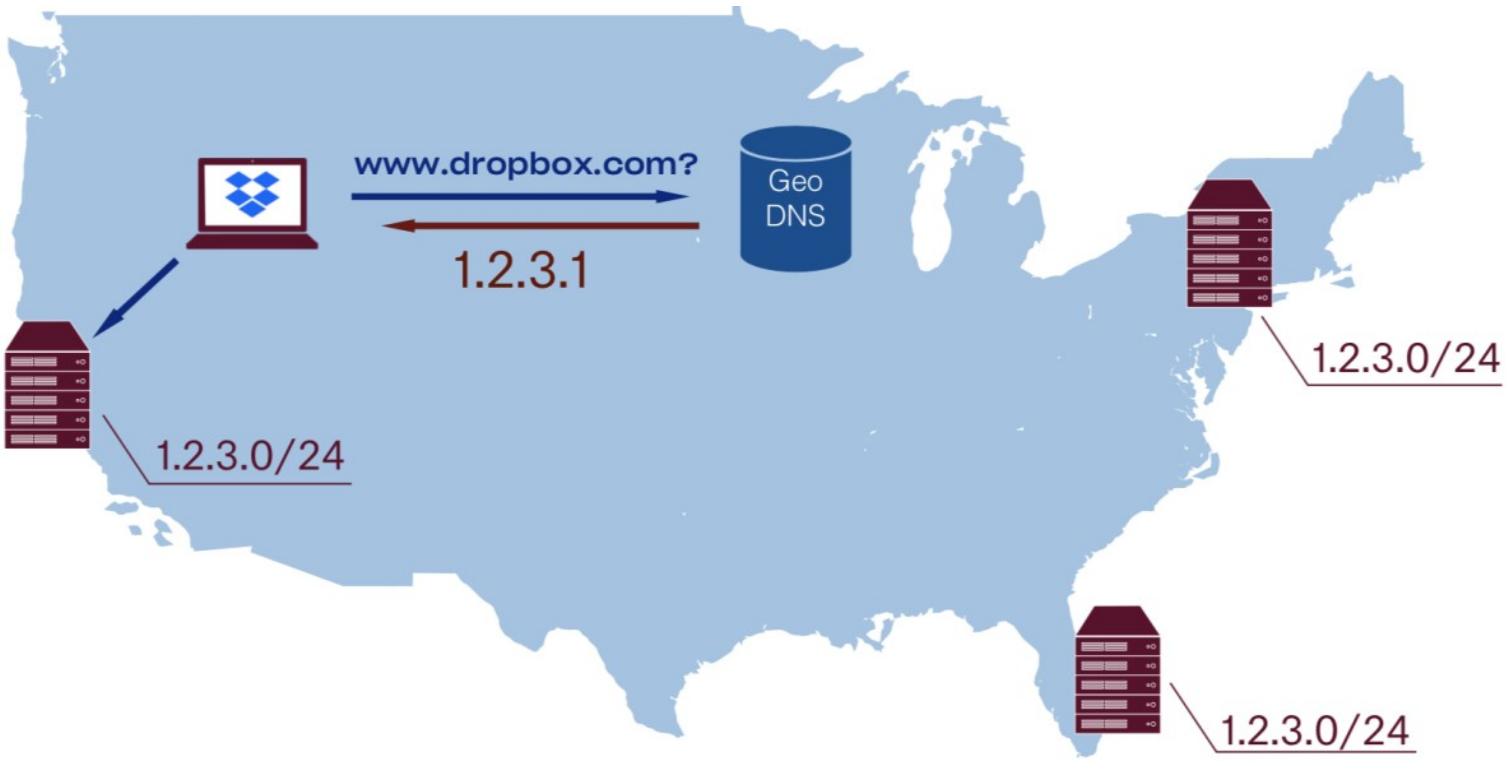
DNS Round Robin



Aliases:
1.2.3.1
1.2.3.2
1.2.3.3

While convenient, DNS allows little mitigation in case of problems like overload, fail, etc. Many clients disregard TTL settings and it takes approx. 15 min. to drain traffic to troubled servers.

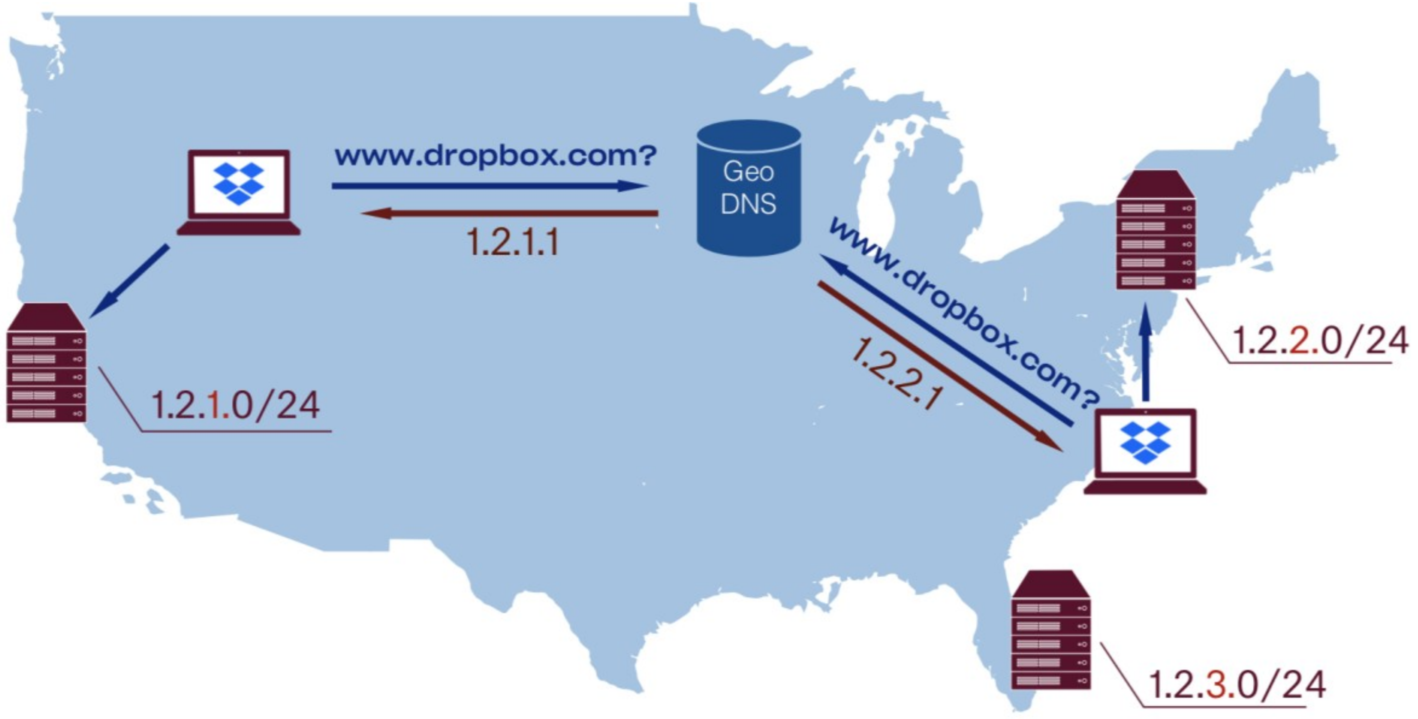
BGP Anycast (hops)



Several PoPs announce the same subnet. Internet does the rest. But: BGP does not know anything about link latency, throughput, packet loss, etc. With multiple routes to the destination, it just selects one with the least number of hops. Troubleshooting very demanding.

https://blogs.dropbox.com/tech/2018/10/dropbox-traffic-infrastructure-edge-network/

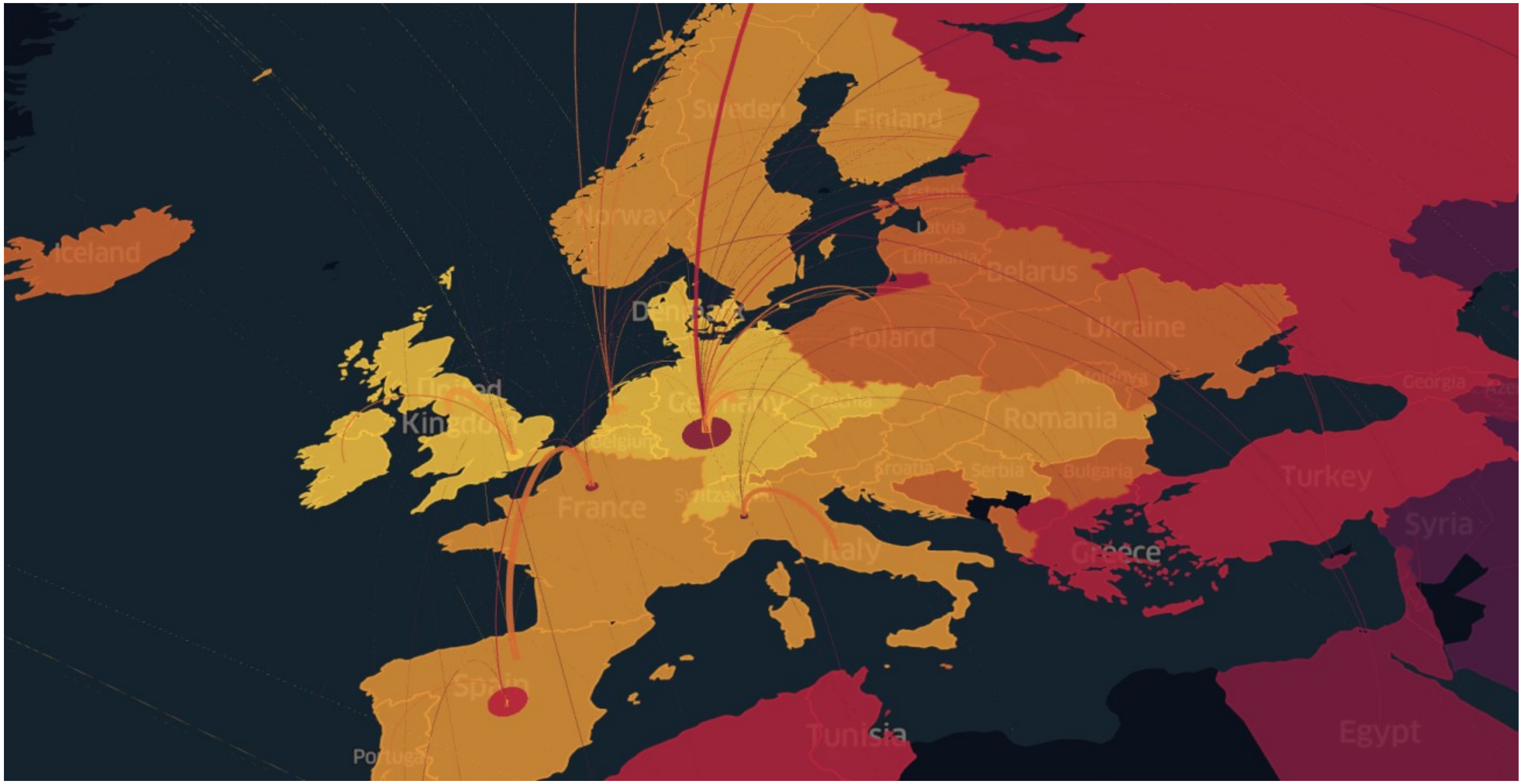
GeoDNS (proximity)



IP addresses are distributed based on geo-location. Relies on a DNS provider guessing user IP by their DNS resolver (or trust EDNS CS data), then guessing user location by their IP address, then approximate physical proximity to latency. Better troubleshooting, TTL is a lie..

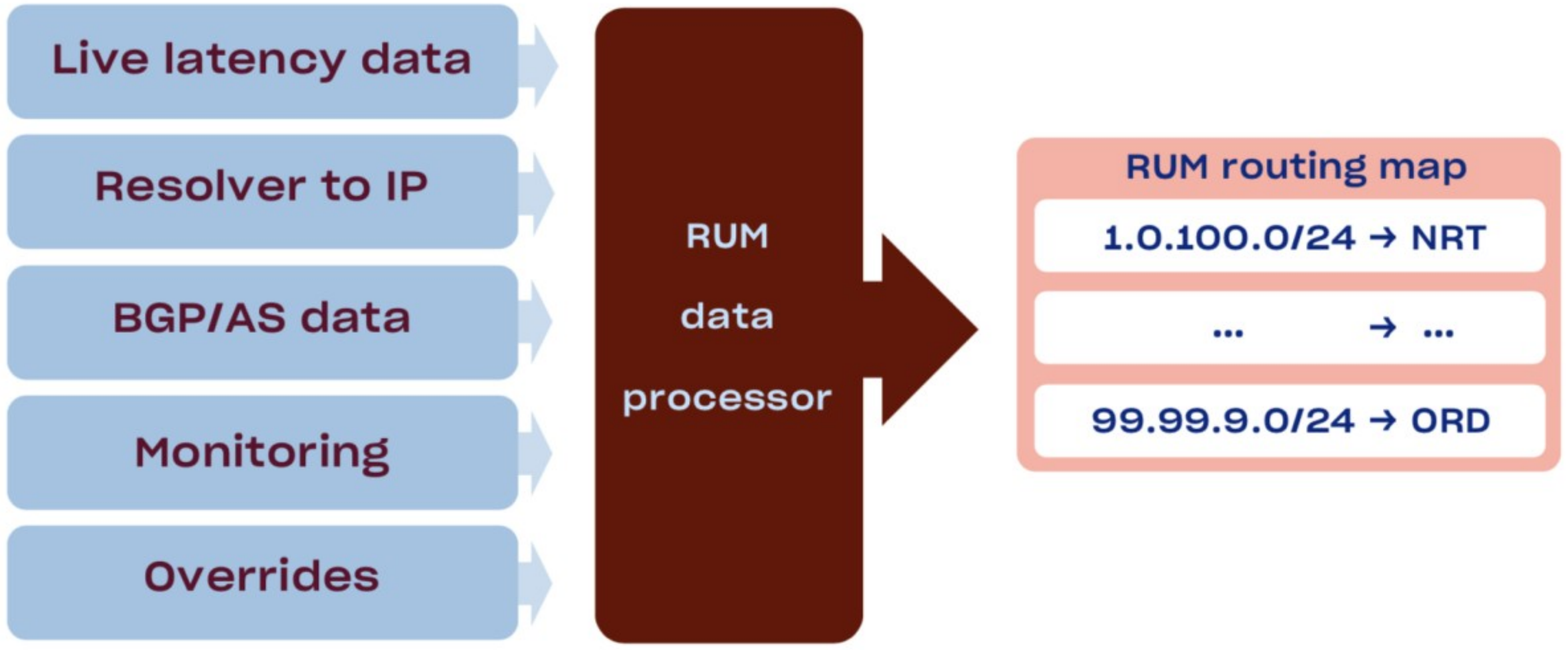
https://blogs.dropbox.com/tech/2018/10/dropbox-traffic-infrastructure-edge-network/

GeoDNS at Dropbox



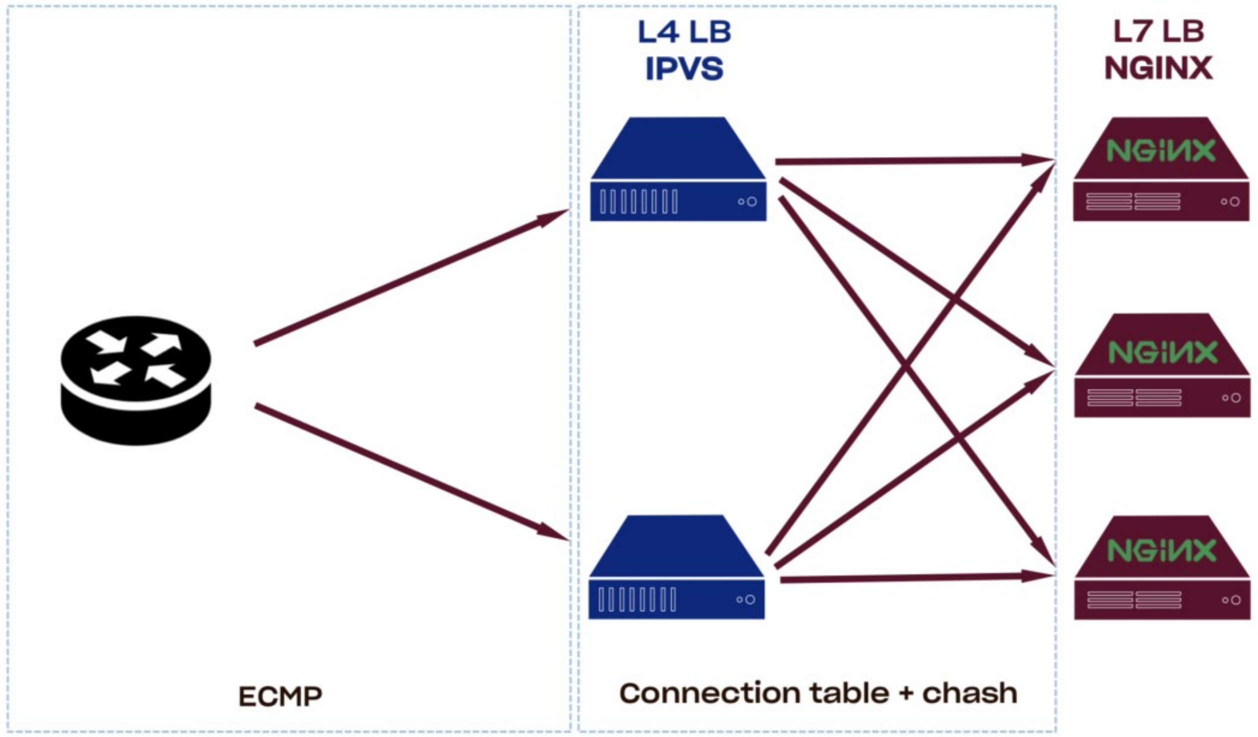
https://blogs.dropbox.com/tech/2018/10/dropbox-traffic-infrastructure-edge-network/

Real-User-Metrics (RUM, client subnet → map)



Embedded client code logs latencies to various DNS Servers and PoPs. This data is uploaded to DNS and compared to anycast and geodns data to create a special map.
https://blogs.dropbox.com/tech/2018/10/dropbox-traffic-infrastructure-edge-network/

Inside PoPs

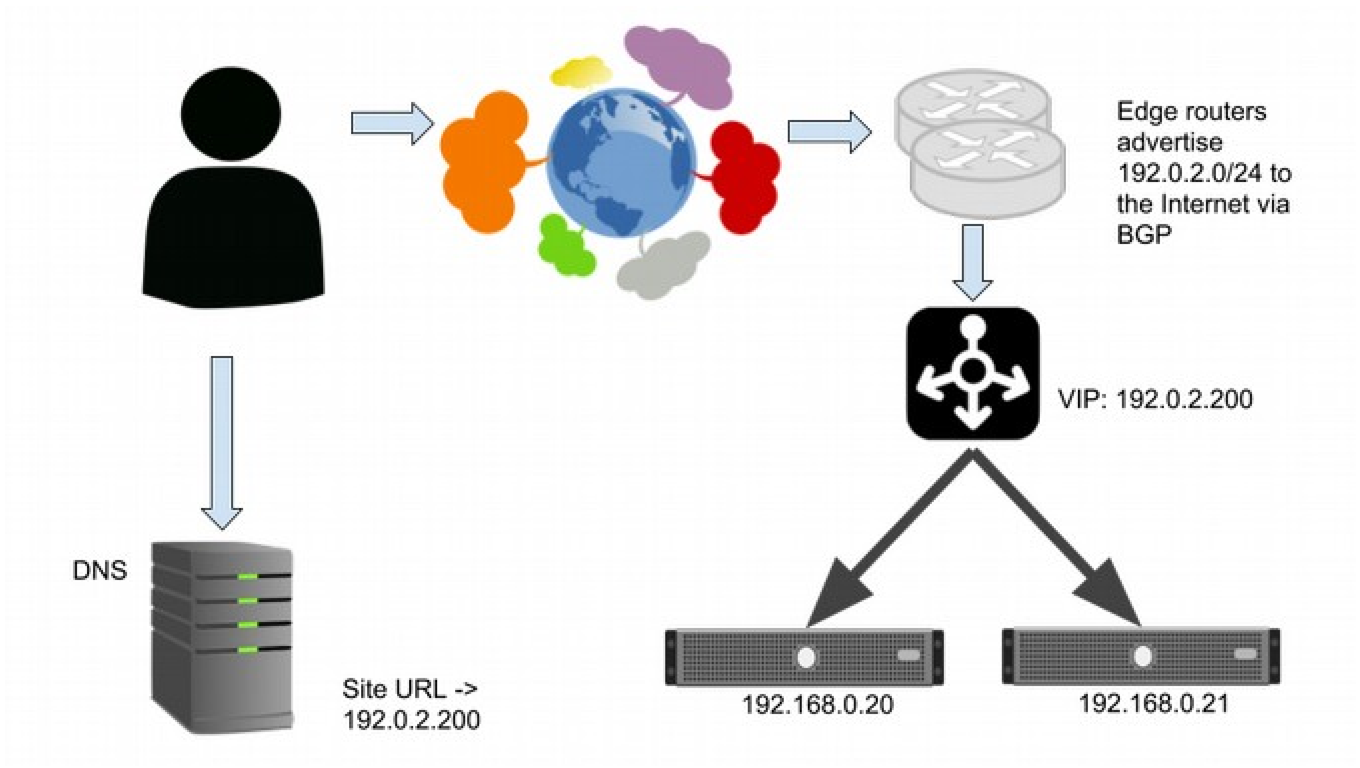


Public and private peering, kernel level routing, consistent hashing are used in typical PoPs. For a description of a software L4LB see the Maglev paper from Google.
https://blogs.dropbox.com/tech/2018/10/dropbox-traffic-infrastructure-edge-network/
https://blog.acolyer.org/2016/03/21/maglev-a-fast-and-reliable-software-network-load-balancer/

Typical Cluster Services

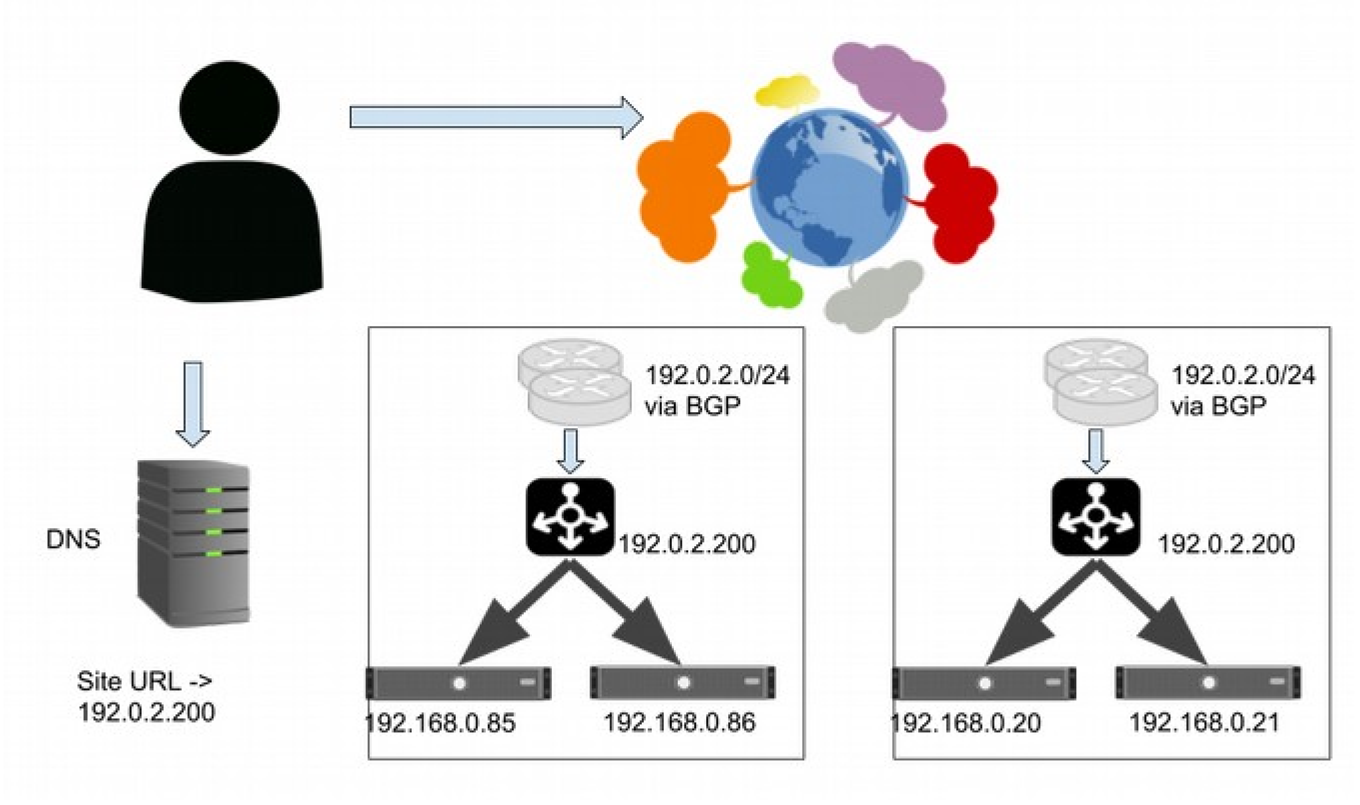
- Fail-over
- Load-Balancing
- Directory Services

Fail-Over with Virtual IP



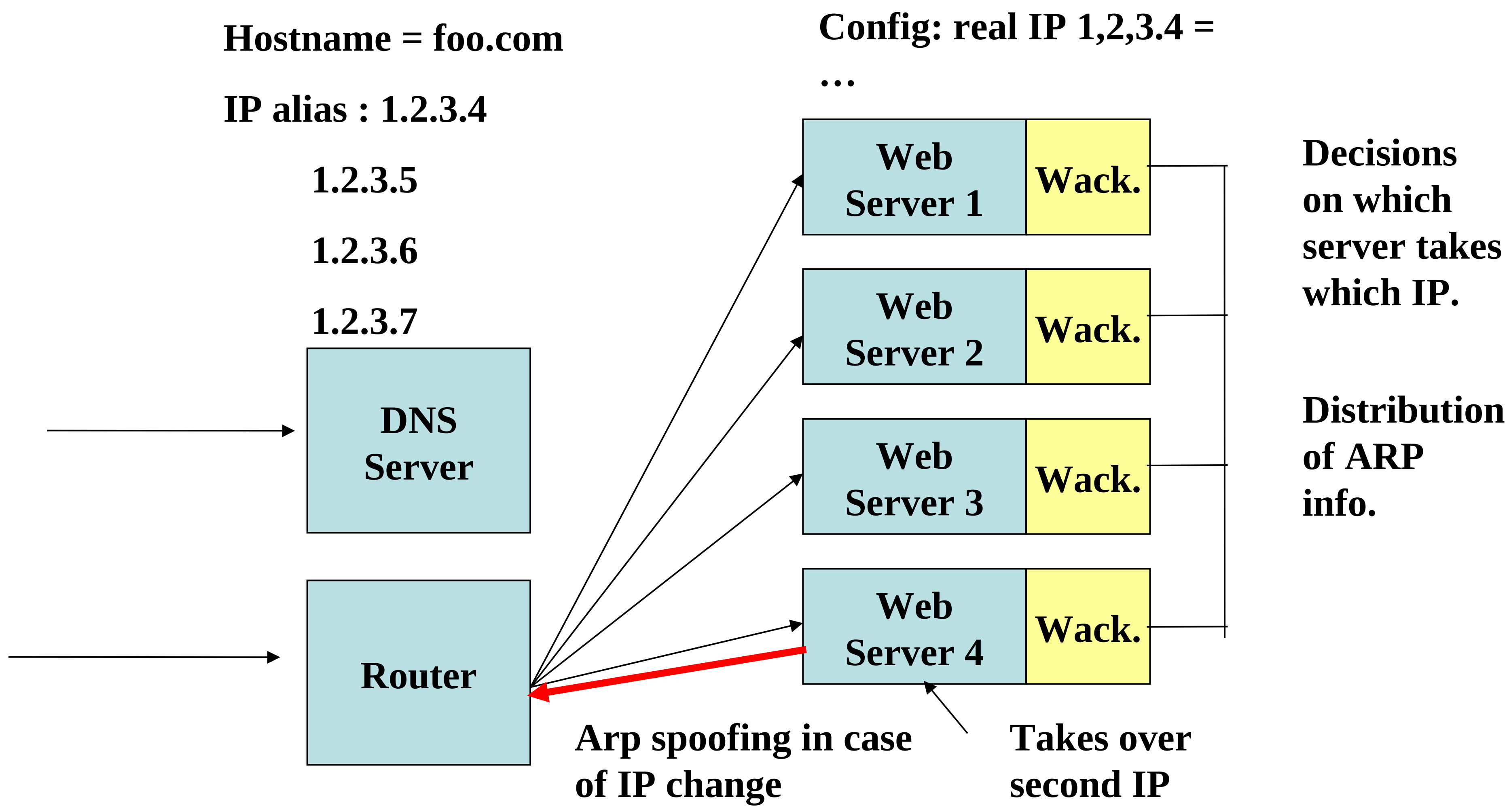
DNS points only a one VIP. In case of a server failure, client sessions to this server are lost but on reconnect clients can establish a new session. No DNS changes/flushes/timeouts involved.
https://opensource.com/article/18/10/internet-scale-load-balancing

Multi-Site Fail-Over with Virtual IP



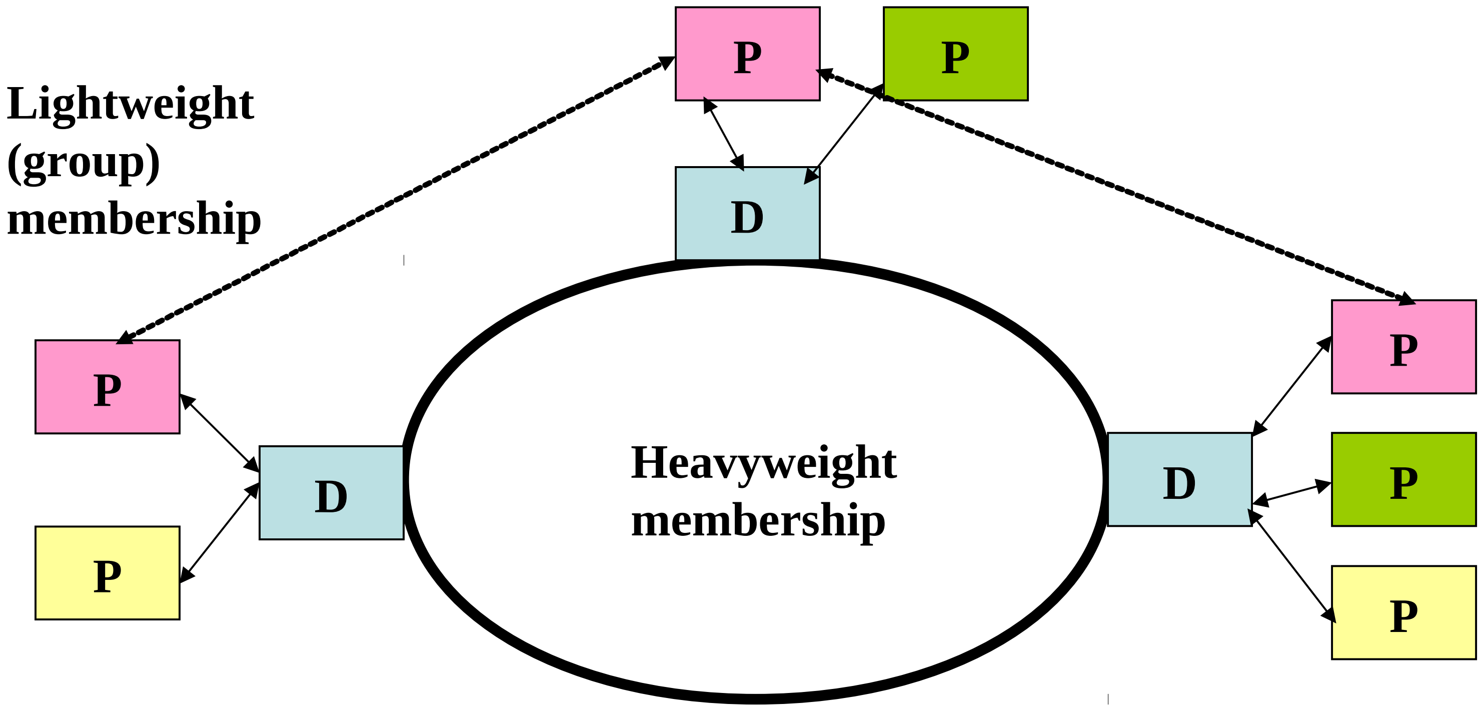
This is a combination of geo-aware DNS (with its problems) and the indirections of a LB/failover front-server which can re-route requests.
https://opensource.com/article/18/10/internet-scale-load-balancing

Peer2Peer HA Distributor for IP Fail-over



“wackamole” from Theo Schlossnagle

Group Communication Middleware



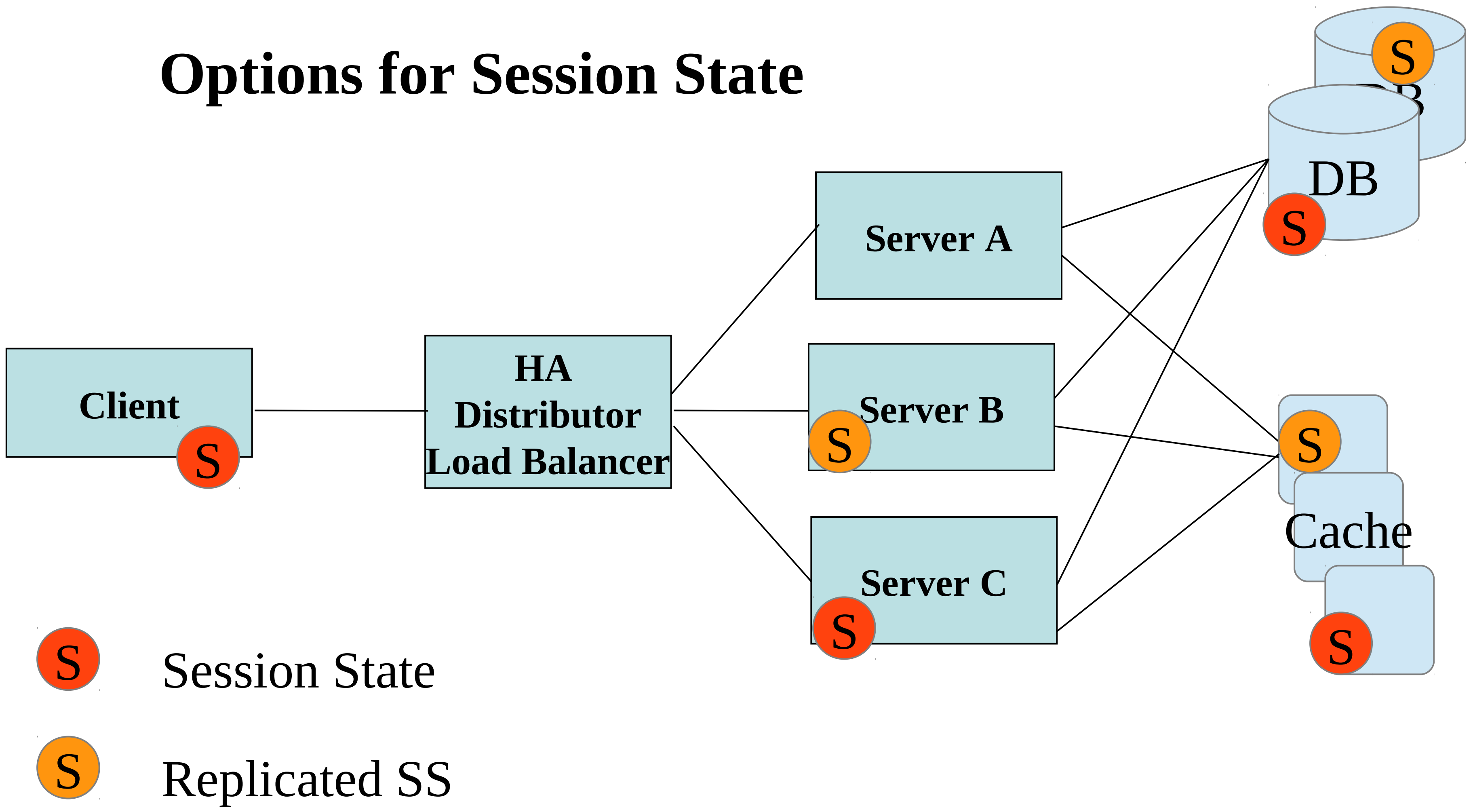
In the Spread group communication framework daemons control membership and protocol behavior (order, flow control). Messages are packed for throughput. Jgroups is another group membership library which could be used.

