



# Hidden in the Clouds

Same Title  
New Talk  
Titles not my strong suit.

Shevek <shevek@nebula.com>

# Cloud Isn't New!



“Little Character”, Control Data / Seymour Cray

# The Systems Administrator's Story

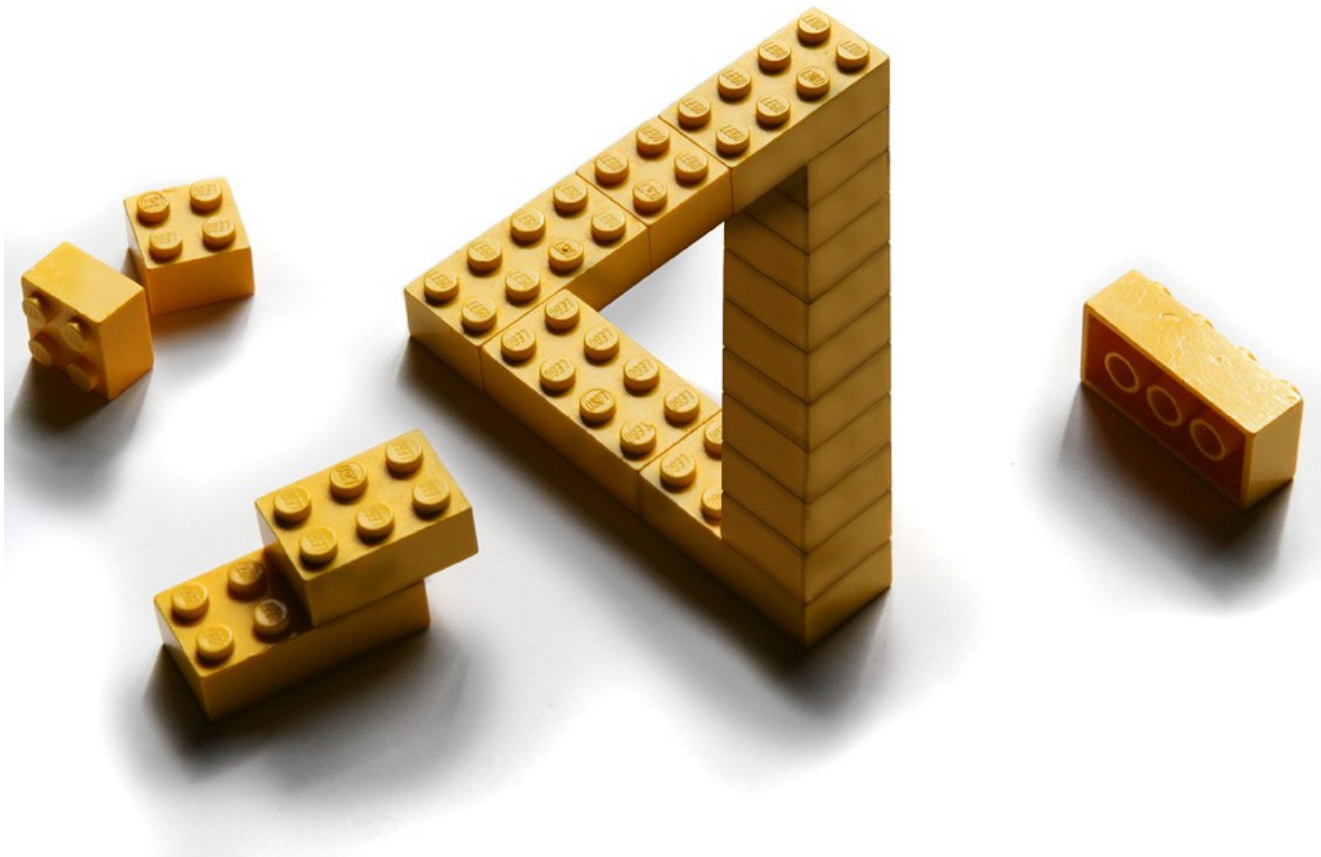
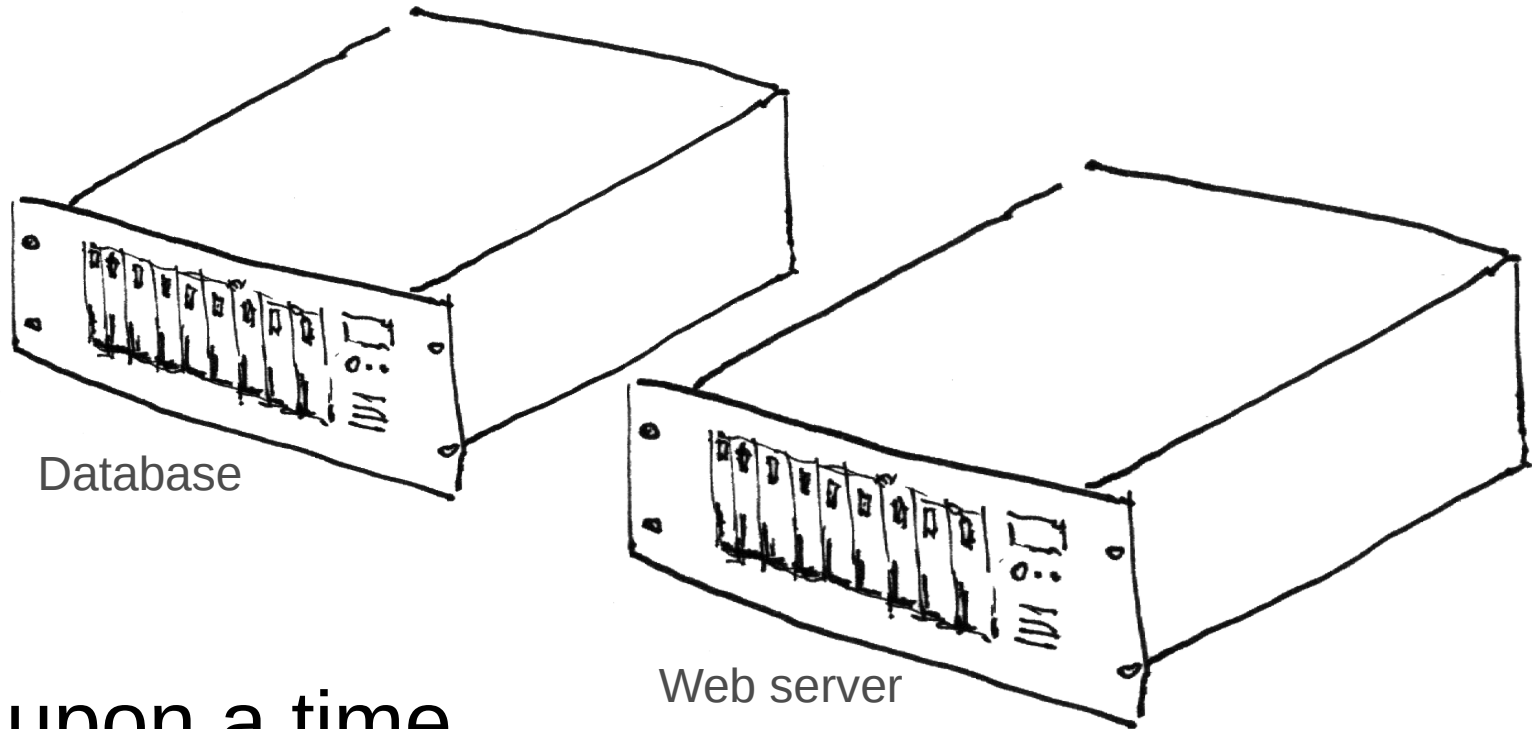


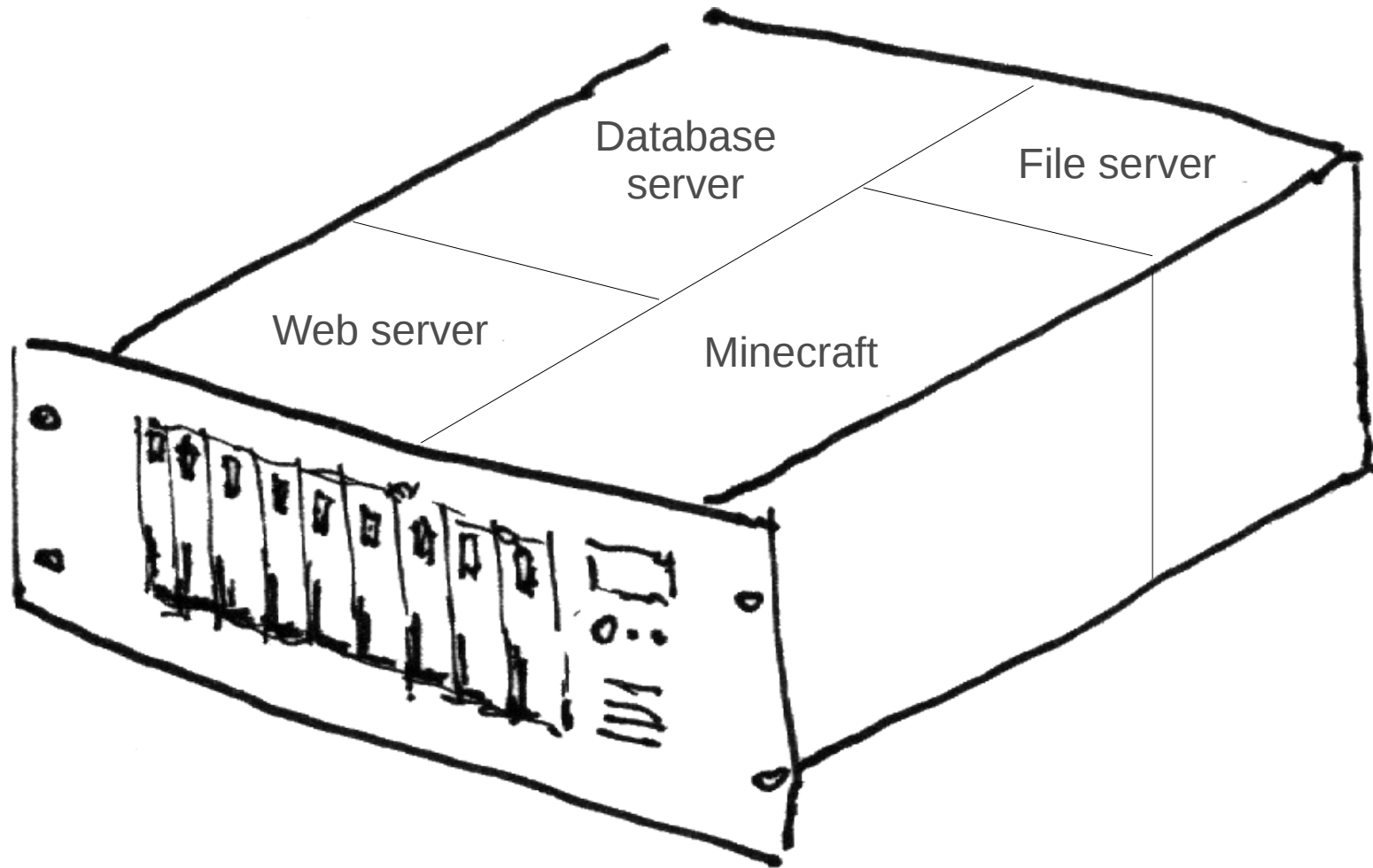
Image © 2012, Erik Johansson

# Servers



- Once upon a time, there was hardware...
- Adding a job required buying a server.
- And all management was manual.

