Scaling Twitter with Open Source

Chris Aniszczyk (**@cra**)

Head of Open Source, Twitter

http://aniszczyk.org

#eumjapan

Reminder: Please Tweet!

@cra

#eumjapan

Agenda

Twitter Scale

Evolution of the Twitter Stack

Twitter Stack Sampling

Concluding Thoughts

What is Twitter?

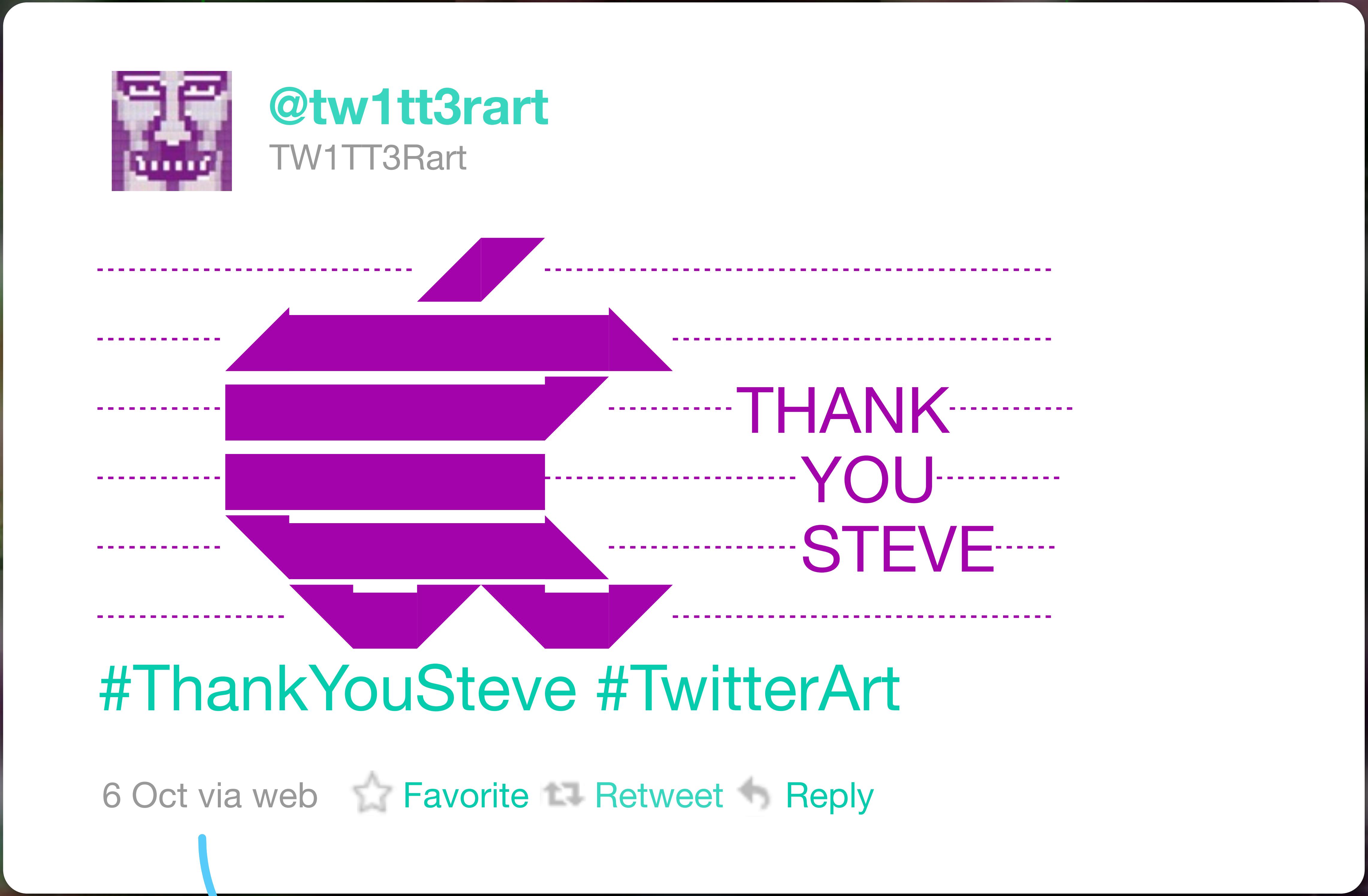
Twitter is a public real-time information network that connects you to what you find interesting

The heart of Twitter: ***tweets***

`sizeof`(1 tweet) = 140 characters

≈ 200 bytes

doesn’t sound like much?



@tw1tt3rart
TW1TT3Rart

THANK
YOU
STEVE

#ThankYouSteve #TwitterArt

6 Oct via web

"Creativity comes from constraint"
"Brevity is the soul of the wit"

What is the scale of Twitter?

500,000,000+ Tweets / Day

3,500,000,000+ Tweets / Week

3.5B Tweets / Week

≈

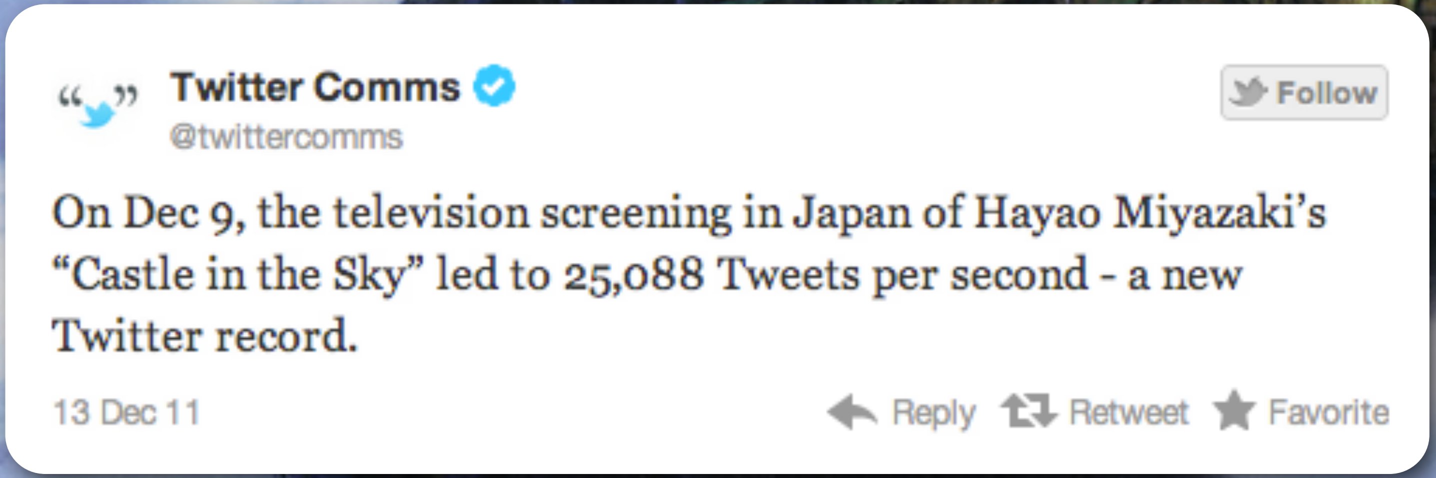
6000+ Tweets / Second

(steady state)

However, there are **peaks**!

Miyazaki 2011

25,088 TPS (NYE 2013: 33,338 TPS)



Twitter Comms
@twittercomms

On Dec 9, the television screening in Japan of Hayao Miyazaki's "Castle in the Sky" led to 25,088 Tweets per second - a new Twitter record.

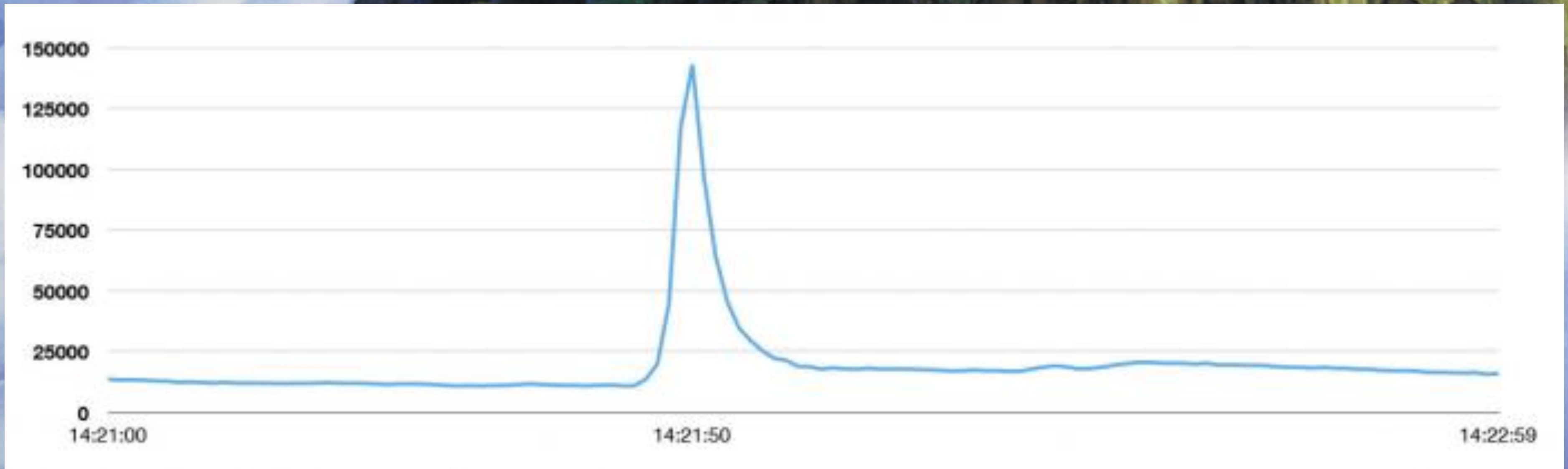
13 Dec 11

バルス! ("Death to Twitter")