Fail-Over, Load-Balancing and Session State

1. “Sticky Sessions
2. Session Storage in DB
3. Session Storage in distributed cache

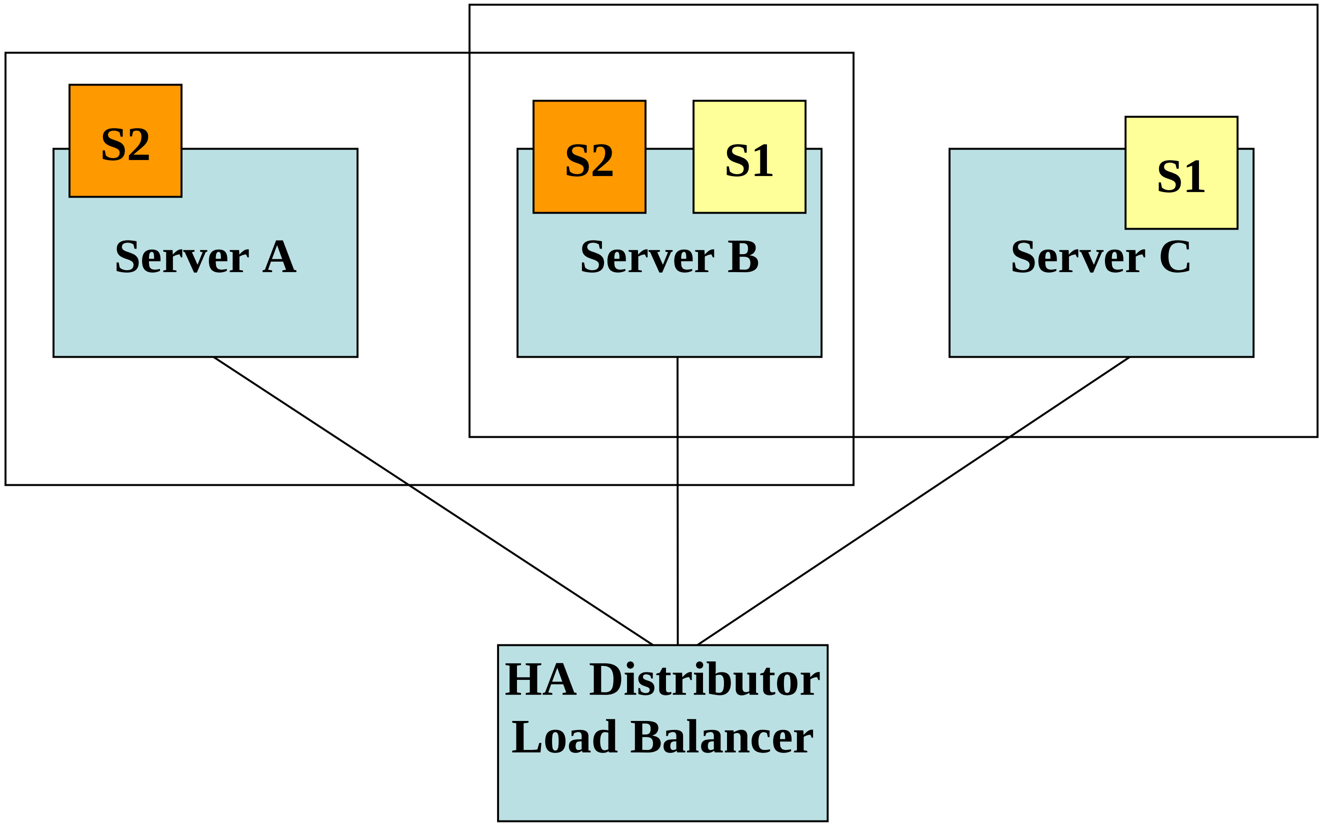
The location of session state determines your options with respect to fail-over and load-balancing. Today, state-less servers with state in distributed backends are the norm. But sticky sessions do have advantages as well due to a non-replicated system of records.

Options for Session State

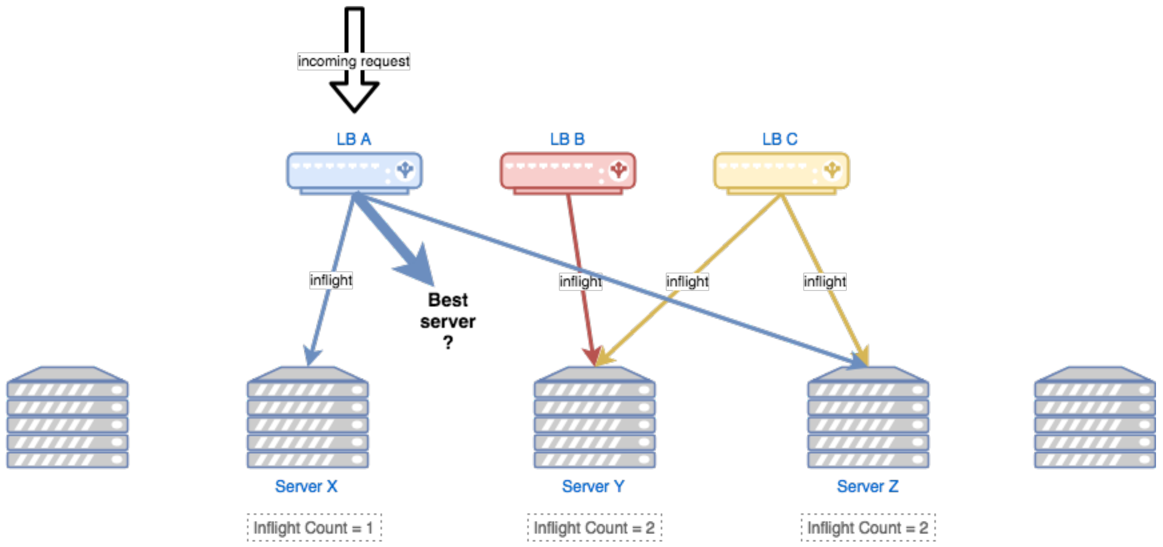


Session State features: Reachability and Replication. With SS only on Server C, we call this “sticky sessions”. The distributor needs to remember the route and there is no fail-over possible.

A compromise: Session Replication Pairs



Distributed Load-Balancing Problems

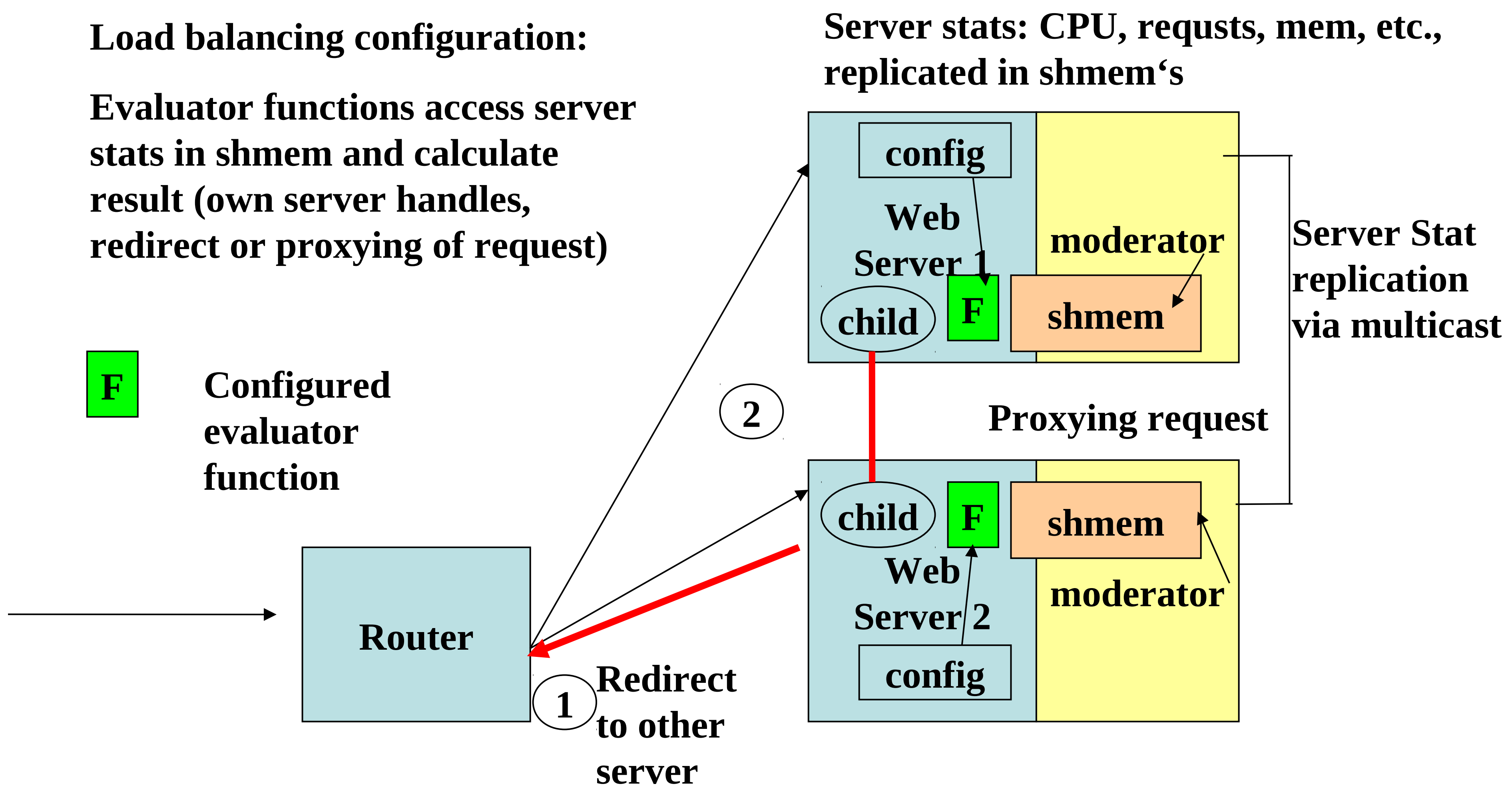


Single Lbs have a distorted view on real server load. The chose-the-shortes-queue alg. (JSQ) combines those views and can create serious herding problems. Client latency should be used as well. https://medium.com/netflix-techblog/netflix-edge-load-balancing-695308b5548c

Peer2Peer Load-Balancer

Load balancing configuration:

Evaluator functions access server stats in shmem and calculate result (own server handles, redirect or proxying of request)



Server stats: CPU, requsts, mem, etc., replicated in shmem‘s
config
Web Server 1
moderator
F
shmem
child
Server Stat replication via multicast
F
Configured evaluator function
2
Proxying request
child
F
shmem
Web Server 2
moderator
config
Router
1
Redirect to other server

“mod backhand” for Apache Server, from Theo Schlossnagle

Middle-LB Architectures: Matt Klein, Introduction to modern network load balancing and proxying

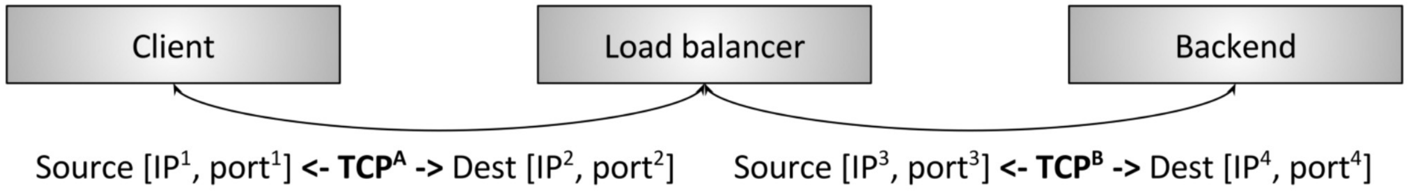

Client
Load balancer
Backend
Source [IP1, port1] <- TCPA -> Dest [IP2, port2]
Source [IP3, port3] <- TCPB -> Dest [IP4, port4]

Figure 2: TCP L4 termination load balancing

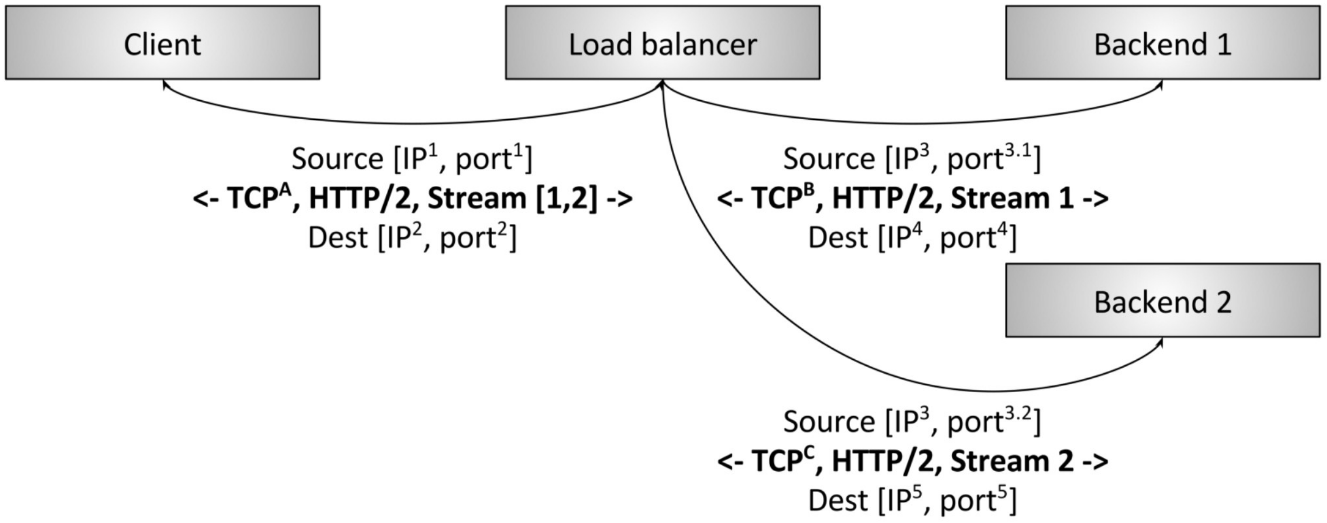

Client
Load balancer
Backend 1
Backend 2
Source [IP1, port1] <- TCPA, HTTP/2, Stream [1,2] -> Dest [IP2, port2]
Source [IP3, port3.1] <- TCPB, HTTP/2, Stream 1 -> Dest [IP4, port4]
Source [IP3, port3.2] <- TCPC, HTTP/2, Stream 2 -> Dest [IP5, port5]

Figure 3: HTTP/2 L7 termination load balancing

Covers direct server return, health checking, observability etc.

Distributed Name/Directory Service

(Non)-functional requirements (aka Systemic Requirements)

Definition: the functions of a system that serve e.g. the business DIRECTLY are called “functional requirements”

“Non-Functional Requirements”: the functions of a system that are necessary to achieve speed, reliability, availability, security etc.

Many programs fulfill the functional requirements but die on the non-functional requirements.

The different views of a system are also called „aspects“ today. Aspect-oriented programming (AOP) tries to keep them conceptually separate but „weaves“ them together at or before runtime execution.

For “Systemic Requirements” see: http://www.julianbrowne.com/article/viewer/systemic-requirements

functional requirements of a naming service

- re-bind/resolve methods to store name/value pairs.

- Provide query interface

-Support for name aliases to allow several logical hierarchies (name space forms a DAG)

-Support for composite names (“path names”)

-Support location – independence of resources by separating address and location of objects.

A naming service offers access to many objects based on names. This requires a sound security system to be in place. Is it a clever system design to offer lots of services/objects only to require a grand access control scheme?

Non-functional requirements of a name service

-Persistent name/value combinations (don’t lose mapping in a crash)

-Transacted: Manipulation of a naming service often involves several entries (e.g. during application installation) This needs to happen in an all-or-nothing way.

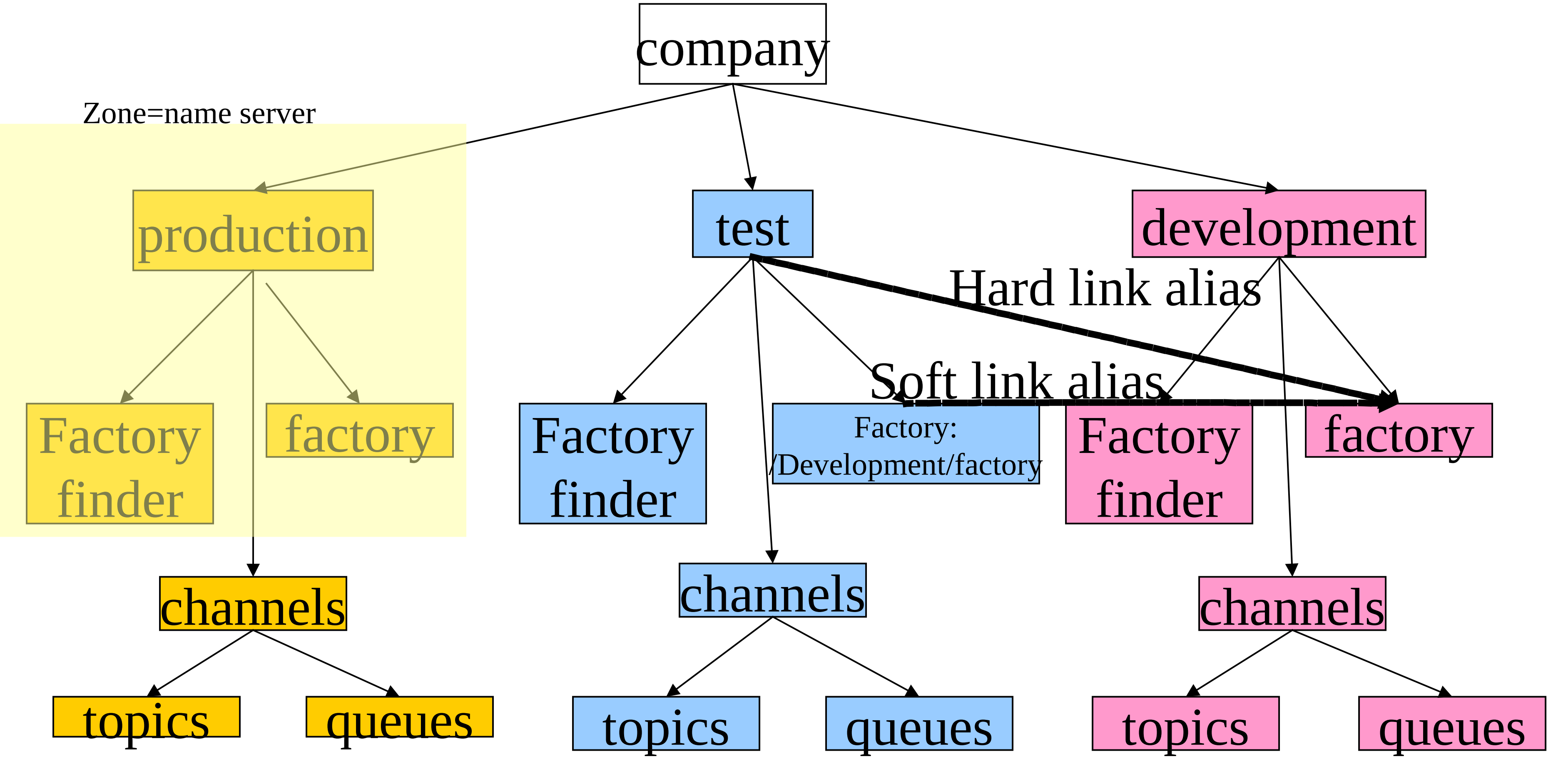
-Federated Naming Services: transparently combine different naming zones into one logical name service.

-Fault-Tolerance: Use replication to guarantee availability on different levels of the name space. Avoid inconsistencies caused by caching and replication.

-Speed: Guarantee fast look-up through clustering etc. Writes can be slower. Client side caching support. Reduce communication costs.

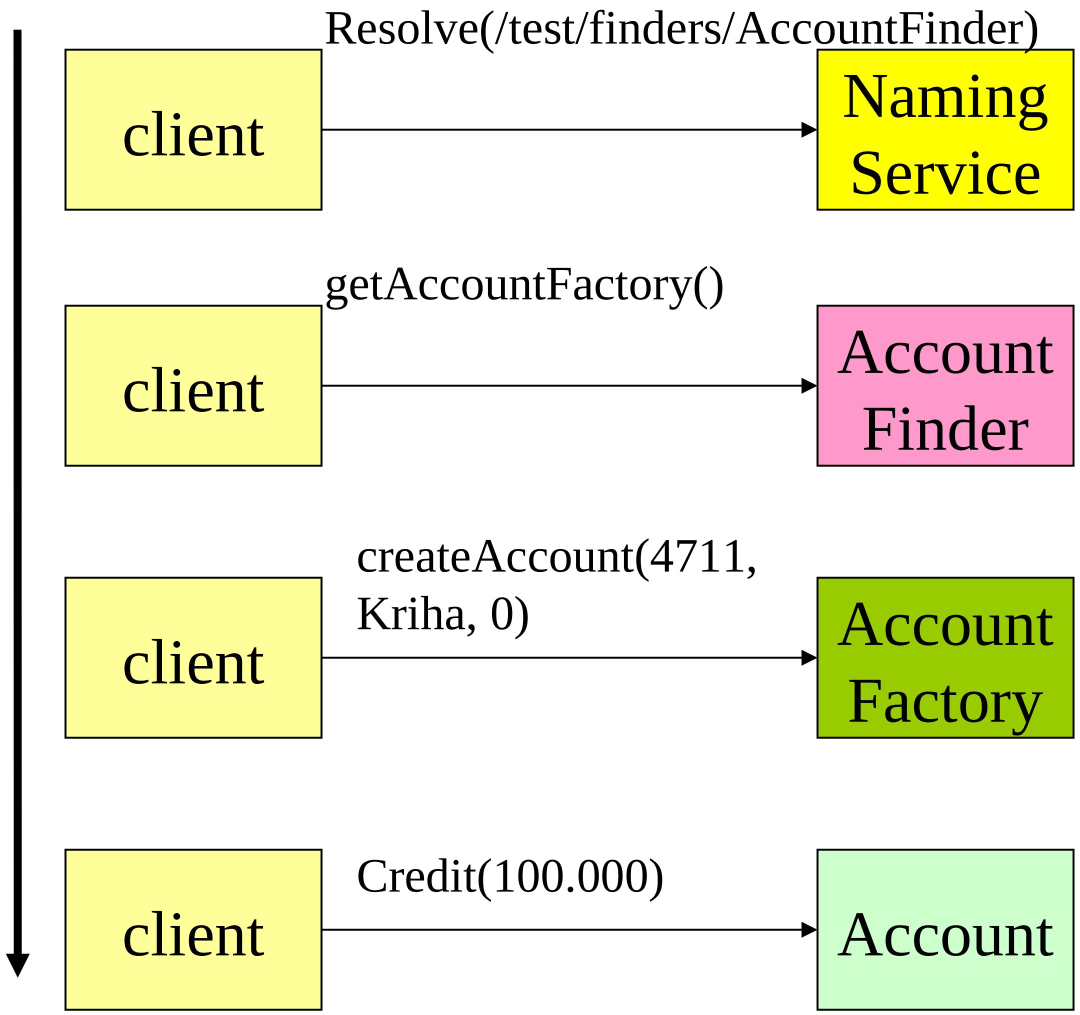
A naming service is a core component of every distributed system. It is easily a Single-point-of-failure able to stop the whole system.

Finding distributed objects : name space design



The organization of a company name space is a design and architecture issue. System Management will typically enforce rules and policies and control changes. Different machines will host parts of the name space (zones)

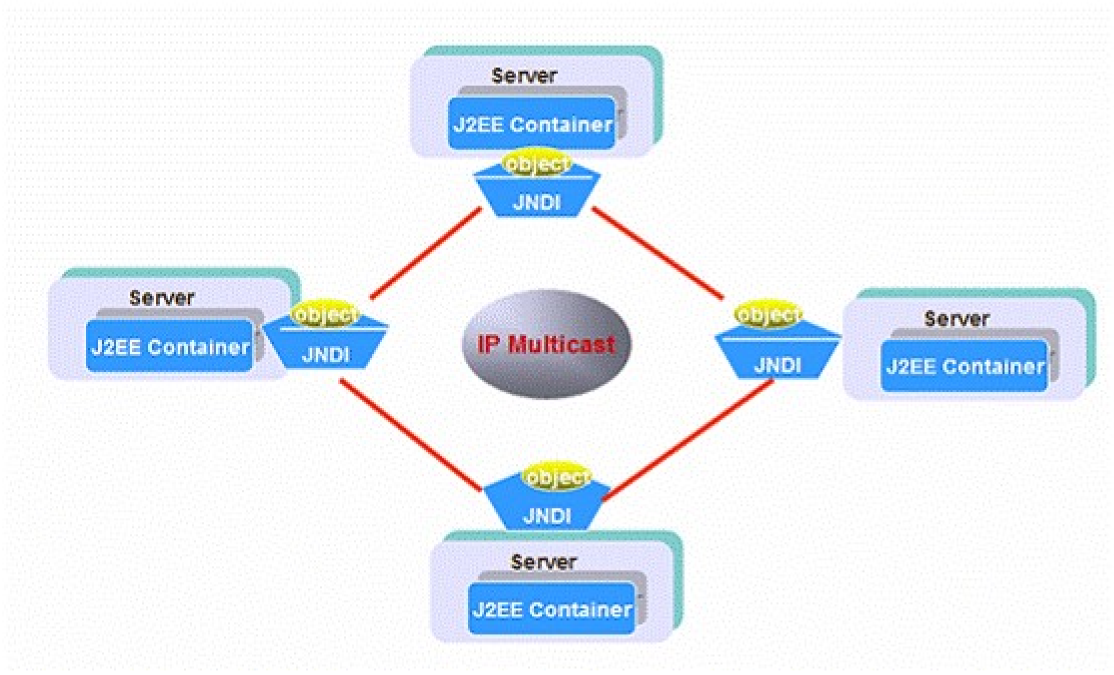
Finding distributed objects : naming service



Using interfaces (naming service, factoryfinder, factory, account) allows system administrators to hide different versions or implementations from clients.

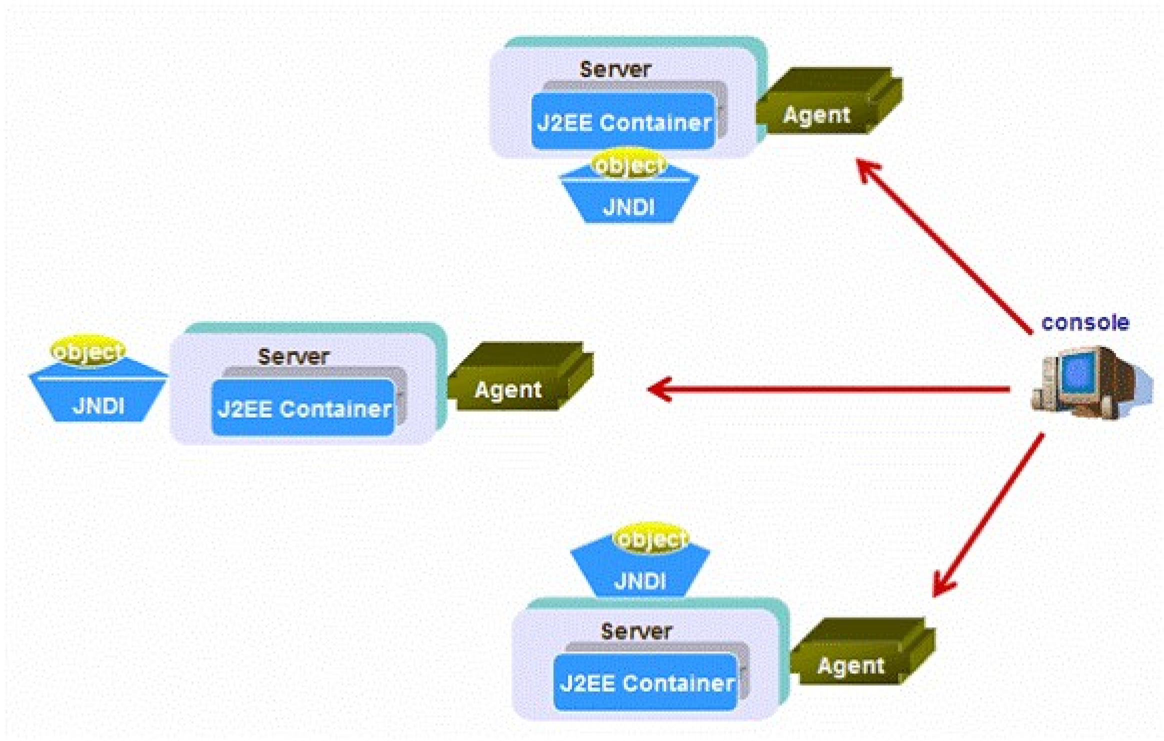
In a remote environment finders also support migration and copy of objects.