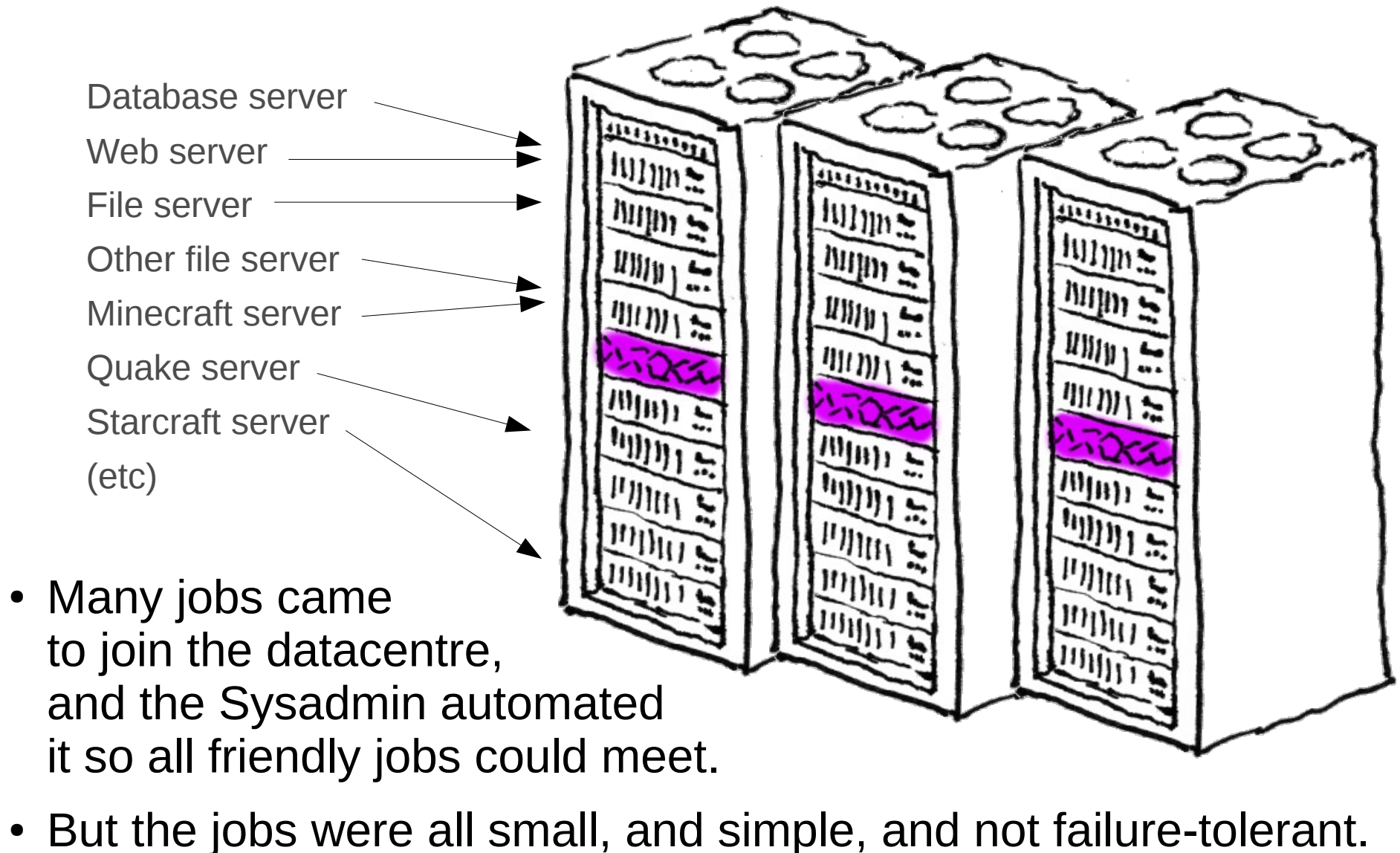
# Virtualization



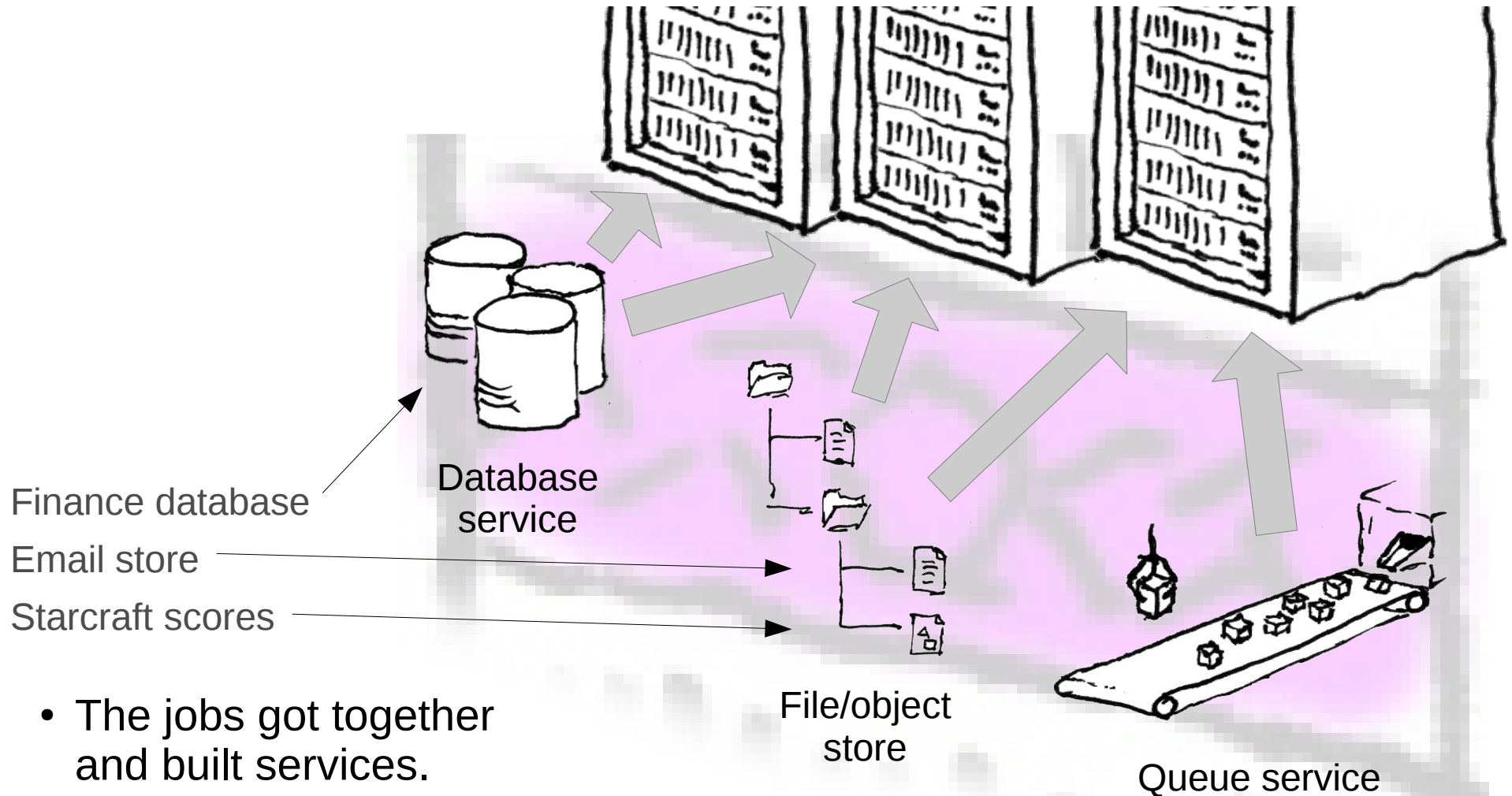
- The King^WSysadmin helped the jobs to make friends, share servers, and Costs were Reduced!.



# Infrastructure as a Service



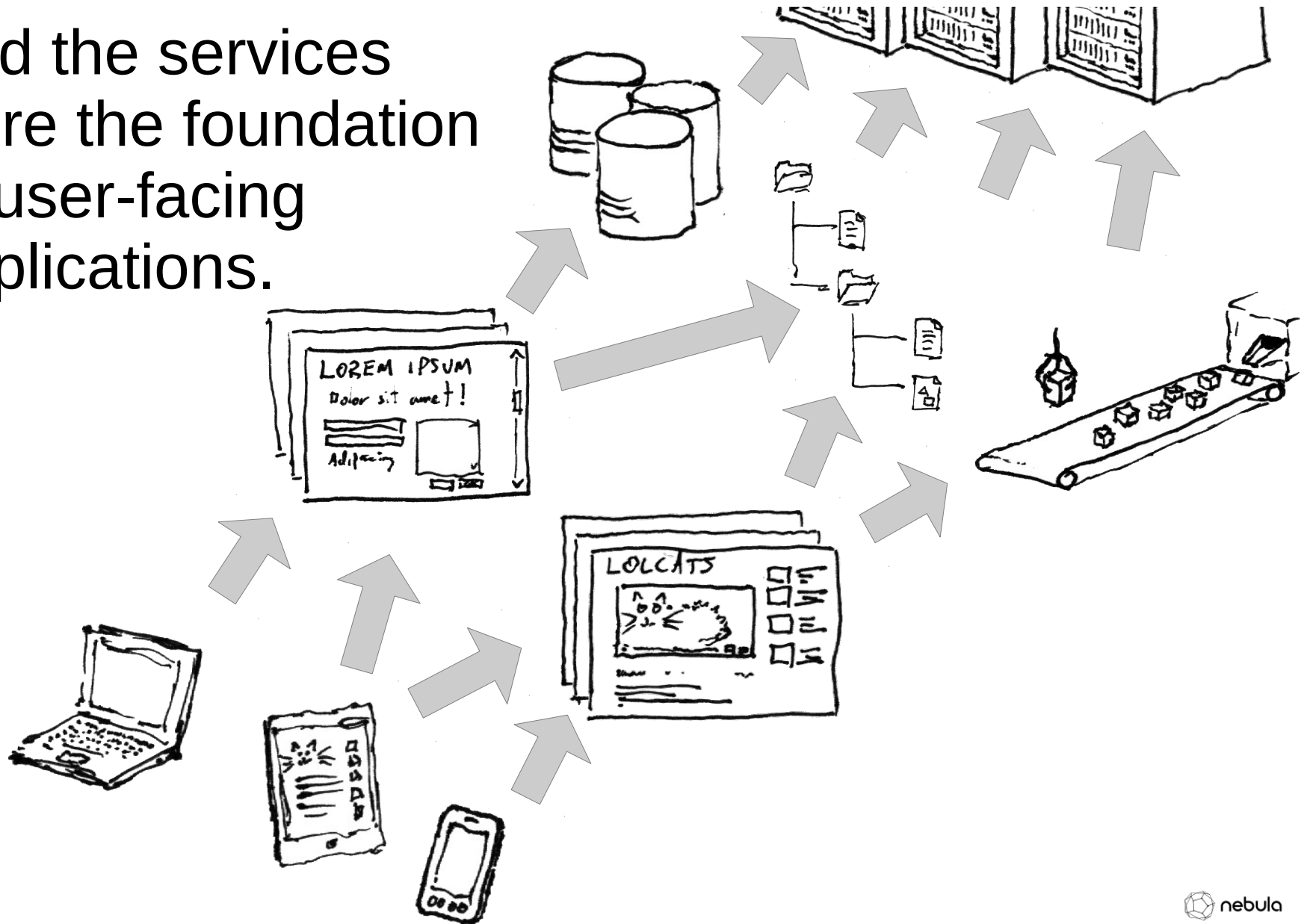
# Platform as a Service



- The jobs got together and built services.
- Services are fault-tolerant, and addressed via the control plane.
- The control plane hides the mapping to hardware.

# Software as a Service

- And the services were the foundation of user-facing applications.

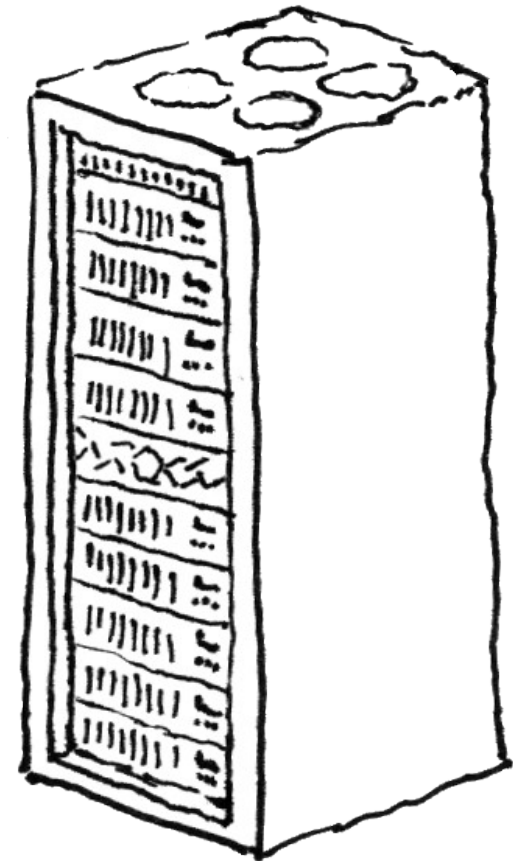
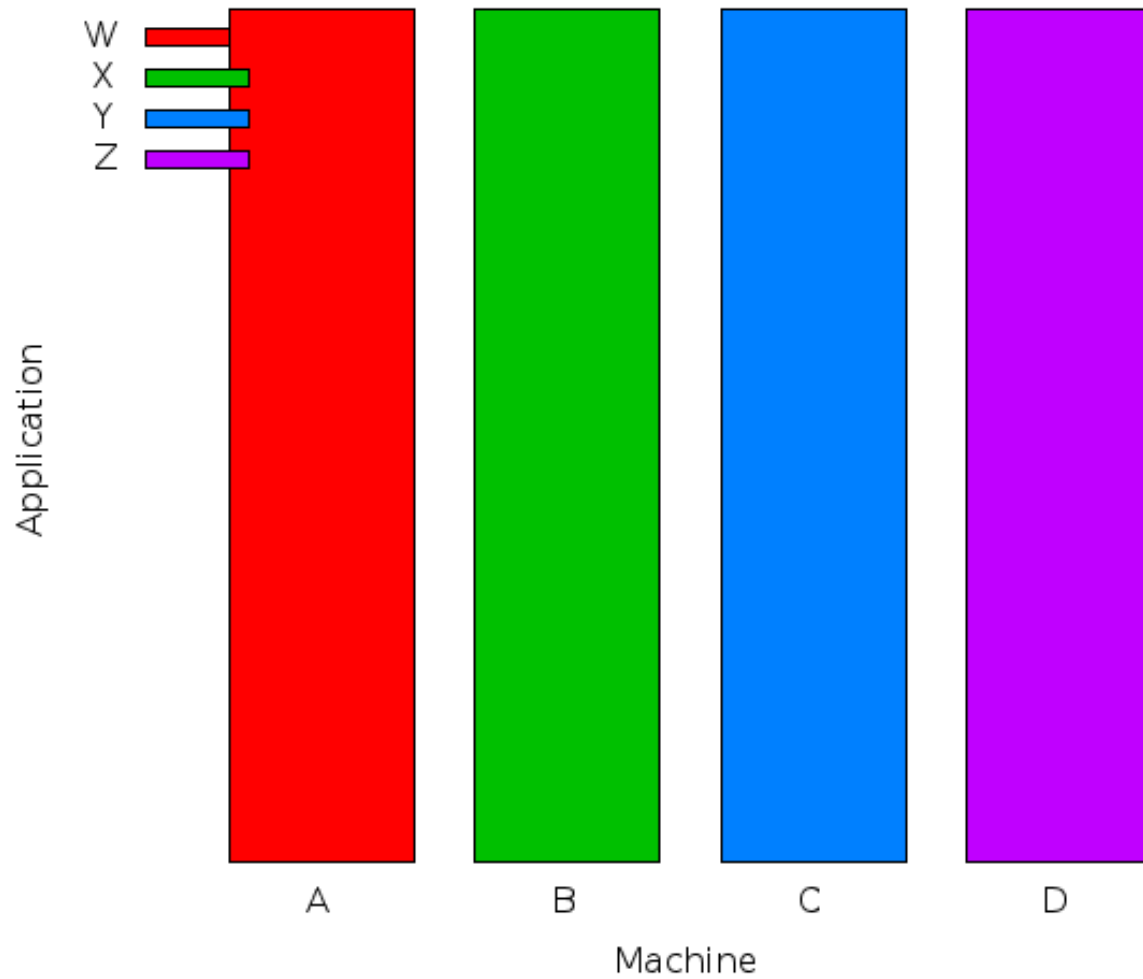