Miyazaki 2013

~~25,088 TPS~~ **143,199 TPS**

https://blog.twitter.com/2013/new-tweets-per-second-record-and-how

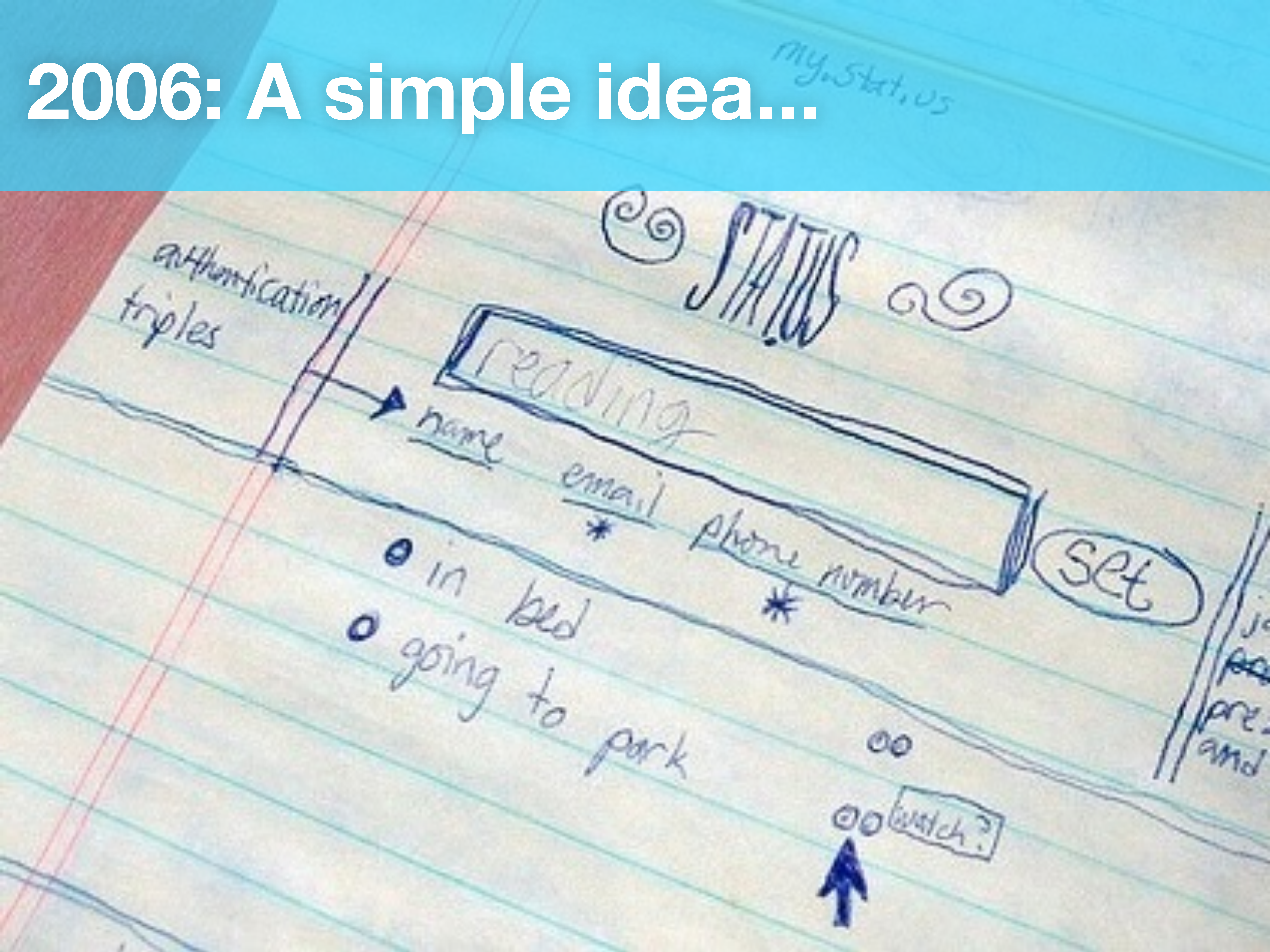


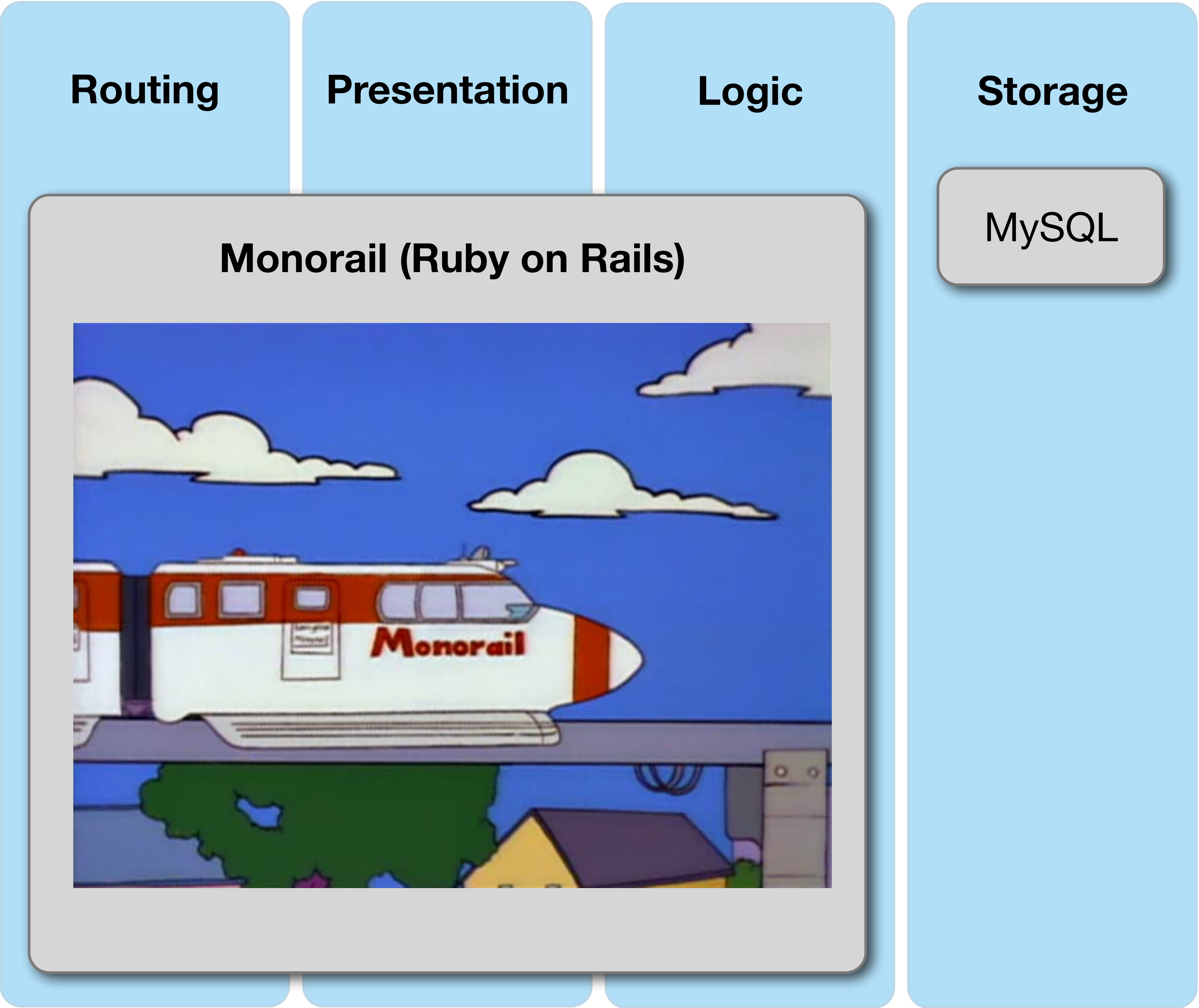
バルス！(“Death to Twitter”)

Twistory

Scaling the Twitter Stack

2006: A simple idea...





Routing

Presentation

Logic

Storage

Monorail (Ruby on Rails)

MySQL

2008: Growing Pains

FAIL WHALE
Twitter: Failure is an option. At least once a day, or whenever you need it.