A fault-tolerant JNDI name service I



From: Wang Yu, uncover the hood of J2EE Clustering,
http://www.theserverside.com/tt/articles/article.tss?l=J2EEClustering

A fault-tolerant JNDI name service II



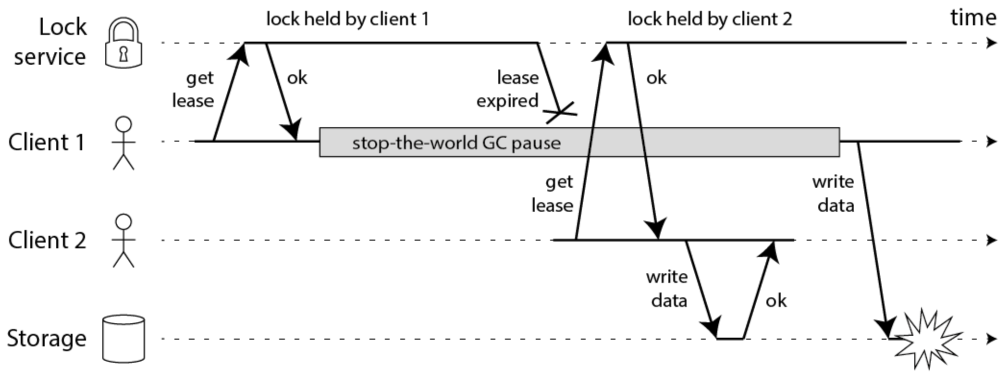
From: Wang Yu, uncover the hood of J2EE Clustering,
http://www.theserverside.com/tt/articles/article.tss?l=J2EEClustering

Examples of Naming Services

- Domain Name System (DNS)
- X.500 Directory
- Lightweight Directory Access Protocol (LDAP)
- CORBA Naming Service
- Java Registry
- J2EE JNDI (mapped to CORBA Naming Service)

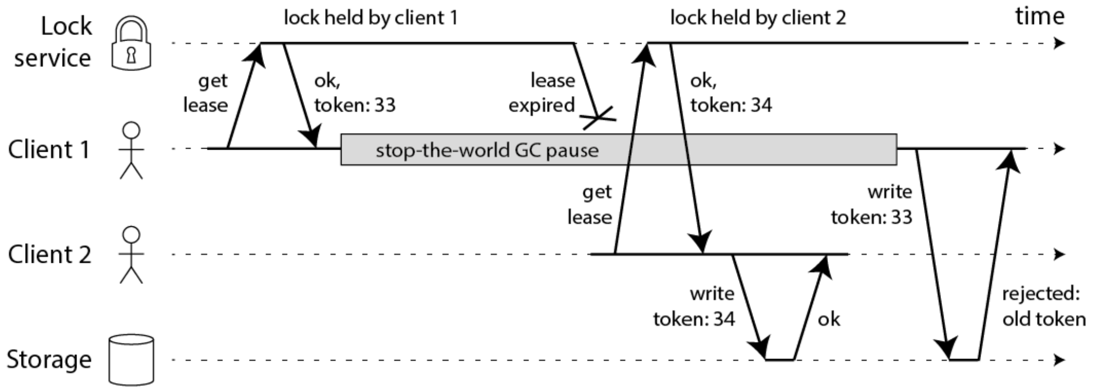
Distributed Locking

```
// THIS CODE IS BROKEN
function writeData(filename, data) {
    var lock = lockService.acquireLock(filename);
    if (!lock) {
        throw 'Failed to acquire lock';
    }
    try {
        var file = storage.readFile(filename);
        var updated = updateContents(file, data);
        storage.writeFile(filename, updated);
    } finally {
        lock.release();
    }
}
```



Time based resource access

Exercise: How does the correct API look?

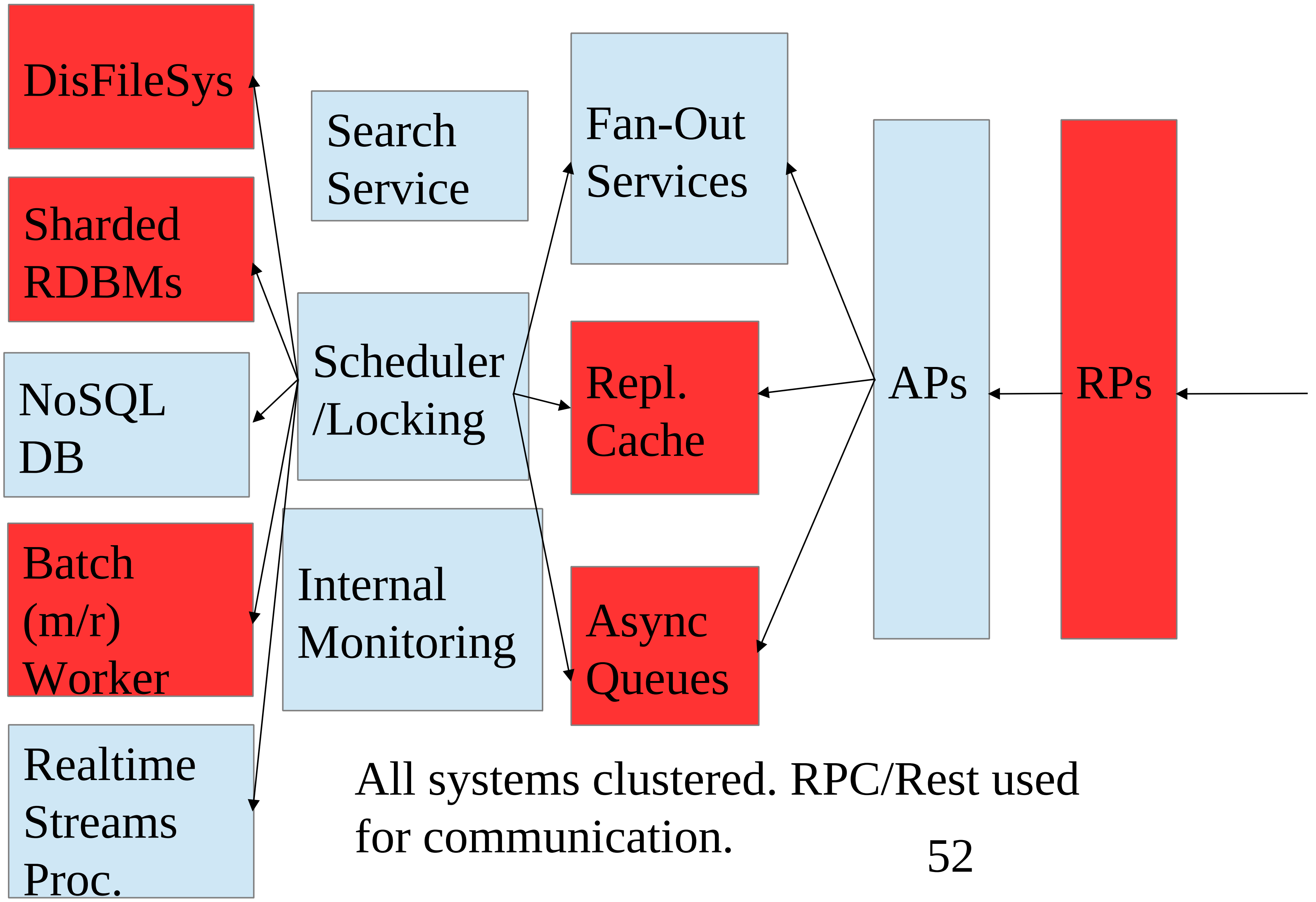


Order based resource access

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

Distributed Operating Systems

Distributed Operating Systems



All systems clustered. RPC/Rest used for communication.

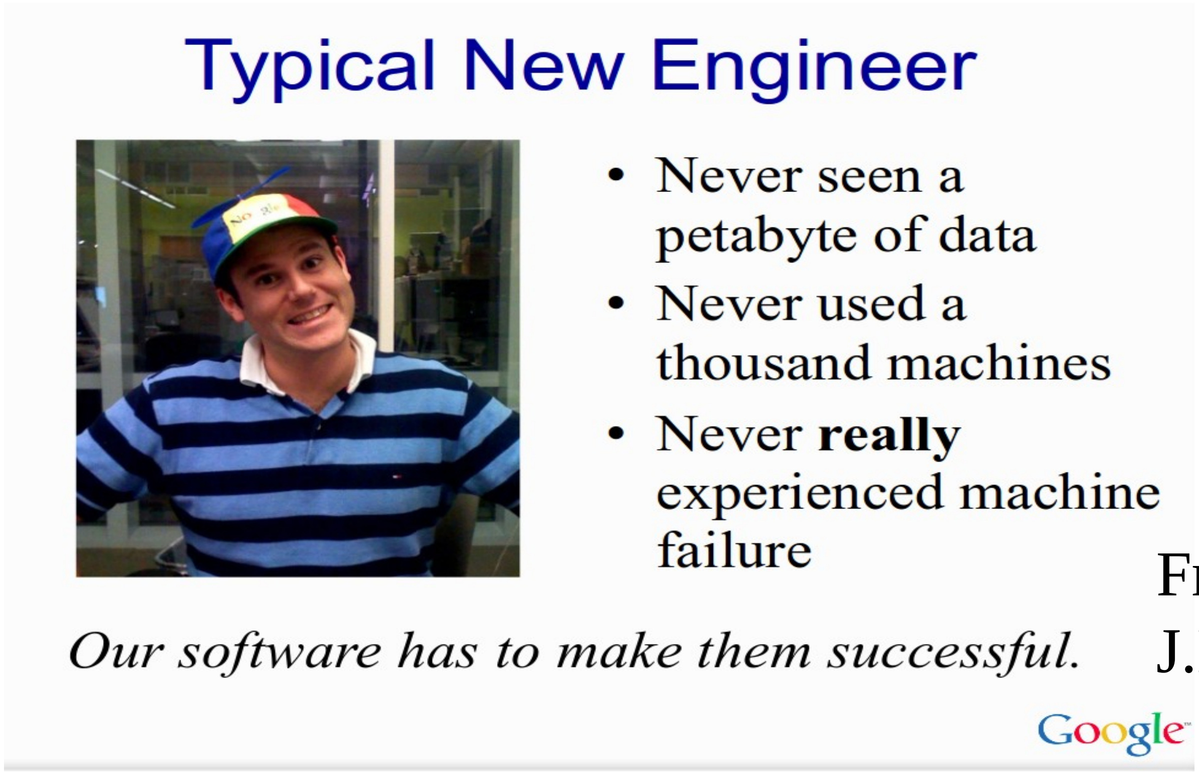
Distributed OS at Amazon

"The big architectural change that Amazon went through in the past five years was to move from a two-tier monolith to a fully-distributed, decentralized, services platform serving many different applications. "

Werner Vogels, Amazon CTO,

At the core of the Amazon strategy are the Web Services. The Amazon team takes the concepts of search, storage, lookup and management the data and turns them into pay-per-fetch and pay-per-space web services. This is a brilliant strategy and Amazon is certainly a visionary company. But what impresses me the most as an engineer is their ability to take very complex problems, solve them and then shrink wrap their solutions with a simple and elegant API. Alex Iskold, SOAWorld Mag.

Distributed Operating Systems



Typical New Engineer

- Never seen a petabyte of data
- Never used a thousand machines
- Never **really** experienced machine failure

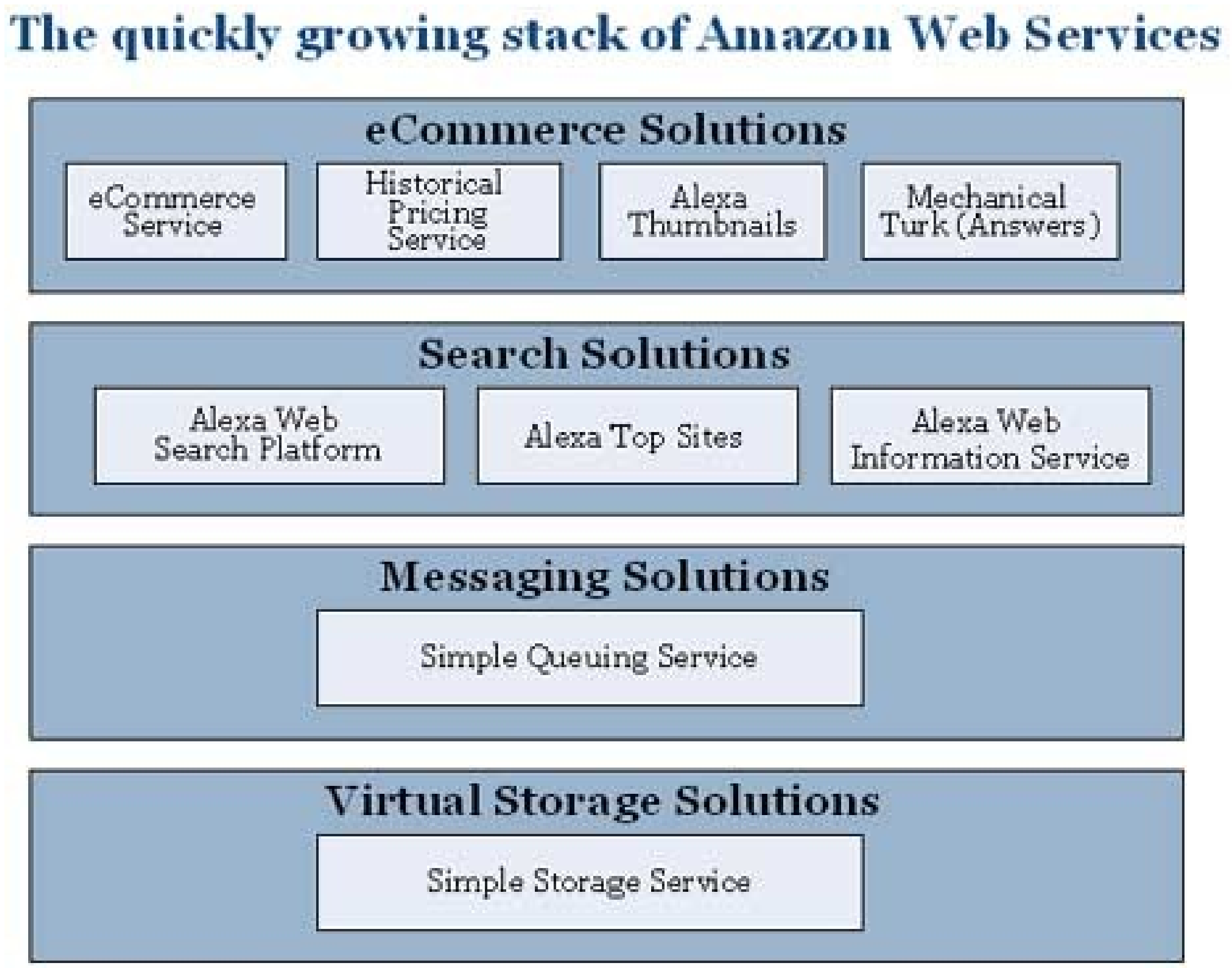
Our software has to make them successful.

Google

From J.Dean talk

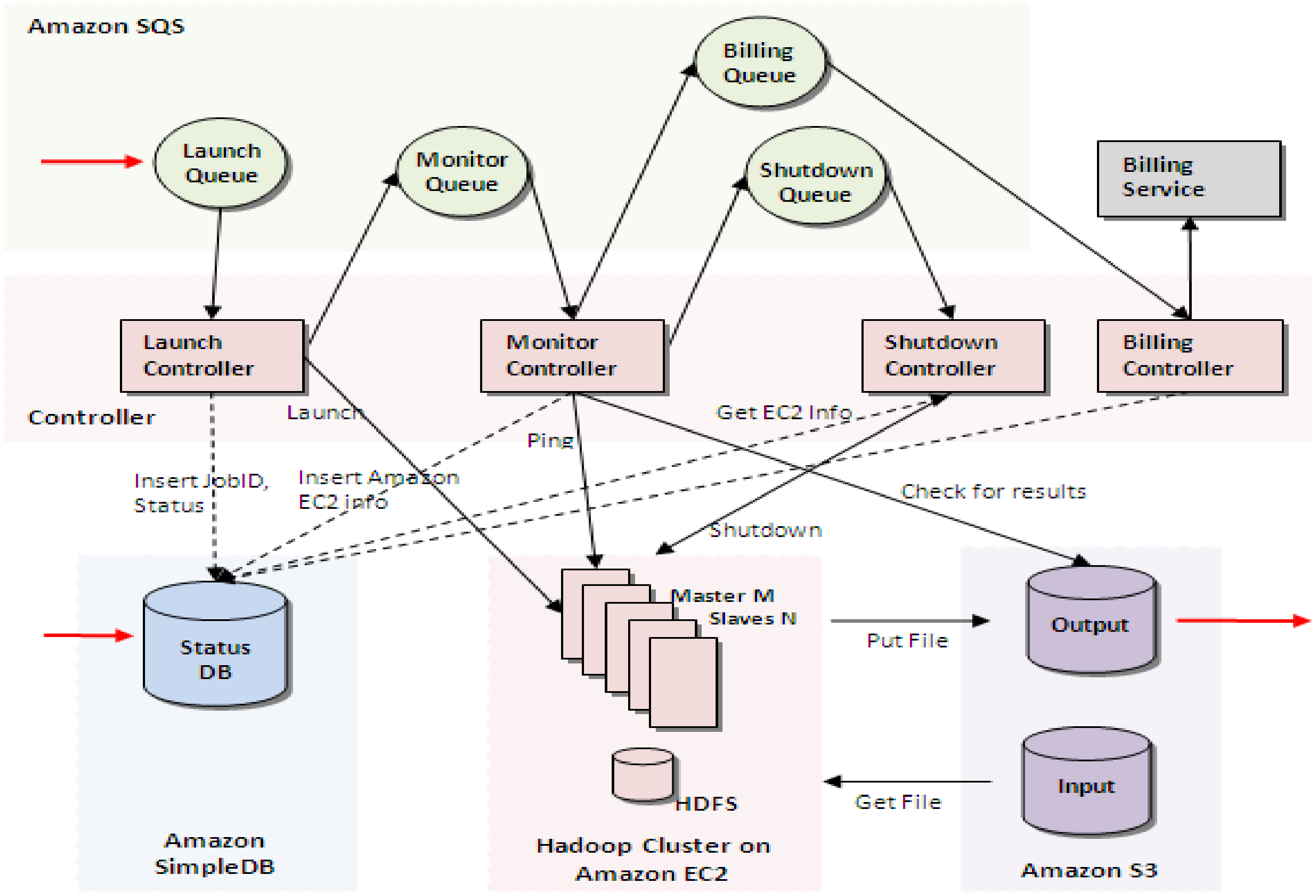
DS Services need to enable application engineers! They need to hide the complexities of distributed algorithms. Google has been especially conservative in the use of APIs.

Amazon Service Arc.


The quickly growing stack of Amazon Web Services
eCommerce Solutions
eCommerce Service
Historical Pricing Service
Alexa Thumbnails
Mechanical Turk (Answers)
Search Solutions
Alexa Web Search Platform
Alexa Top Sites
Alexa Web Information Service
Messaging Solutions
Simple Queuing Service
Virtual Storage Solutions
Simple Storage Service

A.Iskold, SOAWorld

Amazon Service Arc.



http://www.infoq.com/news/2009/02/Cloud-Architectures

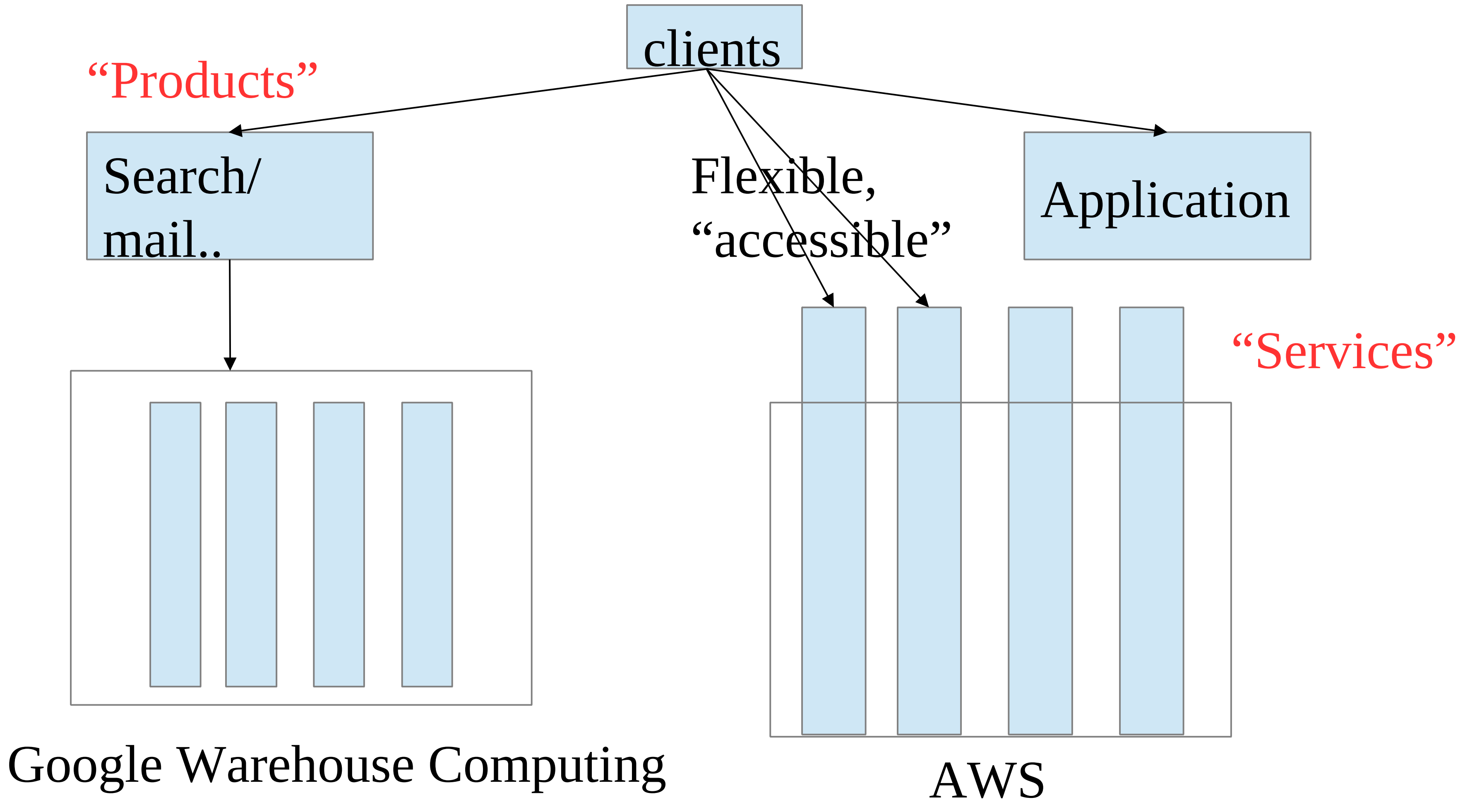
Amazon Client Facing Services

- Storage Services (S3, huge storage capacity, RDS, Aurora, Mongo)
- Computation Services (EC2, Virtual Machines)
- Queuing Services
- Load-balancing services
- Elastic map reduce
- Cloudfront (memcache like fast cache)
- Lambda, Stepfunctions
- AI framework services
- And so on....

Distributed OS at Google

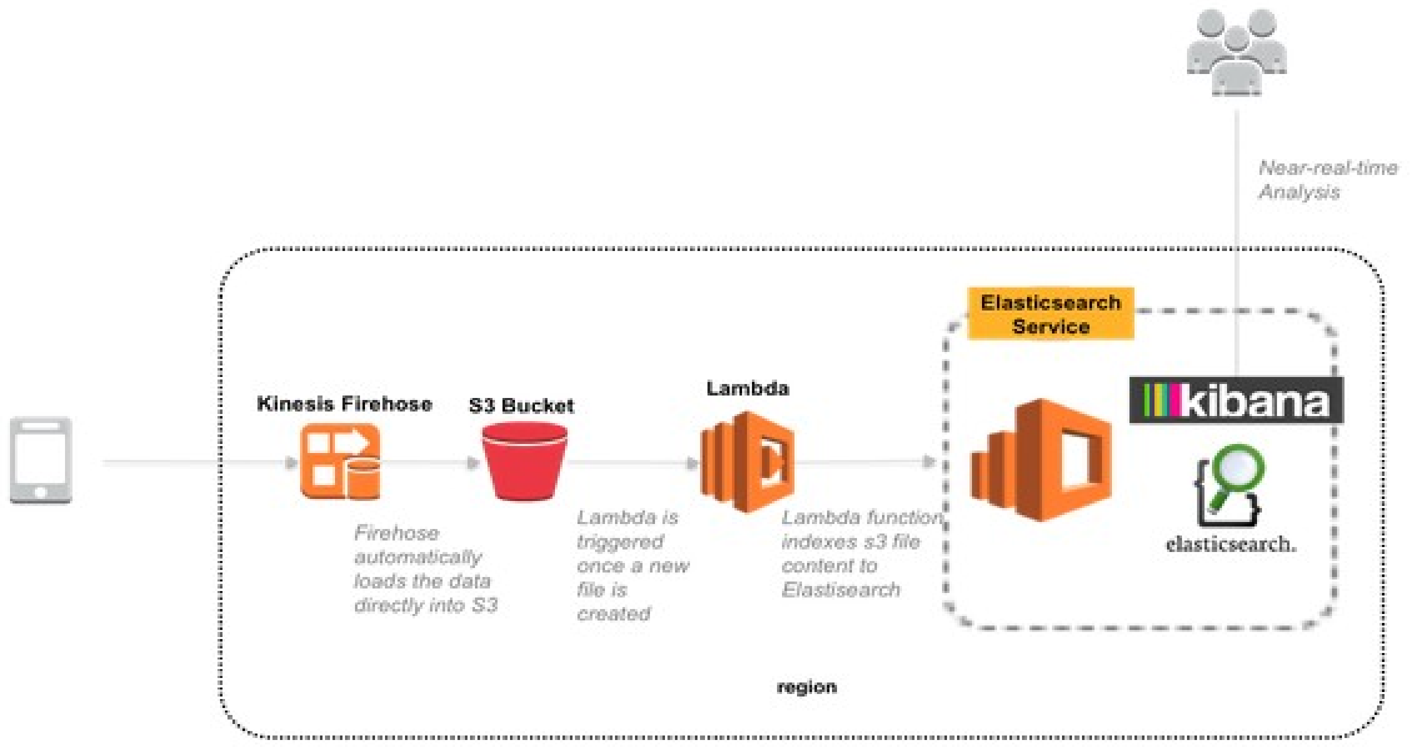
- Scheduling Service
- Map/Reduce Execution Environment
- F1 NewSQL DB
- Spanner Replicated DB
- BigTable NoSQL Storage
- Distributed File System
- Chubby Lock Service
- Pregel, Percolator, Dremel graph processing and SQL
- Dapper distributed locking
- Google compute platform

Google vs. Amazon



According to Steve Yegge,, Jeff Bezos understood 2 things: A platform can be re-purposed. And you can't make it right for everybody! Read: "Yegge's Rant": https://plus.google.com/+RipRowan/posts/eVeouesvaVX

Example: Near RT Discovery Platform on AWS



Near-real-time Analysis
Elasticsearch Service
Kinesis Firehose
S3 Bucket
Lambda
kibana
elasticsearch.
Firehose automatically loads the data directly into S3
Lambda is triggered once a new file is created
Lambda function indexes s3 file content to Elastisearch
region

From: Assaf Mentzer, Building a Near Real-Time Discovery Platform with AWS, http://blogs.aws.amazon.com/bigdata/post/Tx1Z6IF7NA8ELQ9/Building-a-Near-Real-Time-Discovery-Platform-with-AWS

Services: Best Practice

- Keep services independent (see Playfish: a game is a service)
- Measure services
- Define SLAs and QoS for services
- Allow agile development of services
- Allow hundreds of services and aggregate them on special servers
- Stay away from middleware and frameworks which force their patterns on you
- Keep teams small and organized around services
- Manage dependencies carefully
- Create APIs to offer your services to customers

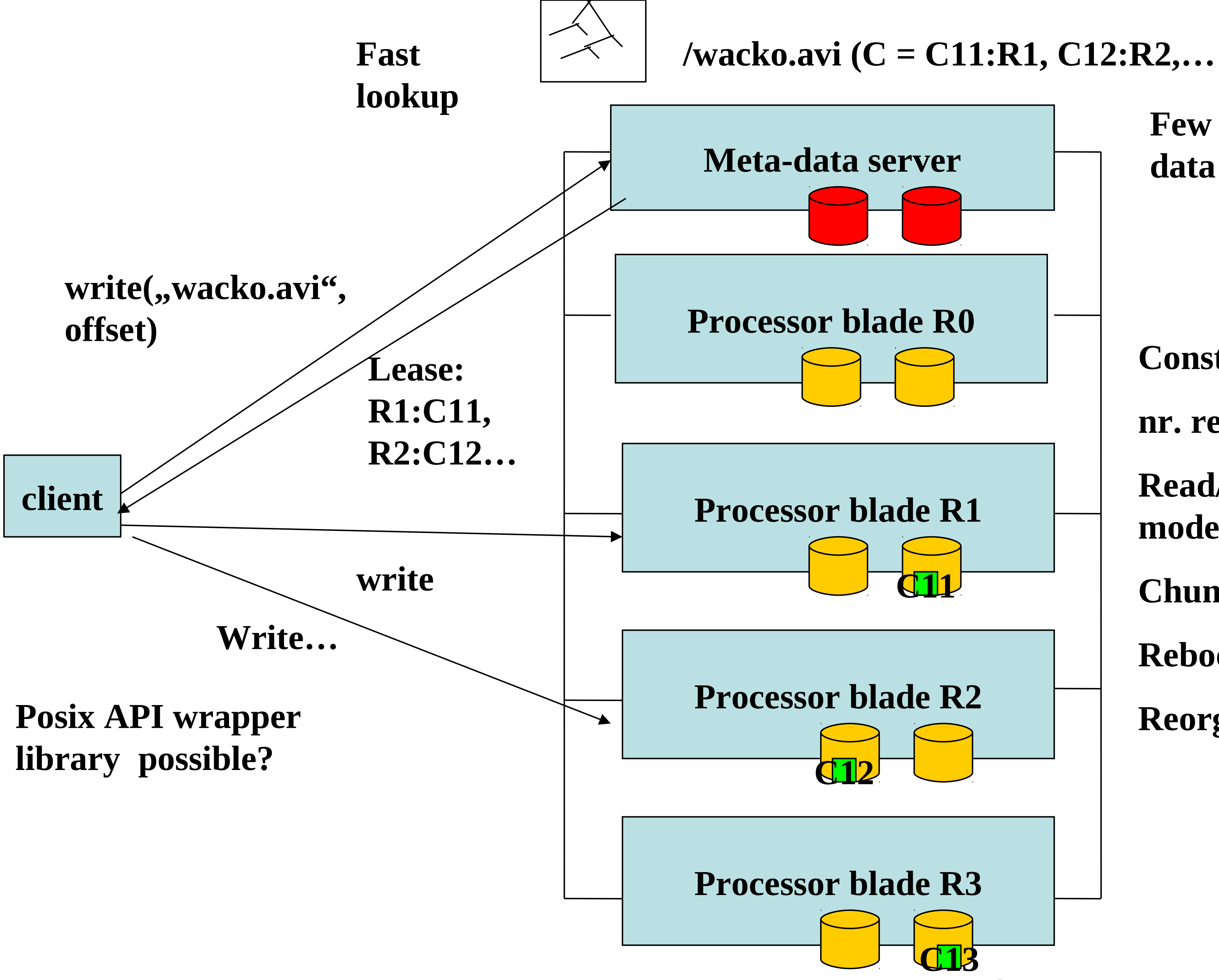
more: www.highscalability.com, Inverview with W.Vogels, allthingsdistributed.com

Distributed File Systems

Example: GFS, Central Master Approach

- made to store large amounts of documents
- processing mostly sequential and batch-oriented
- atomic updates needed, but no distributed transactions
- high bandwidth needed, latency no big issue
- huge chunk-size (block-size) to keep meta-data small
- chunk-replication instead of RAID.
- fast rebuild of broken disks
- clients aware of GFS specifics (no posix, no order between concurrent chunks etc.)