# Aggregation and Disaggregation

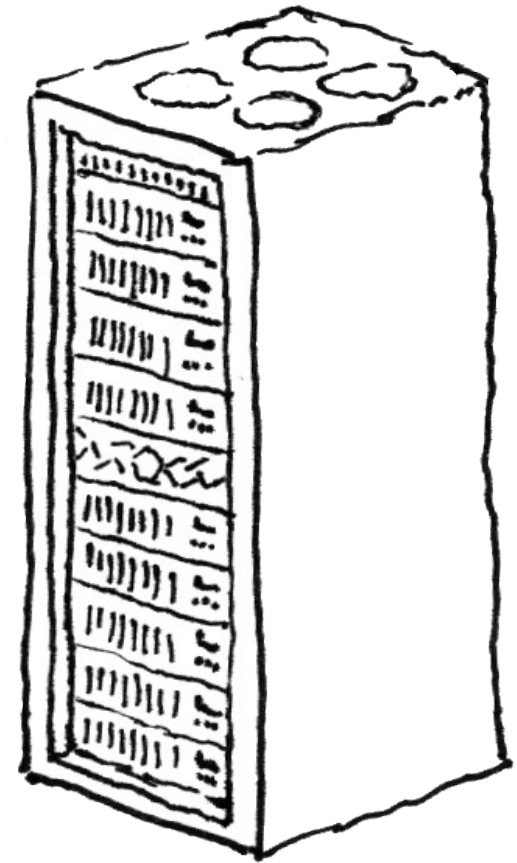
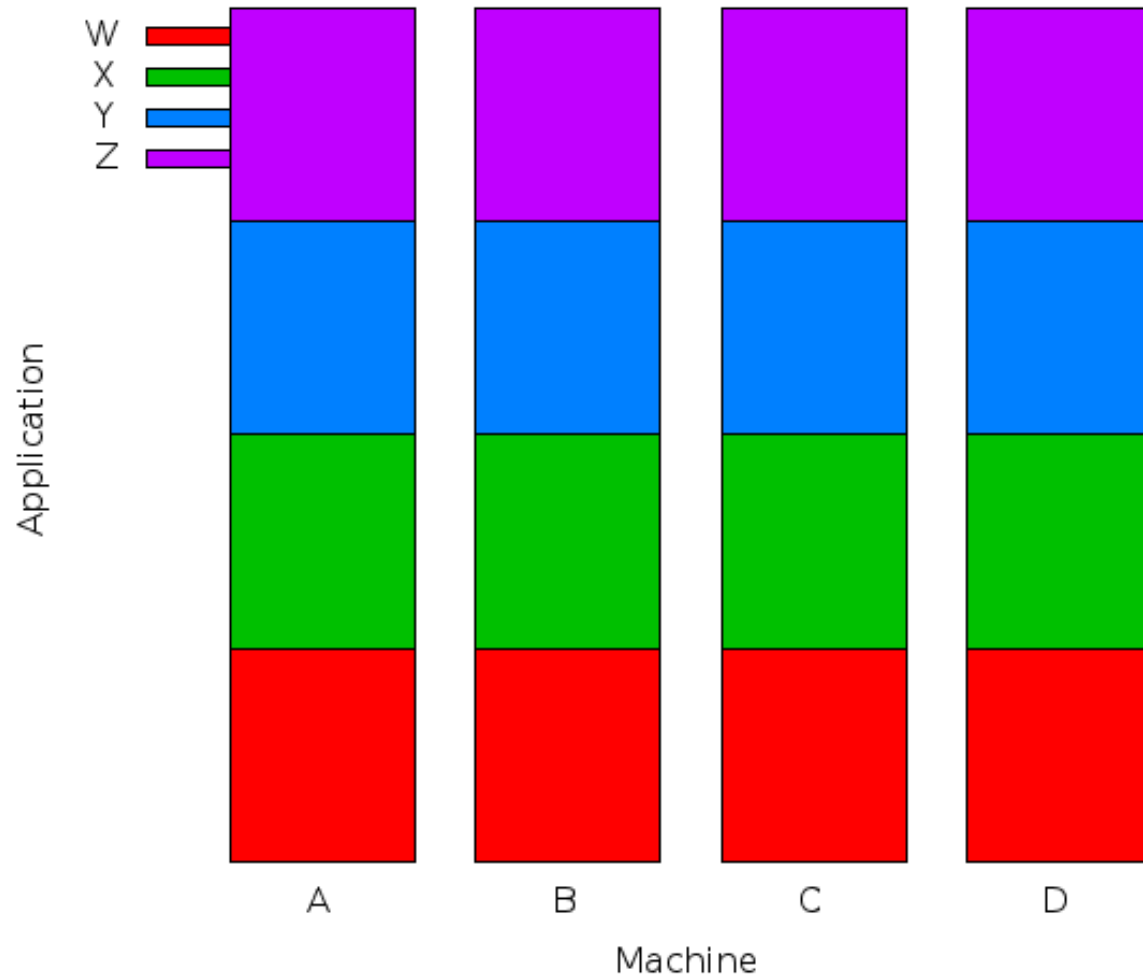
- Virtualization:
  - Disaggregation of hardware allows right-sizing.
- IaaS:
  - Automation of the control plane.
- PaaS:
  - Aggregation of hardware into a service, such as a database or filesystem.
- SaaS:
  - Disaggregation of a software installation into user-sized units.
  - We used to have one per desk.
  - Now we have one per cloud, or one per planet.

# Benefits of Virtualization



... and a machine fails.

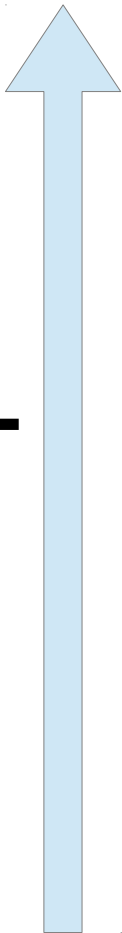
# Benefits of Virtualization



Now we have only two cases!

# Why did XaaS Change Business?

Developers



- Virtualization and low overheads.
- Standardized and uniform administration.
- Automatic system management.
- Resource tracking and accounting.
- Service definitions.
- Scalability with a linear cost model.
- Easy API and portal access.
- Development resources and tools.
- Lower barrier to entry for end users.

Administrators



[http://blogs.forrester.com/james\\_staten/13-02-25-why\\_your\\_enterprise\\_private\\_cloud\\_is\\_failing](http://blogs.forrester.com/james_staten/13-02-25-why_your_enterprise_private_cloud_is_failing)

# The Developer's Story

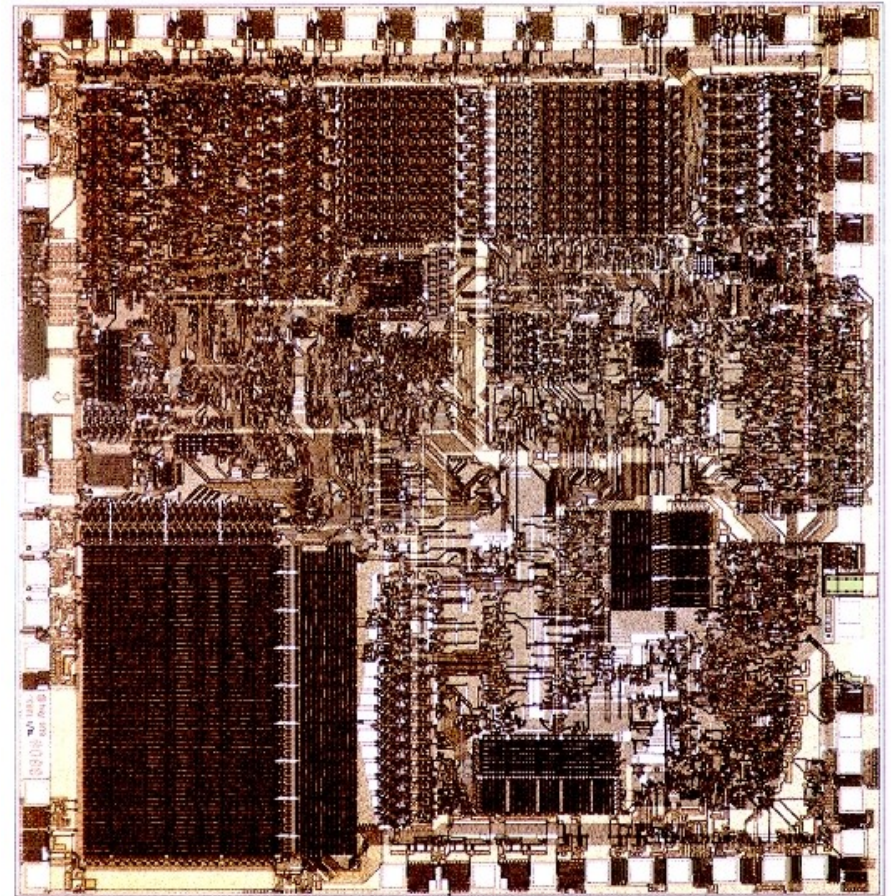
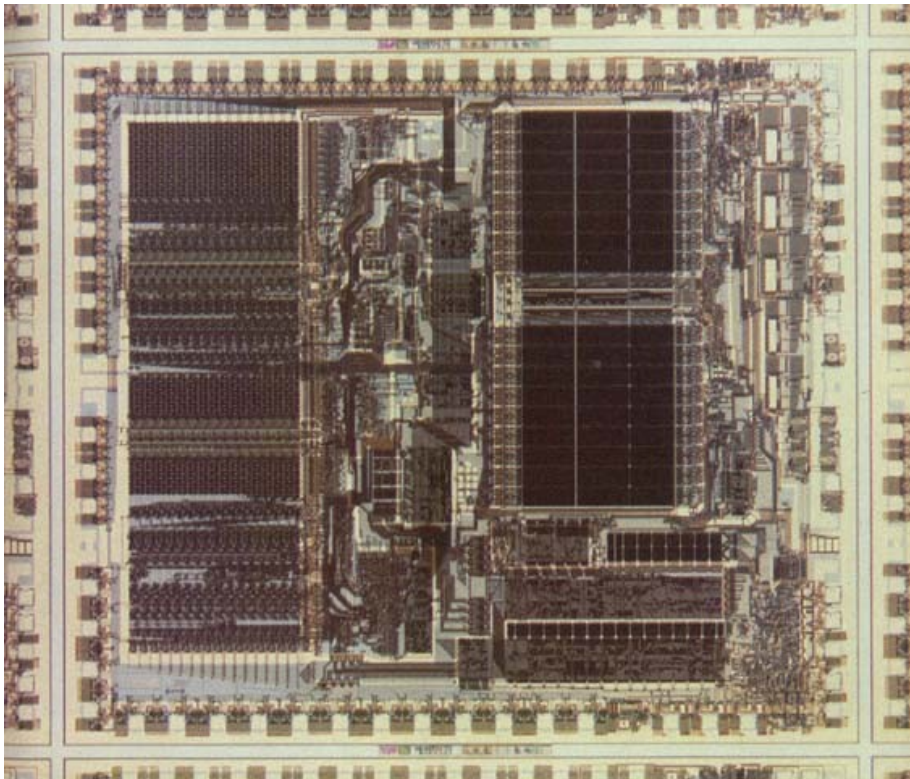




# Building High Performance Systems

- We need to do more work per unit time.
- What if we can't do more basic operations per second on a single CPU?
- Trade-off between number of instructions and complexity of instructions.

# Performance: RISC vs CISC



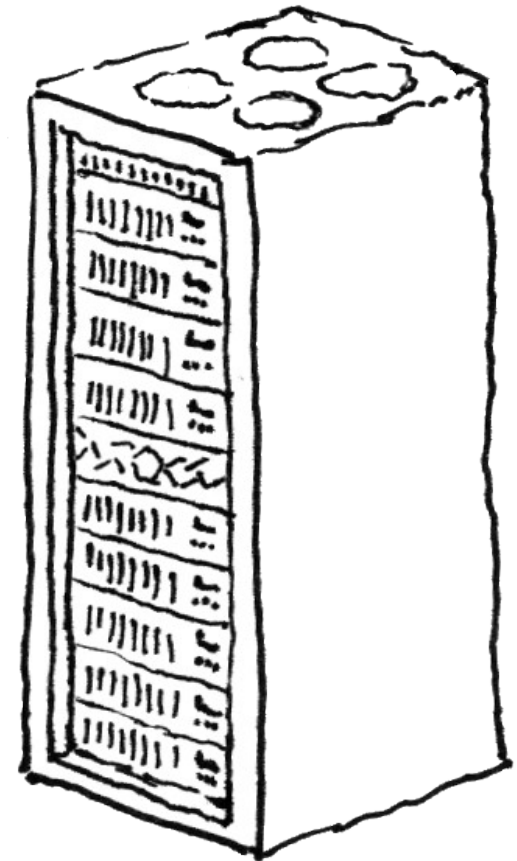
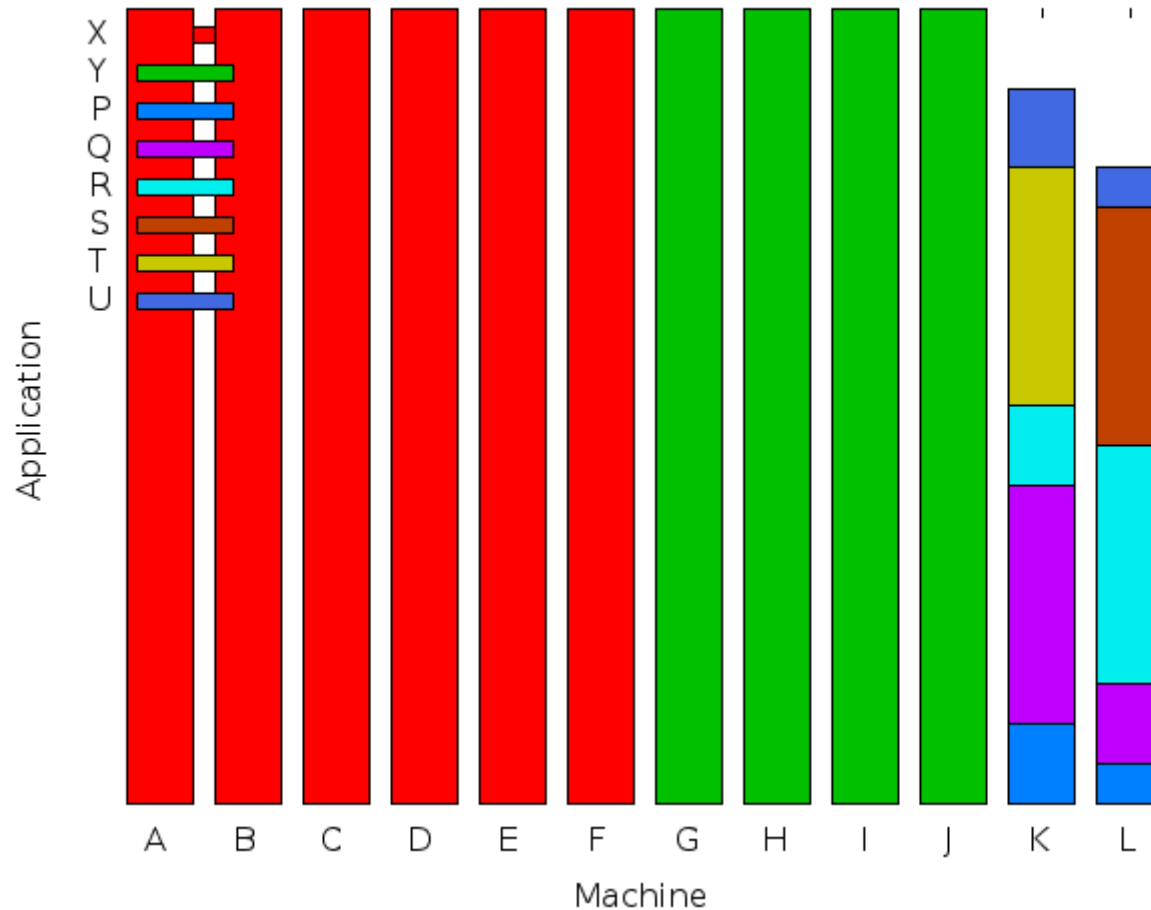
How does this map to the cloud?