Routing

Presentation

Logic

Storage

MySQL

Tweet Store

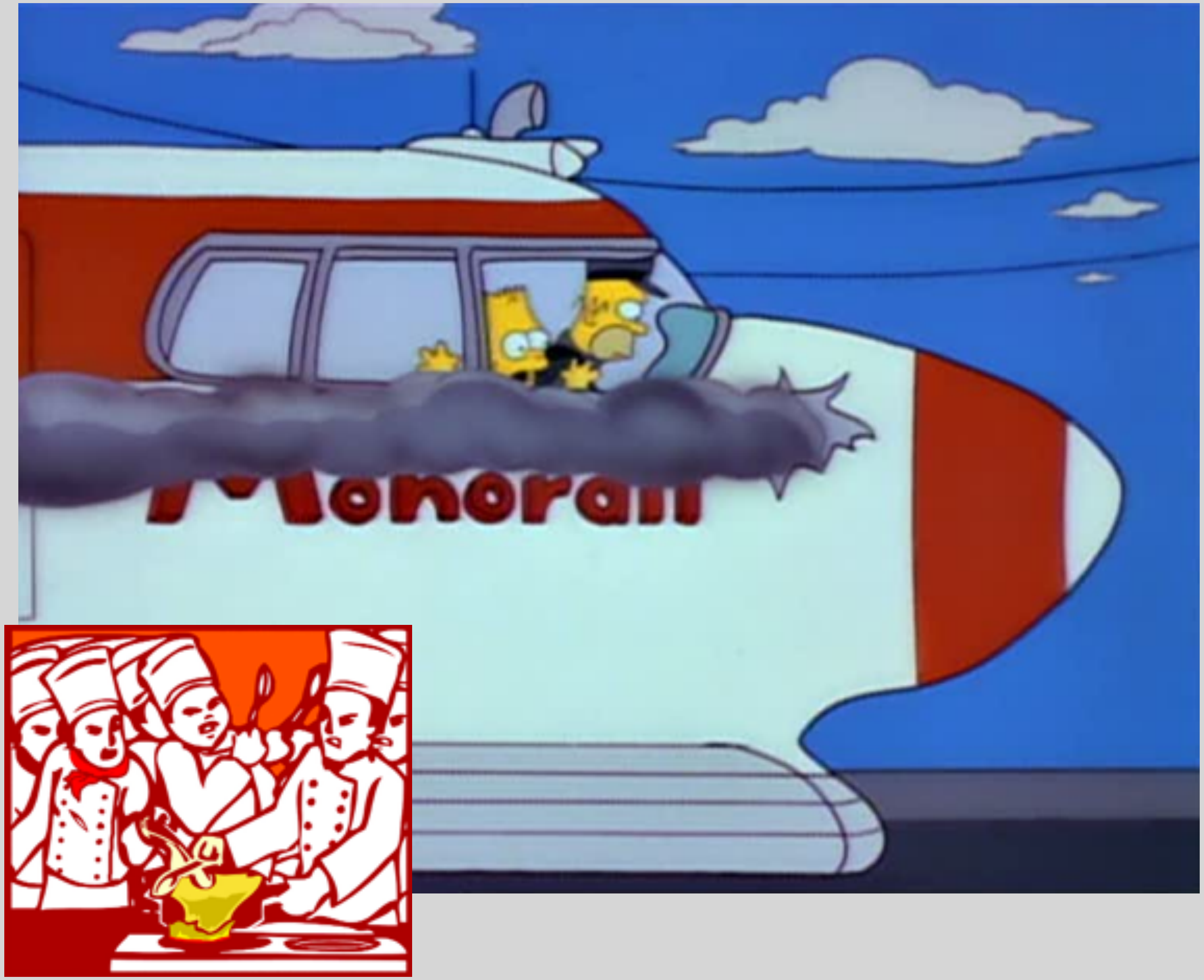
Flock

Cache

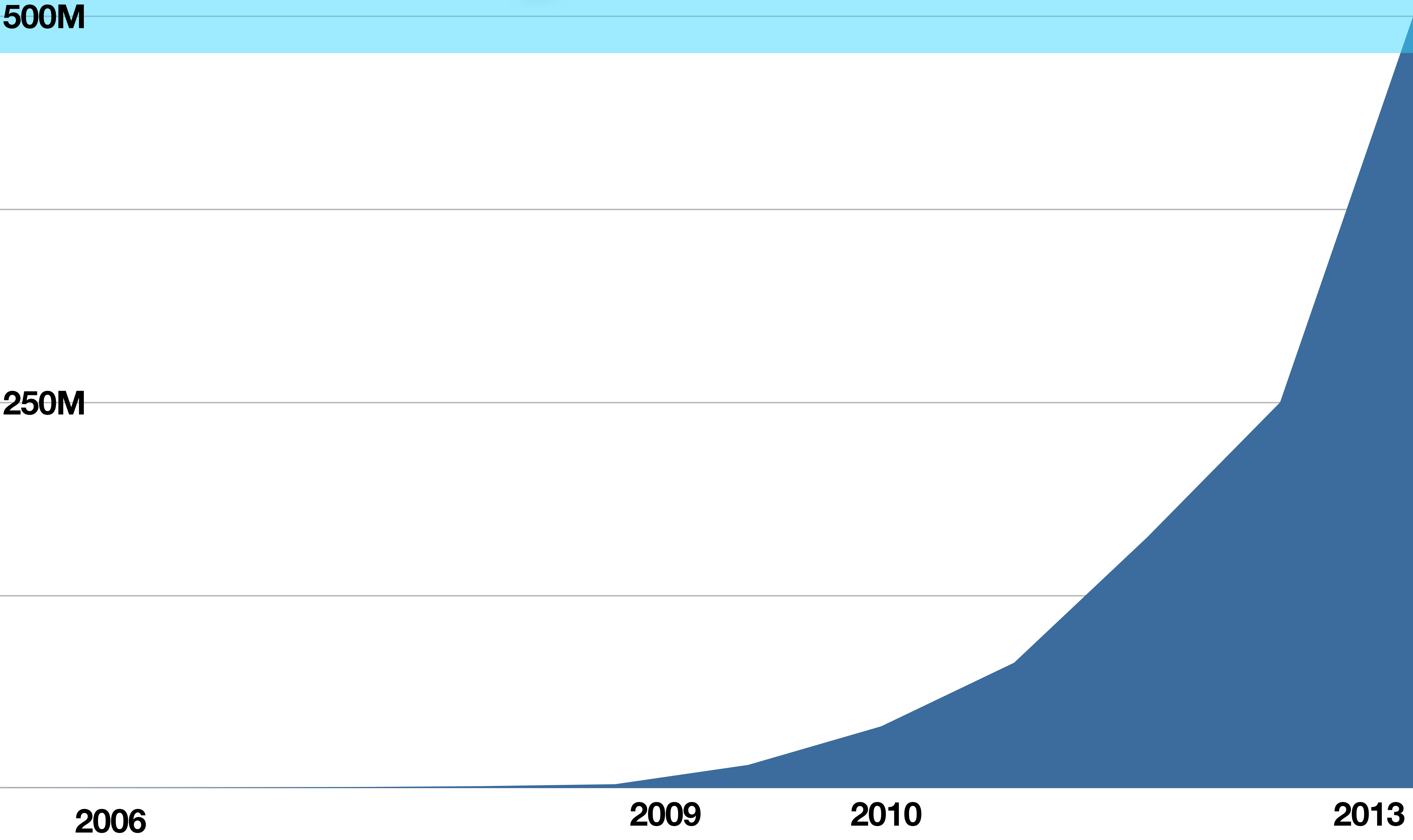
Memcache

Redis

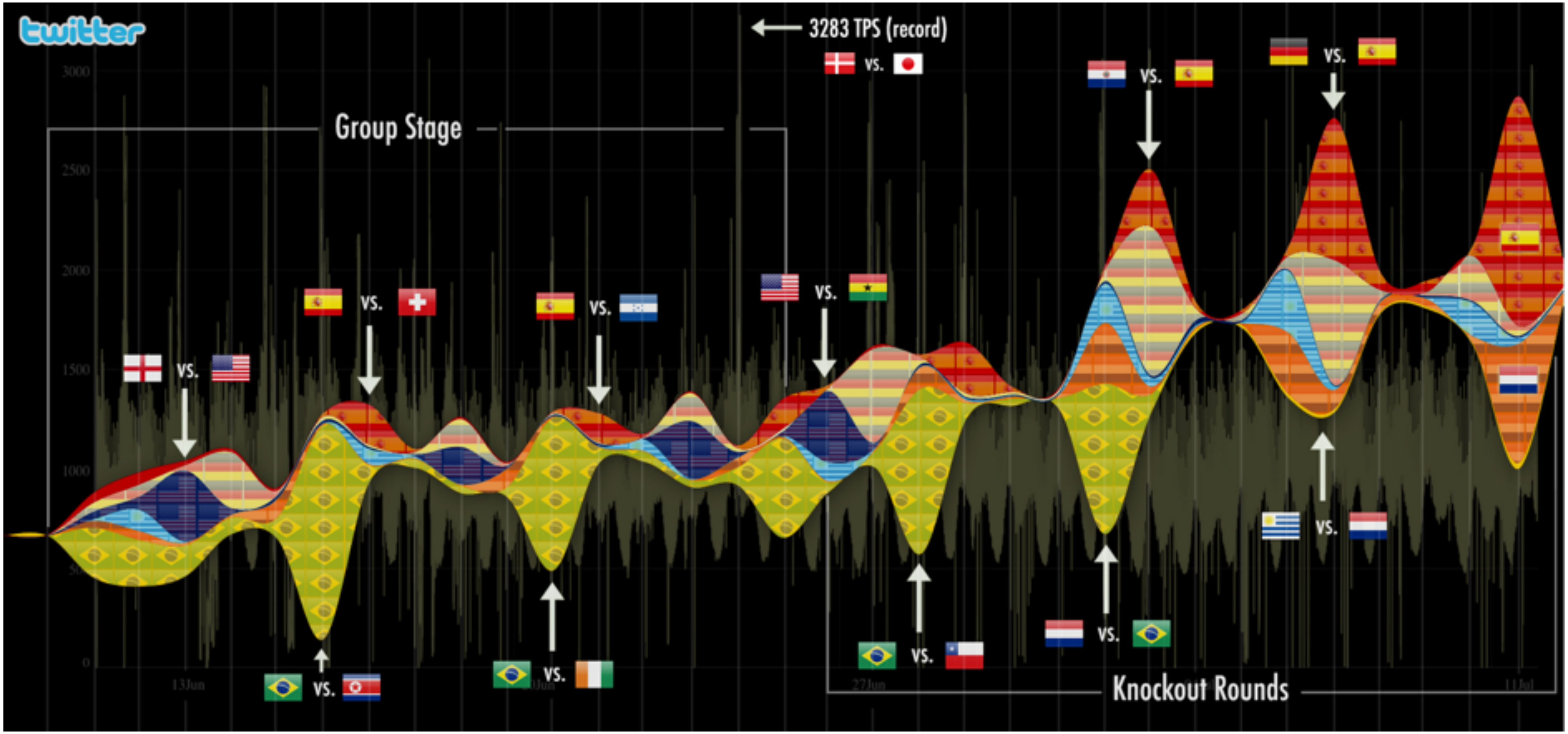
Monorail (Ruby on Rails)



2009+: Crazy Growth



2010: World Cup Woes



https://blog.twitter.com/2010/2010-world-cup-global-conversation
http://bits.blogs.nytimes.com/2010/06/15/twitter-suffers-from-a-number-of-technical-glitches

What was wrong?

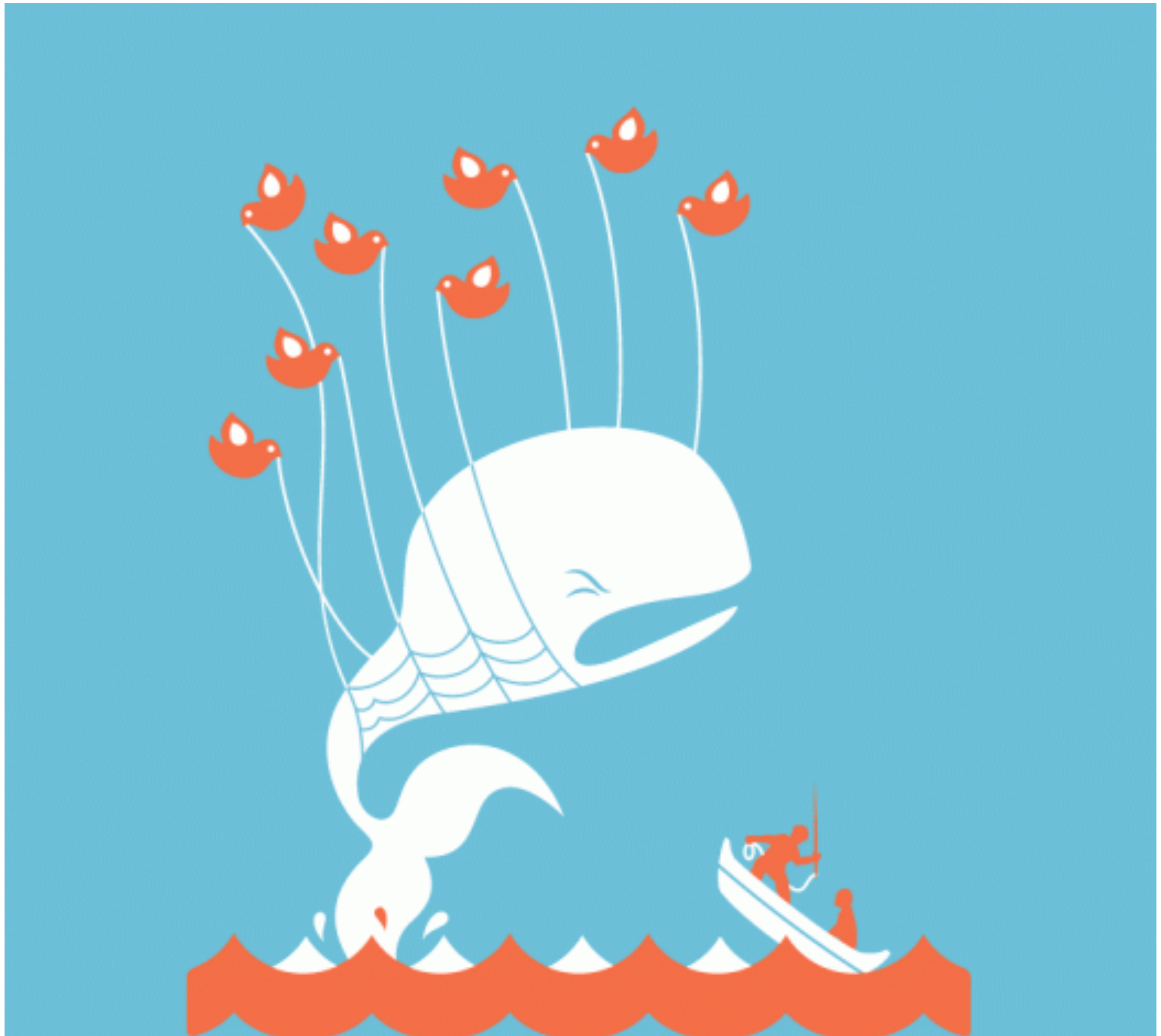
Fragile monolithic Rails code base: managing raw database and memcache connections to rendering the site and presenting the public APIs

Throwing machines at the problem: instead of engineering solutions

Trapped in an optimization corner: trade off readability and flexibility for performance

Whale Hunting Expeditions

We organized archeology digs and whale hunting expeditions to understand large scale failures



Re-envision the system?