GFS was built around Google's needs, which was mostly batch processing at that time (map-reduce processing). This changed drastically over the years and forced Google to build new storage systems on top (e.g. BigTable)



Few meta-data

Constraints:

nr. replicas

Read/update modes

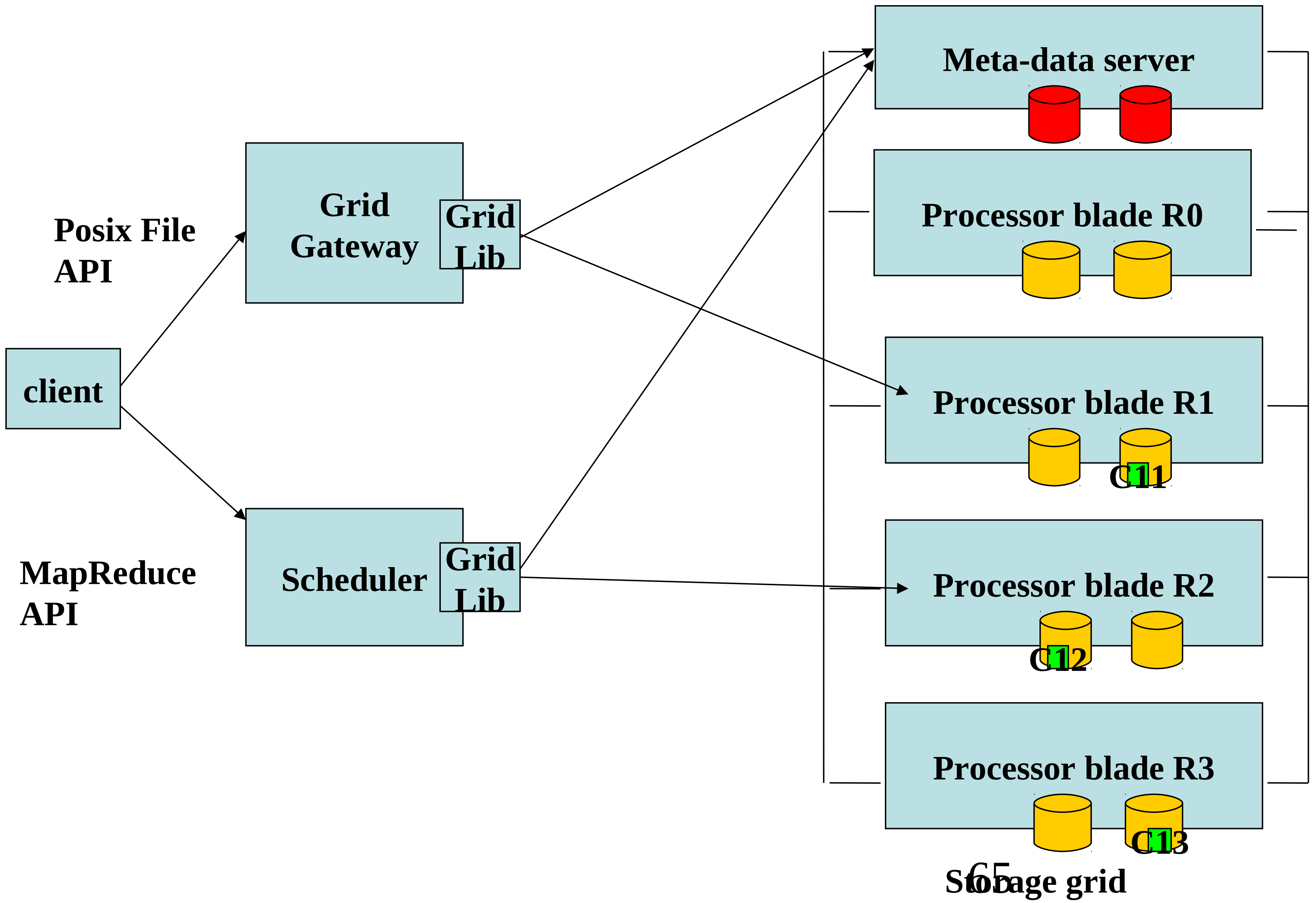
Chunk size

Reboot time

Reorganiz.

Posix API wrapper library possible?

Storage grid


Posix File API
MapReduce API
client
Grid Gateway
Grid Lib
Scheduler
Grid Lib
Meta-data server
Processor blade R0
Processor blade R1
C11
Processor blade R2
C12
Processor blade R3
C13
Storage grid

Storage Backend Technology (e.g. CephFs)

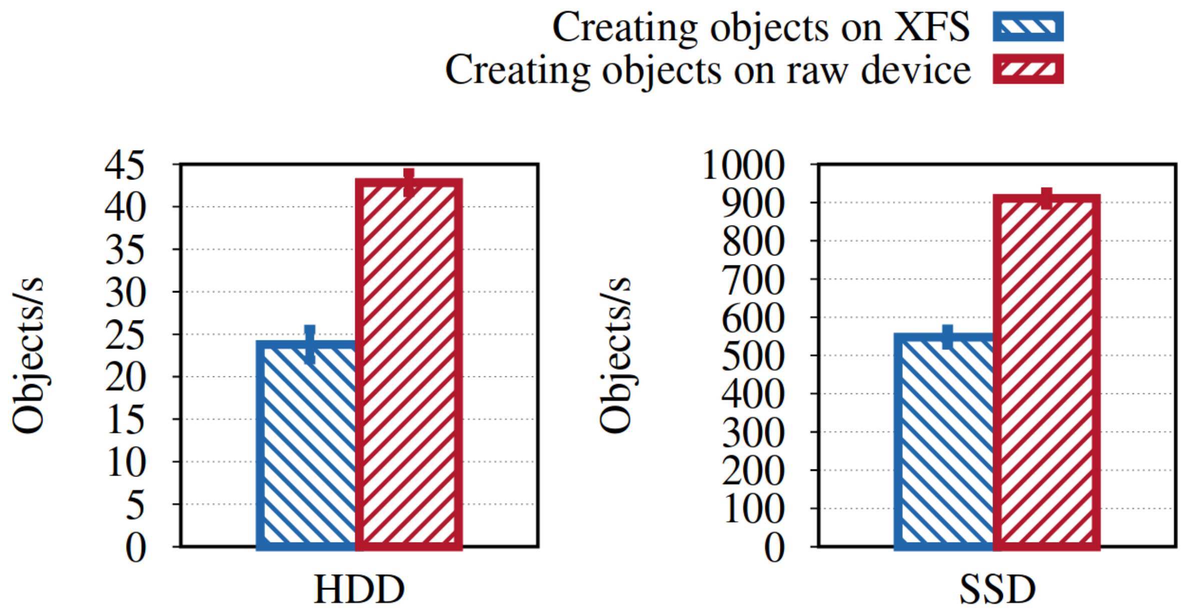
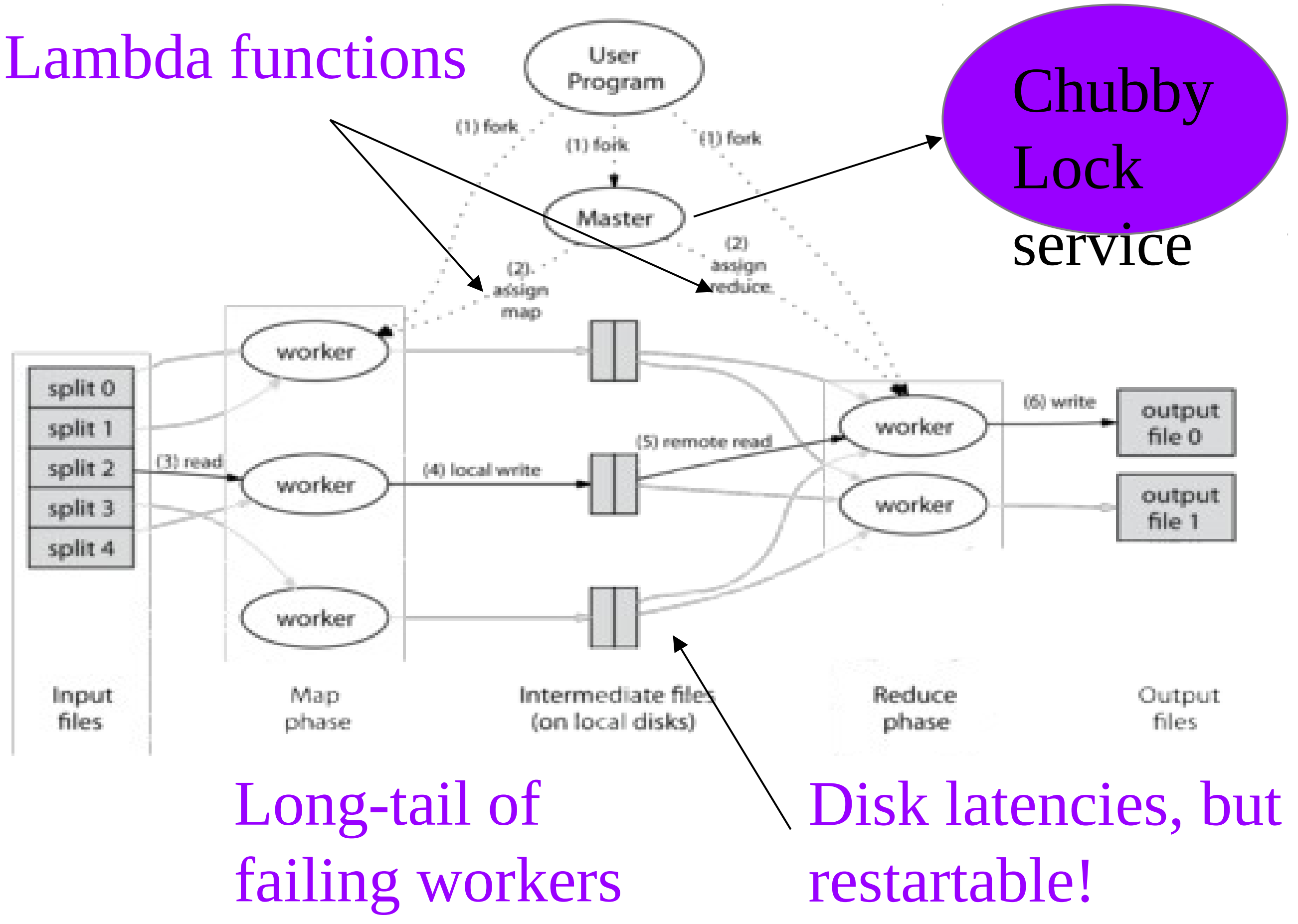


Figure 3. The overhead of running an object store workload on a journaling file system. Object creation throughput is 80% higher on a raw HDD (4 TB Seagate ST4000NM0023) and 70% higher on a raw NVMe SSD (400 GB Intel P3600).

From: https://blog.acolyer.org/2019/11/06/ceph-evolution/

Batch Processing at Scale: Map/Reduce



Diag. From: Simplifieded Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat of Google Inc. My annotations in purple.

Exercise: Use Map/Reduce to calculate PageRank

Define a map and a reduce function to calculate a pagerank from documents!

(define pagerank simply as the number of references (links) to a site)

Grid Storage vs. NAS/SAN

Posix-Grid gateway needed

Special caching possible but not needed for video (read-ahead needed?)

Huge bandwidth and scalable

Maintenance special?

Proprietary?

Parallel Processing possible

Special Applications needed

Questionable compatibility with existing apps.

Disaster revovery across sites?

Standard Lustre use possible? (framestore?)

More electric power and space needed for grids

Posix compatible

Special caching difficult to implement in standard products

Hard limit in SPxx storage interface but plannable and limited lifetime anyway

Simple upgrades

Standard filesystem support

Dynamic growth of file systems via lun-organization

Maintenance effort to balance space/use

Proven, fast technology

Expensive disaster recovery via smaller replicas

Several different filesystem configuration possible

Without virtual SAN hot-spots possible on one drive

Longer drive-rebuild times

Key points with grid storage: watch out for proprietary lock-in with grid storage and applications. Watch out for compatibility problems with existing apps. Without real parallel processing applications there is no use for the CPUs, they just eat lots of power (atom?). You should be able to program your solutions (map/reduce with Hadoop). Definitely more prog. Skills needed with grids. NAS/SAN won‘t go away with grid storage (which is specialized).

The sad Story of Myspace ...

Membership Milestones:
- 500,000 Users: A Simple Architecture Stumbles

1 Million Users:Vertical Partitioning Solves Scalability Woes

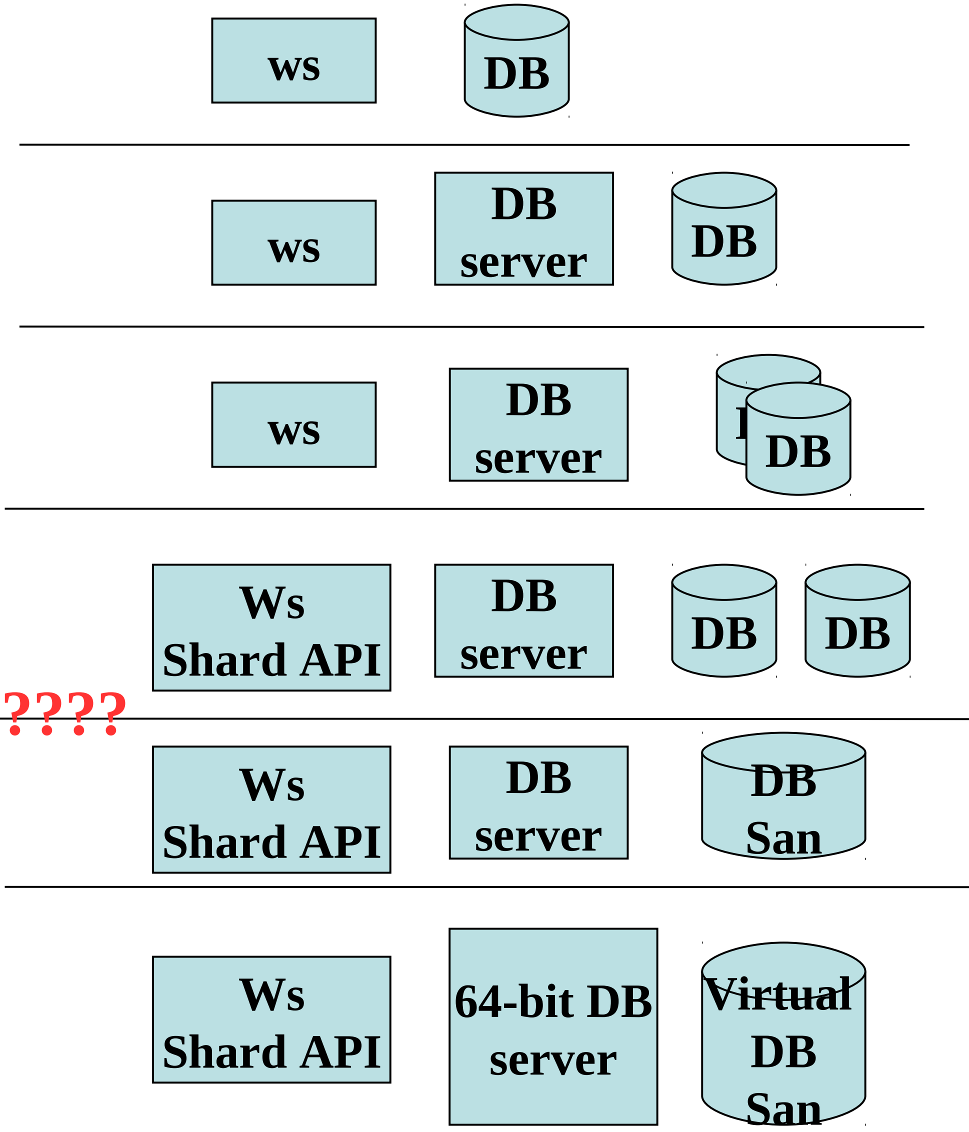
- 3 Million Users: Scale-Out Wins Over Scale-Up

What is MISSING????

- 9 Million Users: Site Migrates to ASP.NET, Adds Virtual Storage

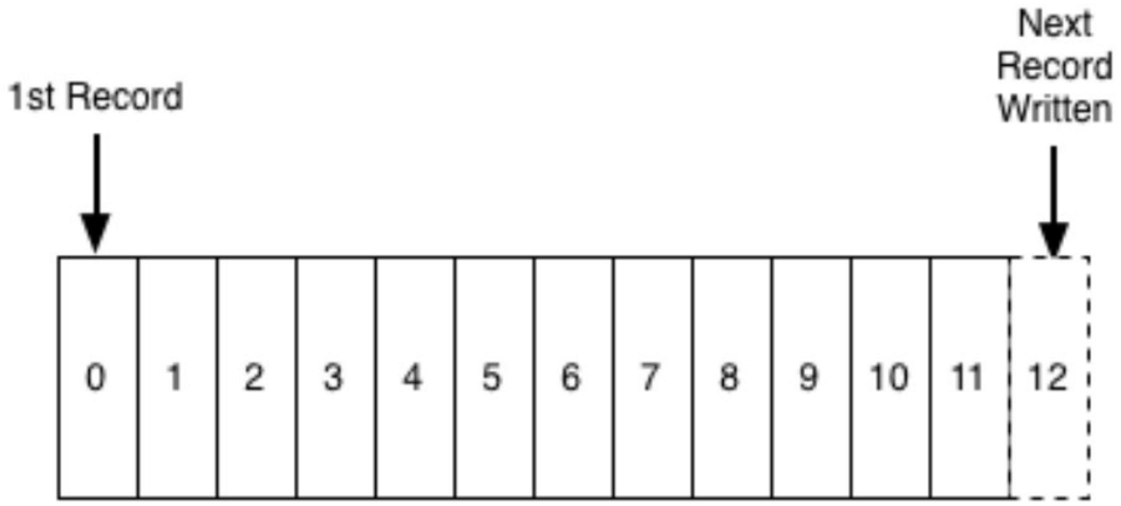
- 26 Million Users: MySpace Embraces 64-Bit Technology

(after Todd Hoff's Myspace article)



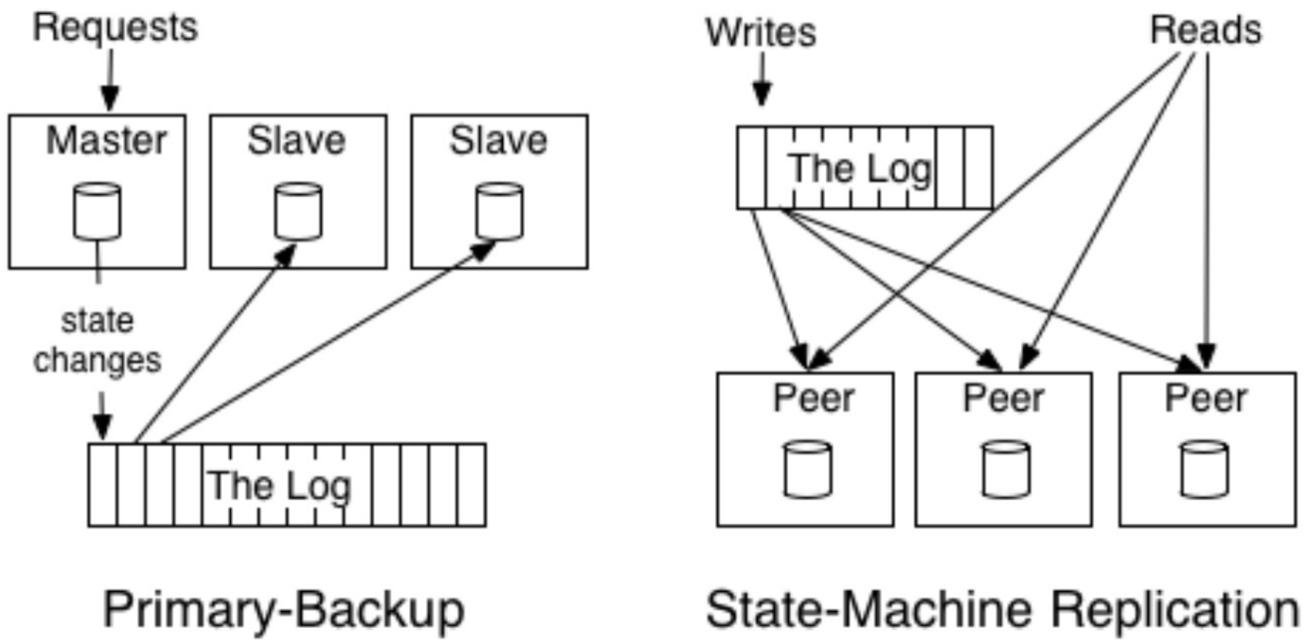
Distributed Logs

What is a Log?



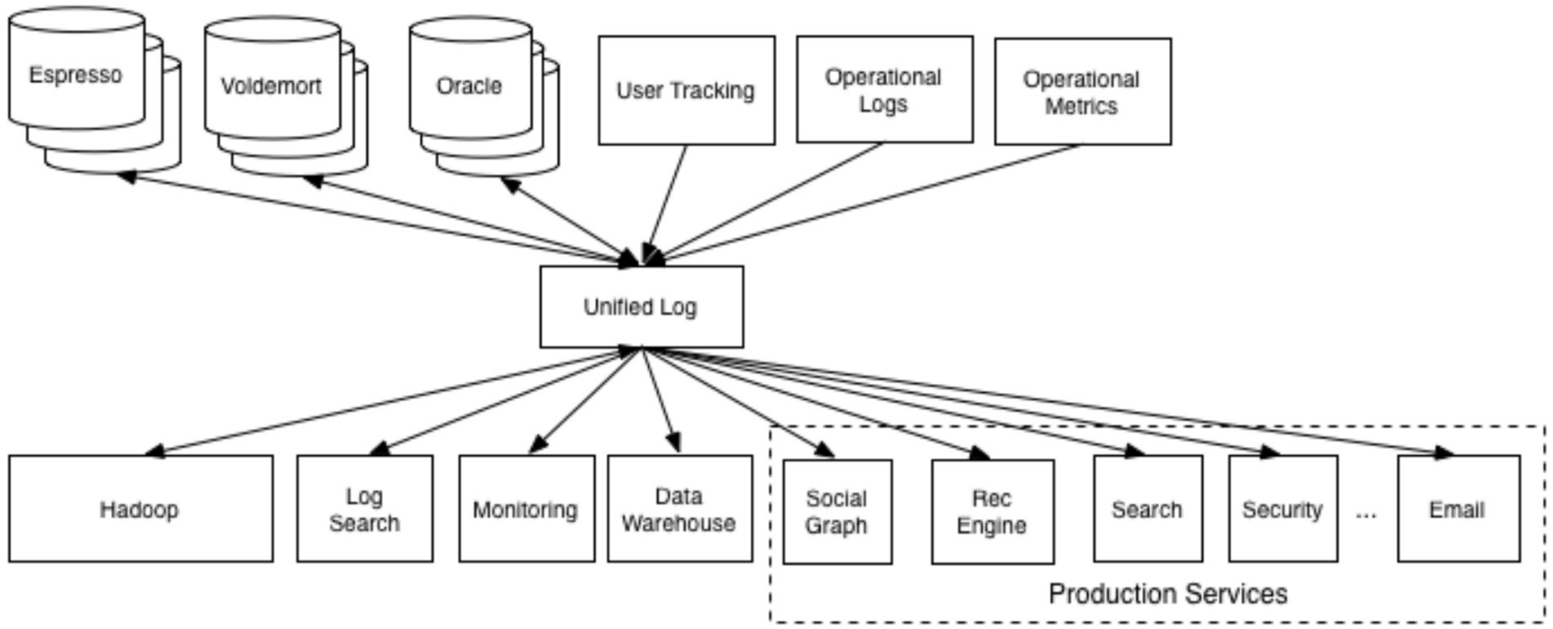
Jay Kreps, The Log: What every software engineer should know about real-time data's unifying abstraction. A log provides total order of events and data distribution. It is the base of many advanced data systems (Dbs, messaging systems, stream processors, pub/sub systems)

Log Architecture



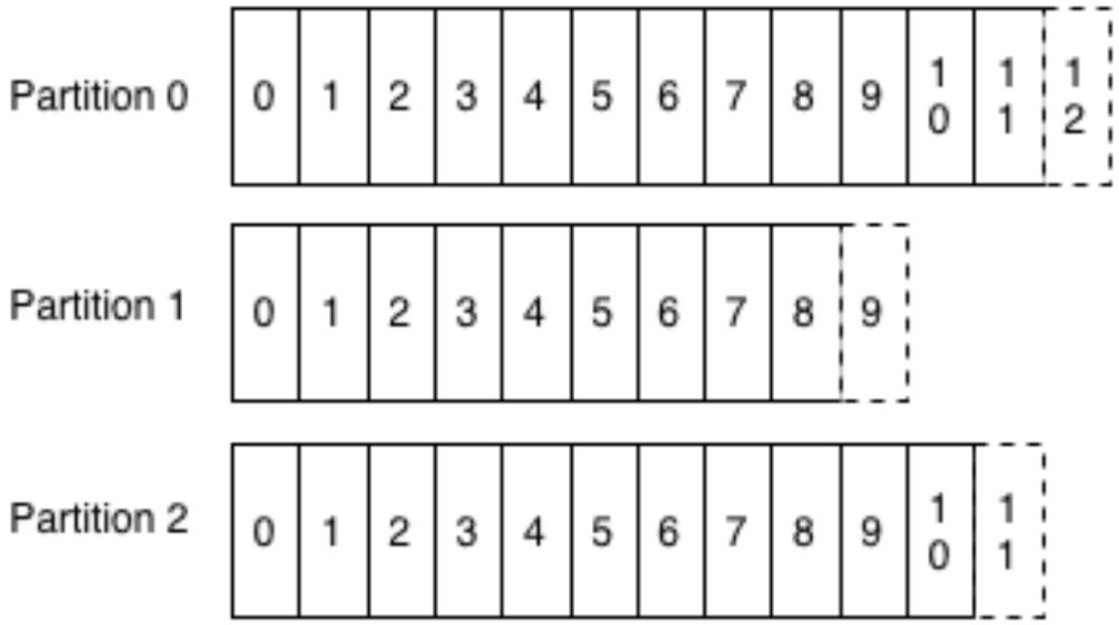
From: Jay Kreps, The Log: What every software engineer should know about real-time data's unifying abstraction. A log can contain raw events (e.g. commands) or the results of a command processed by a master first.

Unified Log



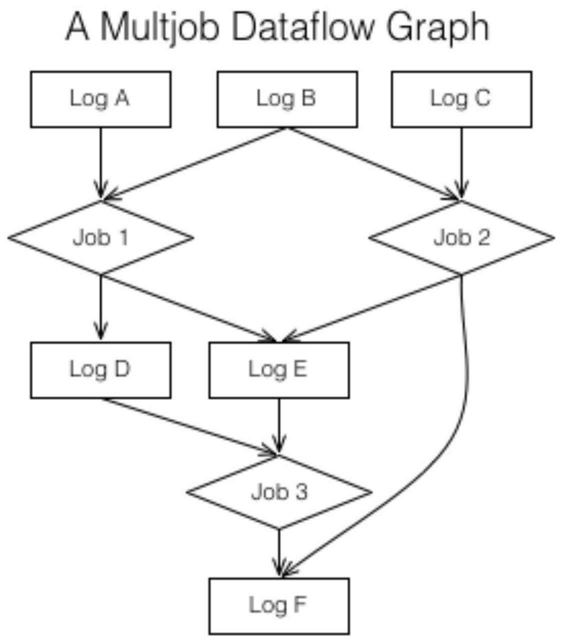
From: Jay Kreps, The Log: What every software engineer should know about real-time data's unifying abstraction. A unified log is a powerful enterprise component. It decouples producers and consumers and lets everybody create derived logs.

Distributed (scalable) Logs: Kafka



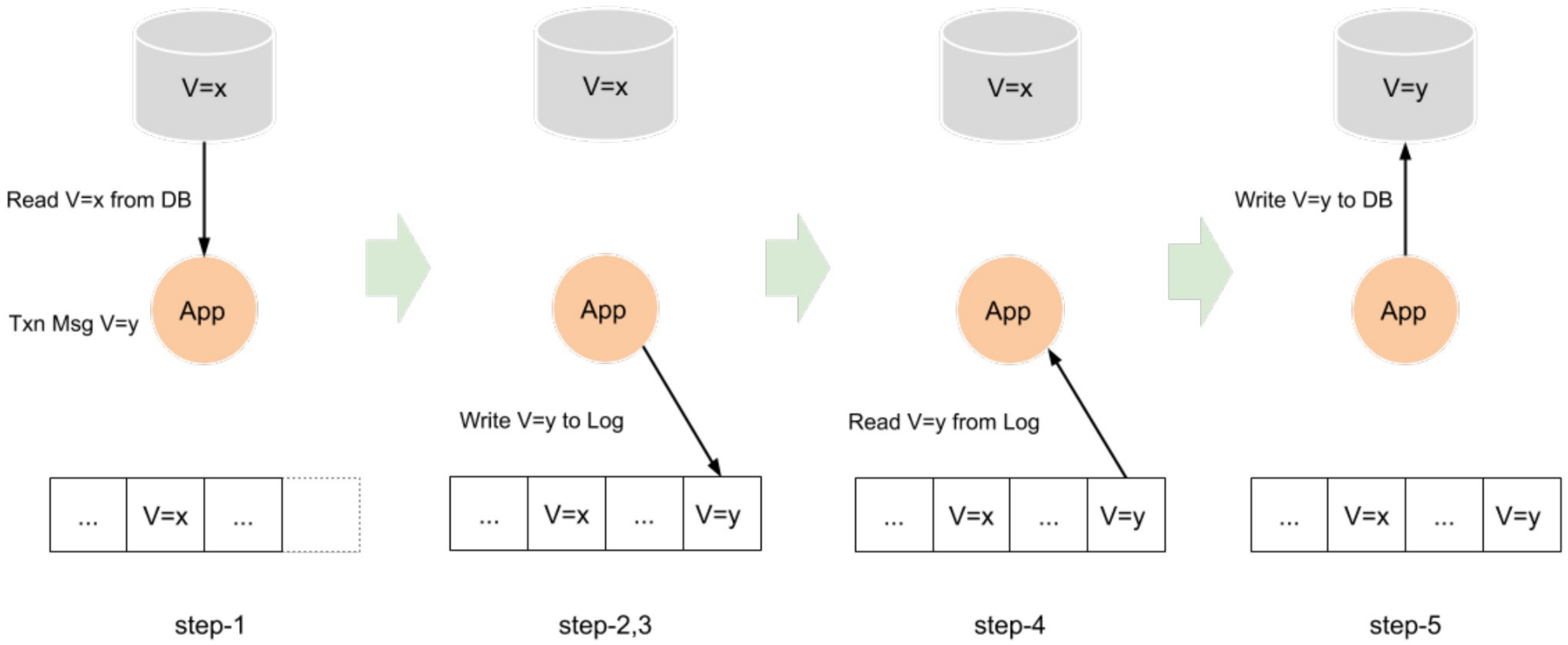
From: Jay Kreps, The Log: What every software engineer should know about real-time data's unifying abstraction. Kafka allows partitioning via keys. Partitions are totally ordered internally but not across partitions. They are replicated and can be read by many consumers. Some systems store data for days in Kafka.

Data-flow (stream) Architecture



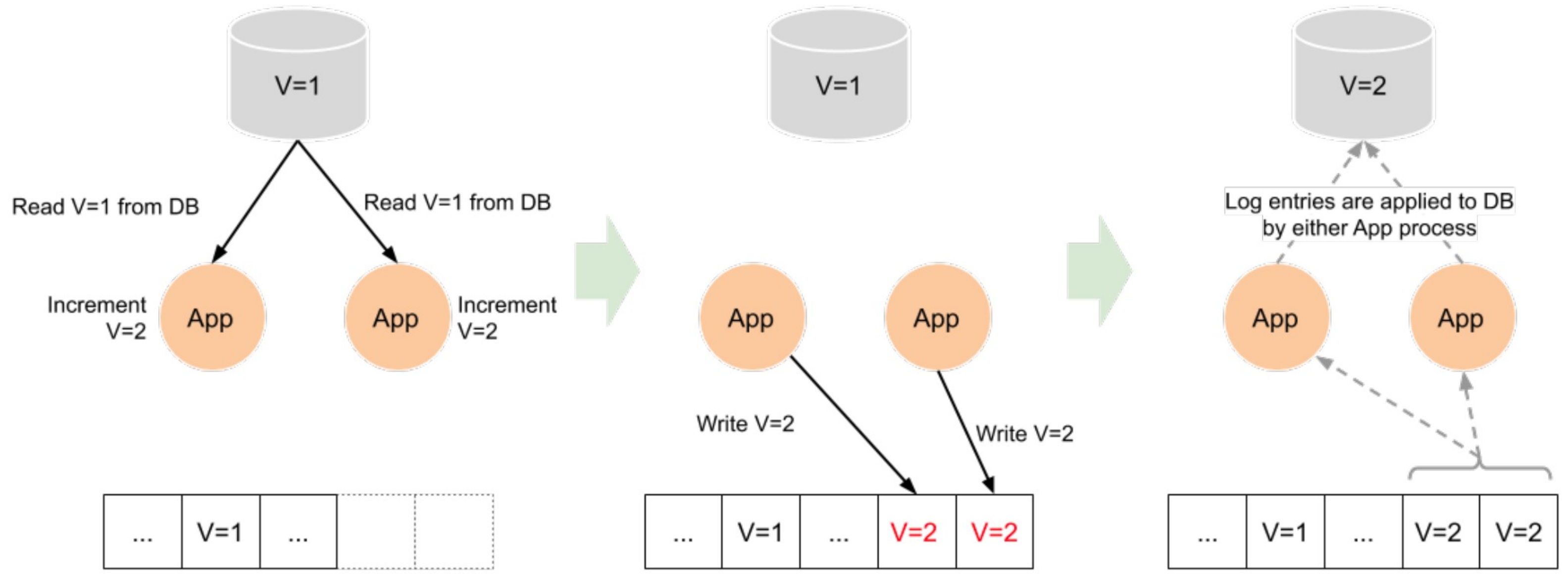
From: Jay Kreps, The Log: What every software engineer should know about real-time data's unifying abstraction. Finally, a stream architecture is Java Streams on steroids (partitioned across many machines). The result is a graph of process stages.