# MPI vs MapReduce

- MPI
  - Small, simple operations
  - No checkpoints
- MapReduce
  - Slow, complex operations
  - Restartable operations
- Both are valid ways to use a cluster.
- Each has its strengths and weaknesses.
- Neither is inherently superior.

So what does our cluster look like?

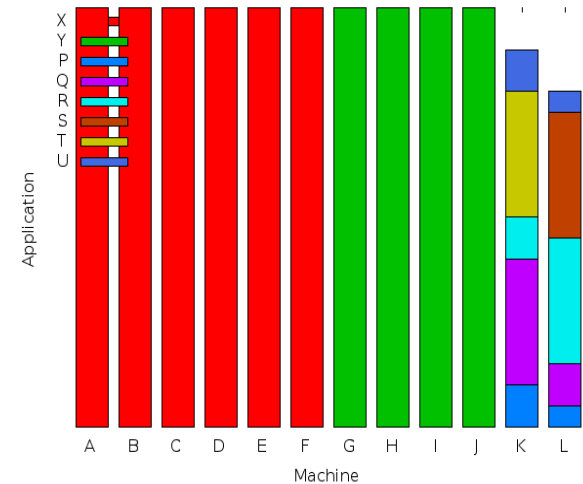
# Production Cluster Usage



Most of the cluster is combined to run larger jobs.

# Production Cluster Usage

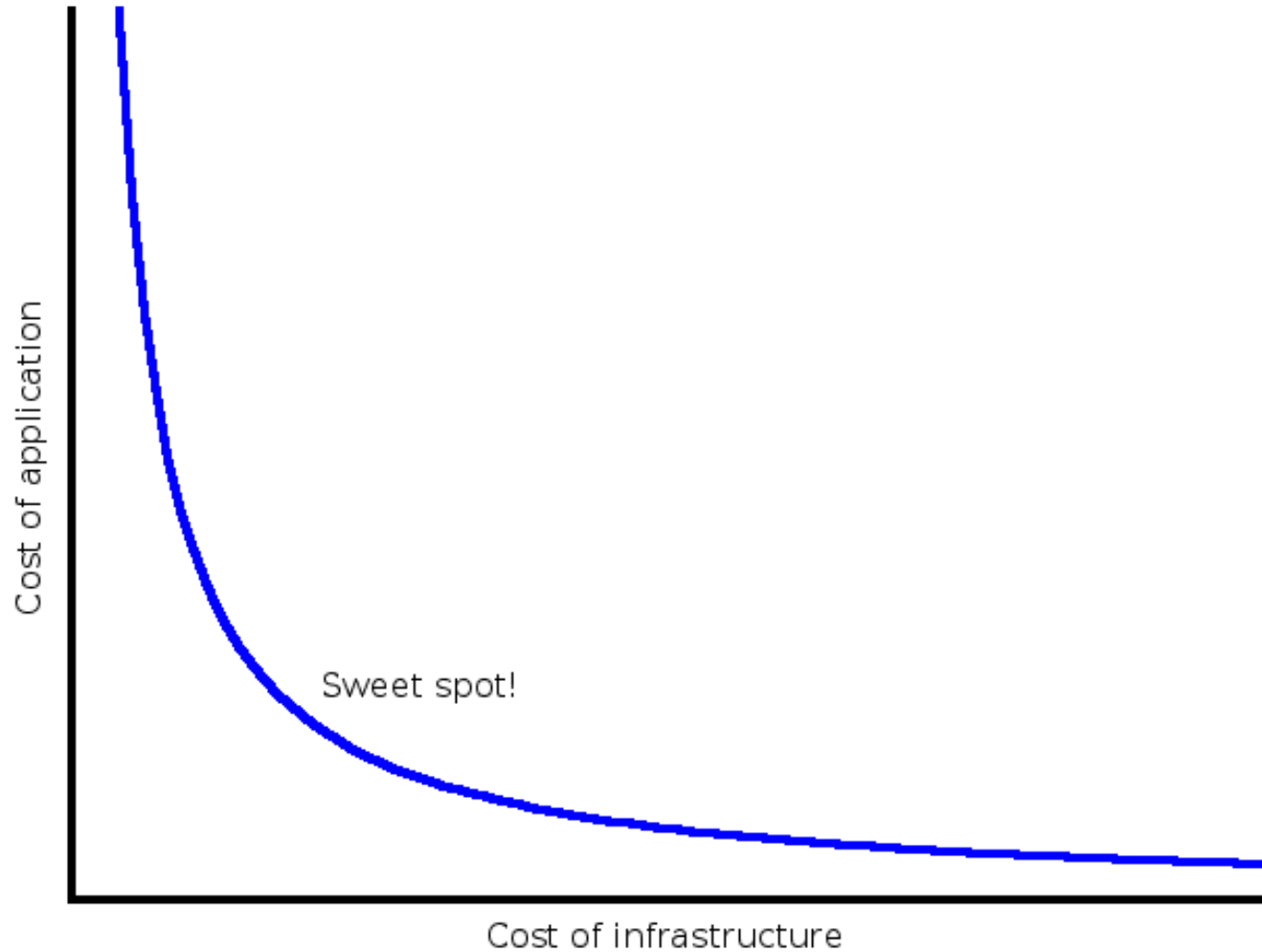
- Production jobs are larger than one node.
  - We have to subdivide the job.
  - The hardware SKU should match the natural subdivision of the job.
- Virtualization is overhead.



How do we design and manage this infrastructure?



# How Cloud Achieves Scale



So what did we win or lose?

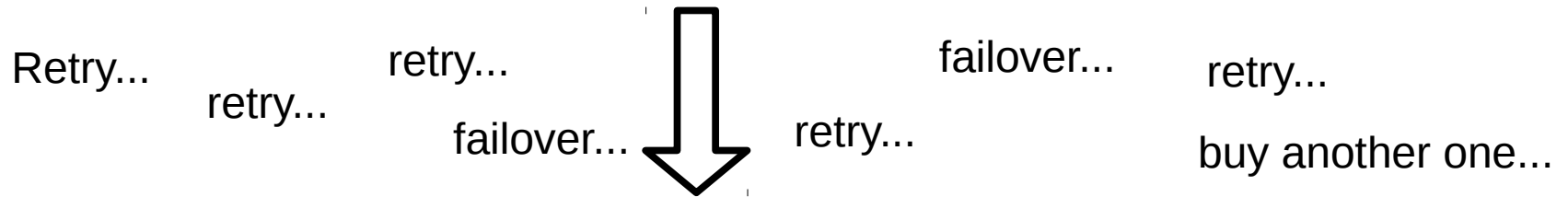
# How Cloud Achieves Scale

- This sweet spot has an associated set of programming techniques:
  - Restricted reliability guarantees.
  - Restricted coordination guarantees.
  - Simpler application contracts.
- As a consequence of this, we get scale!
  - Abstraction of hardware → orthogonality of hardware and software.
  - Automation → elasticity (accessibility) for developers.
  - Simplicity of contract → predictability and ease of programming.
  - Restricted coordination guarantees → scale-out.

Things fall apart...

# Exposition of Underlying Contract

The cost of hiding failure rapidly exceeds the benefit.

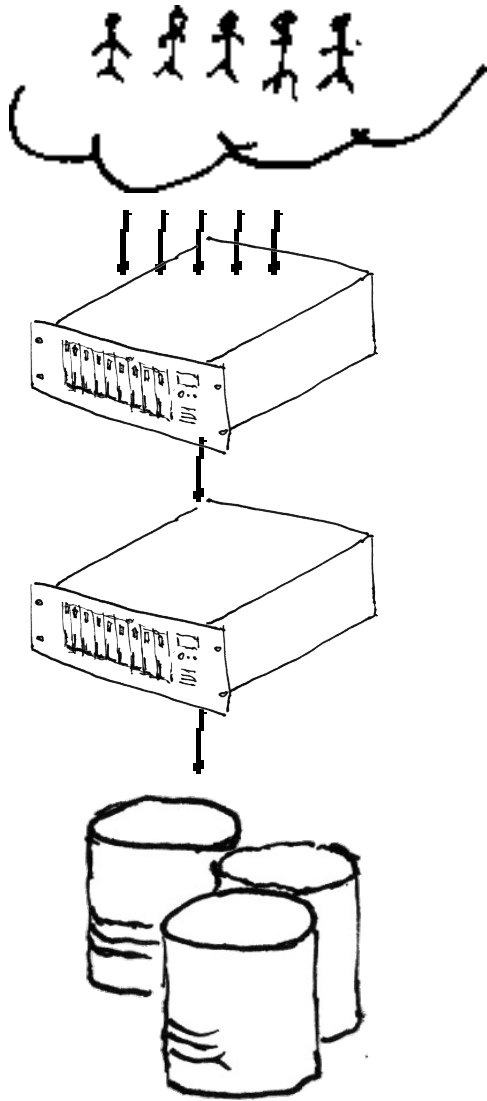


Handle, rather than hide failures.

... handle them in the application layer.

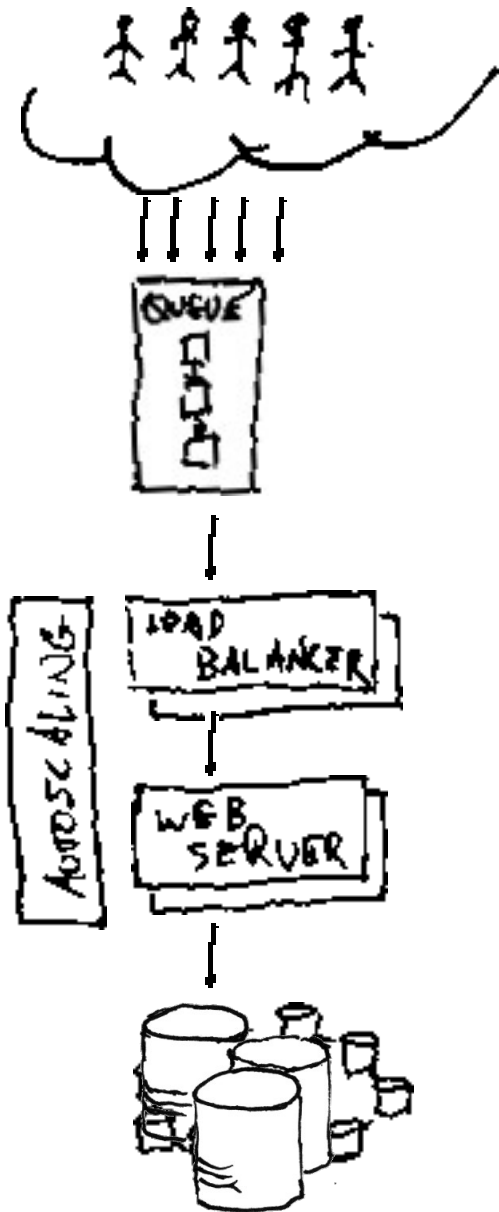
- The resulting robustness of the stack creates a more reliable service overall.
- Consider Netflix vs Oracle...