We wanted big infra wins: in performance, reliability and efficiency (reduce machines to run Twitter by 10x)

Failure is inevitable in distributed systems: we wanted to isolate failures across our infrastructure

Cleaner boundaries with related logic in one place: desire for a loosely coupled services oriented model at the systems level

Ruby VM Reflection

Started to evaluate our front end server tier: CPU, RAM and network

Rails machines were being pushed to the limit: CPU and RAM maxed but not network (200-300 requests/host)

Twitter's usage was growing: it was going to take a lot of machines to keep up with the growth curve

JVM (Java) Experimentation

We started to experiment with the JVM…

Search (Java via Lucene)

http://engineering.twitter.com/2010/10/twitters-new-search-architecture.html

FlockDB: Social Graph (Scala)

https://blog.twitter.com/2010/introducing-flockdb

https://github.com/twitter/flockdb

…and we liked it, enamored by JVM performance!

We weren’t the only ones either: http://www.slideshare.net/pcalcado/from-a-monolithic-ruby-on-rails-app-to-the-jvm

The JVM Solution

Level of trust with the JVM with previous experience

JVM is a mature and world class platform

Huge mature ecosystem of libraries

Polyglot possibilities (Java, Scala, Clojure, etc)

Decomposing the Monolith

Created services based on our core nouns:

Tweet service

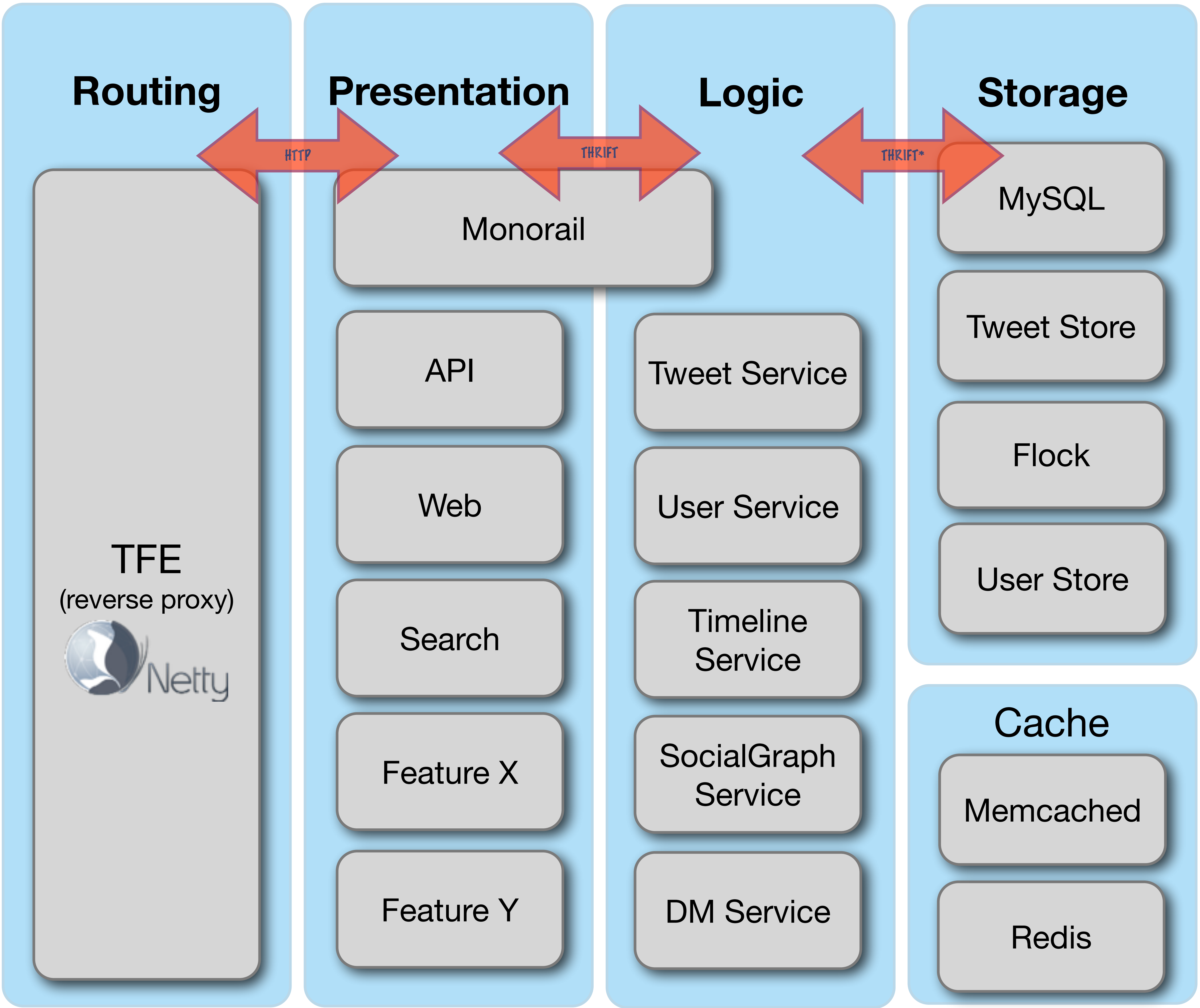
User service

Timeline service

DM service

Social Graph service

….



Routing

Presentation

Logic

Storage

HTTP

THRIFT

THRIFT*

TFE
(reverse proxy)

Netty

Monorail

API

Web

Search

Feature X

Feature Y

Tweet Service

User Service

Timeline Service

SocialGraph Service

DM Service

MySQL

Tweet Store

Flock

User Store

Cache

Memcached

Redis

Twitter Stack

A peak at some of our technology

Finagle, Scalding and Mesos

Services: Concurrency is Hard

Decomposing the monolith: each team took slightly different approaches to concurrency

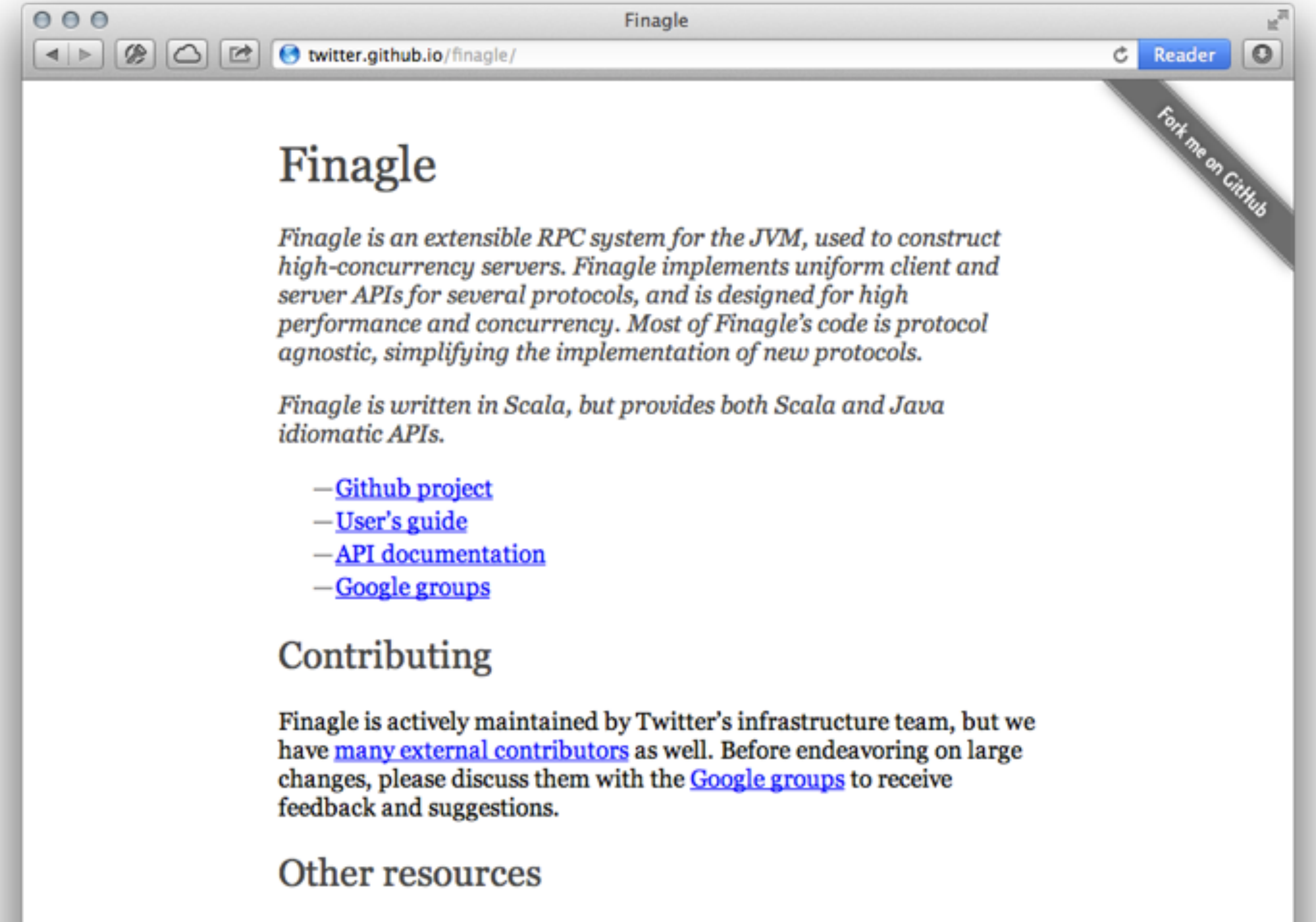
Different failure semantics across teams: no consistent back pressure mechanism

Failure domains informed us of the importance of having a unified client/server library: deal with failure strategies and load balancing

Hello Finagle! (Scala-based)

http://twitter.github.io/finagle

Used by Twitter, Nest, Soundcloud, Foursquare and more!