Other Architectures: Waltz



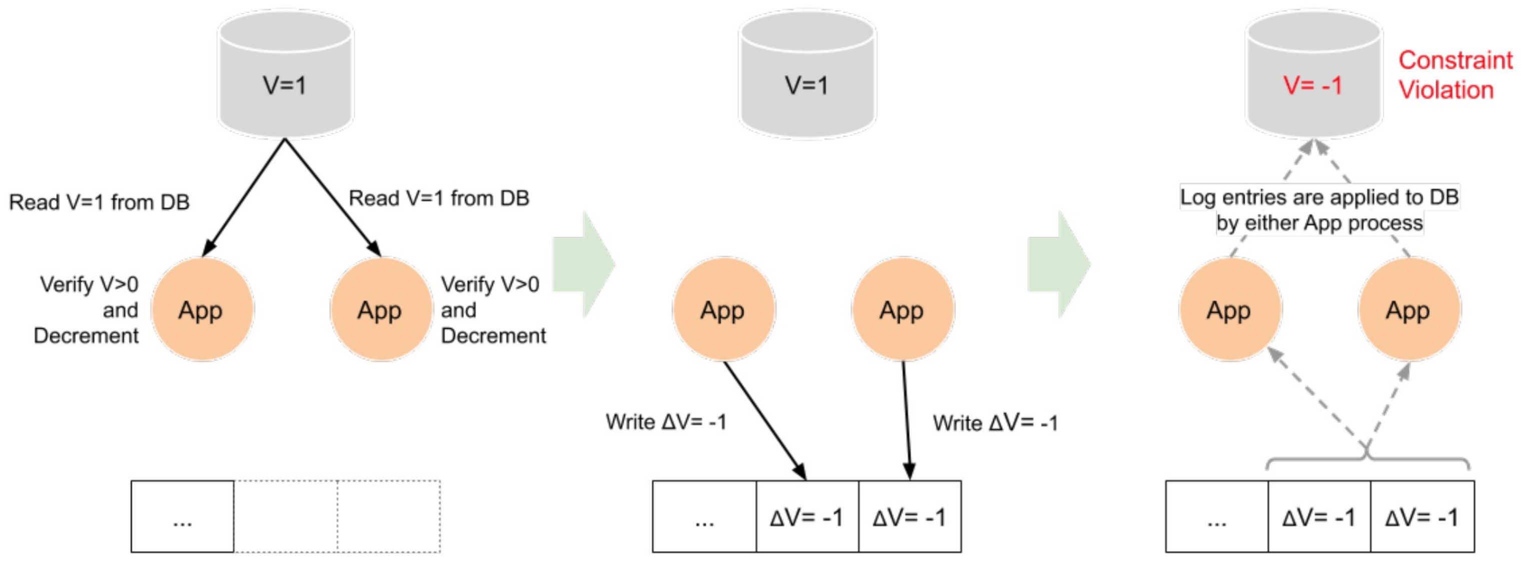
From: Y. Matsuda, Waltz: A Distributed Write-Ahead Log (wepay.com).

Waltz: Lost Update



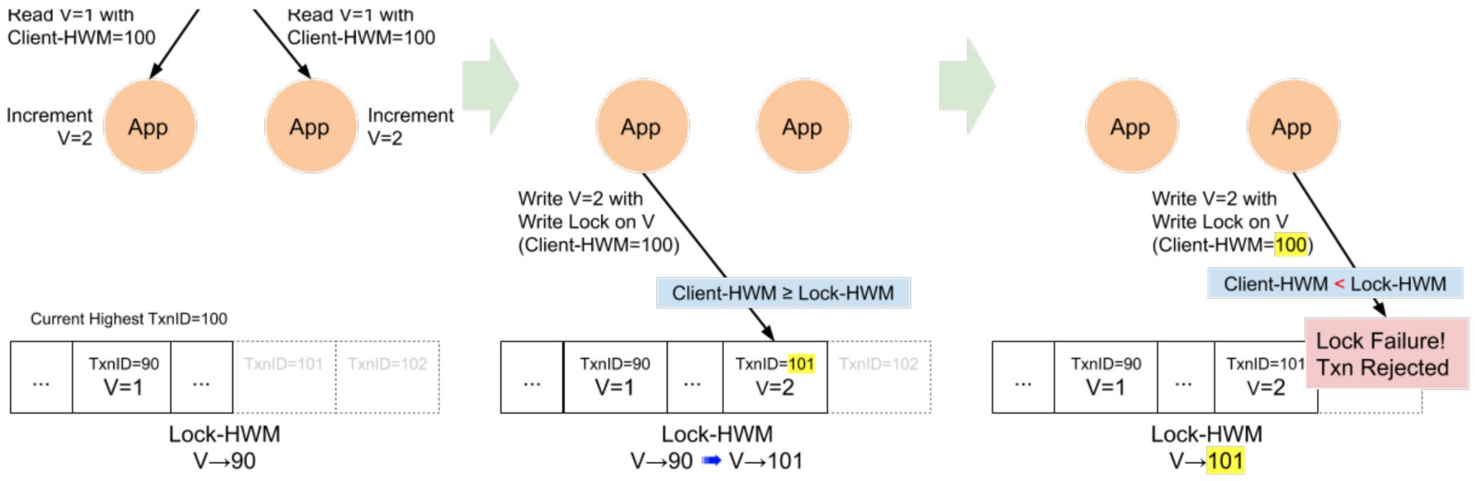
From: Y. Matsuda, Waltz: A Distributed Write-Ahead Log (wepay.com).

Waltz: Constraint Violation



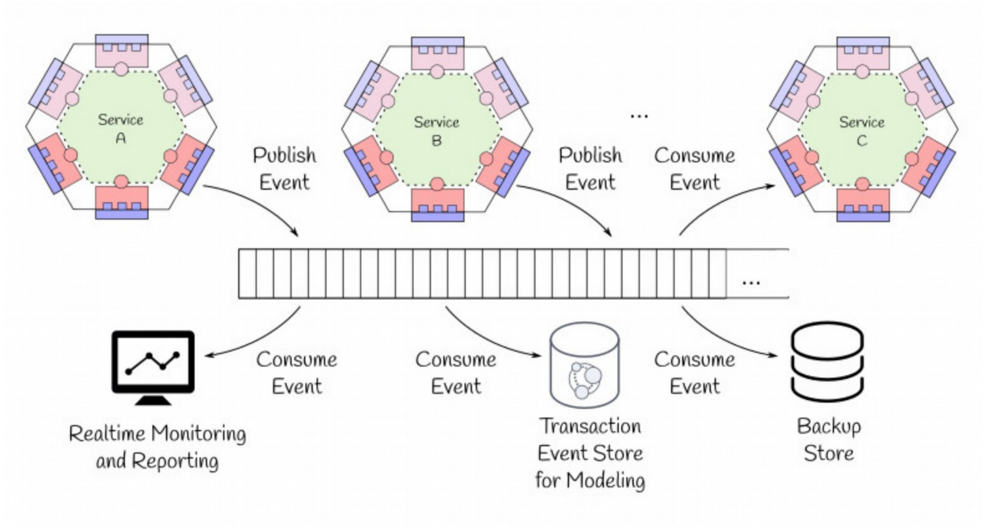
From: Y. Matsuda, Waltz: A Distributed Write-Ahead Log (wepay.com).

Waltz: Optimistic Locking


Read V=1 with Client-HWM=100
Read V=1 with Client-HWM=100
Increment V=2
App
App
Increment V=2
Current Highest TxnID=100
... | TxnID=90 V=1 | ... | TxnID=101 | TxnID=102
Lock-HWM V→90
App
App
Write V=2 with Write Lock on V (Client-HWM=100)
Client-HWM ≥ Lock-HWM
... | TxnID=90 V=1 | ... | TxnID=101 V=2 | TxnID=102
Lock-HWM V→90 ➡ V→101
App
App
Write V=2 with Write Lock on V (Client-HWM=100)
Client-HWM < Lock-HWM
Lock Failure! Txn Rejected
... | TxnID=90 V=1 | ... | TxnID=101 V=2
Lock-HWM V→101

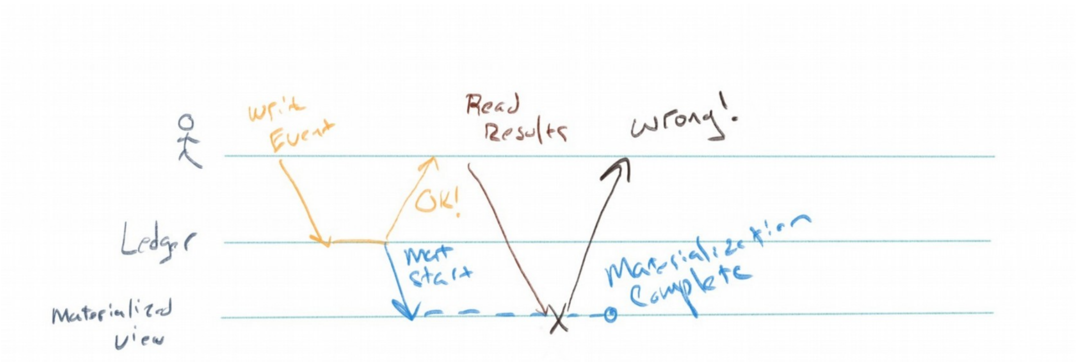
From: Y. Matsuda, Waltz: A Distributed Write-Ahead Log (wepay.com).

Event-Sourcing: Not Easy!



What happens when schemas change?
Materialization cost?
Eventual consistence?
CQRS pattern.

But read-my-writes?



From: https://chriskiehl.com/article/event-sourcing-is-hard

Caching Services

Caching Service

- the fastest request is the one not made!
- storage systems cannot take thousands of concurrent hits
- partitioned storage systems force joins outside DBs
- the social graph is highly connected

Requirements:

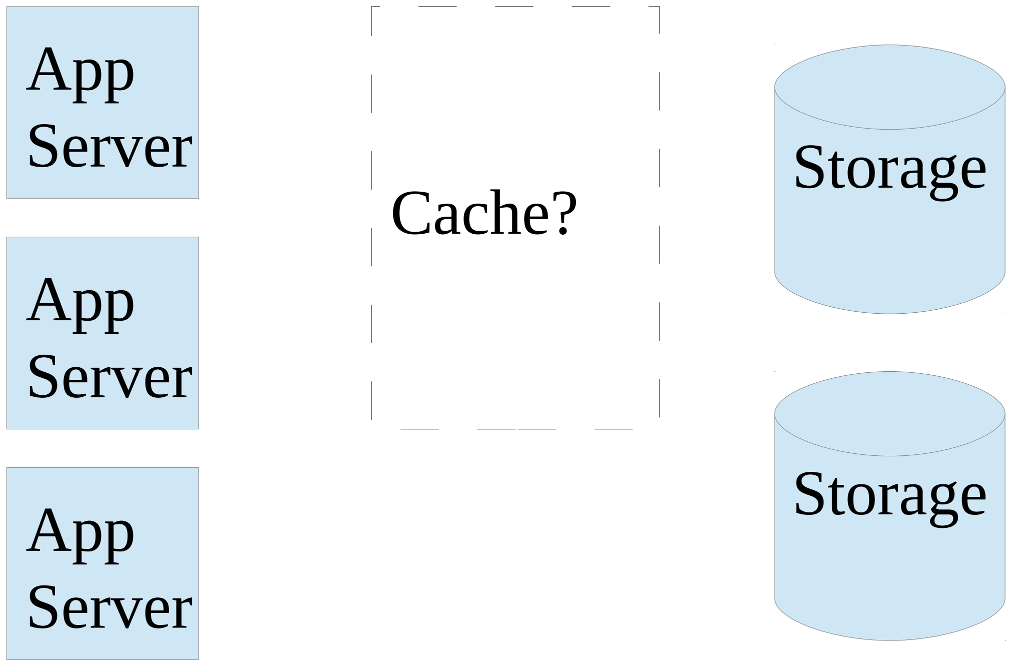
1. Able to add machines to the cache service
2. No “thundering herds” due to placement changes
2. Needs to replicate cache entries
3. Needs to be very fast
4. Optional disk backup
5. From ram to ssd
6. Different cache replacement policies supported (danger!)

Exercise: Design of a Caching Service

API:

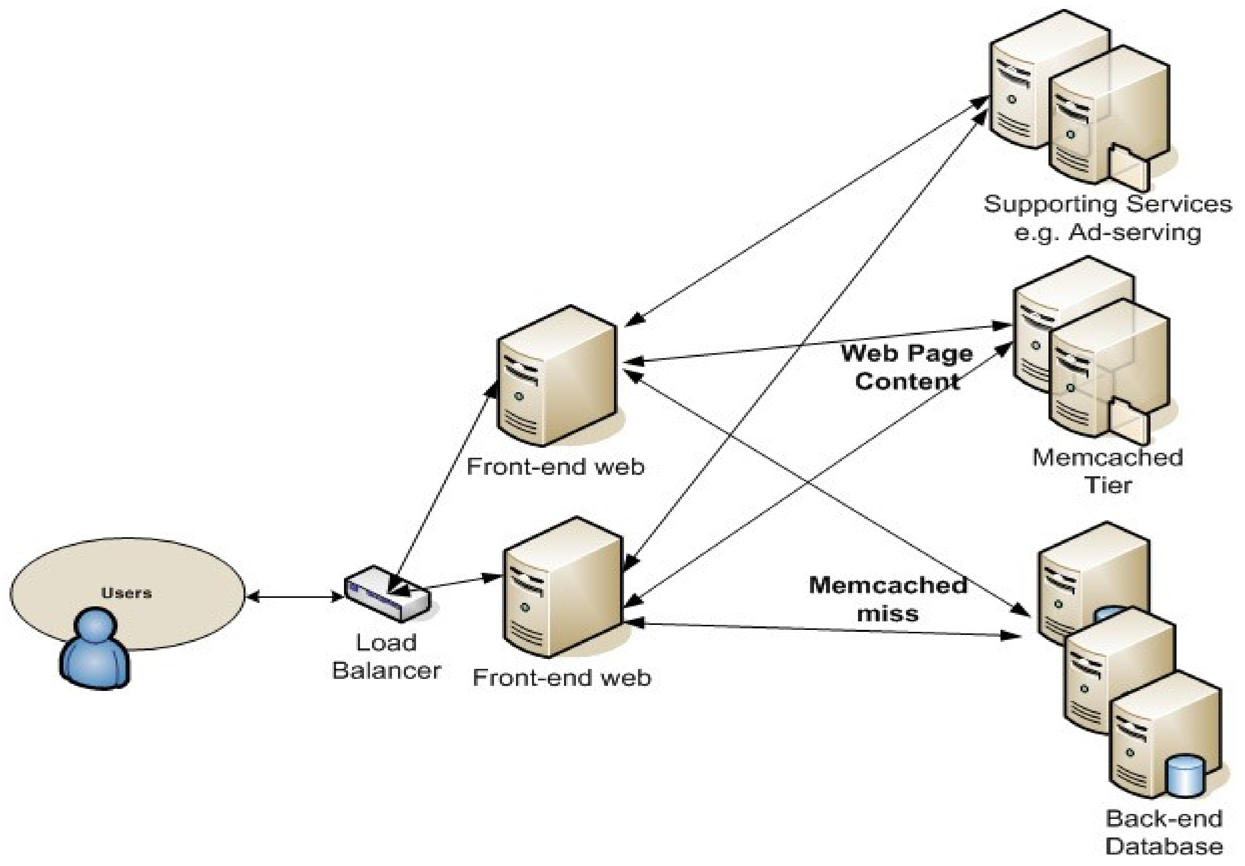
Put (key, value)

Value = get (key)



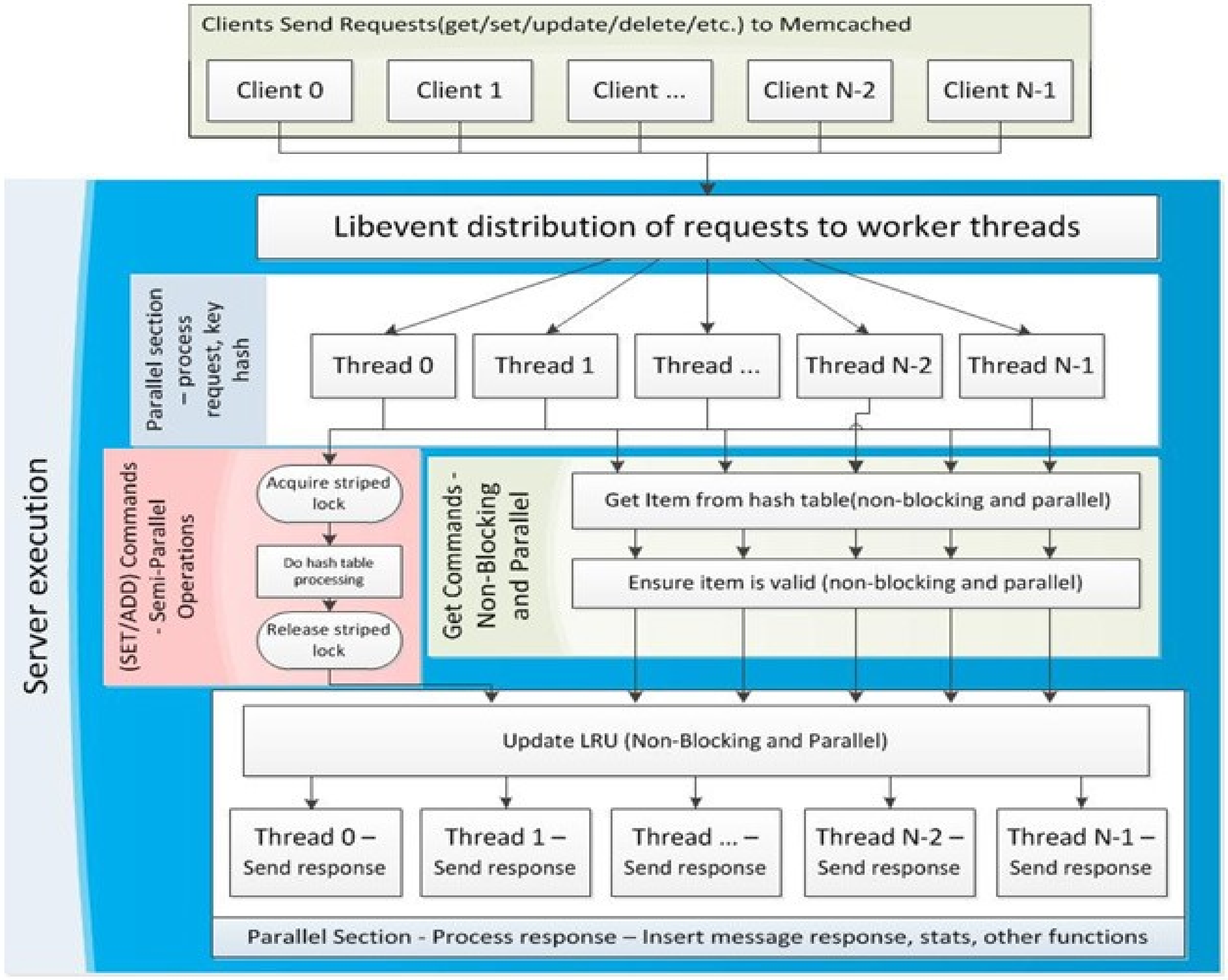
Design a cache architecture and discuss performance, scalability and behavior in case of failures!

Example: Memcached Caching Service



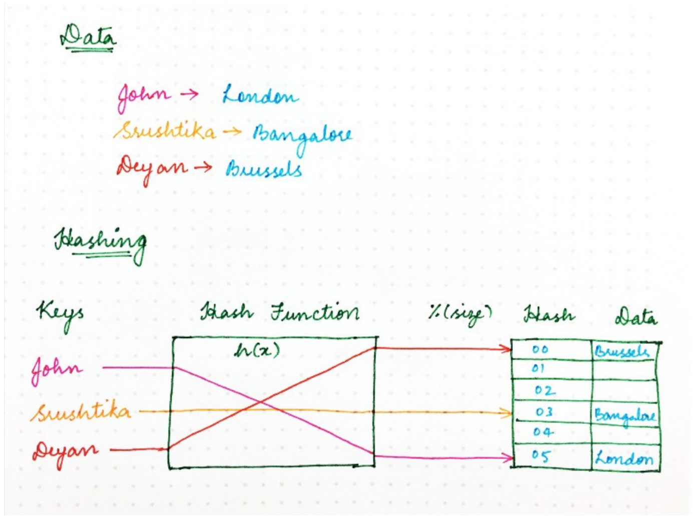
Source: Enhancing the Scalability of Memcached, J. T. Langston Jr , August 16, 2012.
Warning: A cache miss will destroy your storage system at 100k requests/sec! For replication: use two memcached arrays or use a memory based NoSQL store with built in replication.

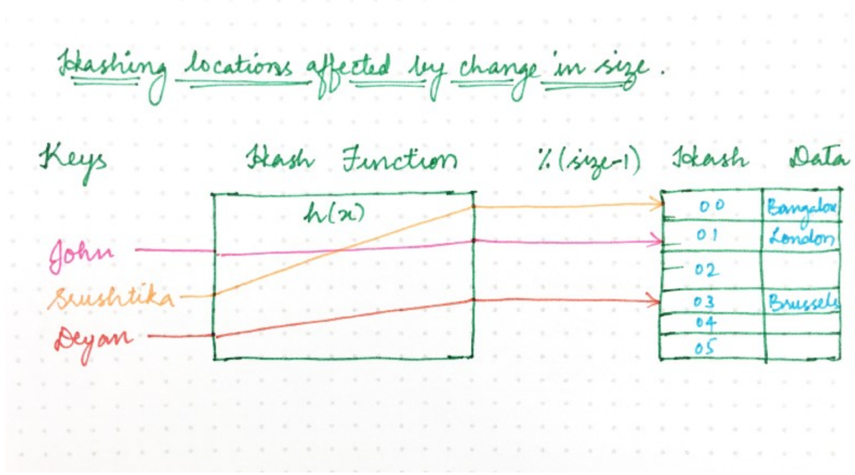
Parallel Memcached Algorithm


Clients Send Requests(get/set/update/delete/etc.) to Memcached
Client 0
Client 1
Client ...
Client N-2
Client N-1
Server execution
Libevent distribution of requests to worker threads
Parallel section – process request, key hash
Thread 0
Thread 1
Thread ...
Thread N-2
Thread N-1
(SET/ADD) Commands - Semi-Parallel Operations
Acquire striped lock
Do hash table processing
Release striped lock
Get Commands - Non-Blocking and Parallel
Get Item from hash table(non-blocking and parallel)
Ensure item is valid (non-blocking and parallel)
Update LRU (Non-Blocking and Parallel)
Thread 0 – Send response
Thread 1 – Send response
Thread ... – Send response
Thread N-2 – Send response
Thread N-1 – Send response
Parallel Section - Process response – Insert message response, stats, other functions

Source: Enhancing the Scalability of Memcached, J. T. Langston Jr , August 16, 2012. Get/Store/Delete locks removed, no CAS in Bag LRU.

The Problem: Changing Machine Count



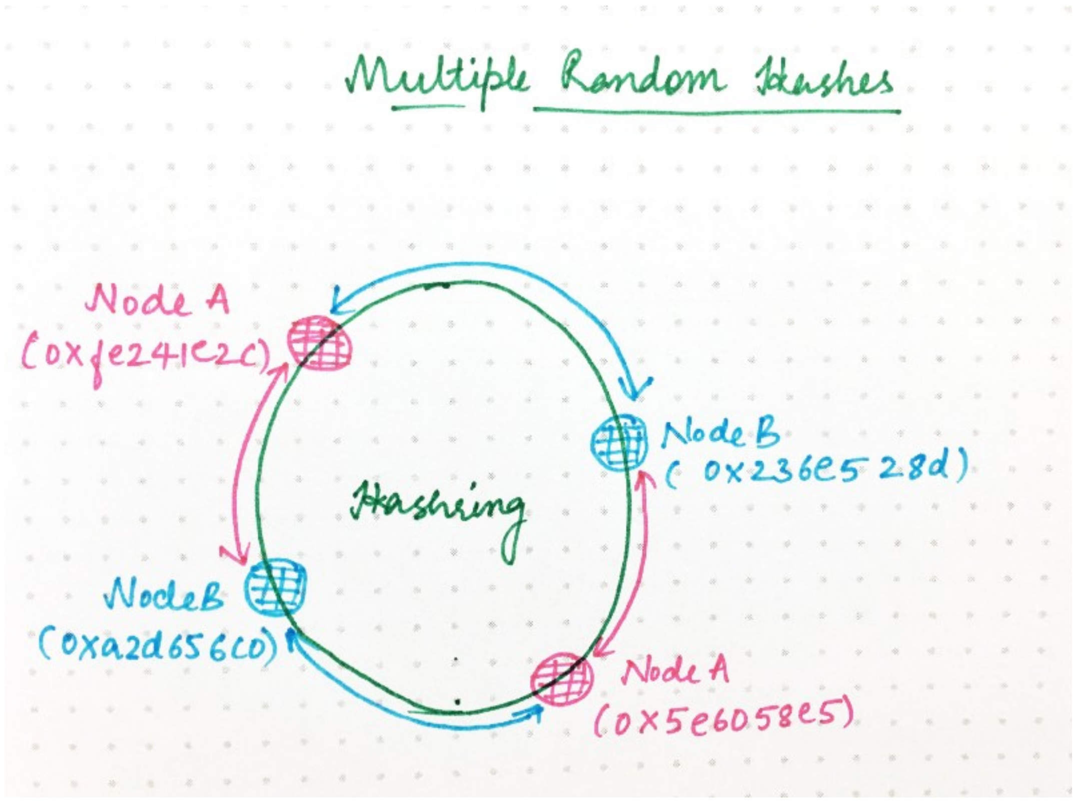


Machine count old

Machine count new

http://highscalability.com/blog/2018/6/18/how-ably-efficiently-implemented-consistent-hashing.html

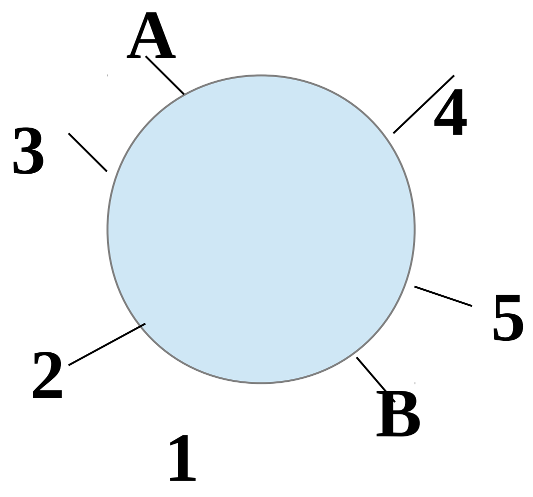
Solution: Consistent Hashing (Ring)

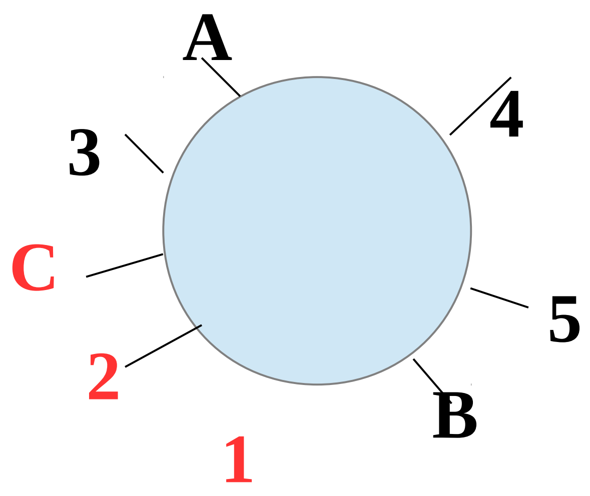


Machines are mapped into a ring. The position decides about the key-space a machine is responsible for. Machines can be mapped to several (virtual) positions

http://highscalability.com/blog/2018/6/18/how-ably-efficiently-implemented-consistent-hashing.html

Simple Consistent Hashing Algorithm

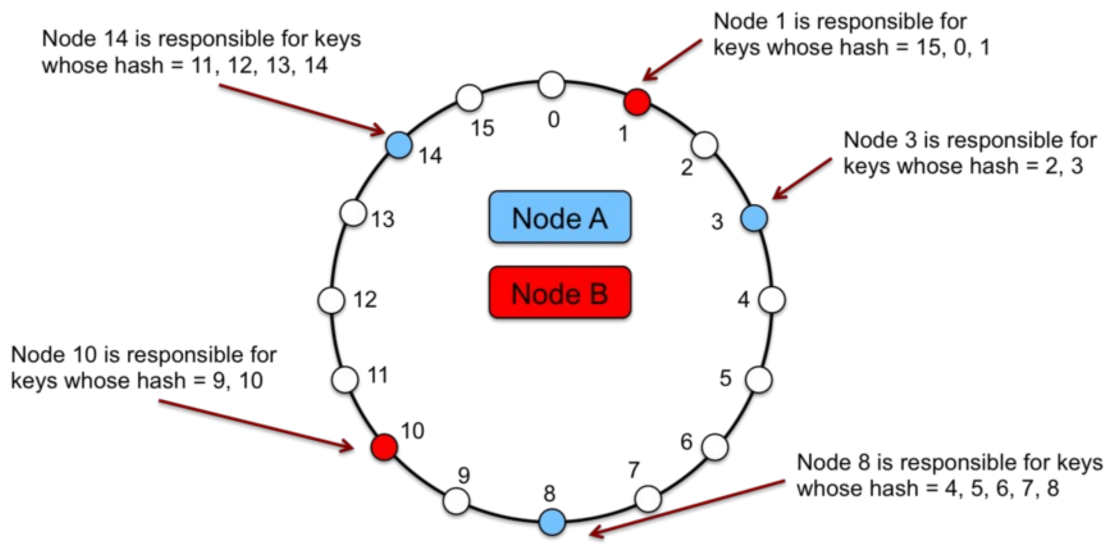




„(i) Both URLs and caches are mapped to points on a circle using a standard hash function. A URL is assigned to the closest cache going clockwise around the circle. Items 1, 2, and 3 are mapped to cache *A*. Items 4, and 5 are mapped to cache *B*. (ii) When a new cache is added the only URLs that are reassigned are those closest to the new cache going clockwise around the circle. In this case when we add the new cache only items 1 and 2 move to the new cache *C*. Items do not move between previously existing caches. „[Kager et.al., Web Caching with Consistent Hashing, MIT]

On average key/slots elements will have to be moved when a new node joins.

Dynamo Consistent Hashing Algorithm

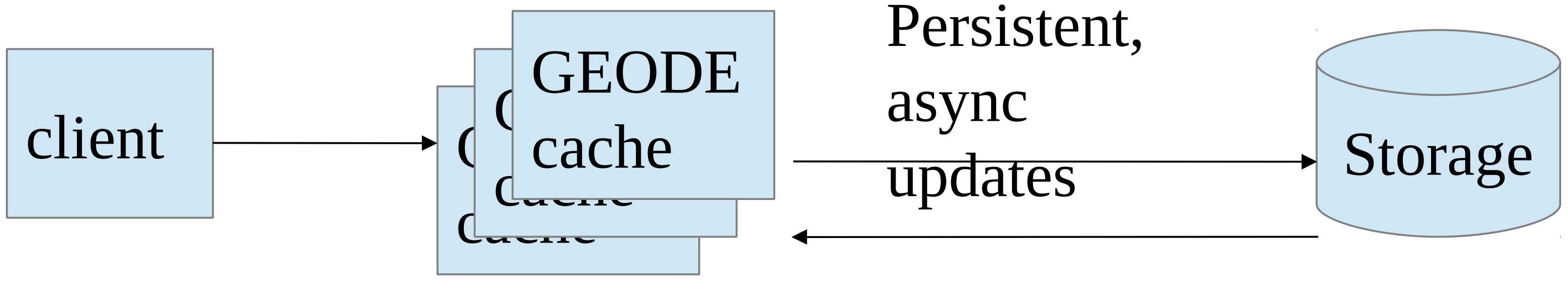

Node 14 is responsible for keys whose hash = 11, 12, 13, 14
Node 1 is responsible for keys whose hash = 15, 0, 1
Node 3 is responsible for keys whose hash = 2, 3
Node 10 is responsible for keys whose hash = 9, 10
Node 8 is responsible for keys whose hash = 4, 5, 6, 7, 8
Node A
Node B
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Dynamo separates placement from partitioning and allows much more flexibility with respect to nodes and locations. The trick is to create virtual nodes and assign those to real machines. In the above diagram, virtual node B is responsible for node 1 an 10. The placement vs. partitioning issue will come up again when we talk about sharding.