# Web Service: Traditional Model

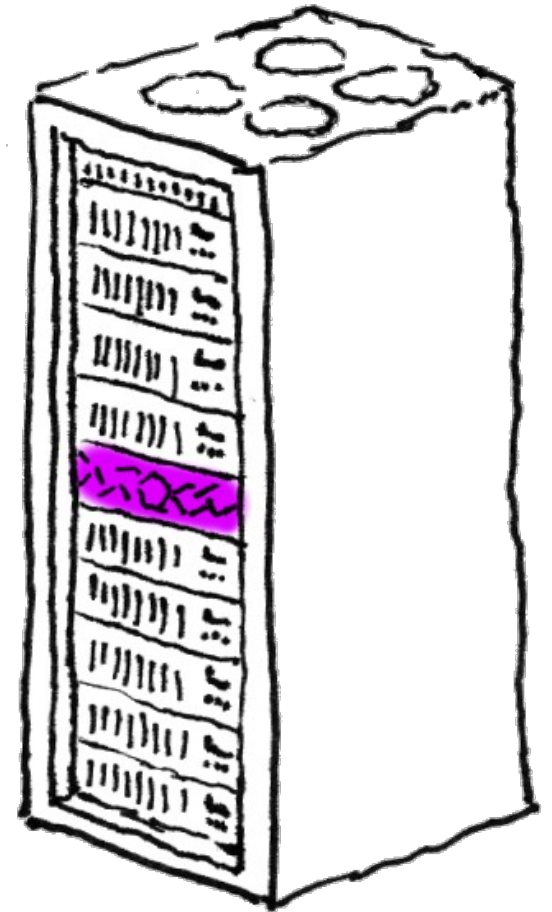


- Users
- Load balancer
- Web server
- Database

# Web Service: Cloud Model

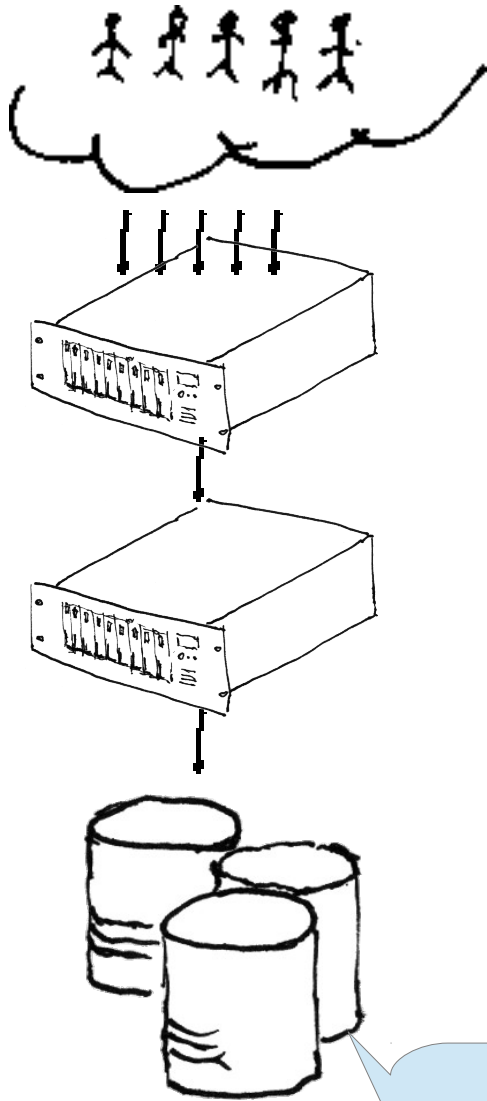


- Users
- Queue
- Load balancer
- Web server
- Storage ring



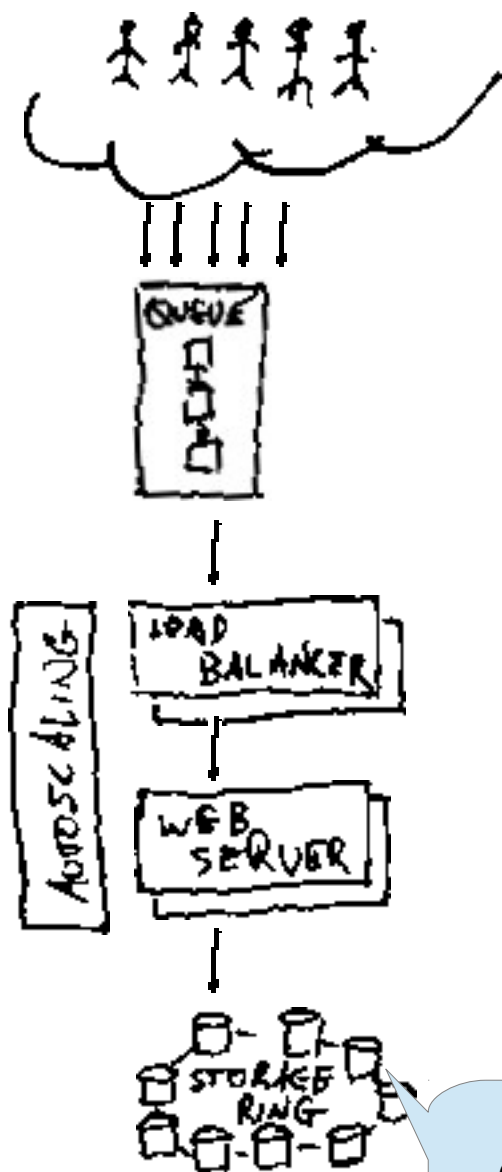


# Failure Analysis

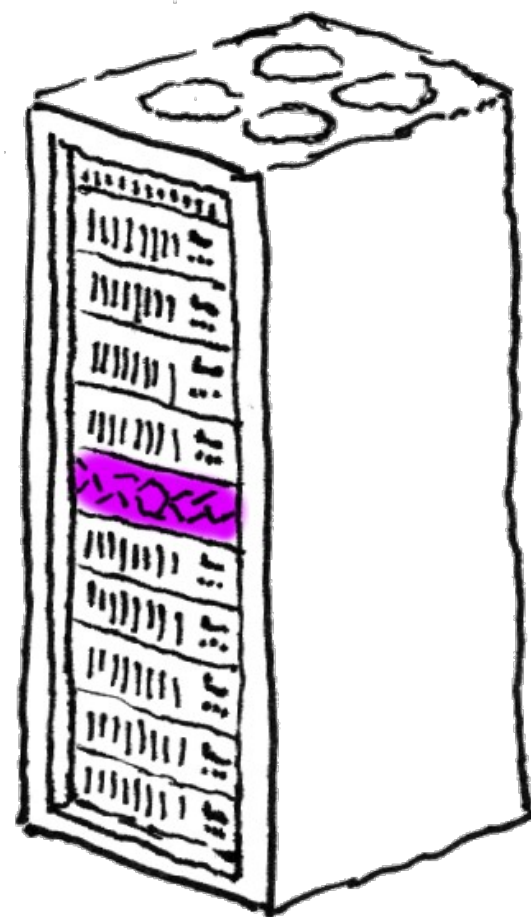


- Database crash
  - The database is not failure tolerant.
  - OK, OK, you paid for a failure tolerant database. Ouch!
- Database hiccup
  - The stack is synchronous.
  - Any failure is exposed to user.

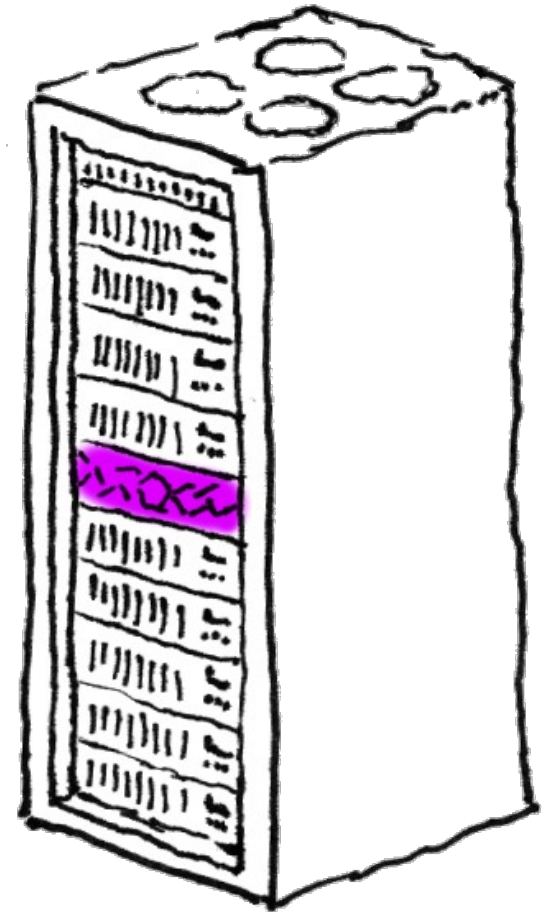
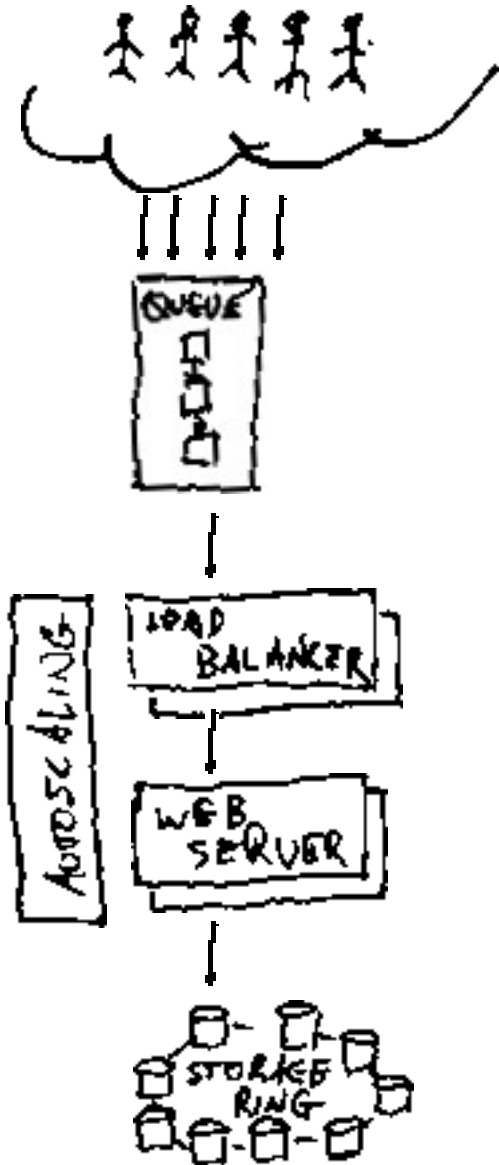
# Failure Analysis



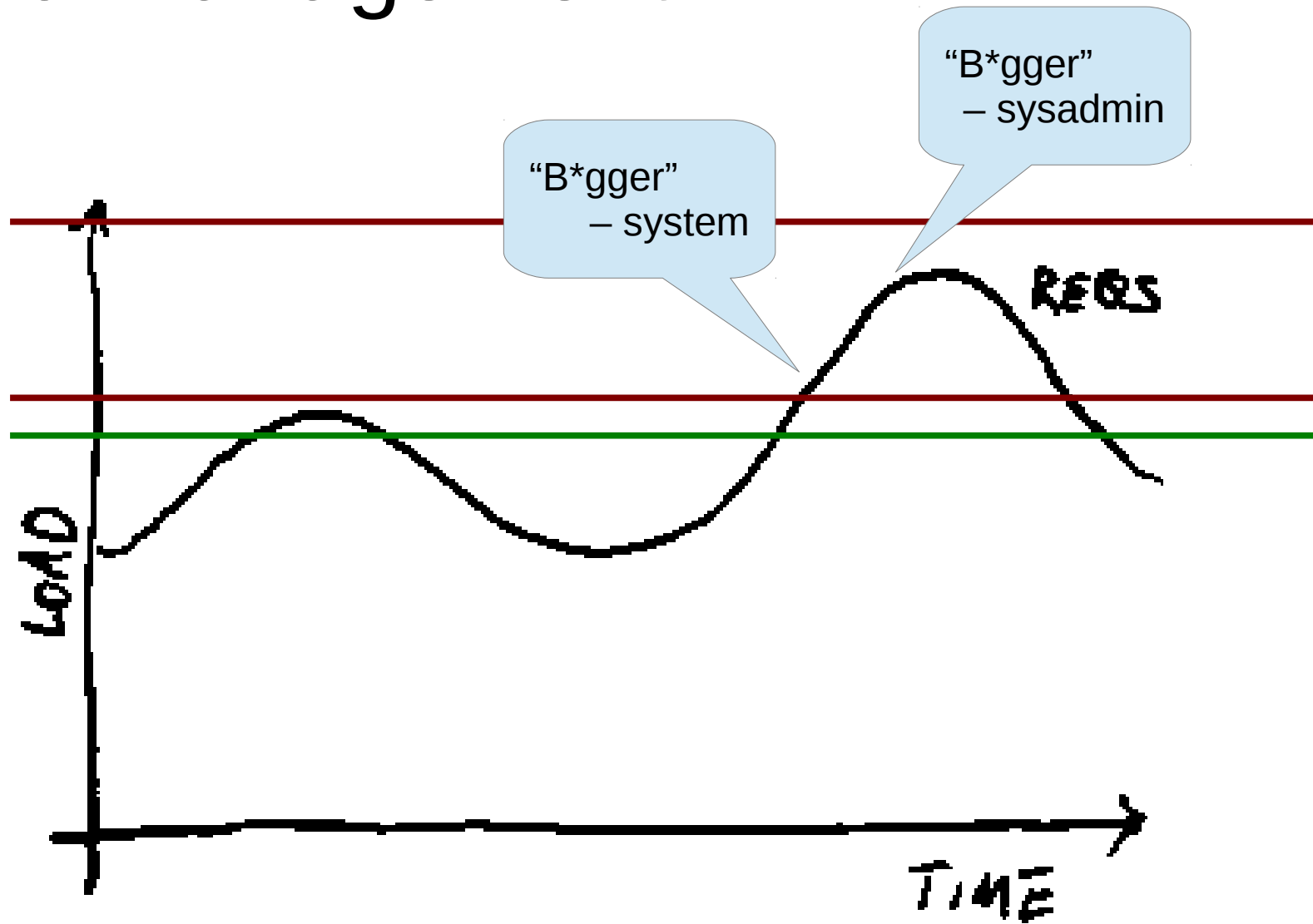
- A node in the storage ring crashed.
  - Who cares?
  - Not the storage ring, nor its clients.
- Storage hiccup.
  - Hiccups are tunable.
  - Either report failure to user, or  
CRASH!!! retry procesing from queue.



# Failure Analysis Questions

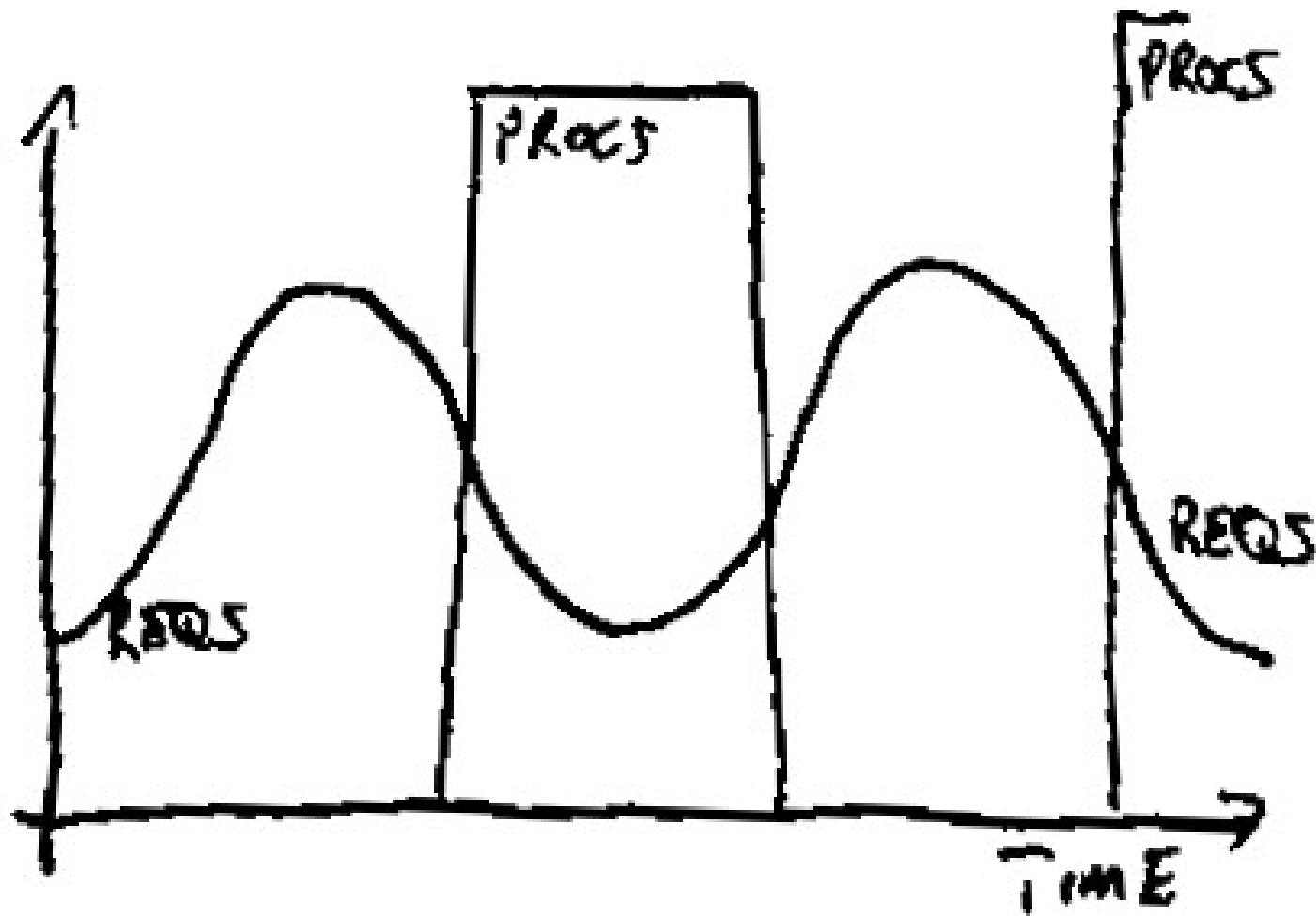


# Load Management



We can do better...!