Finagle

Finagle is an extensible RPC system for the JVM, used to construct high-concurrency servers. Finagle implements uniform client and server APIs for several protocols, and is designed for high performance and concurrency. Most of Finagle's code is protocol agnostic, simplifying the implementation of new protocols.

Finagle is written in Scala, but provides both Scala and Java idiomatic APIs.

—Github project
—User's guide
—API documentation
—Google groups

Contributing

Finagle is actively maintained by Twitter's infrastructure team, but we have many external contributors as well. Before endeavoring on large changes, please discuss them with the Google groups to receive feedback and suggestions.

Other resources

Finagle Programming Model

Takes care of: service discovery, load balancing, retrying, connection pooling, stats collection, distributed tracing

`Future[T]`: modular, composable, async, non-blocking I/O

http://twitter.github.io/effectivescala/#Concurrency

Hadoop with Scalding

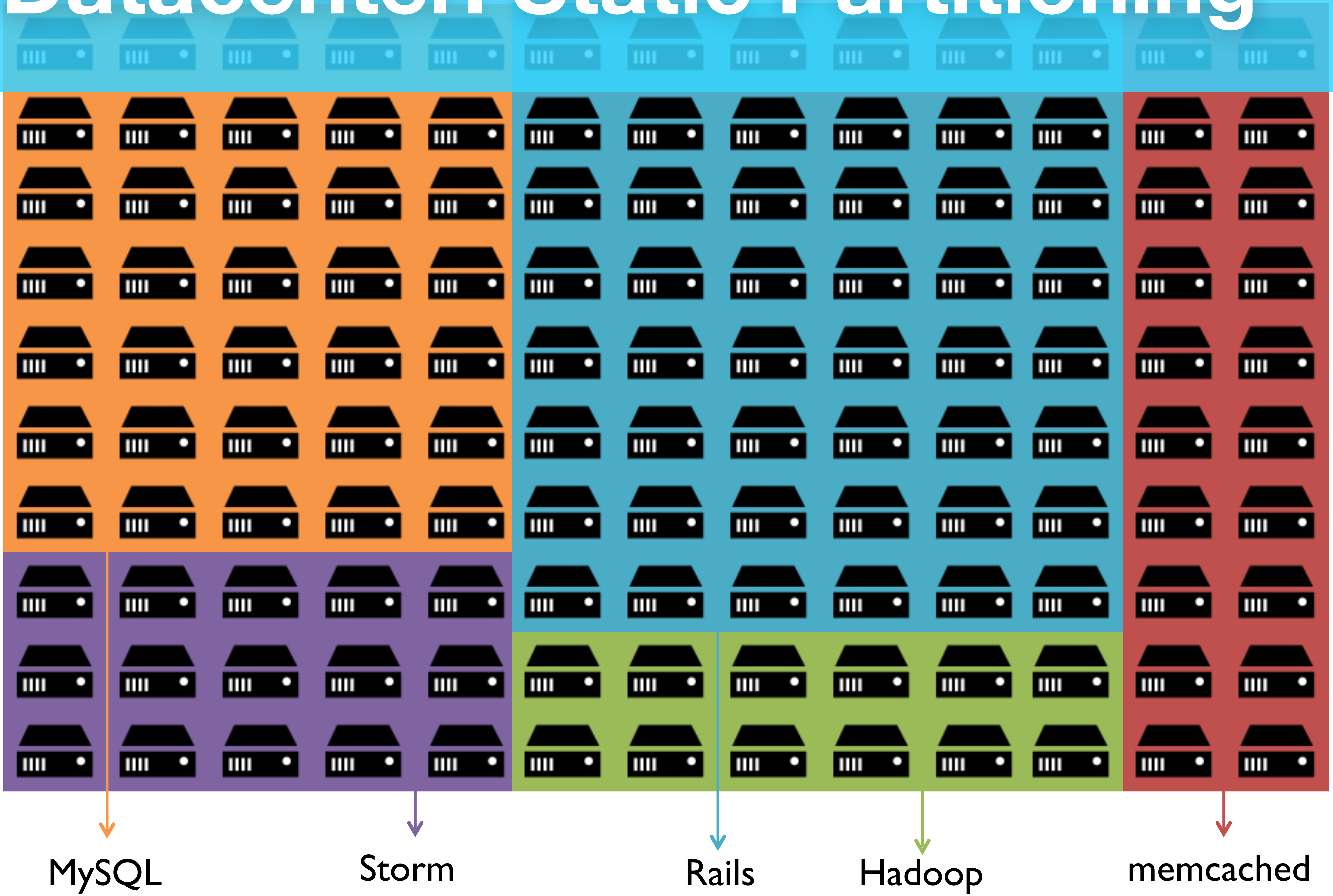
Services receive a ton of traffic and generate a ton of use log and debugging entries.

@Scalding is a open source Scala library that makes it easy to specify MapReduce jobs with the benefits of functional programming!