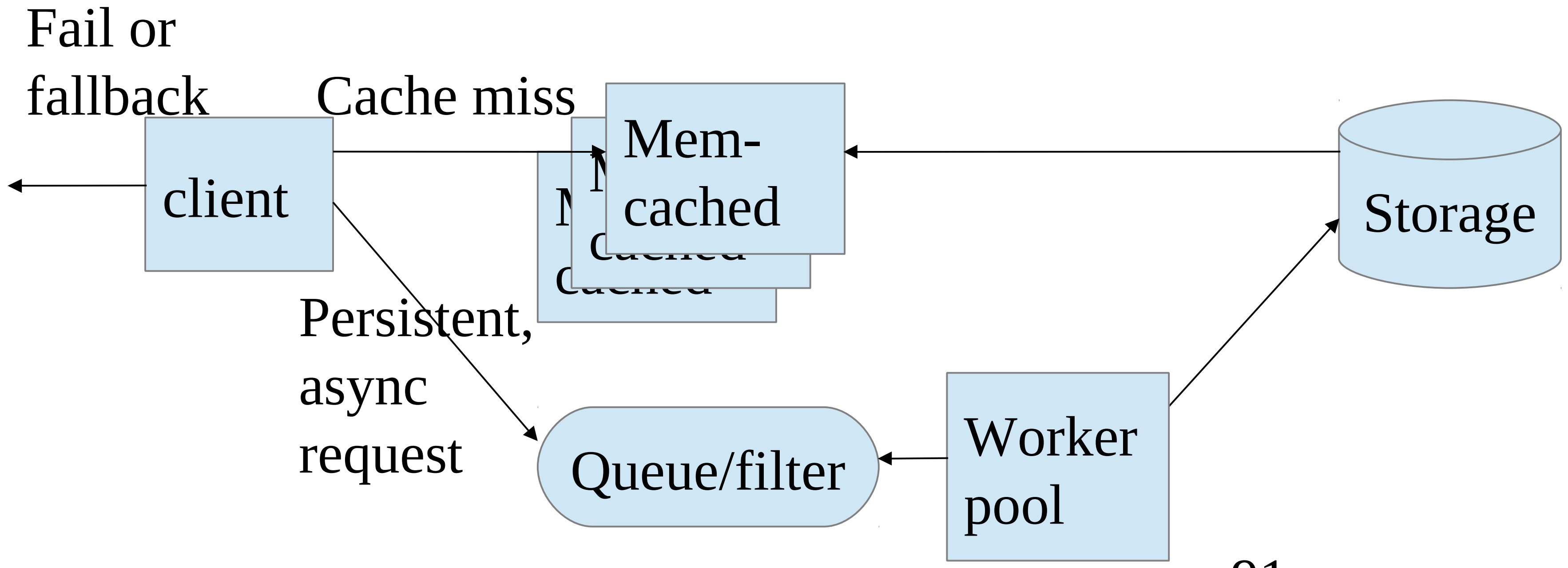
Load Balancing is much easier due to the additional indirection. Dynamo also replicates data over the next n-1 nodes if requested. Lit: DeCandia et.al., Dynamo: Amazon's Highly Available Key-value Store, SOSP'07, October 14–17, 2007, Stevenson, Washington, USA. Copyright 2007 ACM.

WARNING: Dynamo is "eventual consistent" (last-write-wins)

Alternative Cache Architectures



Write through cache, pull



request cache, push

Cache-Patterns

Pull: during request time, concurrent misses, client crashes lead to outdated caches, complicated handling of concurrent misses and updates, slow and dangerous for backends.

Push: Automated push can lead to updates for cached values, which are no longer needed. Only use it for values ALWAYS needed.

Pre-warmed: The system loads the cache before the application accepts client requests. (Big applications with pull caches could not even boot without this)

In all cases: Beware of LRU or clocked invalidations! Your cache is mission critical!

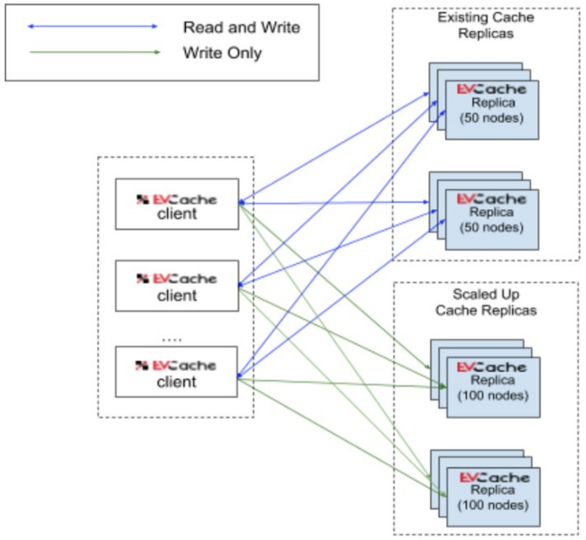
Cache-Design

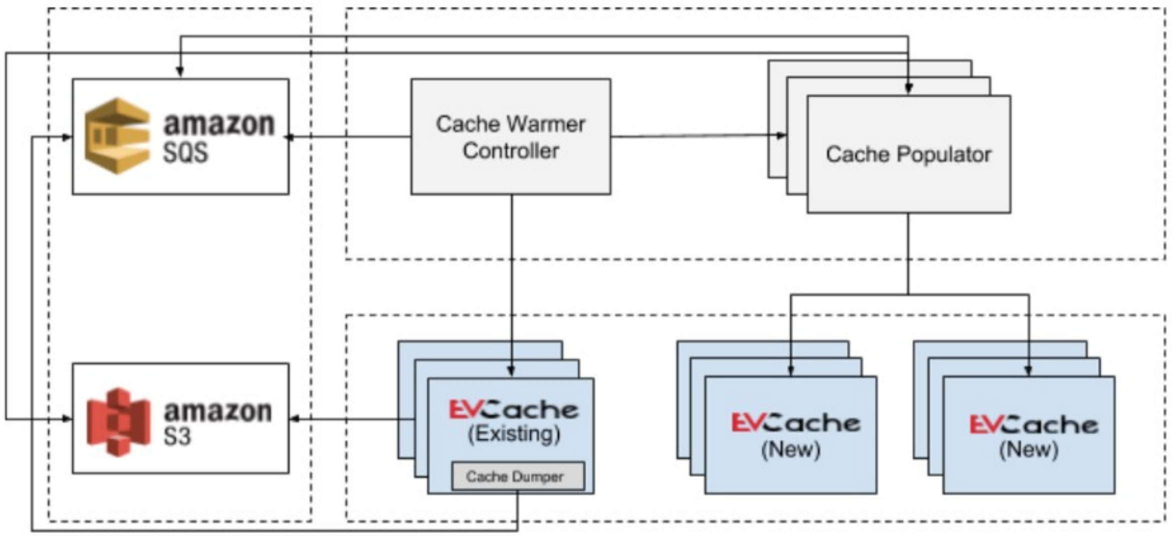
Your cache design absolutely depends on the information architecture of your system:

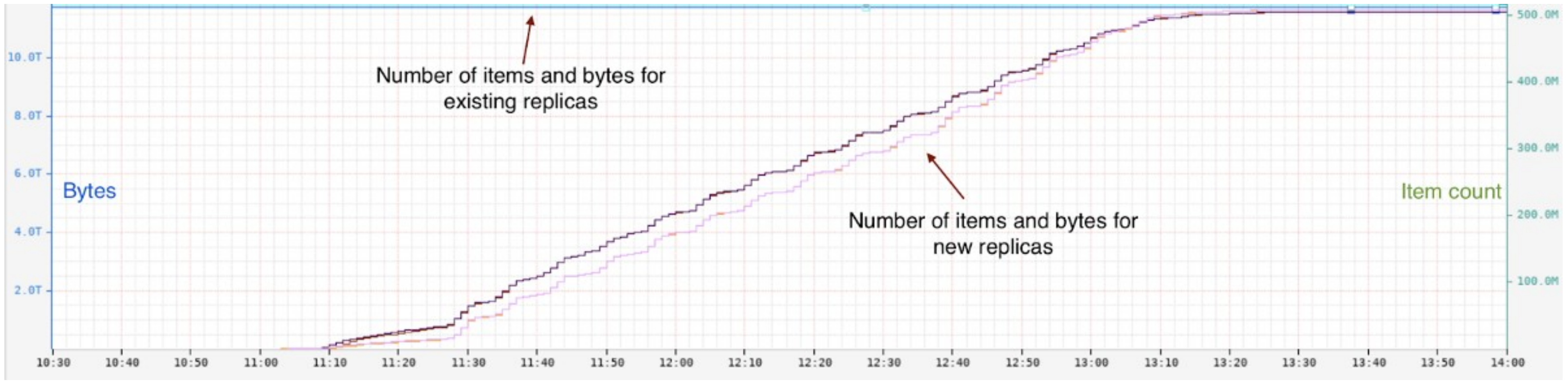
- what kind of information fragments exist?
- what is the lifecycle of those fragments?
- how long are the fragments valid?
- what kind of effects does an invalidate of a value cause?
- are there dependencies between fragments, pages etc.?

We will discuss an example IA in our design session! See "Design of a Modern Cache" by B.Manes for advanced eviction policies, probabilistic data structures (CountMin sketch) and high concurrency buffers.

Netflix Cache-Design for Replication and Scale



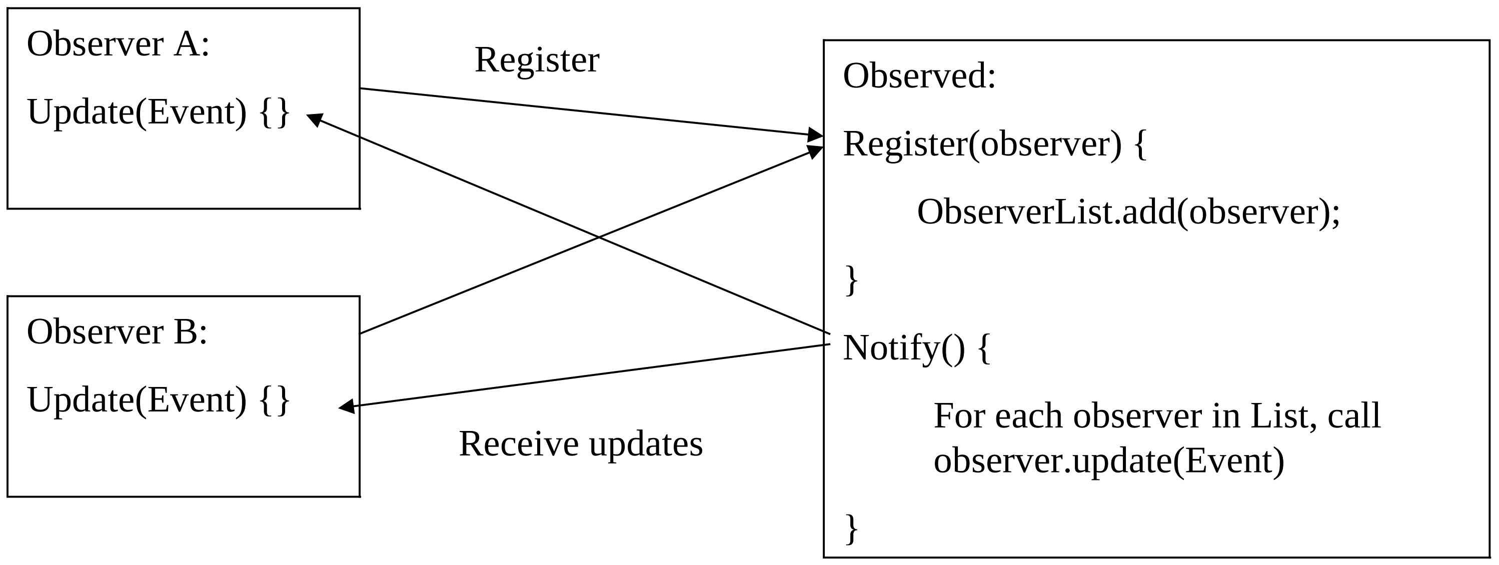




Dump keys and data to external storage (batch) to avoid disturbing client traffic. Send meta-data to populator for cache warming. Watch out for changes during this process… D. Jayaraman, S. Madappa, S. Enugula, I. Papapanagiotou, https://medium.com/netflix-techblog/cache-warming-agility-for-a-stateful-service-2d3b1da82642

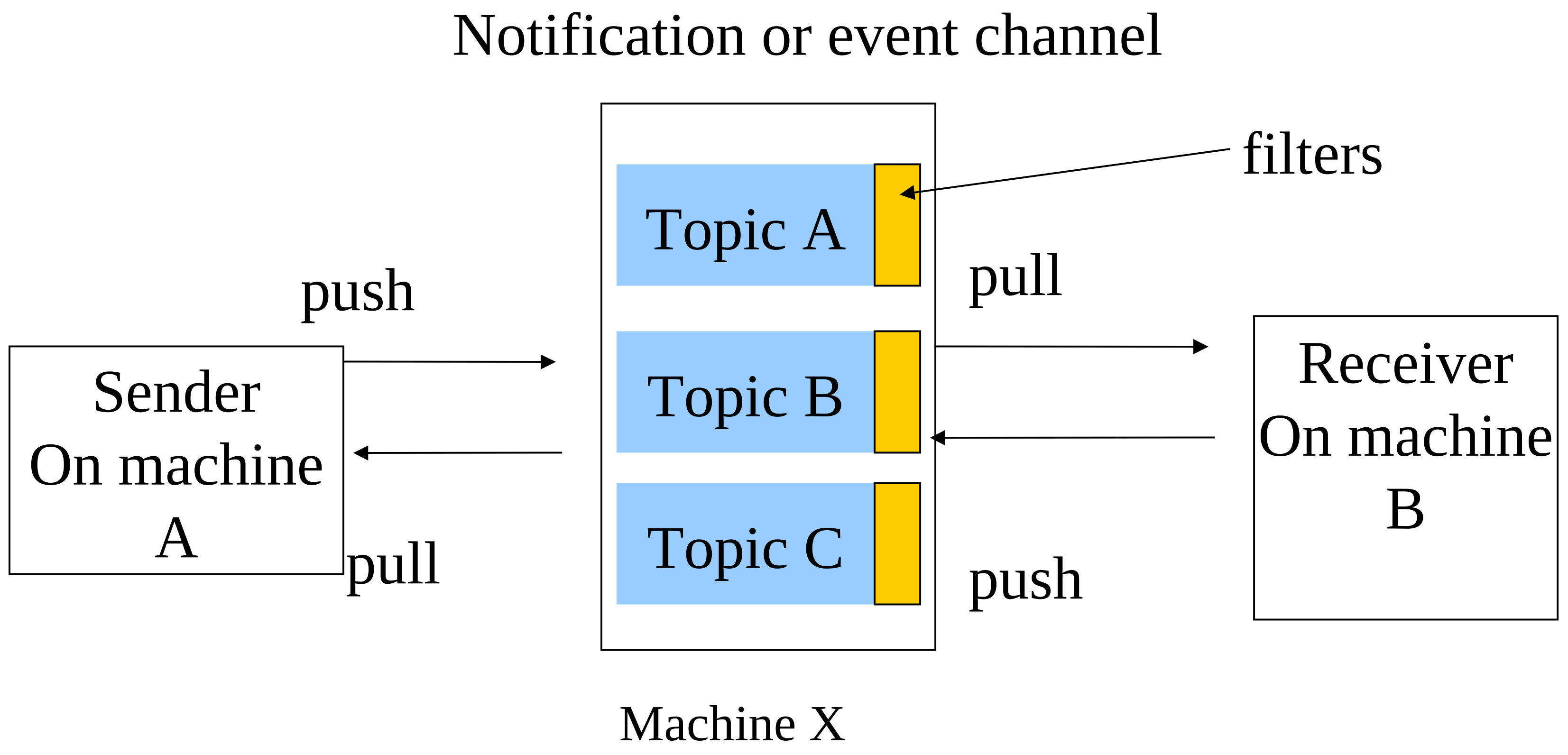
Async Rulez: Event Processing

Local, Synchronous Events: Observer Pattern



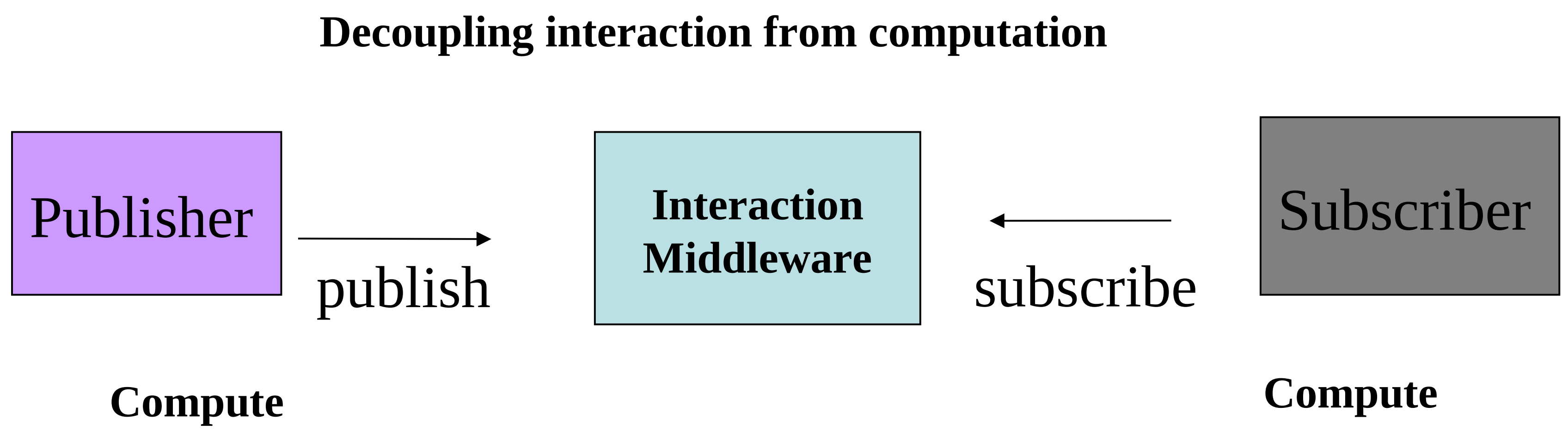
Even in the local case these observer implementations have problems: Updates are sent on one thread. If an observer does not return, the whole mechanism stops. If an observer calls back to the observed DURING an update call, a deadlock is easily created. The solution does not scale and is not reliable (if the observer crashes, all registrations are lost). And of course, it does not work remotely because the observer addresses are local.

Distributed Events



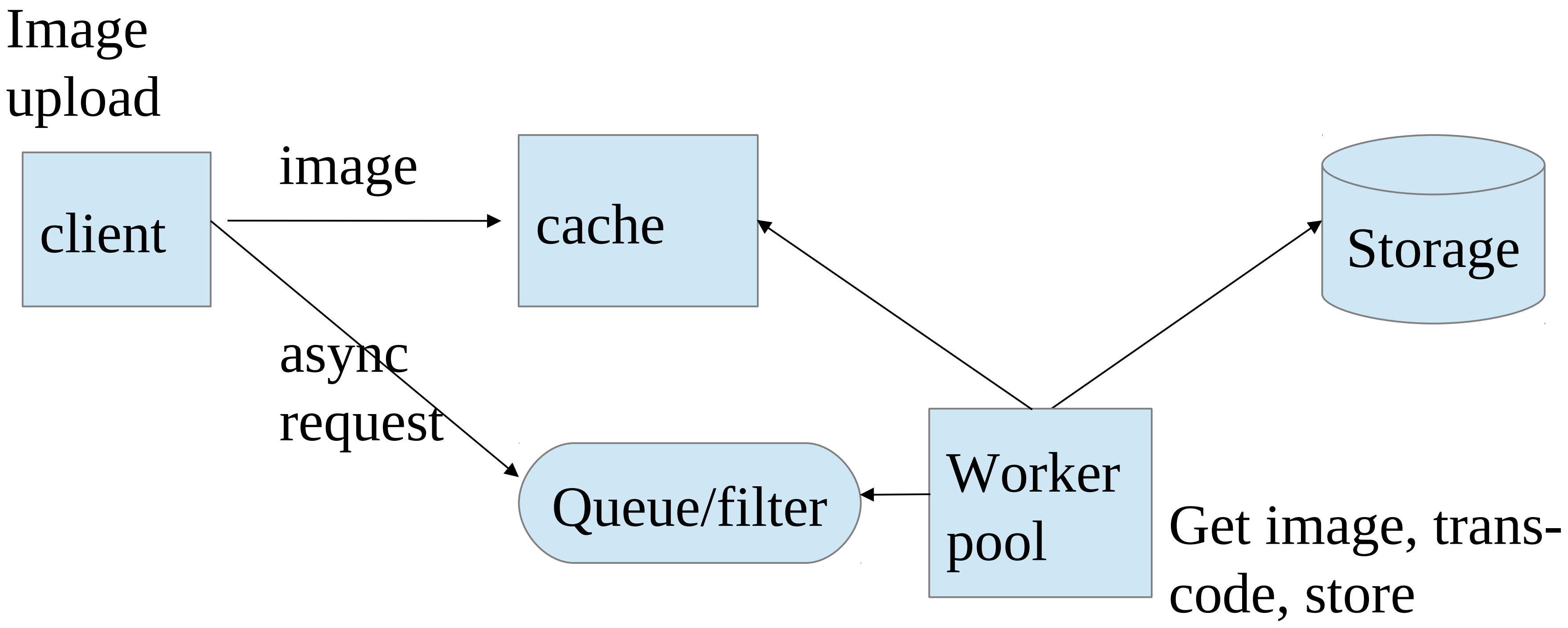
Various combinations of push and pull models are possible.
Receivers can install filters using a constraint language to
filter content. This reduces unwanted notifications.

Asynchronous Event-Processing



- Programming Style used to de-couple components
- Also used to de-couple asynchronous sub-requests from synchronous main requests
- Implemented as Message-Oriented-Middleware (MOM) or socket-based communication library
- Broker-less or brokered mode

Use Case: De-Couple Slow Requests



Today, message queues are used to de-couple requests, create background jobs, store overflow data and spikes etc.

EP-Interaction-Models

Adressee	Consumer initiated	Producer initiated
Direct	Request/Reply	callback
Indirect	Anonymous Request/Reply	**Event-based**

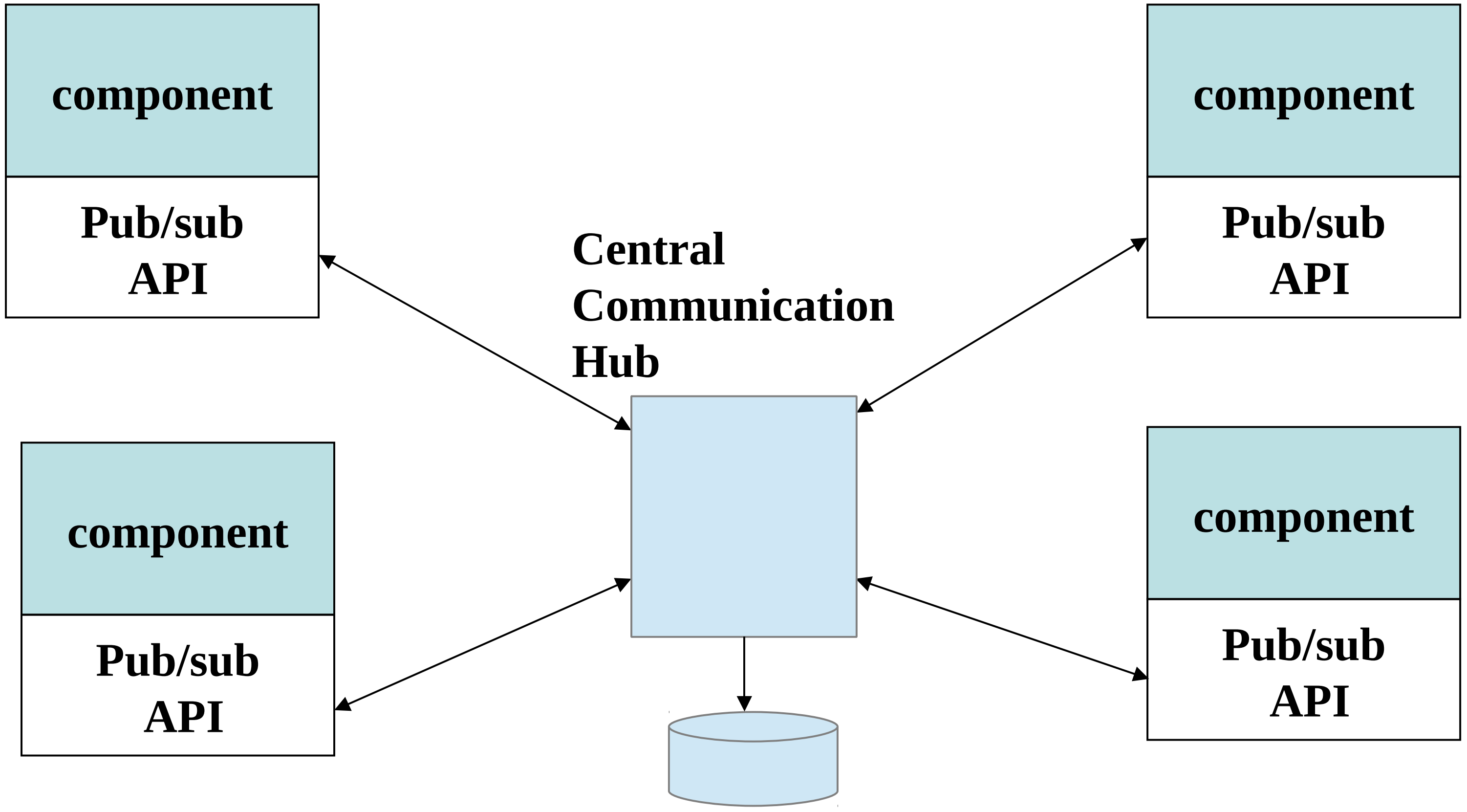
Expecting an immediate „reply“ makes interaction logically synchronous – NOT the fact that the implementation might be done through a synchronous mechanism. This makes an architecture synchronous by implementation (like with naive implementations of the observer pattern).

Interaction Models according to Mühl et.al.

Features of event-driven interaction

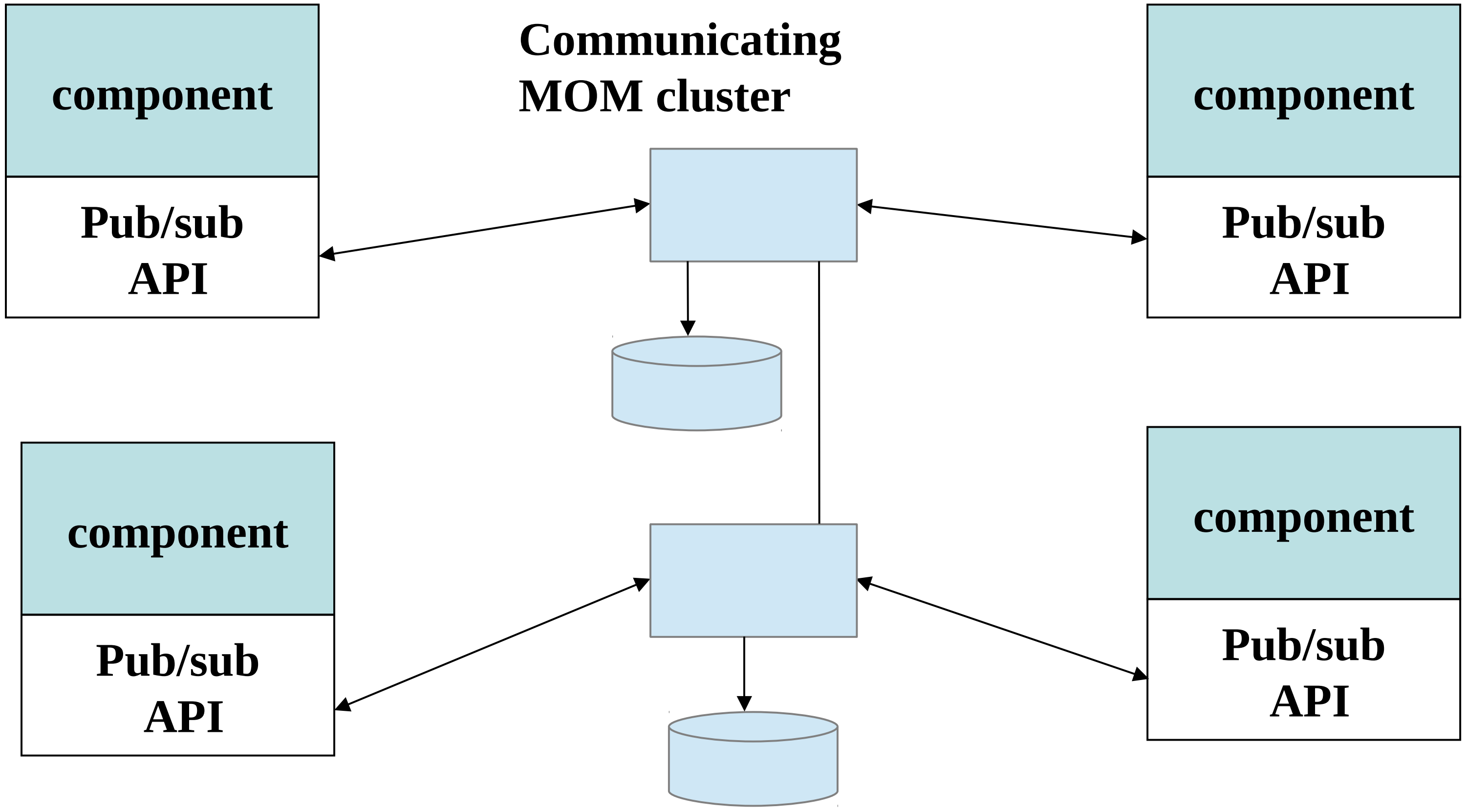
- **Basic event: Everybody can send events, everybody can receive events - no restrictions, filtering etc.**
- **Subscription: Optimization on receiver side. Only events for which a subscription exists are forwarded to receiver. Can trigger publishing too.**
- **Advertisement: Optimization on sender side. Informs receivers about possible events. Avoids broadcast of every event in connection with subscriptions.**
- **Content-based filtering can be used for any purpose. Can happen at sender side, in the middleware or at receiver side.**
- **Scoping: manipulation of routes through an administrative component. Invisible assembly of communicating components through routes.**

Centralized Message-Oriented-Middleware



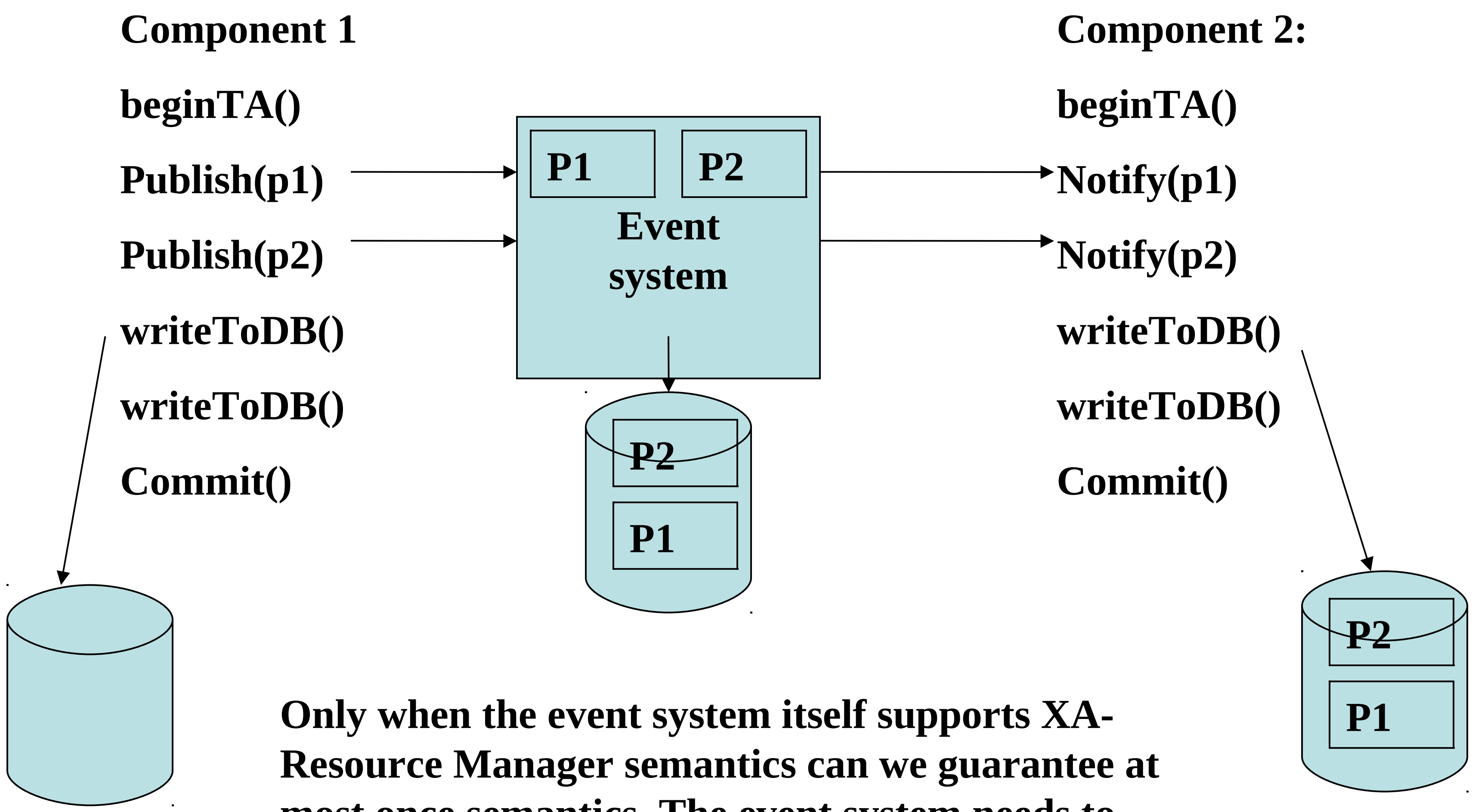
The system collects all notifications and subscriptions in one central place. Event matching and filtering are easy. Creates single-point-of-failure and scalability problems. High degree of control possible. No security/reliability problems on clients. Scalability problems.