# Turning the Knob



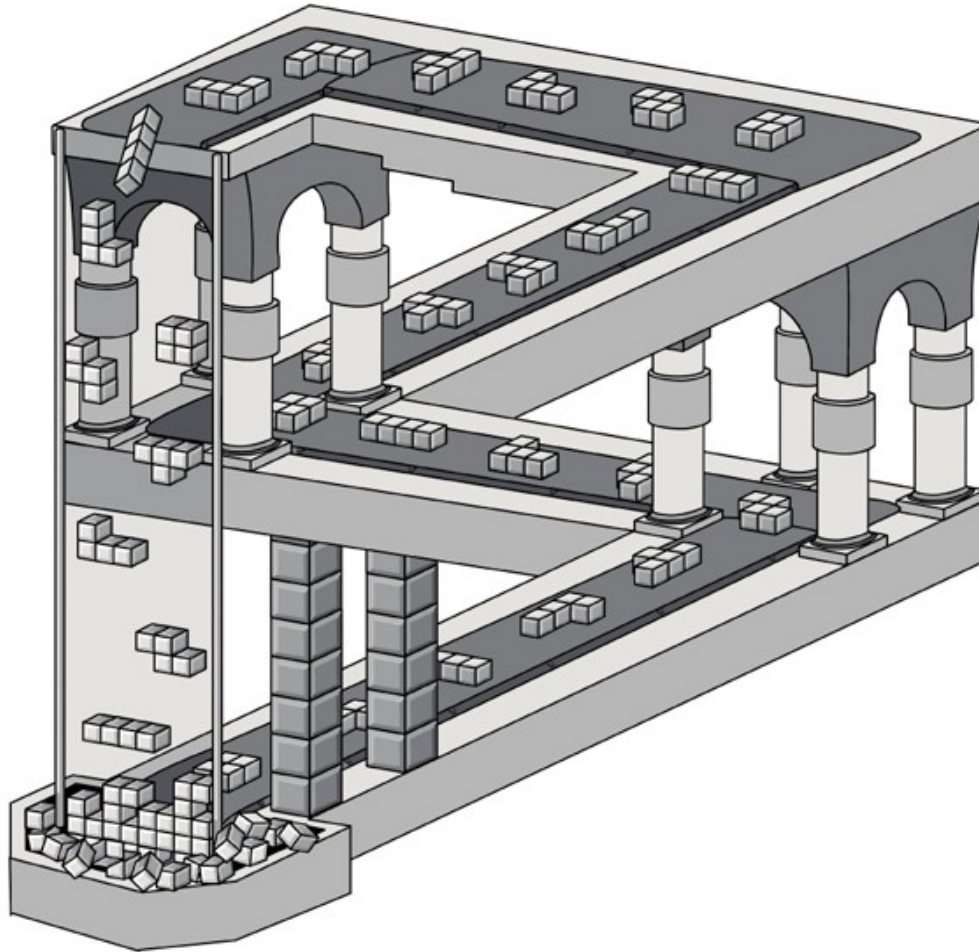
Next example.



# Cloud vs Scale-Up?

- Lazy algorithms with bad access patterns.
  - Many of these are bad in any case.
  - See Mechanical Sympathy.
- Shared-anything vs shared-nothing.
  - Do we need to go as far as shared-nothing?
  - Remote memory / RDMA.
  - MapReduce vs Bloom-Filter feedback.
- The same as NUMA, but more so.
  - Distances are larger.
  - Not many people can really program NUMA.
  - Think Cray again?
- One basket, and watch that basket.
  - Not practical or realistic, at any scale.
  - Ask anyone who has a home server.

# How to be Successful in the Cloud



“Would you rather fight 1 horse-sized duck or 100 duck-sized horses?”

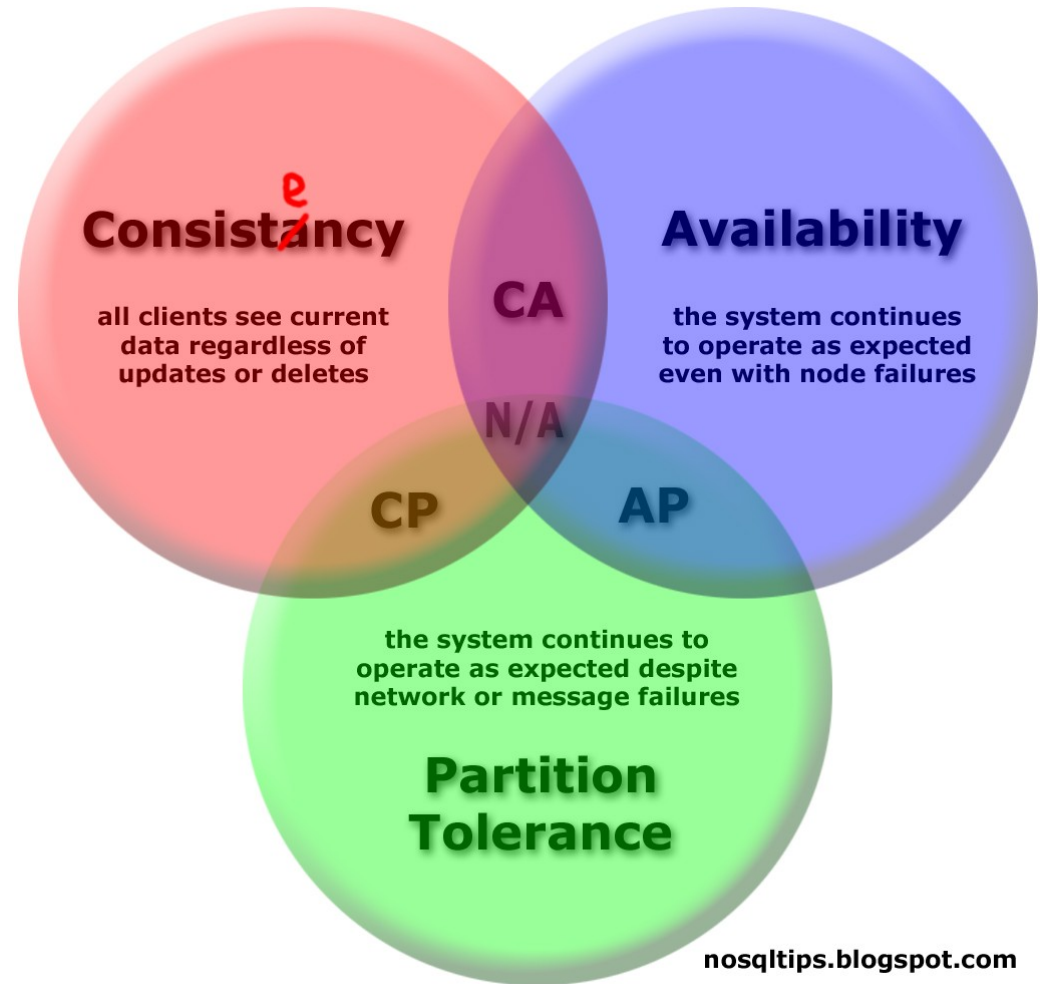
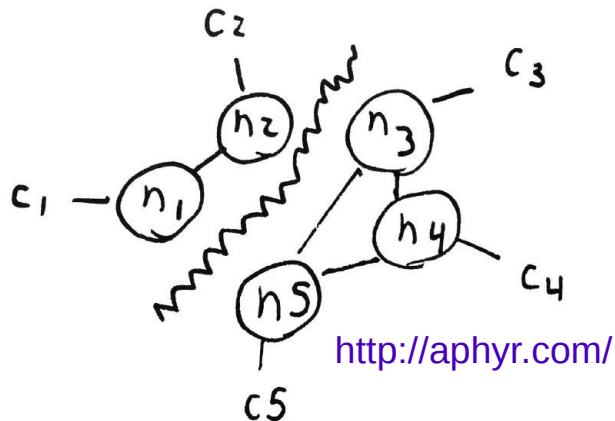
# Attributes of Cloudy Applications

- Of systems:
  - Stateless components
  - Failure tolerance, failover, circuit-breakers
  - Replication
  - Independent, loosely coupled components
- Of processes:
  - Independence of datasets
  - Repeatability

# Brewer's CAP Theorem

Any distributed system must either

- fail, or
- give the wrong answer.



Yes, I fixed the typo in the image.

# Architectural Guidelines

- Separate long term storage.
  - This is the only “reliable” component.
  - All other components should be stateless.
- Subdivide your dataset or workload.
  - The I/O layout will be tightly coupled to your algorithm.
  - Allow for re-execution of a unit.
- Checkpoint computations.
  - Decide how much (of what) you are willing to lose.
- Consider approximation algorithms.
  - You can compute the correct answer even with incorrect intermediates.

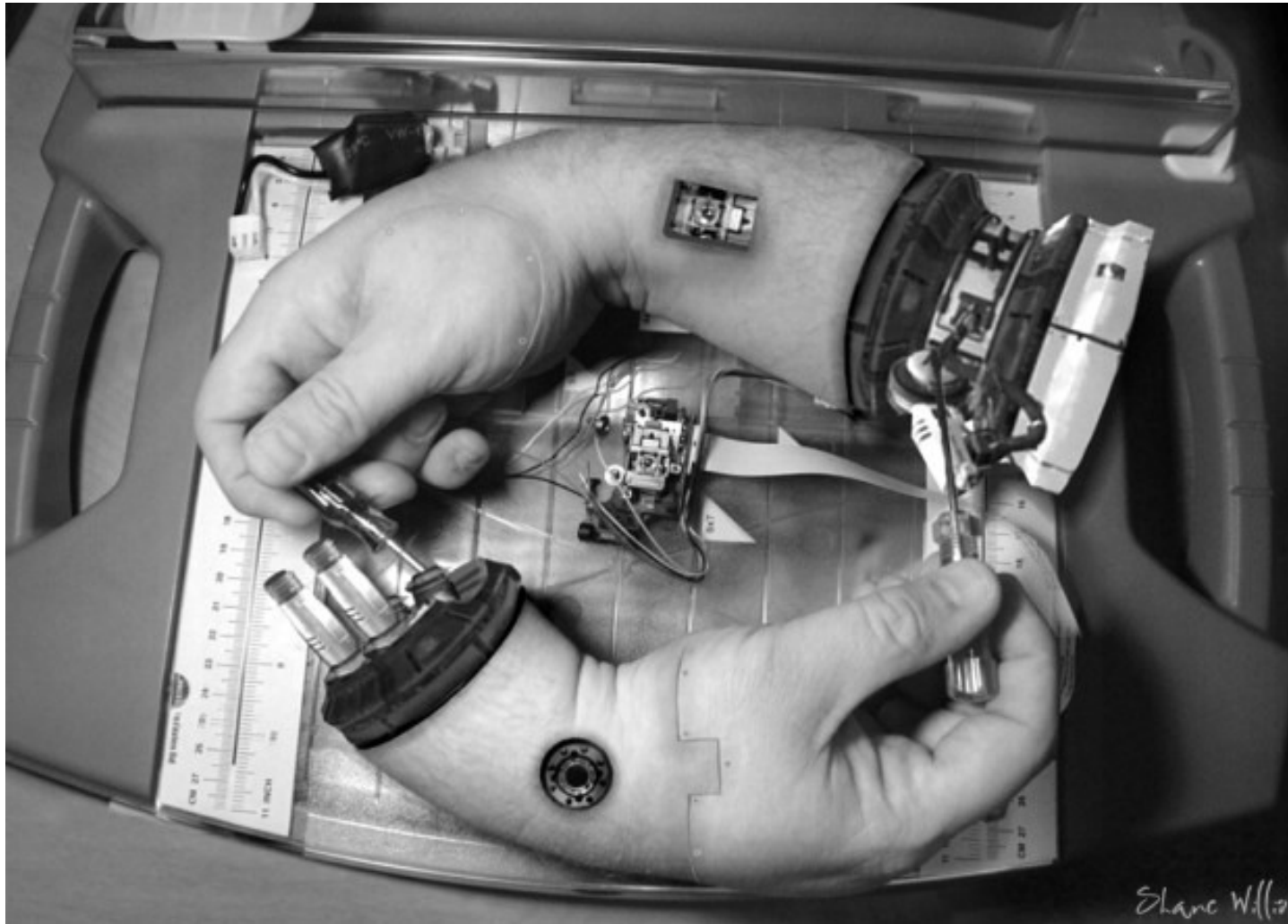
# Mistaken Requests

- Some common, but (usually) mistaken requests:
  - Transactions.
  - Hot fail-over.
  - Process migration.
  - Strongly consistent ordering or clocks.
  - Cluster-wide truths.

Let's talk some examples, while we're here.



# Implementations and Examples



# Existing Building Blocks

- Building Blocks
  - Cassandra
  - ZooKeeper
  - MapReduce/Hadoop
  - JGroups
- Tools and Management:
  - Dapper
  - Hystrix
  - Scribe
  - ChaosMonkey
- Counterexamples:
  - MPI, MySQL, Mosix, DRBD

# Cassandra (Facebook/Netflix)

- High performance distributed hash table.
  - Replaces the relational database.
  - Optional consistency.
    - Allows a performance/consistency trade-off.
  - Schema-free long term storage.
  - Denormalized data.
    - Seeks cost more than reads.
  - No transactions!
  - No shutdown procedure!
    - All the focus is on crash recovery.
- And the real meat:
  - Dynamo, hinted handoff, reconstruction, ...