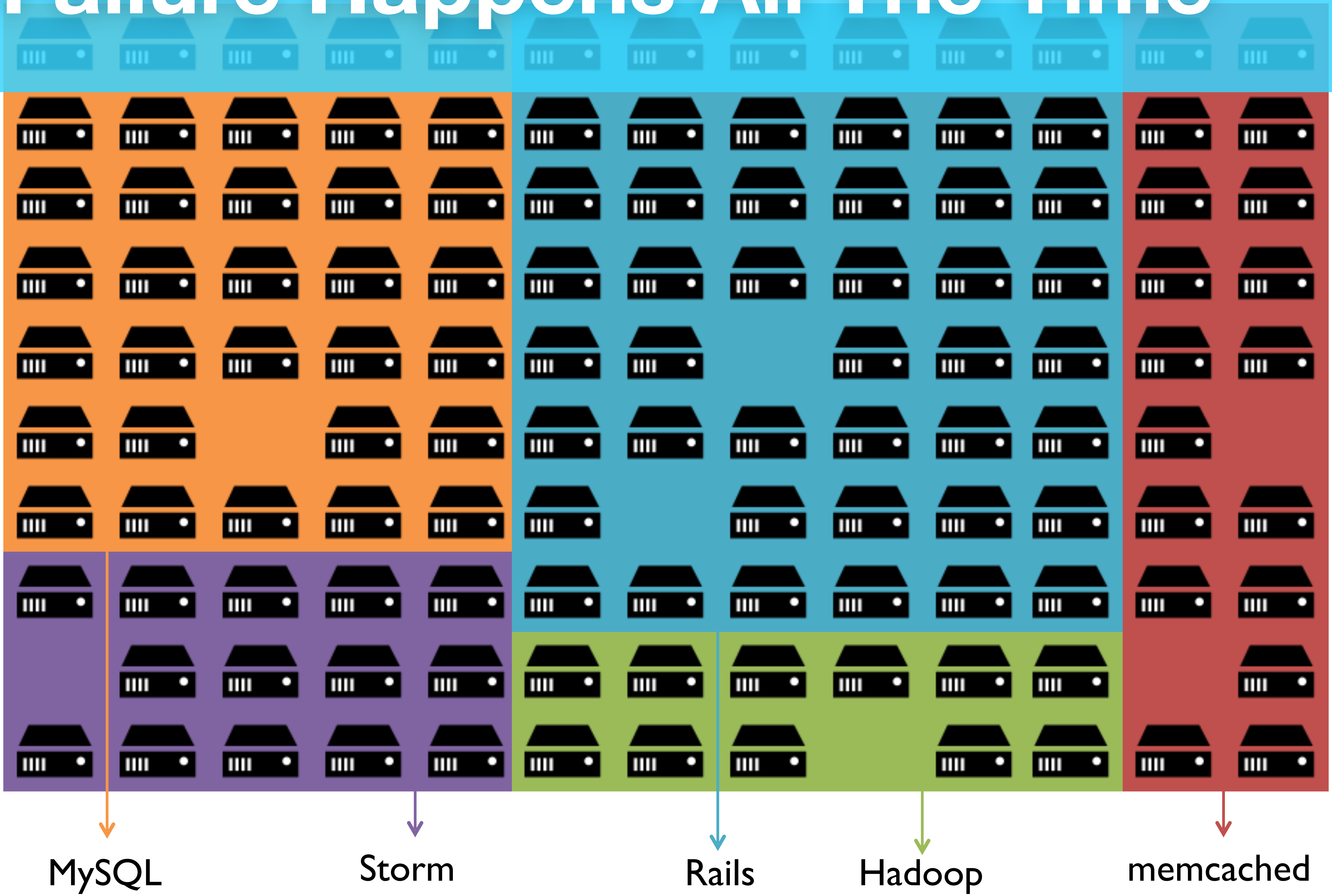
https://github.com/twitter/scalding

https://github.com/twitter/scalding/wiki/Rosetta-Code

Datacenter: Static Partitioning



Failure Happens All The Time

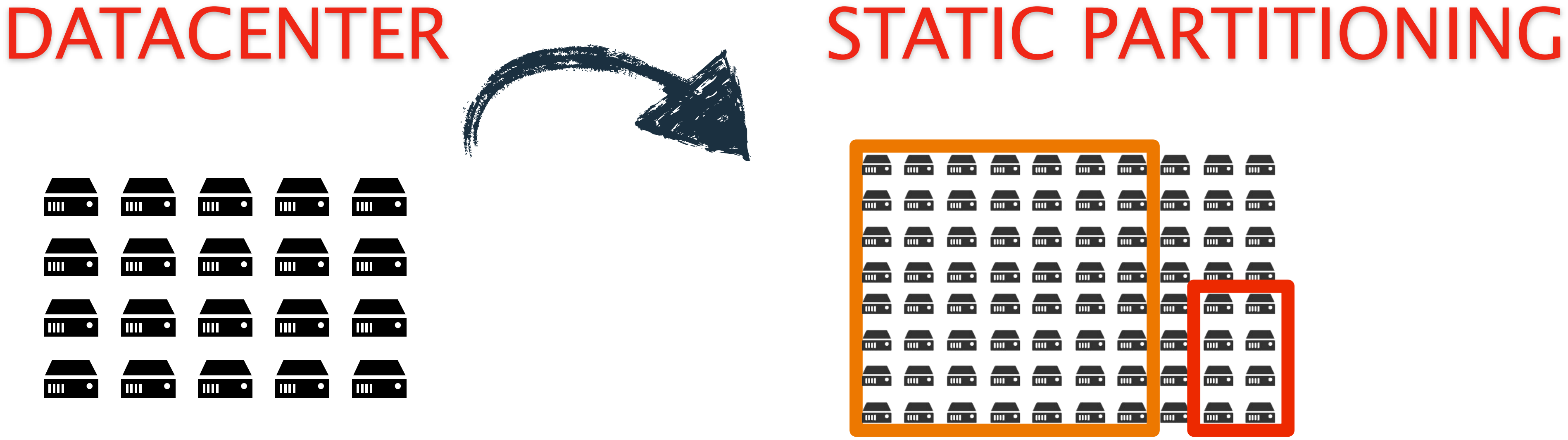


Data Center Evils

The evils of single tenancy and static partitioning

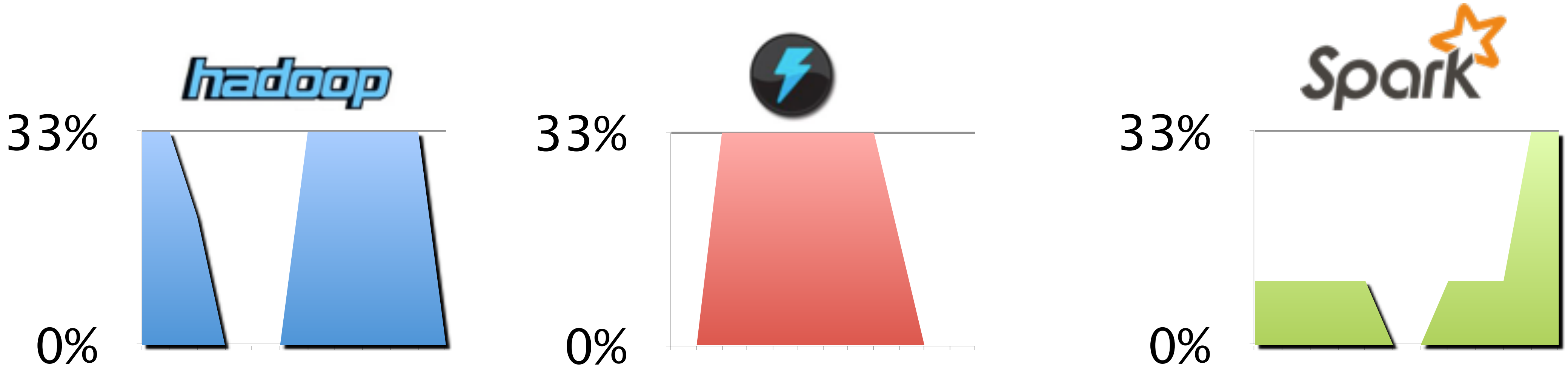
Different jobs… different utilization profiles…

Can we do better?



DATACENTER

STATIC PARTITIONING



Borg and The Birth of Mesos

Google was generations ahead with Borg/Omega

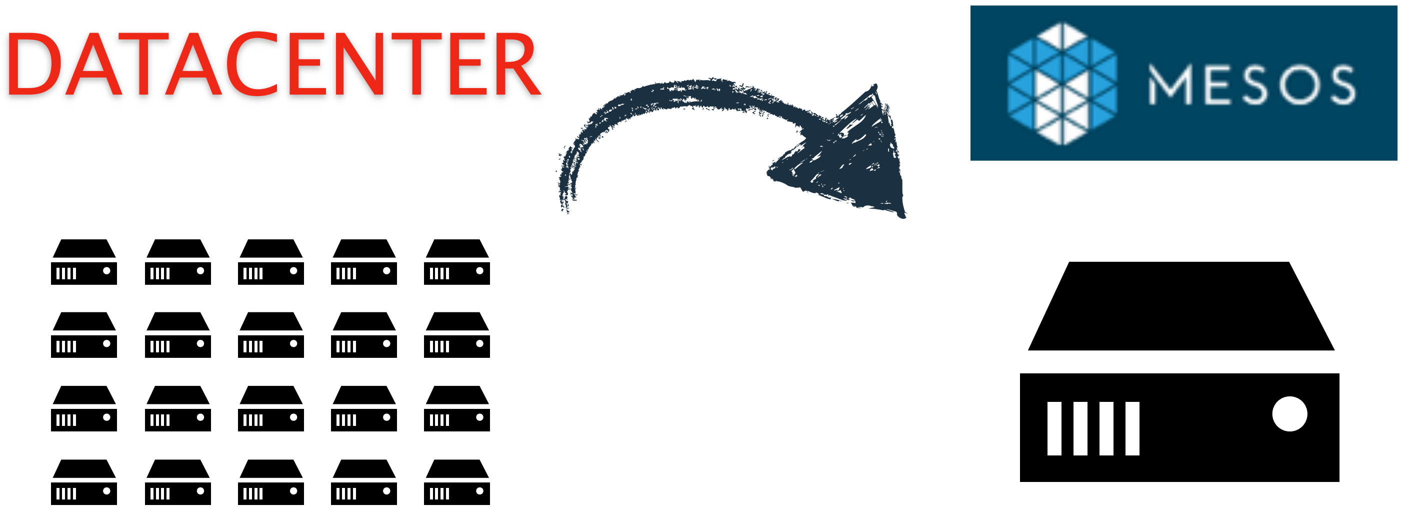
"The Datacenter as a Computer"

http://research.google.com/pubs/pub35290.html (2009)

engineers focus on resources needed; mixed workloads possible

Learn from Google and work w/ university research!

http://wired.com/wiredenterprise/2013/03/google-borg-twitter-mesos



Mesos, Linux and cgroups

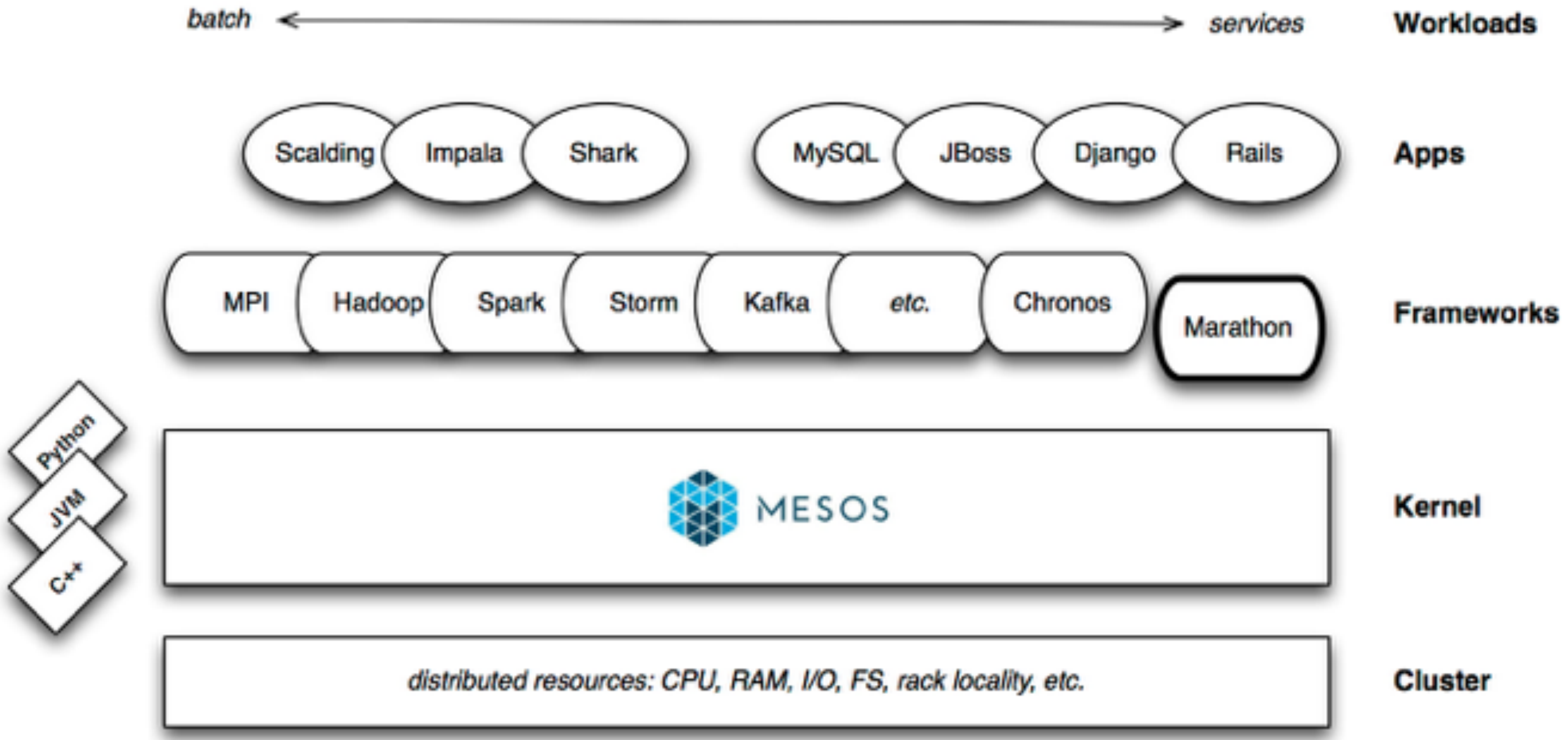
Apache Mesos: kernel of the data center

obviates the need for VMs* (aggregation; not virtualization)

isolation via Linux cgroups (CPU, RAM, network, FS)