Clustered Message-Oriented-Middleware



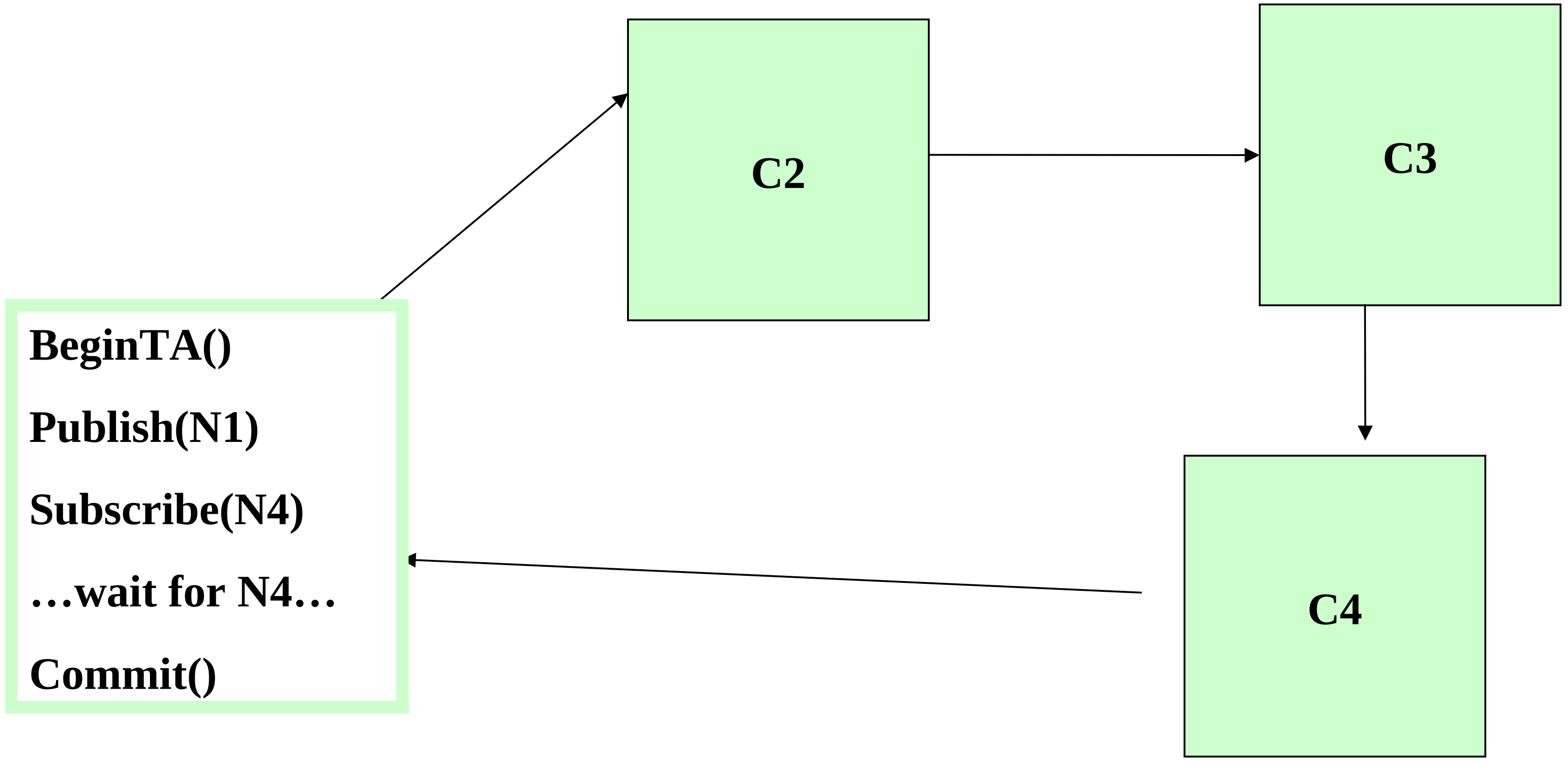
Clustering provides scalability at higher communication costs for consistency. Filtering and routing of notifications can become expensive due to lots of routing/filter-tables at cluster nodes.

External Transactions in event-driven systems



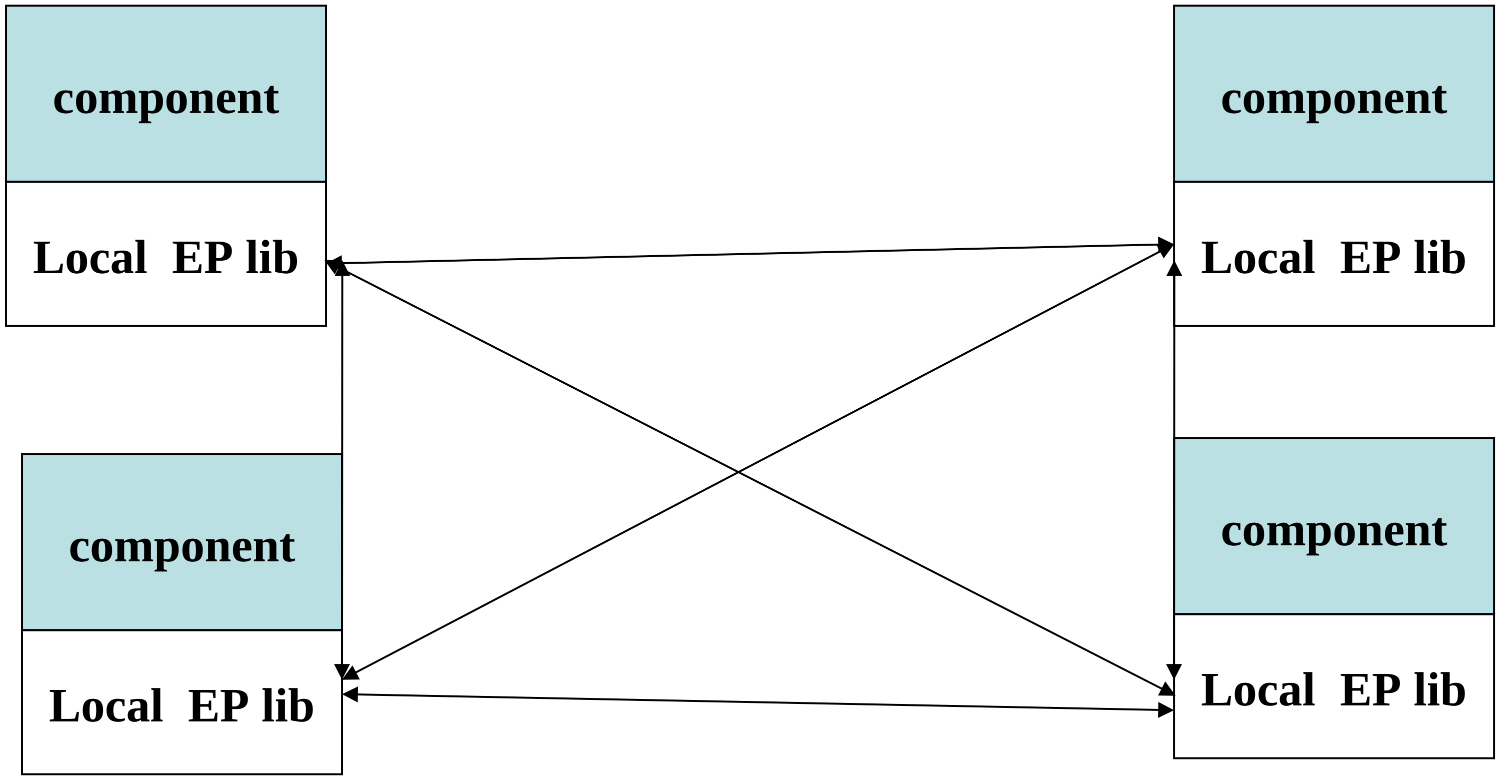
Only when the event system itself supports XA-Resource Manager semantics can we guarantee at most once semantics. The event system needs to store the events in safe storage like a DB. Even if publisher or subscribers crash, events won't get lost!

Wrong feedback expectations with a transaction



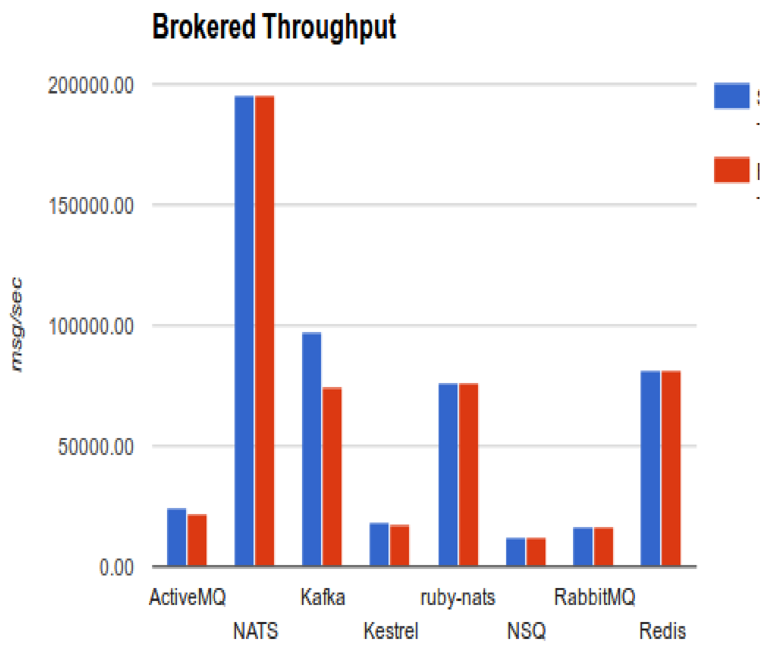
Code that will not work. N4 causally depends on N1 being published by C1 and received by C2. But as publishing by C1 is done within a transaction the notification N1 does not become visible until the end of the transaction – which will not be reached because of the wait for N4 – which will never come.

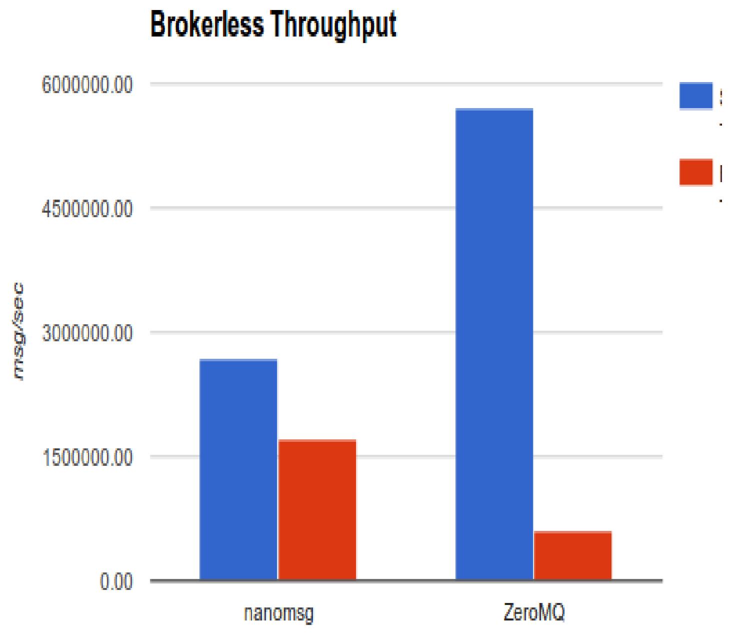
Simple P2P Event Library



The local libraries know about each other, but the components are de-coupled. This broker-less architecture is much faster than brokered ones. It does not provide at-most-once semantics or protection against message loss. Only atomicity and perhaps fifo is guarantieed. ZeroMQ, Aaron and Nanomsg are examples.

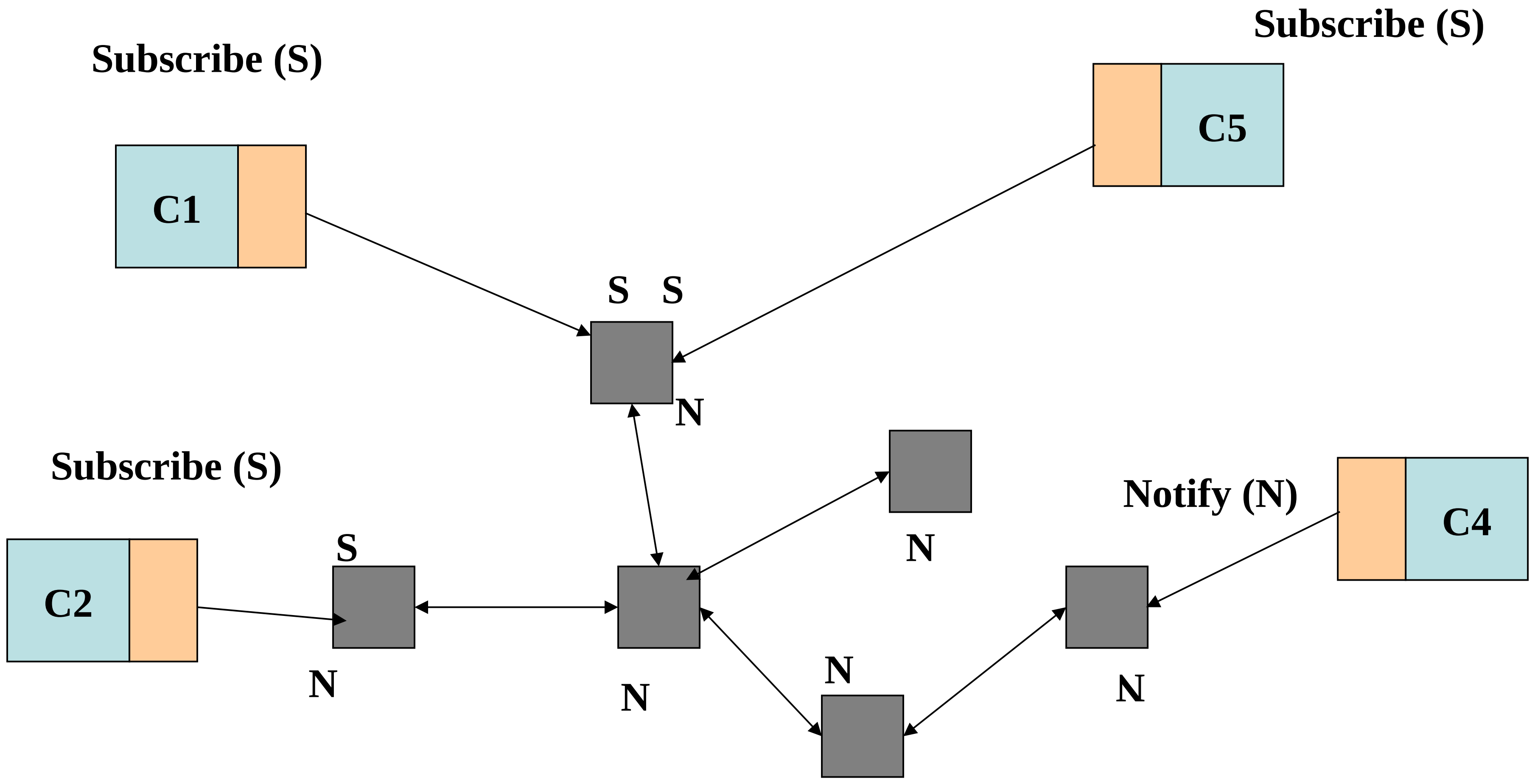
Brokered vs. Broker-less Throughput





http://bravenewgeek.com/dissecting-message-queues/

Special Case: flooding protocols for distribution



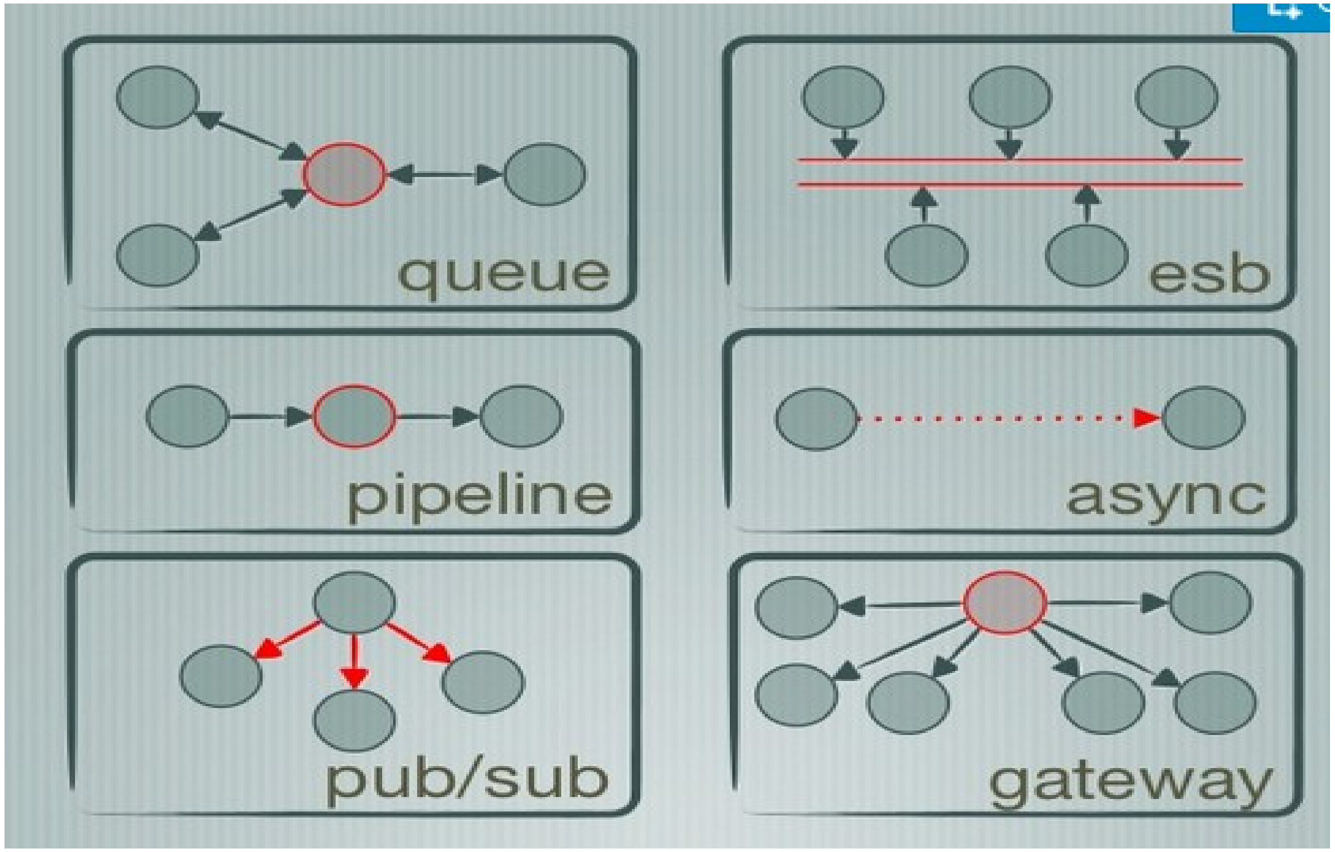
With flooding notifications travel towards subscriptions which are only kept at leaf brokers. See: Mühl et.al. Pg. 22 ff. Advantages are that subscriptions become effective rather quickly and notifications are guaranteed to arrive everywhere. The price is a large number of unnecessary notifications to leaf nodes without subscribers

Example: ZeroMQ

- Brokerless, fast and small socket library for messaging,
- message filtering possible
- connection patterns like pipeline, pub/sub, multi-worker
- various transports (in process, across local process, across machines and multicast groups)
- message-passing process model without need for synchronization
- multi-platform and multi-language
- “suicidal snail” fail-fast mechanism to kill slow subscribers

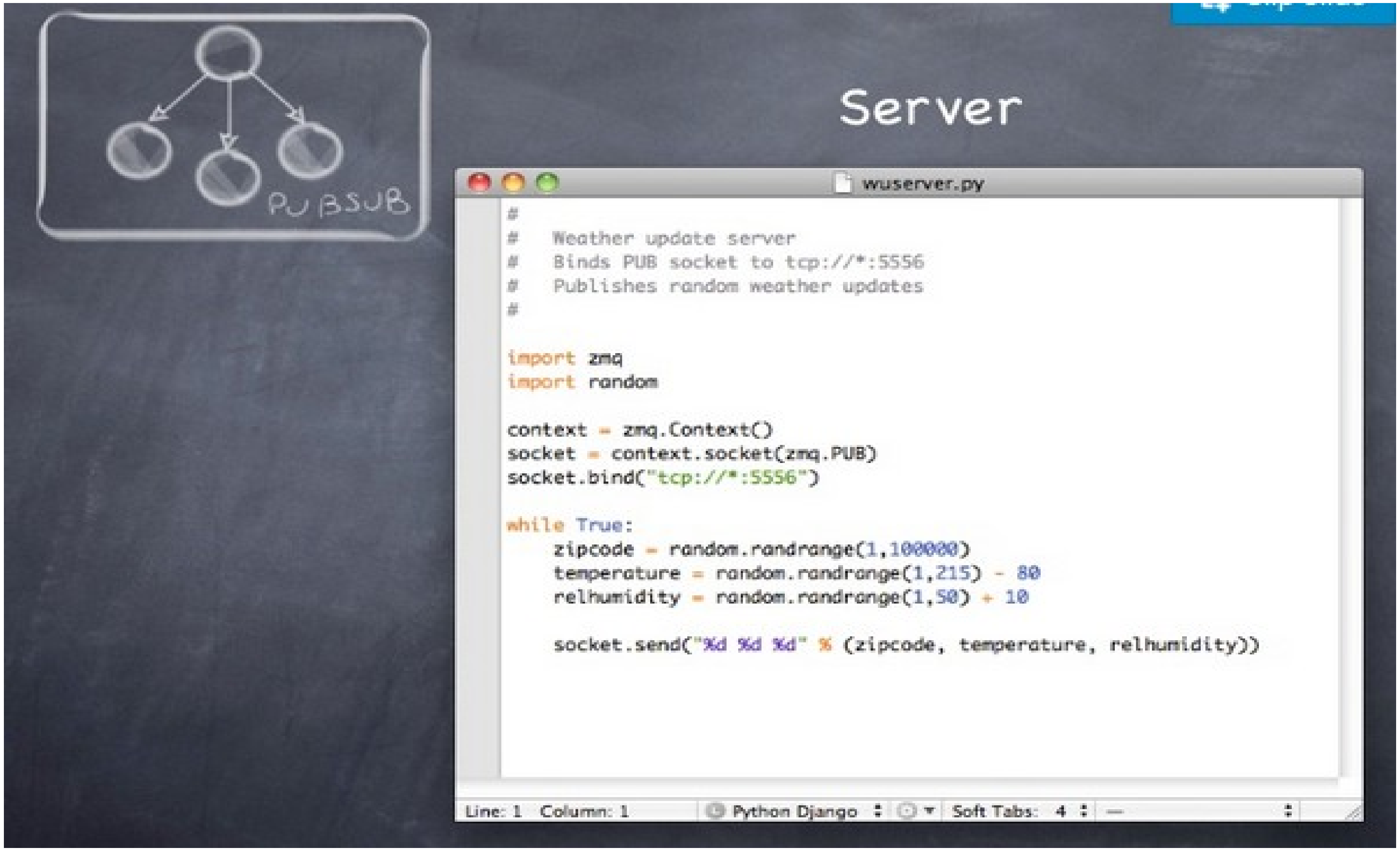
After: P.Hintjens, iMatix, Introduction to ZeroMQ
http://www.slideshare.net/pieterh/overview-of-zeromq

Connection Patterns in ZeroMQ



After: I.Barber, http://www.slideshare.net/IanBarber/zeromq-is-the-answer?qid=e9d81d72-45dd-4b28-a6b0-49fce5617a35&v=default&b=&from_search=1

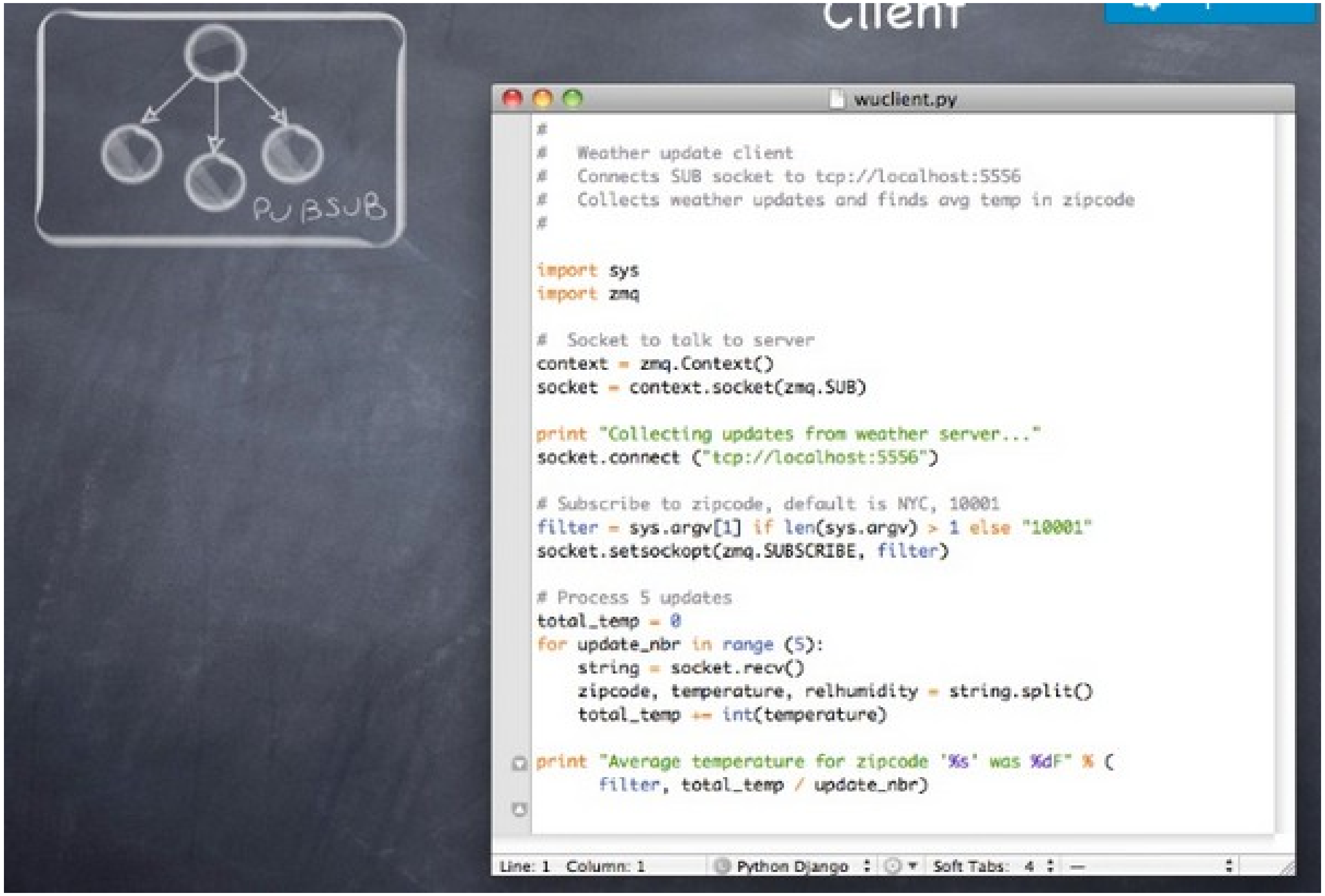
Pub-Sub Server in ZeroMQ



```
#
#   Weather update server
#   Binds PUB socket to tcp://*:5556
#   Publishes random weather updates
#

import zmq
import random

context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.bind("tcp://*:5556")

while True:
    zipcode = random.randrange(1,100000)
    temperature = random.randrange(1,215) - 80
    relhumidity = random.randrange(1,50) + 10

    socket.send("%d %d %d" % (zipcode, temperature, relhumidity))
```

After: Francesco Crippa, http://www.slideshare.net/fcrippa/europycon2011-implementing-distributed-application-using-zeromq

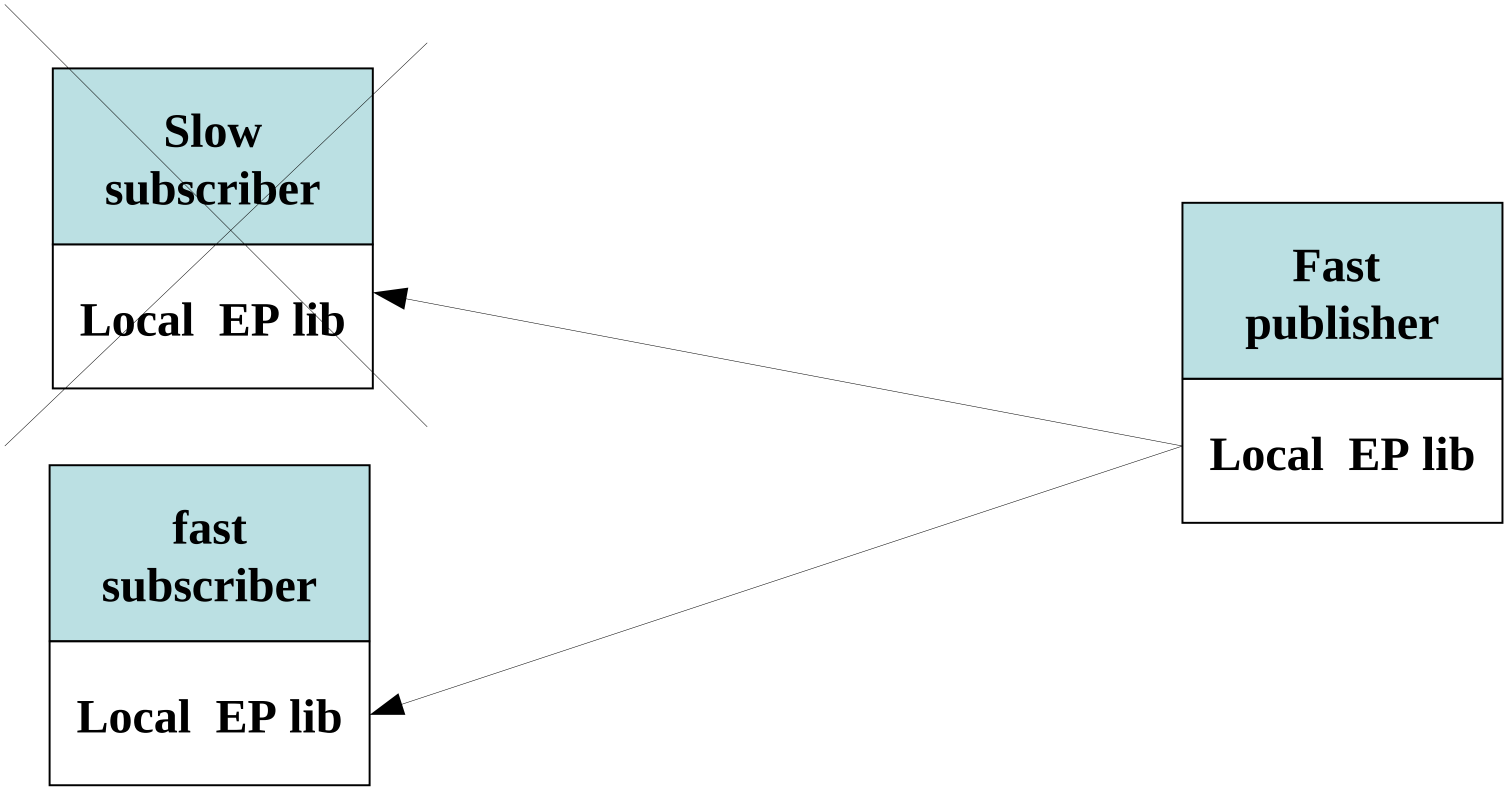
Pub-Sub Client in ZeroMQ



```python
#
#   Weather update client
#   Connects SUB socket to tcp://localhost:5556
#   Collects weather updates and finds avg temp in zipcode
#

import sys
import zmq

#  Socket to talk to server
context = zmq.Context()
socket = context.socket(zmq.SUB)

print "Collecting updates from weather server..."
socket.connect ("tcp://localhost:5556")

# Subscribe to zipcode, default is NYC, 10001
filter = sys.argv[1] if len(sys.argv) > 1 else "10001"
socket.setsockopt(zmq.SUBSCRIBE, filter)

# Process 5 updates
total_temp = 0
for update_nbr in range (5):
    string = socket.recv()
    zipcode, temperature, relhumidity = string.split()
    total_temp += int(temperature)

print "Average temperature for zipcode '%s' was %dF" % (
      filter, total_temp / update_nbr)
```

After: Francesco Crippa, http://www.slideshare.net/fcrippa/europycon2011-implementing-distributed-application-using-zeromq

„Snail Suicide"



There is no broker to buffer large amounts of messages. ZeroMQ uses a surprising solution for this problem: A subscriber which gets overrun, commits suicide! This prevents the fan-out from becoming slow for all participants and makes buffering (except for a little) unnecessary.