# ZooKeeper (Everybody)

- A distributed agreement system.
  - Atomic operations across multiple machines.
  - Twitter use it for configuration.
  - Netflix wrote the Curator client.
  - Assume it useful, but do not assume it reliable.
  - Does not scale.

# Hadoop (Yahoo, World+Dog)

- HDFS:
  - Distributed filesystem, reasonably robust.
  - Very restricted API.
- MapReduce:
  - Restartable and repeatable computation.
- Hive:
  - A MapReduce-based SQL engine.
- HBase:
  - A distributed hash table.
- Other projects:
  - Varying levels of maturity and reliability.

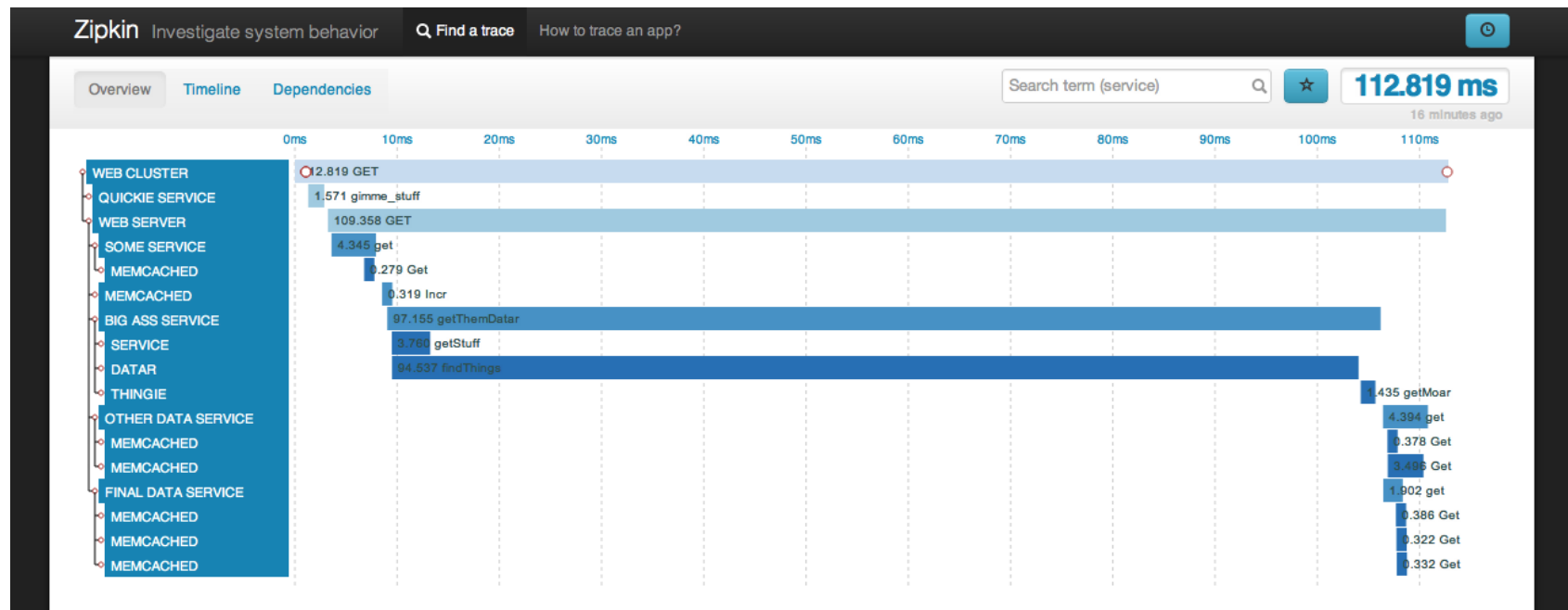
# Implementations of Tools

- Distributed Tracer: Dapper/Zipkin
- CircuitBreaker: Hystrix
- Logger: Scribe
- Exception centralizer: ???
- Fault Injection: ChaosMonkey



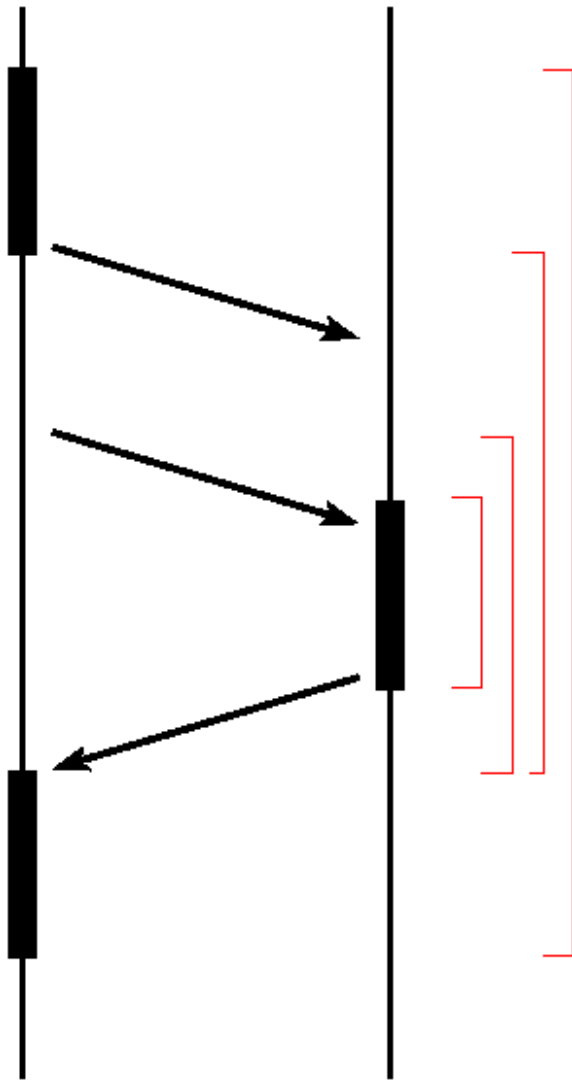
# Tools: Zipkin (Twitter)

- A holistic view of system behaviour.
- What happened, and when?
- Adaptive rate sampling



See: <http://engineering.twitter.com/2012/06/distributed-systems-tracing-with-zipkin.html>

# Aside: Large System Effects

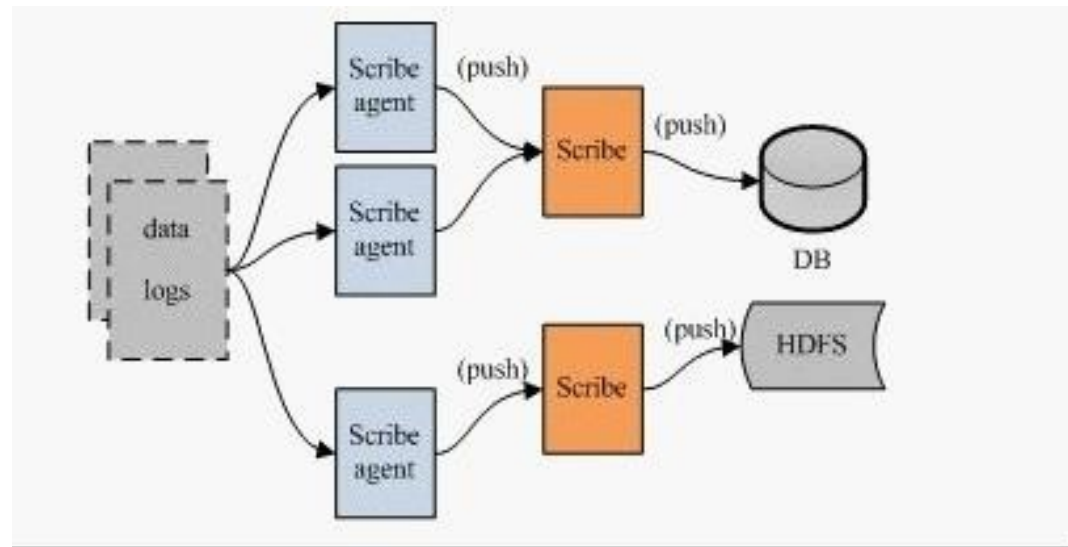


- In a large system, overheads matter.
- We must account for:
  - Setup time.
  - Failed calls.
  - Network delay.
  - Tear-down time.

See: <http://research.google.com/pubs/pub36356.html>

# Logging (Facebook)

- Scribe: A fault tolerant log-routing framework.



The median latency for trace data collection is less than 15 seconds. The 98th percentile latency is itself bimodal over time; approximately 75% of the time, 98th percentile collection latency is less than two minutes, but the other approximately 25% of the time it can grow to be many hours. – Sigelman et al, Google

See: [http://www.facebook.com/note.php?note\\_id=32008268919](http://www.facebook.com/note.php?note_id=32008268919)

# Transactional Logging

- Reduces log volume from successful calls.

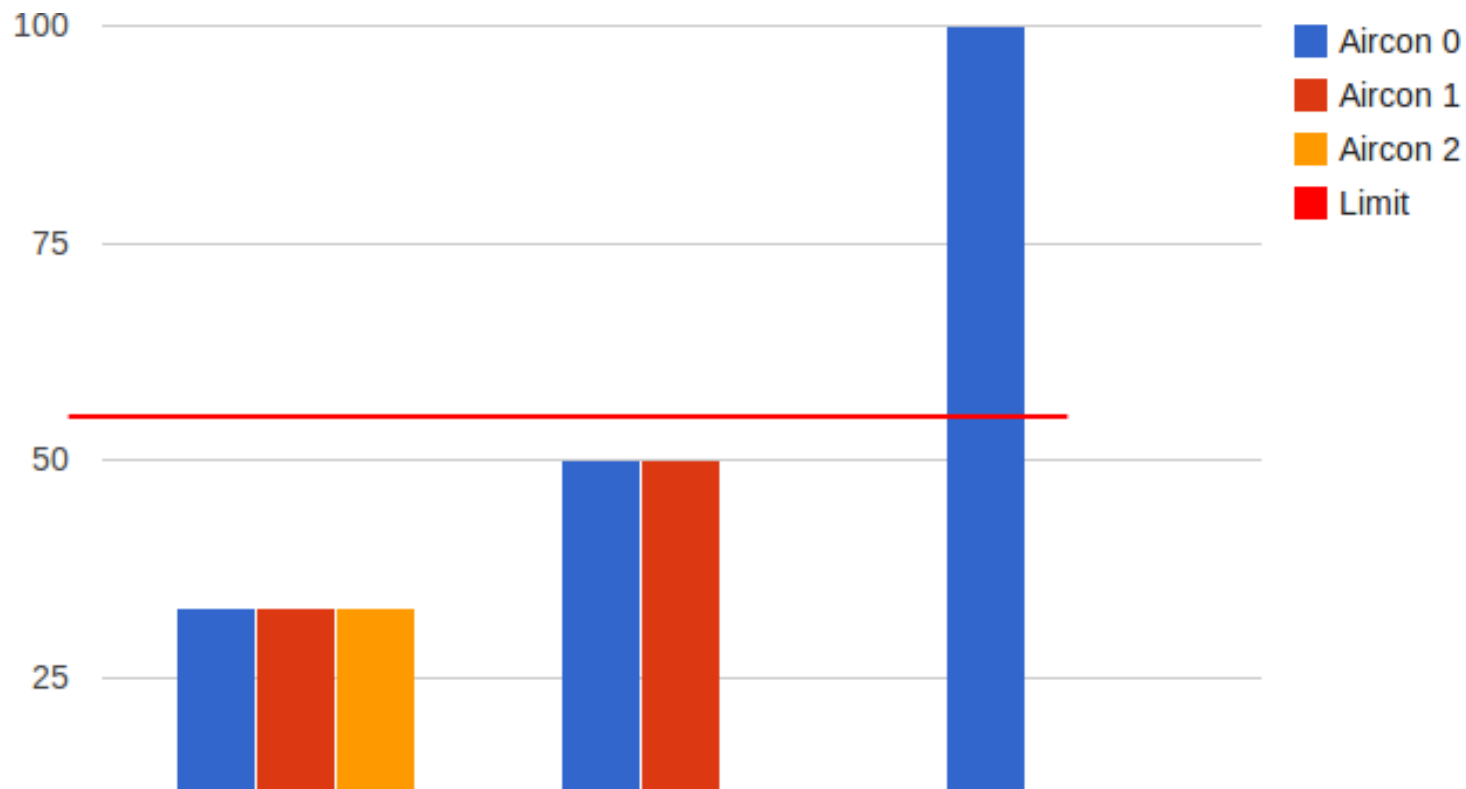
```
LogContext ctx = LOG.begin();
try {
    ...
    if (LOG.isDebugEnabled())
        LOG.debug(...);
    ...
    ctx.discard();
} finally {
    ctx.close();
}
```

- If an exception is thrown, messages in the context are not discarded.

See: <http://pragprog.com/magazines/2011-12/justintime-logging>

# Failures Cascade

- Failure of one mirror transfers load to others.



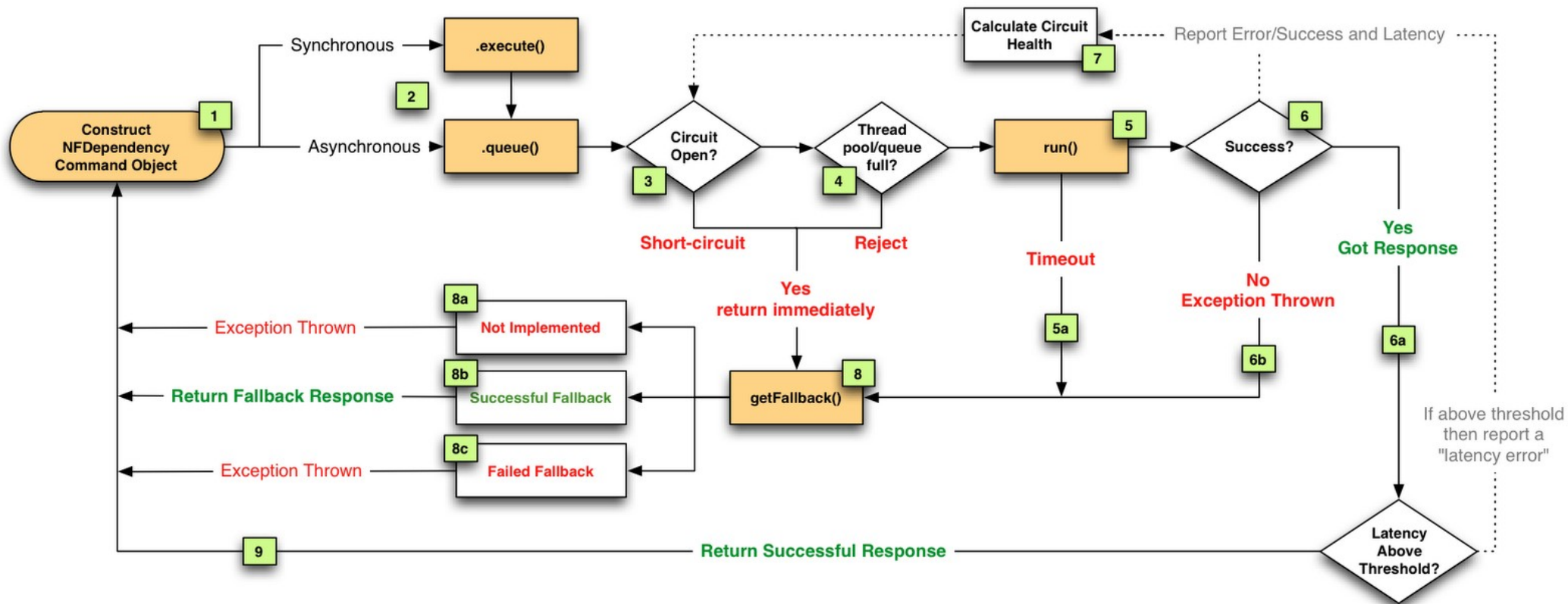
See: <http://upalc.com/google-amazon.php>

# Tolerance of Failure

- Even without cascading failure, if each component is 95% reliable, a 10-component system is 60% reliable.
- We must handle failures in upstream systems.