reshape clusters dynamically based on resources

multiple frameworks; scalability to 10,000s of nodes



batch ⟷ *services* **Workloads**

Scalding Impala Shark MySQL JBoss Django Rails **Apps**

MPI Hadoop Spark Storm Kafka *etc.* Chronos Marathon **Frameworks**

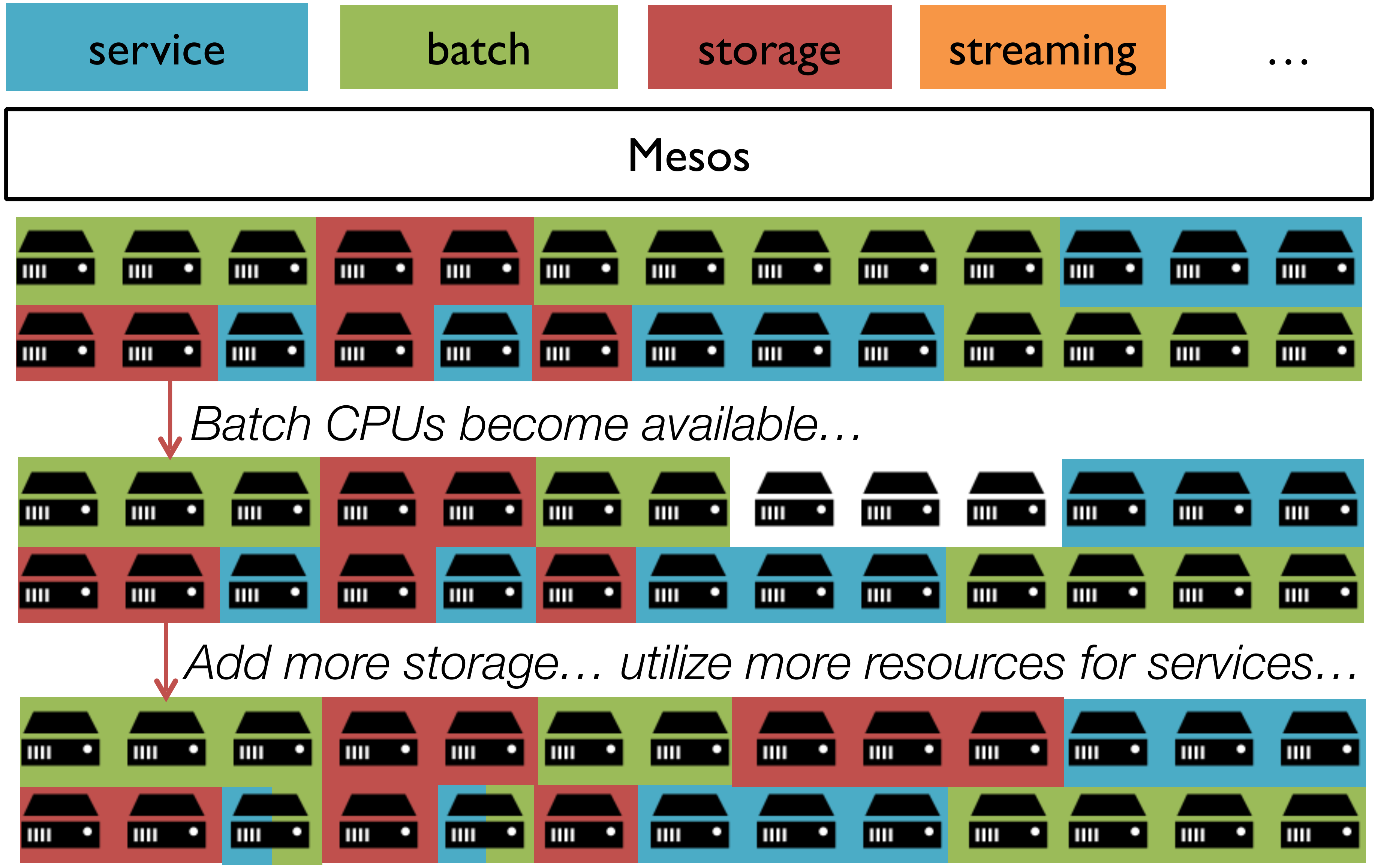
Python JVM C++

MESOS **Kernel**

distributed resources: CPU, RAM, I/O, FS, rack locality, etc. **Cluster**

Datacenter Operating System

Apache Mesos: kernel of the data center



service

batch

storage

streaming

…

Mesos

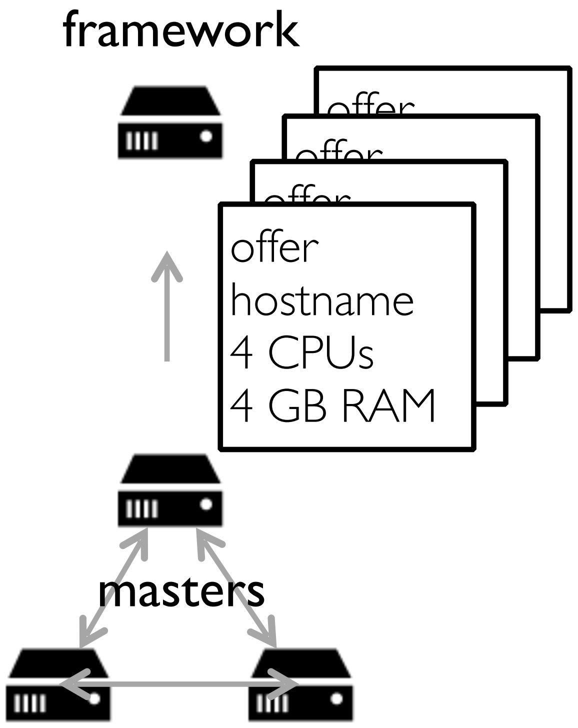
Batch CPUs become available…

Add more storage… utilize more resources for services…

For more details, watch: https://www.youtube.com/watch?v=r7qN8QwGv2w

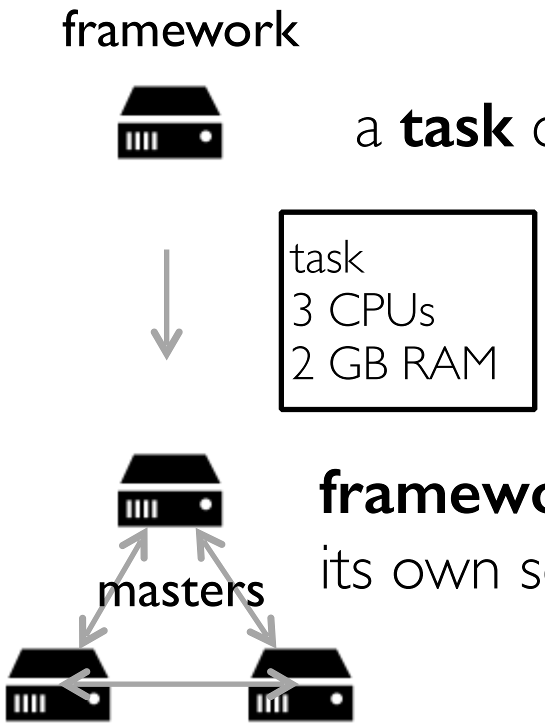
Two Level Scheduling

Think non-blocking sockets in the kernel!



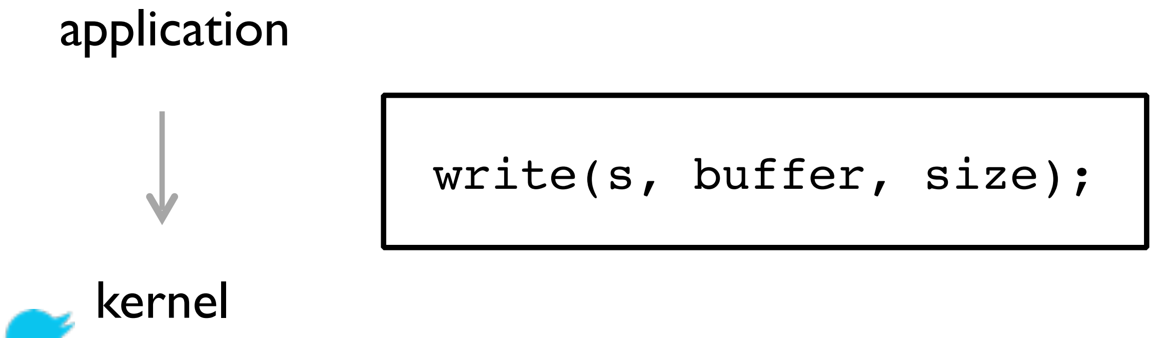
resources are allocated via ***resource offers***

a resource offer represents a snapshot of available resources (one offer per host) that a **framework** can use to run tasks

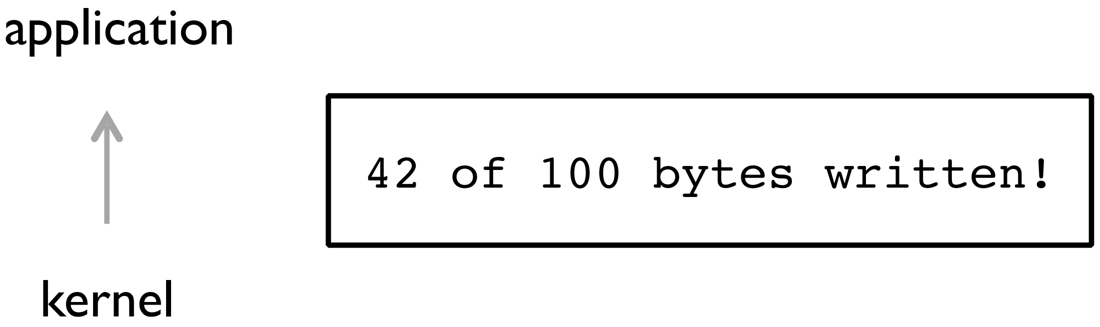


a **task** can use a subset of an offer;

frameworks use offers to perform its own scheduling



```
write(s, buffer, size);
```

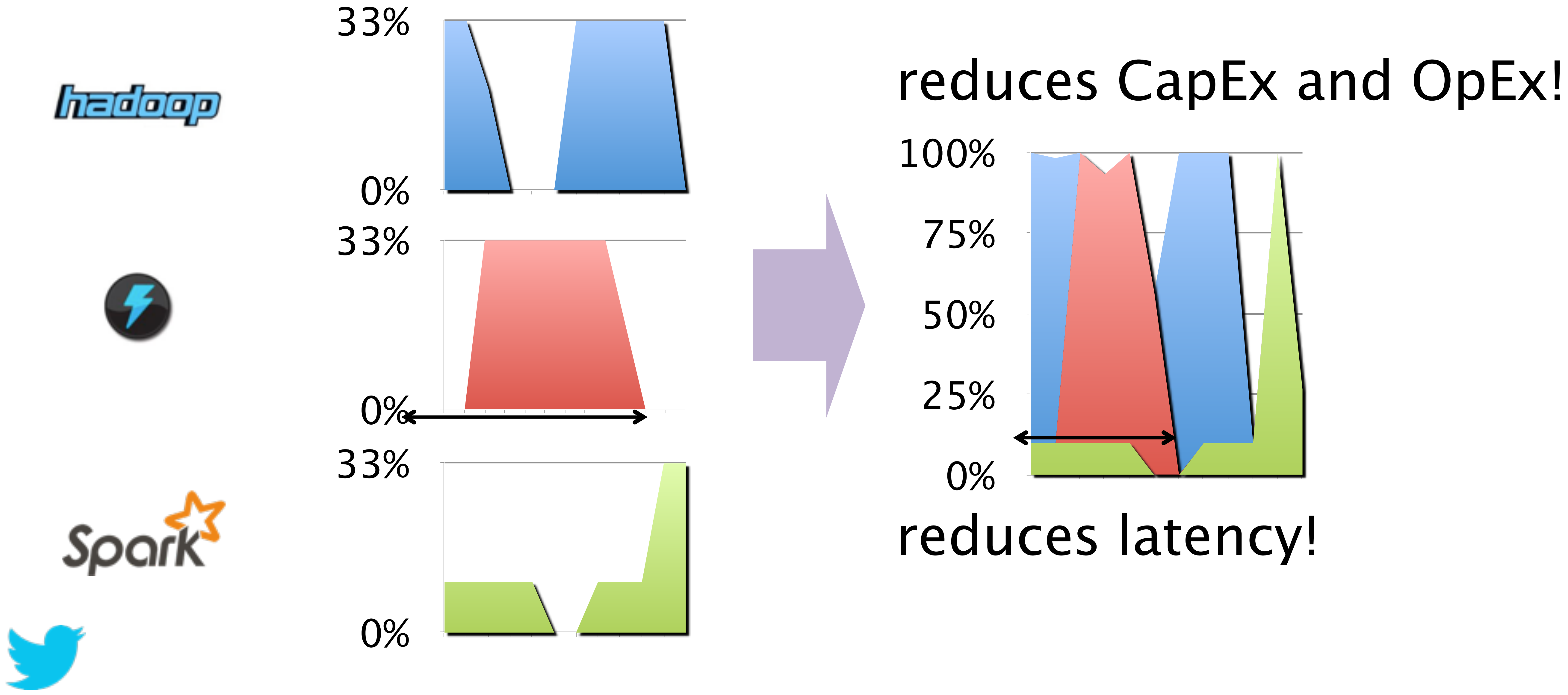


```
42 of 100 bytes written!
```

Data Center Computing

Reduce CapEx/OpEx via efficient utilization of HW

http://mesos.apache.org



reduces CapEx and OpEx!