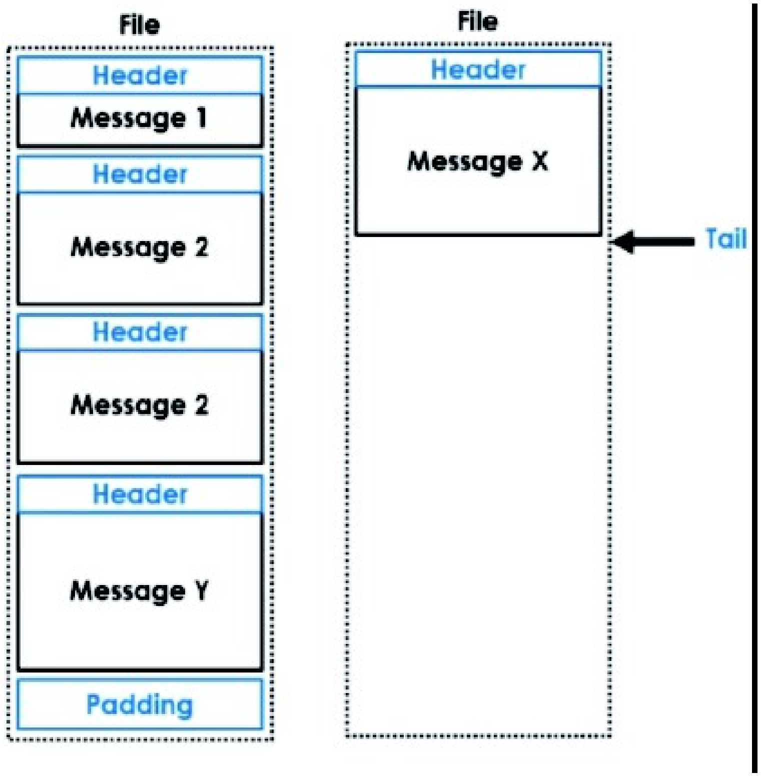
Nanomsg – a re-write of ZeroMQ

- zero-copy mechanism to bypass CPU
- sockets and threads are de-coupled (allows user-level threads)
- fast Radix-trie to maintain subscriptions
- scalability patterns provided (PAIR, REQREP, PIPELINE, BUS, PUBSUB, and SURVEY)
- Posix compliant sockets,
- pluggable interfaces for transports and messaging patterns
- thread-safe sockets, interaction as sets of state machines

From: Tyler Treat, A Look at Nanomsg and Scalability Protocols, http://bravenewgeek.com/a-look-at-nanomsg-and-scalability-protocols/

Aaron – extremely fast P2P messaging

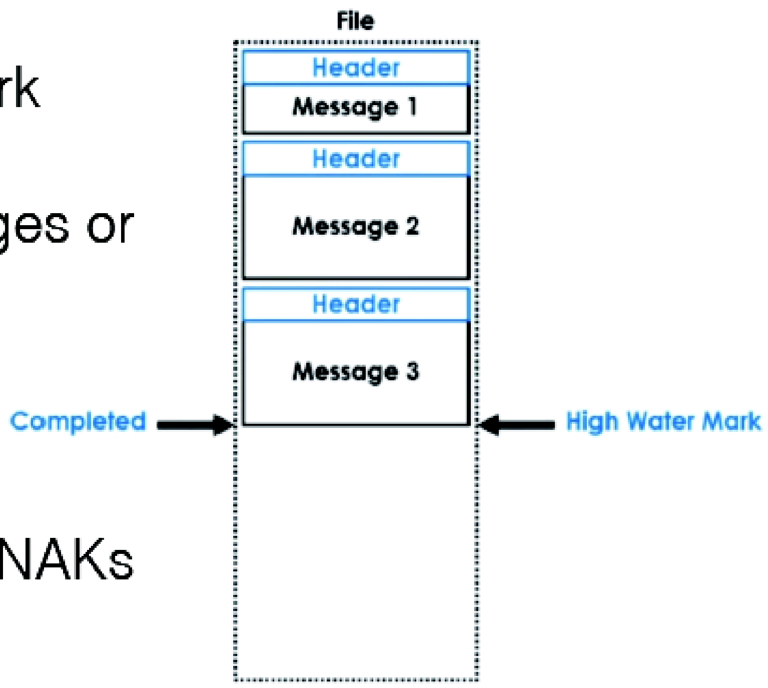
- 2 threads write messages simultaneously
- Y wins the race to increment the tail
- X increments tail & runs past buffer
- Y can use the space it allocated
- X puts padding record to complete buffer
- X moves to next buffer



Wait-free concurrent writing of incoming message stream with no head of line blocking. Further optimiziations: Direct write of header structure to network, memory mapped files. >6M/sec messages with 40 bytes each

Aaron – Interaction Protocol

- 2 counters: completed & high water mark
- no need to keep lists of missing messages or gaps
- log buffers are linear in memory
- conductor is looking for a gap & sends NAKs with random future offset



No need for ACK messages. Daemon process (conductor) checks for Gaps in receive structure and requests missing messages

The Coming of the Shard

One of the all-time-classics from http://highscalability.com/blog/2009/8/6/an-unorthodox-approach-to-database-design-the-coming-of-the.html

It gives you all the good reasons why you SHOULD shard – and why you should NOT do it ever!

Sharding – divide and conquer!

“The evil wizard Mondain hat attempted to gain control over Sosaria by trapping its essence in a crystal. When the Stranger at the end of Ultima I defeated Mondain and shattered the crystal, the crystal shards each held a refracted copy of Sosaria.”

[http://www.ralphkoster.com/2009/01/08/database_sharding_came_from_uo/] (found in [Scheurer])
Game developers early on discovered the need to partition things, because systems could not carry the load.

Sharding as Partitioning

What: Data types (pictures, comments etc., HORIZONTAL),
Data values (paying customers vs. non-paying, VERTICAL)
Functions: login, upload, query etc.

How: Static sharding functions (time, hash, RR)
Dynamic sharding via meta-data indexes

Where: placement strategy fixed because of sharding function. Virtualized placement supporting lifecycle Changes in data behavior.

One of the best introductions to sharding and partitioning that I found is made by Jurriaan Persyn of Netlog. “Database Sharding at Netlog” is a presentation held at Fosdem 2009 http://www.jurriaanpersyn.com/archives/2009/02/12/database-sharding-at-netlog-with-mysql-and-php/

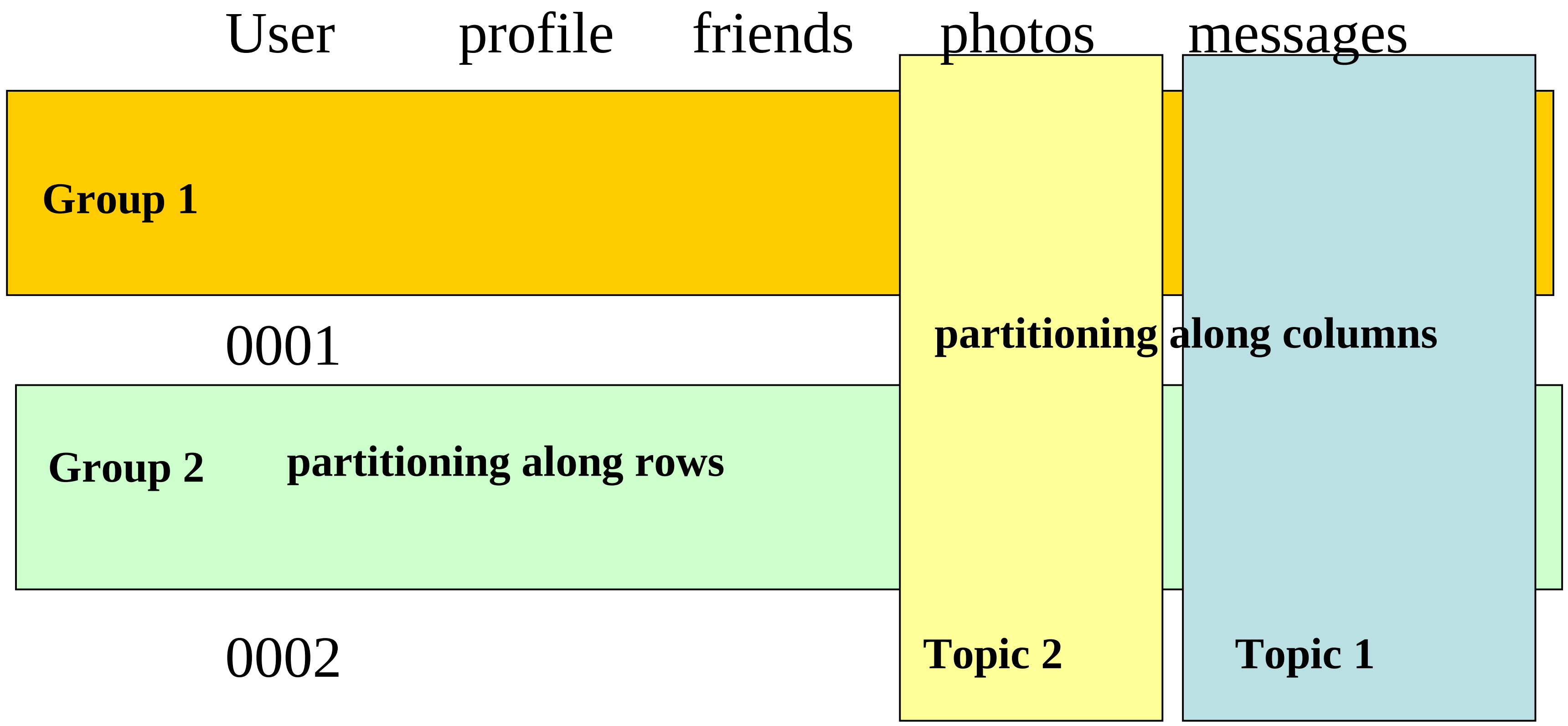
But first: Know your numbers! ("Load-Parameters")

- requests/sec, data/sec
- what is your read/write ratio?
- what is more expensive? Reads or writes, or both?
- what are your traffic patterns? Spikes?
- how much bandwidth to the data do you need? Changing?
- do you need random or sequential access?
- which data grow fastest?
- which functions grow fastest?
- how much replication safety do you need?
- Did you simplify queries? Push them to analytics?

And finally: How will those numbers change, when your caching services are in place? When you fixed queries?

Do not repeat the myspace error of optimizing in one place only! Hint: If you have a bottleneck in your system and your optimizations cannot push the bottleneck to a different spot in your architecture at least for some period of time – then you should re-think your architecture end-to-end! Do you need an edge-cache (e.g. Akamai)? Change your business requests? Add more async services?

Horizontal vs. Vertical Sharding



Sharding Strategies

Persyn lists requirements for a sharding scheme and implementation:

1. allow flexible addition of possibly heterogeneous hardware to balance growth
2. keep application code stable even in case of sharding changes.
3. Allow mapping between application view and sharding view (e.g. using shard API against a non-sharded database)
4. Allow several sharding keys
5. Make sharding as transparent as possible to the application developer by using an API.

And I would like to add:

6. Keep sharding and placement strategies separat!

Sharding Functions

Example: Horizontal sharding of users
A typical key is e.g. the userID given to your customers. Several algorithms can be applied to this key to create different groups (shards).
- A numerical range (users 0-100000, 100001-200000 etc.)
- A time range (1970-80, 81-90, 91-2000 etc.)
- hash and modulo calculation
- directory based mapping (arbitrary mapping from key to shard driven by meta-data table)

A directory-based mapping will allow transparent load-balancing between shards, e.g. when disks with older users become idle! Ideally, the client-storage library will learn the proper function dynamically (stored e.g. in zookeeper)

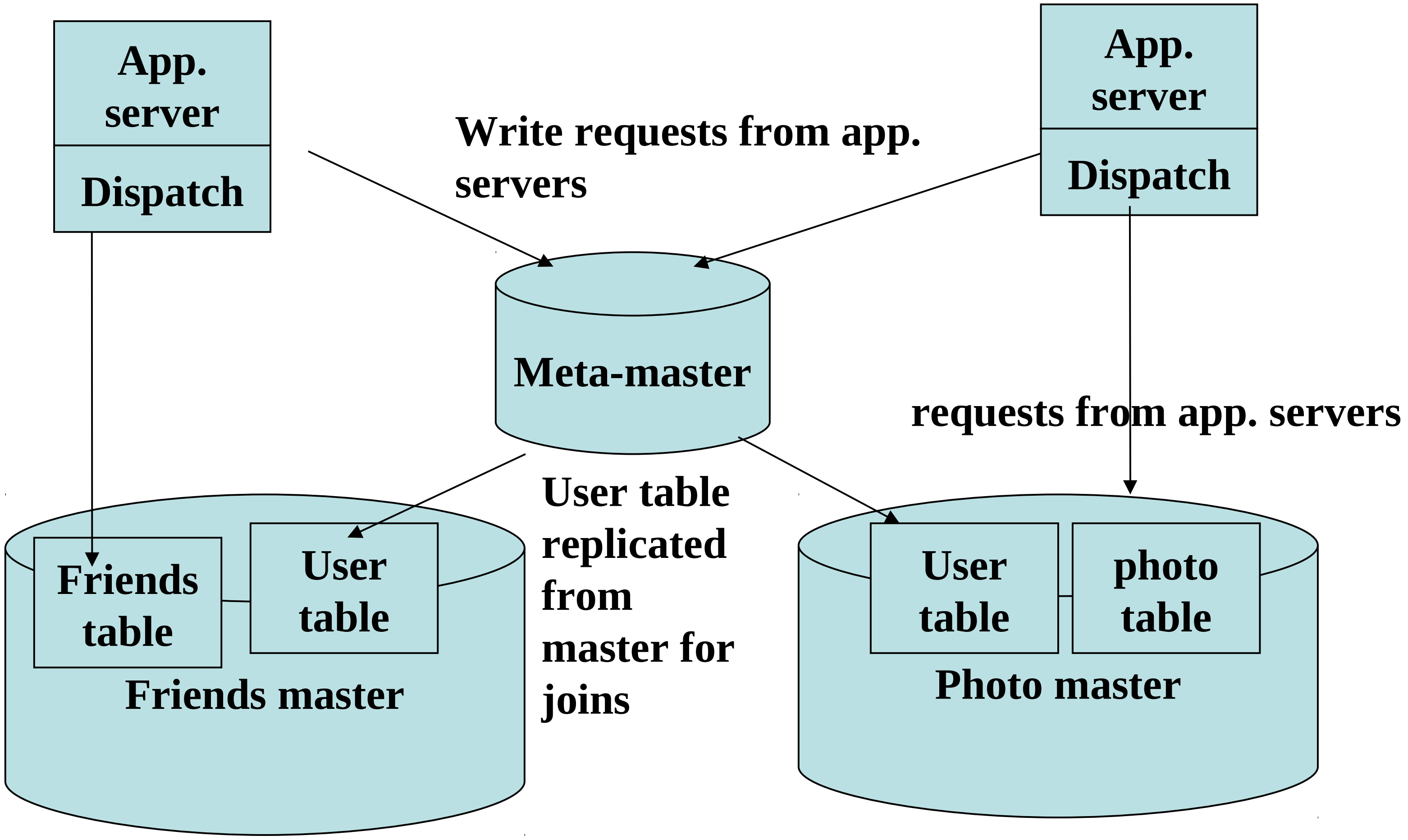
Consequences of HS and VS

- no more DB-Joins with SQL. Lots of copied data!
- a lot more partial requests needed for data aggregation
- expensive distributed transactions for consistency (if needed)
- vertical sharding distributes related data types from one user
- horizontal sharding distributes related users from each other (bad for social graph processing)
- SQL further limited because of mostly key/value queries and problems with automatic DB-Sequences

And: every change needs application changes as well!!!!

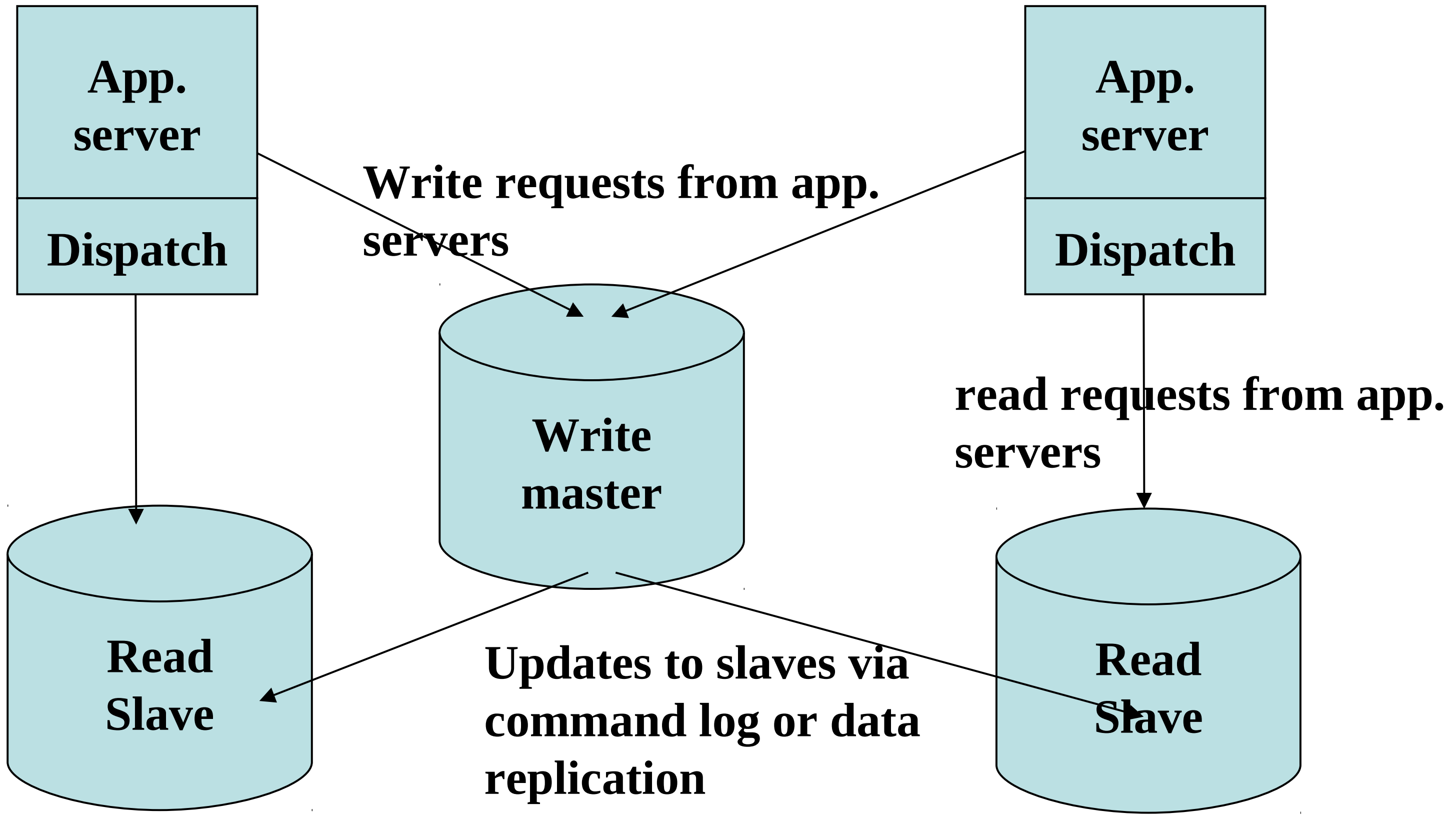
It is questionable, whether splitting RDBMs into several shards is really beneficial. As most of the aggregation is done in the cache and application server levels anyway, you might think about using a NoSQL. Consistency will be an issue then, but most sharded systems do not run dist. Transactions anyway. USE AN INDIRECTION LAYER IN APPLICATIONS TO HIDE STORAGE CHANGES!

Vertical Split with De-Normalization



Not shown: read slaves per master

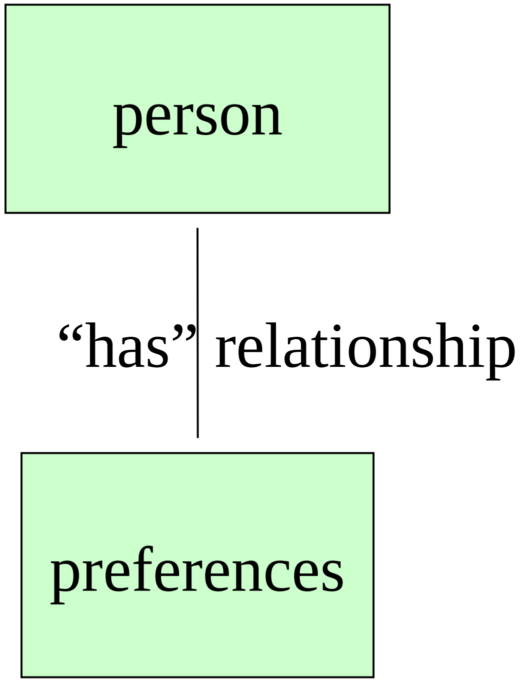
Partitioning over Functions: Read-Slaves


App. server
Dispatch
App. server
Dispatch
Write requests from app. servers
Write master
read requests from app. servers
Read Slave
Read Slave
Updates to slaves via command log or data replication

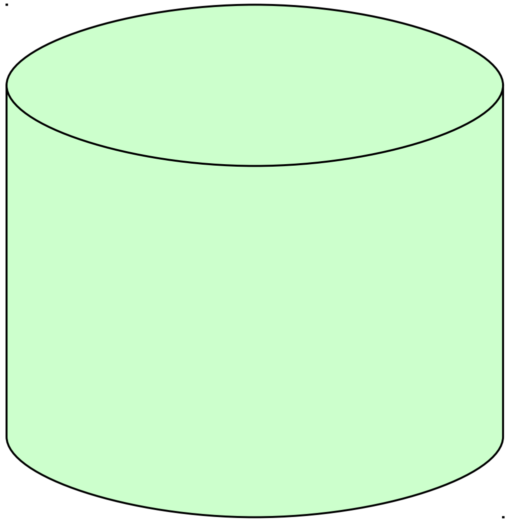
1. Make sure, that those reads result from compulsory cache misses
2. This won't help your writes at all, but suffers from lots of them.
3. How consistent is this?

Relationships

Relationships



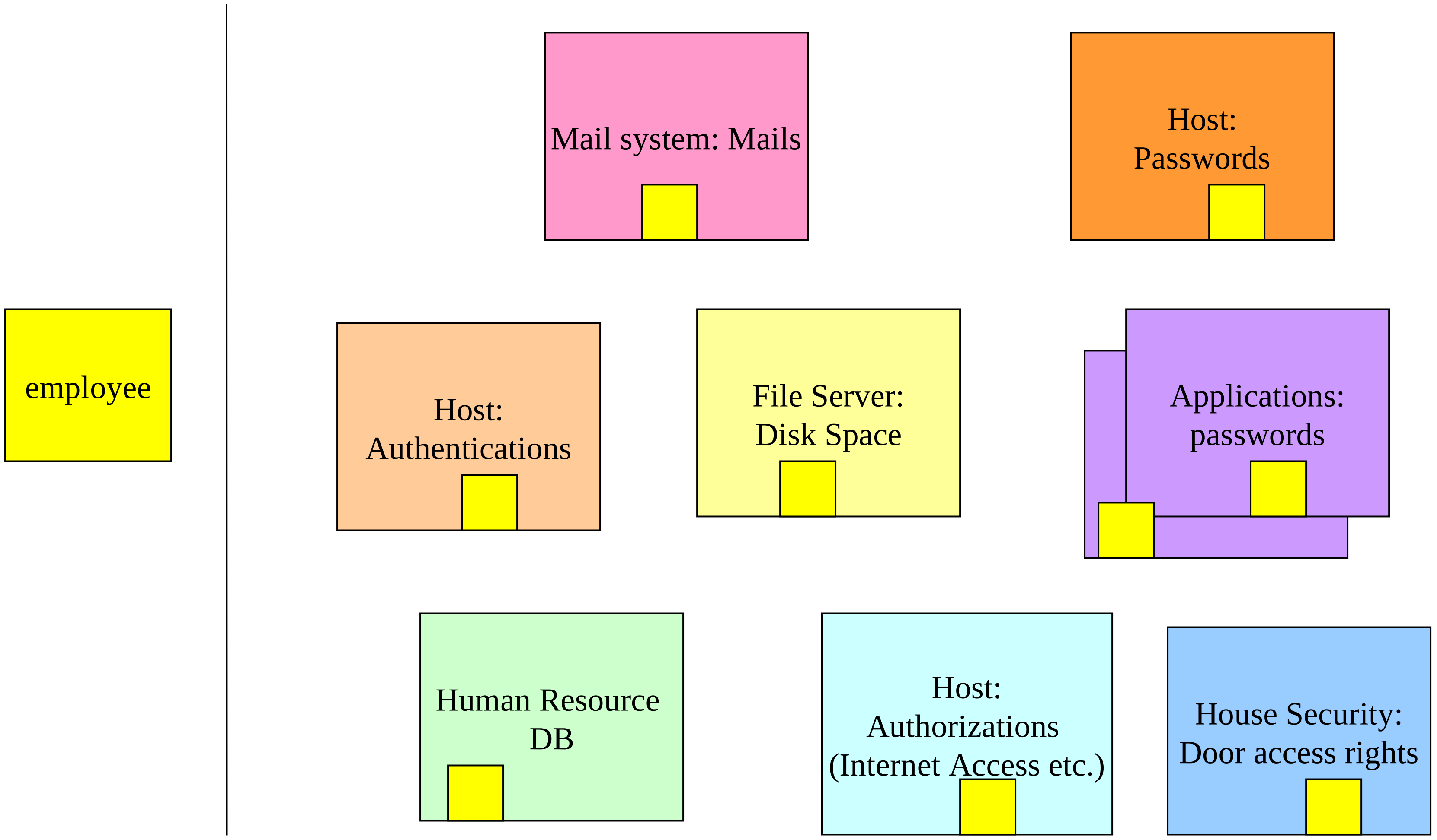
Referential integrity rules



"Person" and "Preferences". If person is deleted, the preferences should go too.

Within the database referential integrity rules protect e.g. containment relationships. We've got nothing like this in object space. And distributed?

Relationships: employee



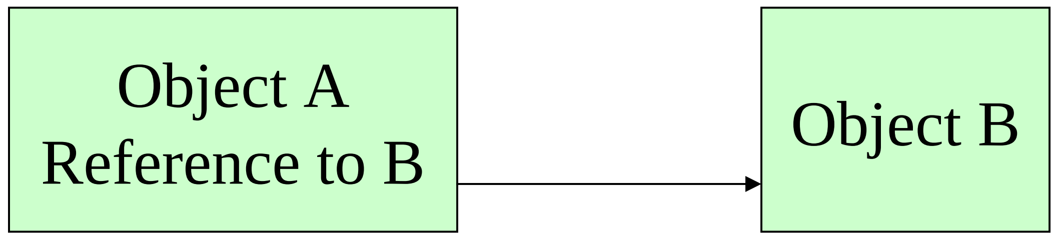
Can you make sure that once an employee leaves, ALL her rights are cancelled, her disc-space archived and erased, the databases for authentication etc. are updated, application specific DBs as well? And last but not least that the badge does no longer work? That all equipment has been returned? This are RELATIONS between an employee and resources which need to be expressed in a machine readable form.

Functional Requirements for a Relationship Service

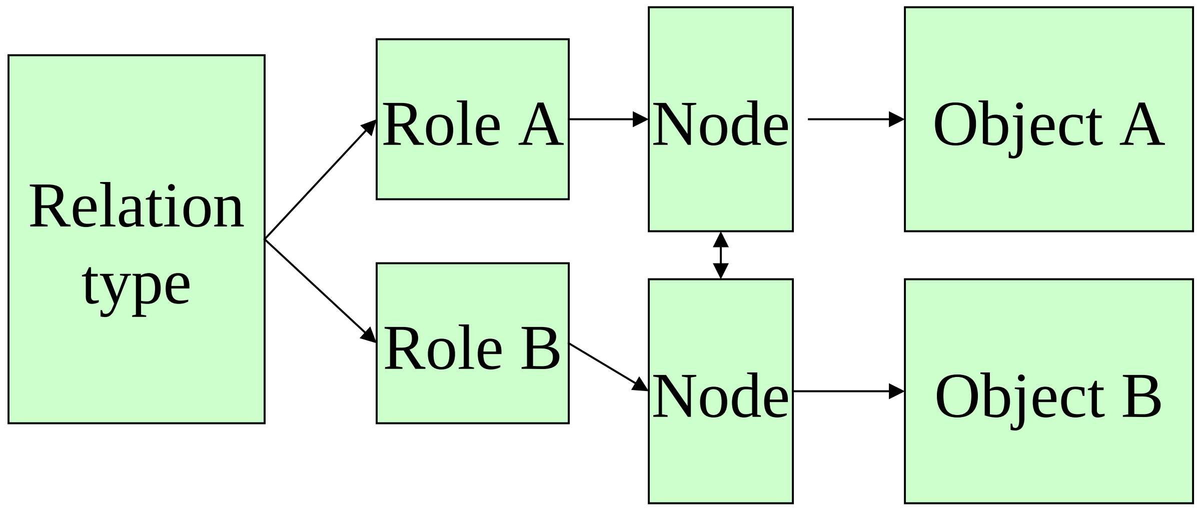
- Allow definition of relations between objects without modifying those objects
- Allow different types of relations
- Allow graphs of relations
- Allow the traversal of relationship graphs
- Support reference and containment relations

More on relationships: W.Emmerich, Engineering Distributed Objects

Relationship Modeling



Programmed relationship



Dynamic relationship

The objects A and B are not aware of any relations defined on them.

Relationship Service: A Failure

The good:

- Powerful modeling tool for meta-information
- Helps with creation, migration, copy and deletion of composite objects and maintains referential integrity

The bad:

- Tends to create many and small server objects
- Performance killer (many CORBA vendors did not implement this service for a long time). EJB: supported with local objects only (in same container)

Today: Graph Processing

- Modern graph-processing databases (e.g. Neo4j)
- Distributed graph-processing algorithms (Google's Pregel)
- Loosely-consistent replicated stores without TAs

The problems with a distributed relationship service were a sign of things to come: Processing friend-relations across 1.5 billion people....

Next Sessions:

When the truth is too expensive: Dealing with Uncertainty in Replicated Systems!

Literature:

- Gray/Reuter, Transaction Processing (Chapter on Transaction Models)
- Peter Bailis, Dissertation on Coordination-free Consistency
- Distributed Systems for fun and profit (Chapter on CRDTs)

Resources

- Understanding LDAP, www.redbooks.ibm.com
- www.io.de distributed web search paper
- Van Steen/Tanenbaum, Chapter on Naming
- Van Steen/Tanenbaum, Chapter on Security (homework for next session)
- Martin Fowler et al., UML Distilled (small and nice)

Werner Vogels, Amazon Architecure, http://queue.acm.org/detail.cfm?id=1142065

Alex Iskold, SOAWorld on Amazon – the real Web Services Comp. http://soa.sys-con.com/node/262024

Resources

D. Bailey, E. Frank-Schultz, P. Lindeque, and J. L. Temple III, Three reliability engineering techniques and their application to evaluating the availability of IT Systems: An Introduction. IBM Systems Journal Vol. 47, Nr. 4, 2008

R. R. Scadden, R. J. Bogdany, J. W. Clifford, H. D. Pearthree, and R. A. Lock, Resilient hosting in a continuously available virtualized environment, IBM Systems Journal Vol 47, Nr. 4

Jeff Dean, Handling Large Datasets at Google, Current Systems and Future Directions, http://prof.ict.ac.cn/DComputing/uploads/2013/DC_2_2_Google.pdf

Distributed Lookup Services - Distributed Hash Tables, Paul Krzyzanowski, December 5, 2012, https://www.cs.rutgers.edu/~pxk/417/notes/23-lookup.html

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels, Dynamo: Amazon’s Highly Available Key-value Store, SOSP’07, October 14–17, 2007, Stevenson, Washington, USA. Copyright 2007 ACM.

Benjamin Manes, Design of a Modern Cache (Caffeine), http://highscalability.com/blog/2016/1/25/design-of-a-modern-cache.html

Resources

• Broker vs. Brokerless

http://zeromq.org/whitepapers:brokerless

• A Look at Nanomsg and Scalability Protocols

http://www.bravenewgeek.com/a-look-at-nanomsg-and-scalability-protocols/

• Message Broker

http://en.wikipedia.org/wiki/Message_broker

• The Log: What every software engineer should know about real-time data's unifying abstraction

http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineershould-know-about-real-time-datas-unifying

Aeron

• GitHub (Efficient reliable unicast and multicast transport protocol)

https://github.com/real-logic/Aeron/

Strangeloop

https://thestrangeloop.com/sessions/aeron-open-source-high-performance-messaging

Resources

Edge Routing, Internet-Scale load-balancing:

Laura Nolan, Murali Suriar, Directing traffic: Demystifying Internet-scale load balancing,

https://opensource.com/article/18/10/internet-scale-load-balancing

Oleg Guba and Alexey Ivanov, Dropbox traffic infrastructure, https://blogs.dropbox.com/tech/2018/10/dropbox-traffic-infrastructure-edge-network/

Mike Smith, Rethinking Netflix’s Edge Load Balancing, https://medium.com/netflix-techblog/netflix-edge-load-balancing-695308b5548c