# Handling Failure (Netflix)

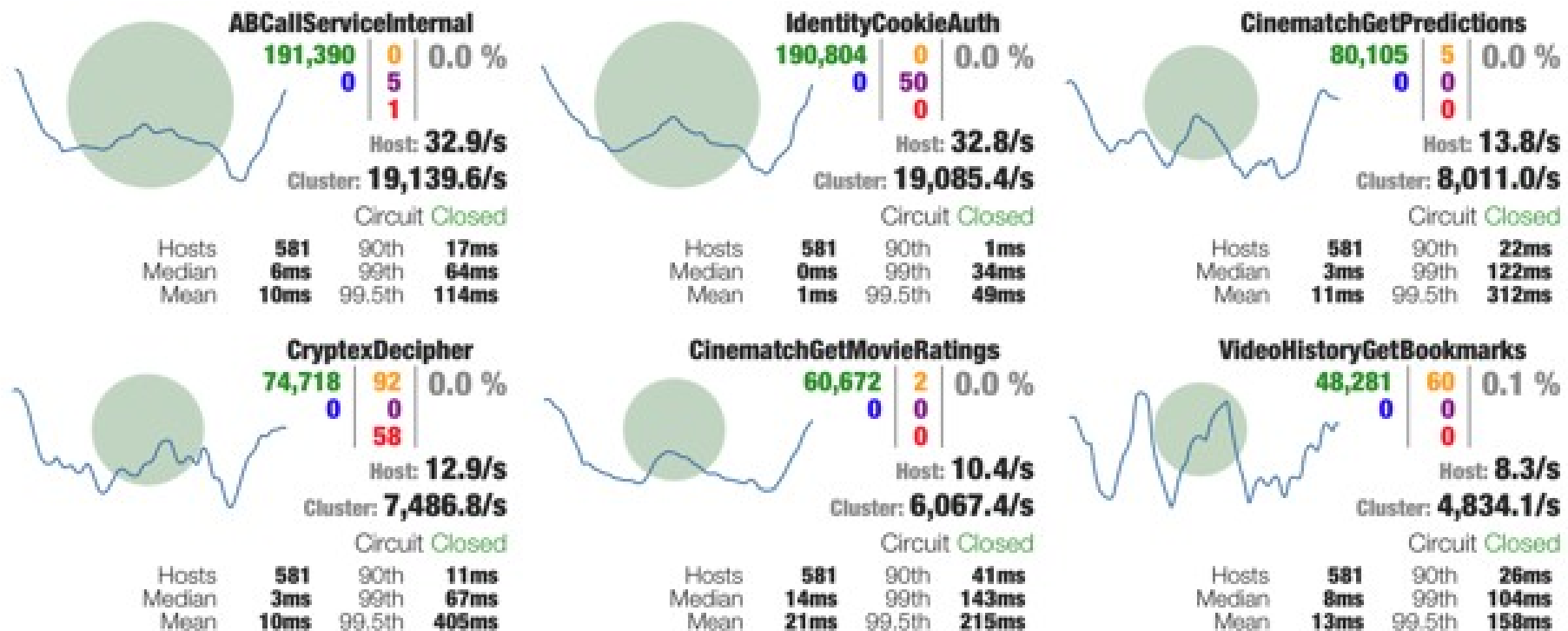


# Aside: Steve Yegge's Google Rant

- Every single one of your peer teams suddenly becomes a potential DOS attacker.
- Monitoring and QA are the same thing: [...] It may well be the case that the only thing still functioning in the server is the little component that knows how to say "I'm fine [...]" in a cheery droid voice.
- A ticket might bounce through 20 service calls before the real owner is identified.
- Debugging problems with someone else's code gets a LOT harder.

See: <http://upalc.com/google-amazon.php>

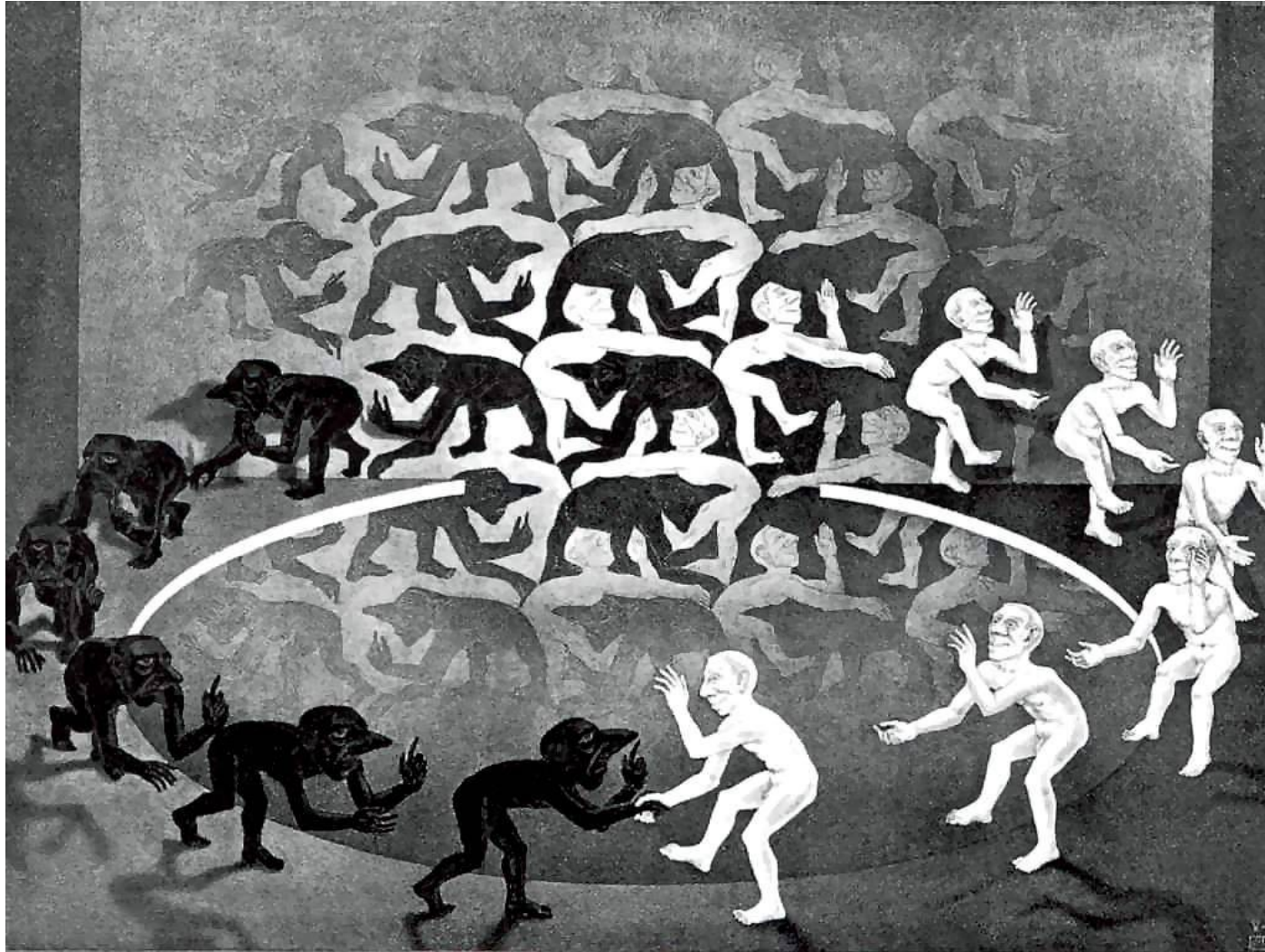
# Monitoring Failure (Netflix)



- Hystrix tells you when it broke.
- Zipkin tells you where and why it broke.

See: <http://techblog.netflix.com/2012/11/hystrix.html>

# Fabric Services



Where your software meets the cloud.

# Fabric Services

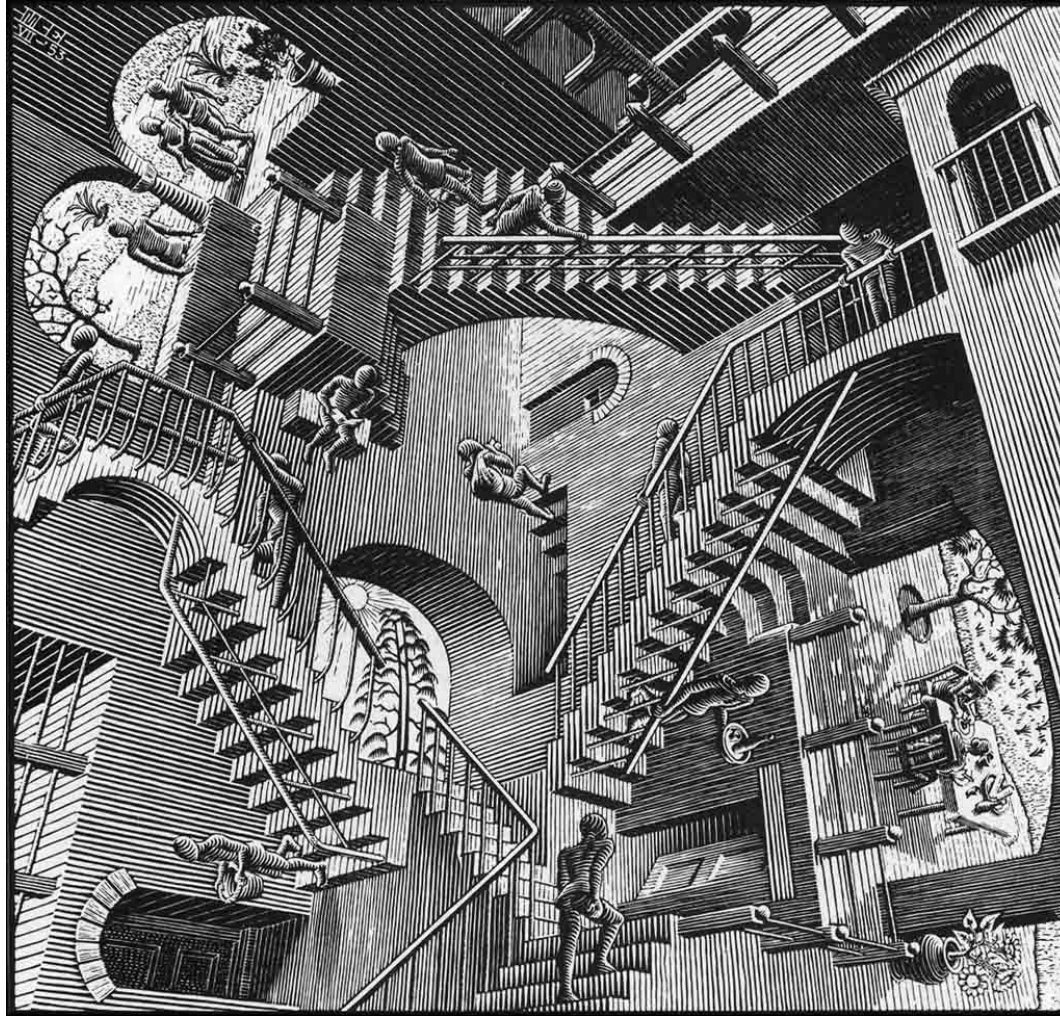
- Storage (Object/DHT)
- Compute
- Network
- Queue
- Service discovery and registration
- Load balancing
- DNS, autoscaling, management, ...

# Using Fabric Services

- The best cloud architects take a set of fabric services and build an application out of them.
- The best product designers create fabric services such that applications can be built out of them.
- It's like an algorithms book, but with different elements.



# Implementations of Cloud



Looks like one of *my* implementations.



# Why Buy Cloud?

- Not Invented Here?
  - If you're going to scale out, you have to build a cloud anyway.
  - A lot of companies did just that before Amazon made it a public commodity.
  - Most of python is just reinventing Java.
    - But python has not yet reinvented most of Java.
    - Give it another 20 years...

# Implementations: Amazon

- Started as a dog-food system.
- Very rich set of fabric services.
- Data import/export is a challenge.
- Probably crossed the overload threshold.
- Expensive.

# Implementations: Google

- Primarily a PaaS offering.
- Presumably also based on dog-food.
- Allows Google greater efficiency in resource management.
  - Comes out in application cost comparisons, but we haven't seen many of those.

# Implementations: Azure

- Azure is a mixture of IaaS, PaaS, SaaS.
- Imagine the customer is an application builder.
  - Amazon sells IaaS with optional PaaS services.
  - Google sells PaaS services with optional IaaS.
  - Azure managed to create a confusion.

# Implementations: Red Hat

- Download and build your own.
- Based on open source components.
- Mostly not very mature.

# Implementations: Nebula

- Delivered on a truck.
- Plug in, turn on.

I will now sing the company song...

# Other Companies to Watch

- Experts at using the cloud!
  - Google
  - Netflix
  - Twitter
  - LinkedIn
  - Facebook
  - Yahoo
- All have papers or publications.

# Conclusions

- I just came to inspire a discussion.
  - The conclusions aren't canned.
  - Please argue with each other / me now.