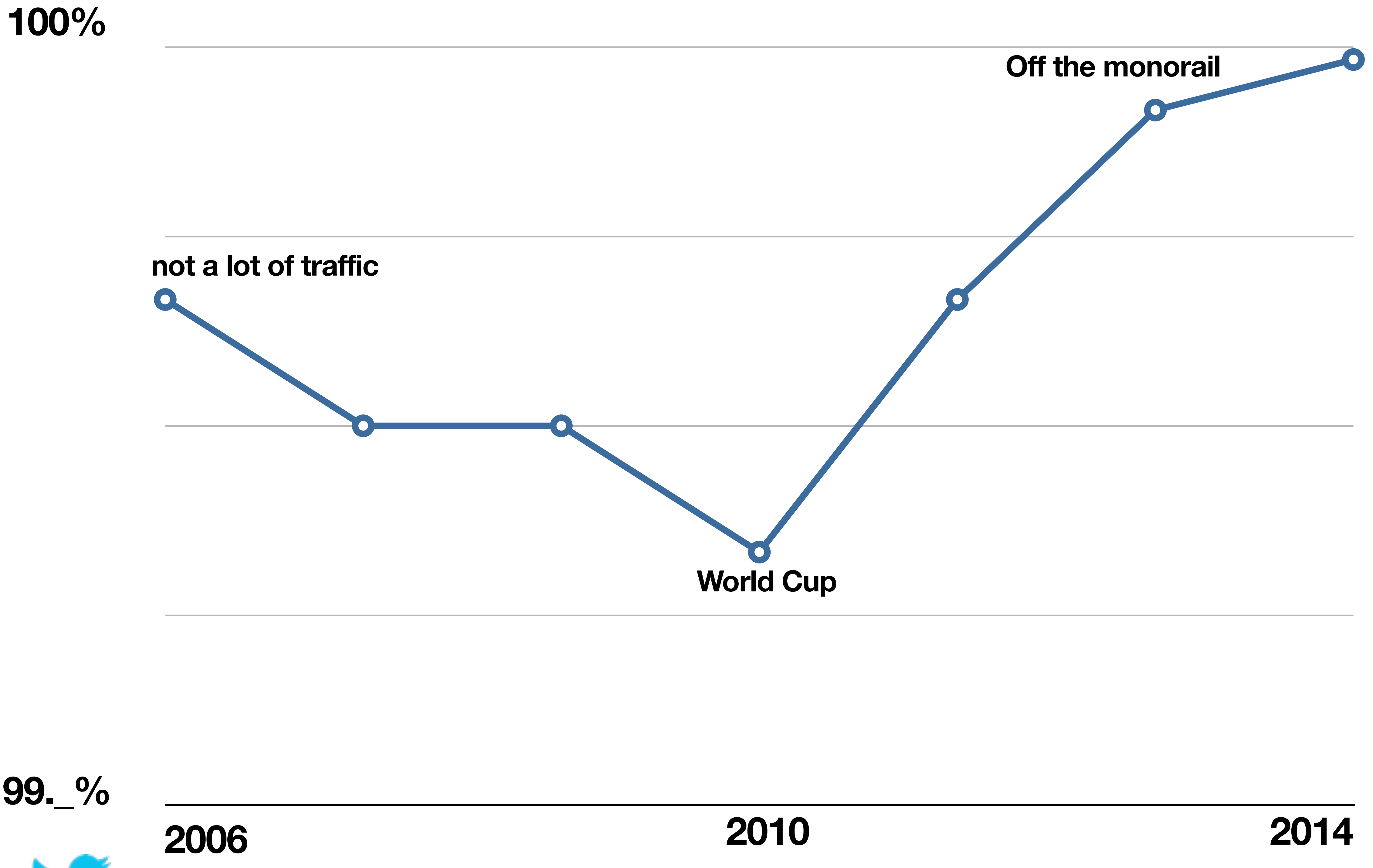
reduces latency!

How did it all turn out?

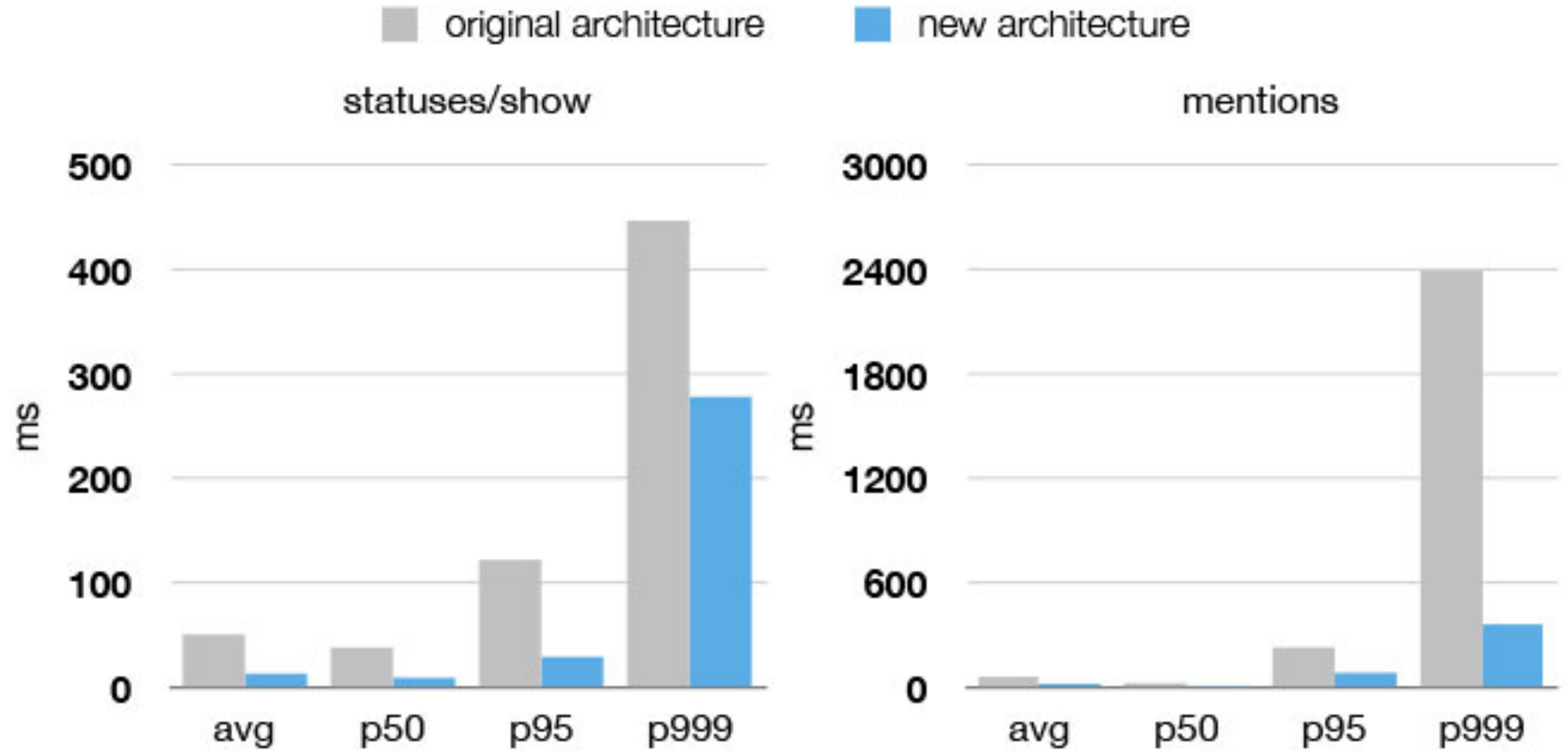
Not bad... not bad at all...

Where did the fail whale go?

Site Success Rate Today :)



Performance Today :)



Growth Continues Today...

3500+ Employees Worldwide

50% Employees are Engineers

284M+ Active Users

500M+ Tweets per Day

35+ Languages Supported

78% Active Users are on Mobile

200+ Open Source Projects

Concluding Thoughts

Lessons Learned

Lesson #1

Embrace open source

best of breed solutions are open these days

learn from your peers code and university research

don’t only consume, give back to enrich ecosystem:

http://twitter.github.io

Lesson #2

Incremental change always wins

increase chance of success by making small changes

small changes add up with minimized risk

loosely coupled micro services work

Lesson #3

“Data center as a computer” is the future direction of infrastructure

Efficient use of hardware saves money

Better programming model (large cluster as single resource)

Thanks for listening!

(hope you learned something new, see opensource.twitter.com)

remember, feel free to tweet me #eumjapan

@cra / @TwitterOSS

zx@twitter.com

Resources

https://github.com/twitter/finagle

https://github.com/twitter/zipkin

https://github.com/twitter/scalding

http://mesos.apache.org

http://wired.com/wiredenterprise/2013/03/google-borg-twitter-mesos

http://mesosphere.io/2013/09/26/docker-on-mesos/

http://typesafe.com/blog/play-framework-grid-deployment-with-mesos

http://strata.oreilly.com/2013/09/how-twitter-monitors-millions-of-time-series.html

http://research.google.com/pubs/pub35290.html

http://nerds.airbnb.com/hadoop-on-mesos/

http://www.youtube.com/watch?v=0ZFMlO98Jk