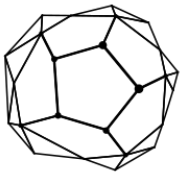


nebula

Hidden in the Clouds

New Ideas in Cloud Computing

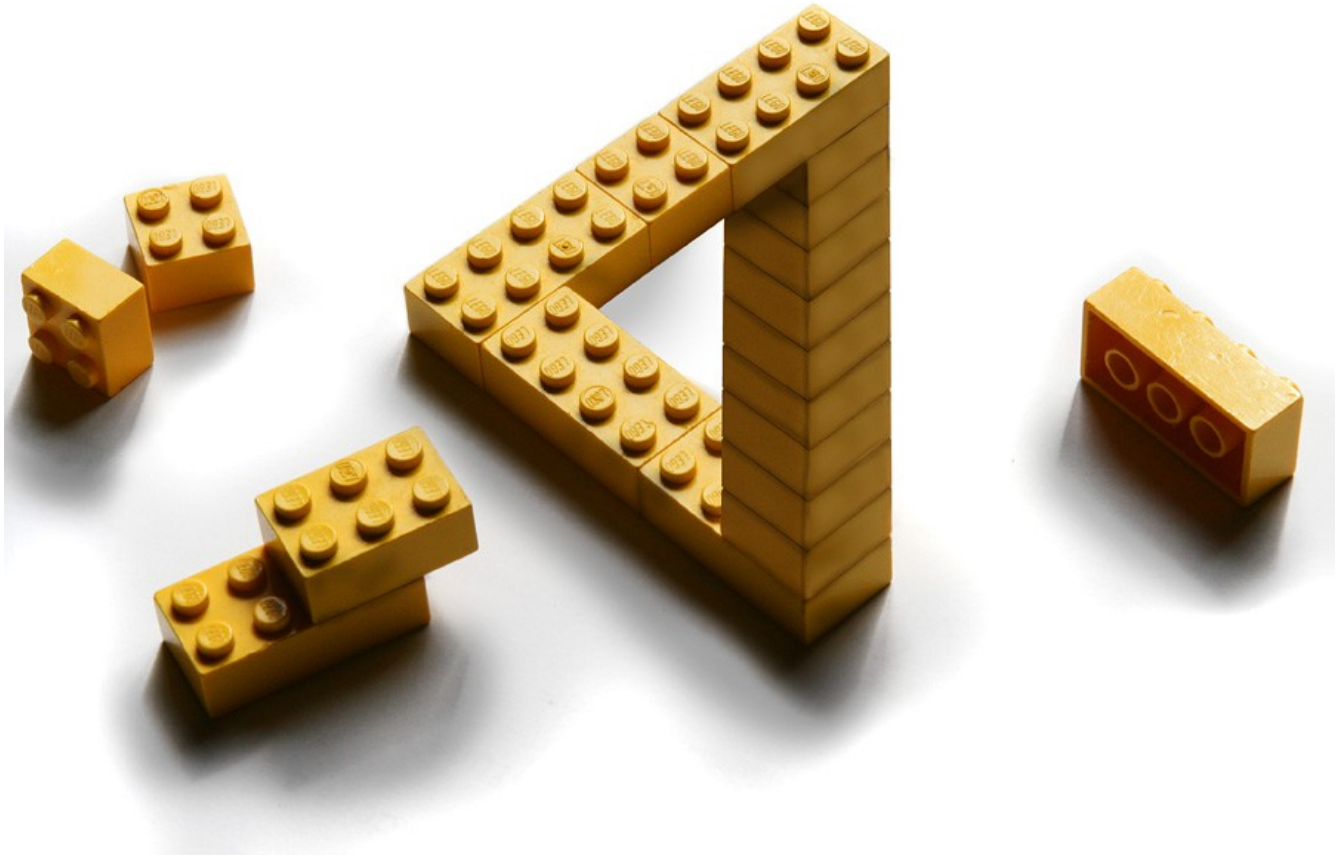
Shevek <shevek@nebula.com>

Introduction

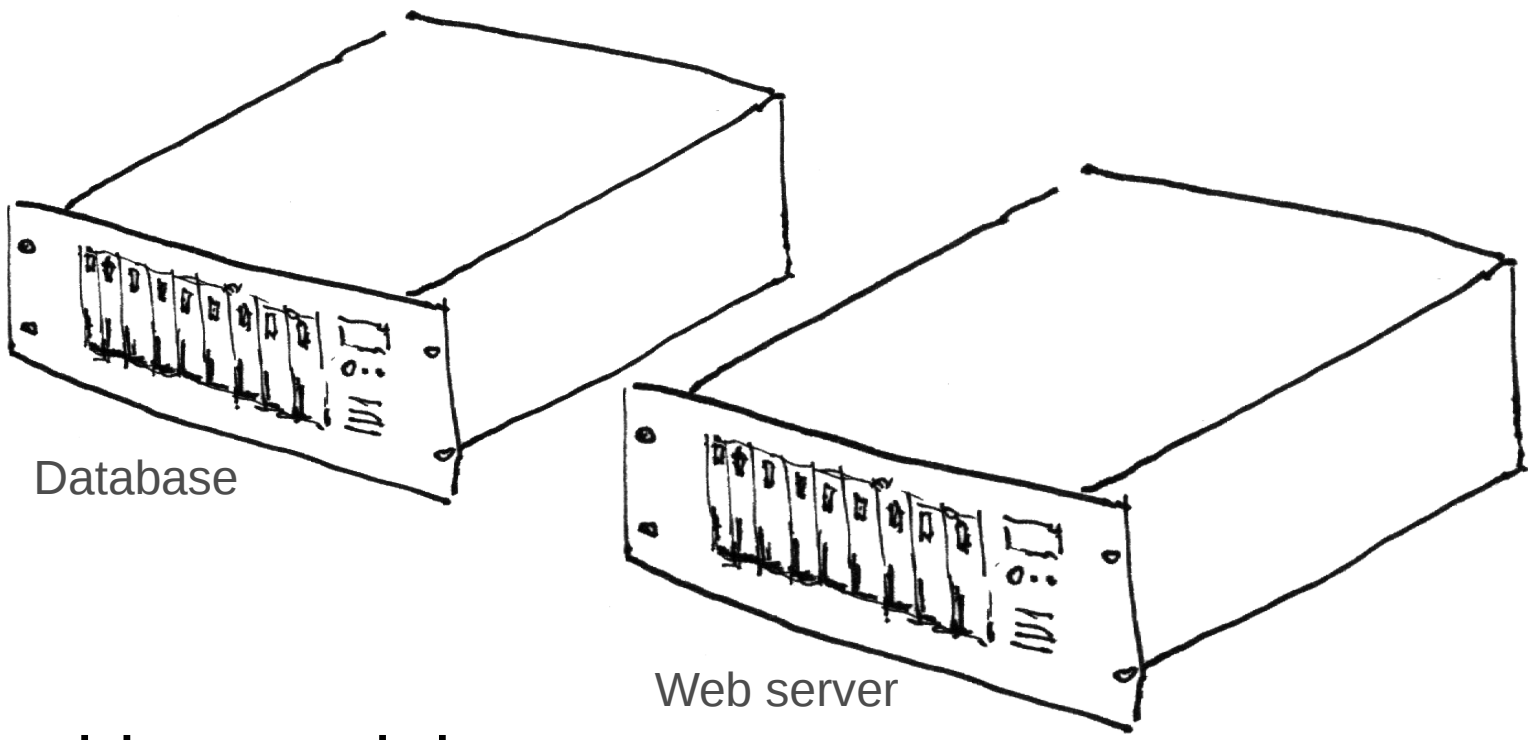
- The conceptual foundation of the cloud.
- Some interesting corollaries.
- Consequences for the business.
- How to be successful in the cloud.
- Examples and implementations.

Bullet points *are* punctuation!

Building the Cloud

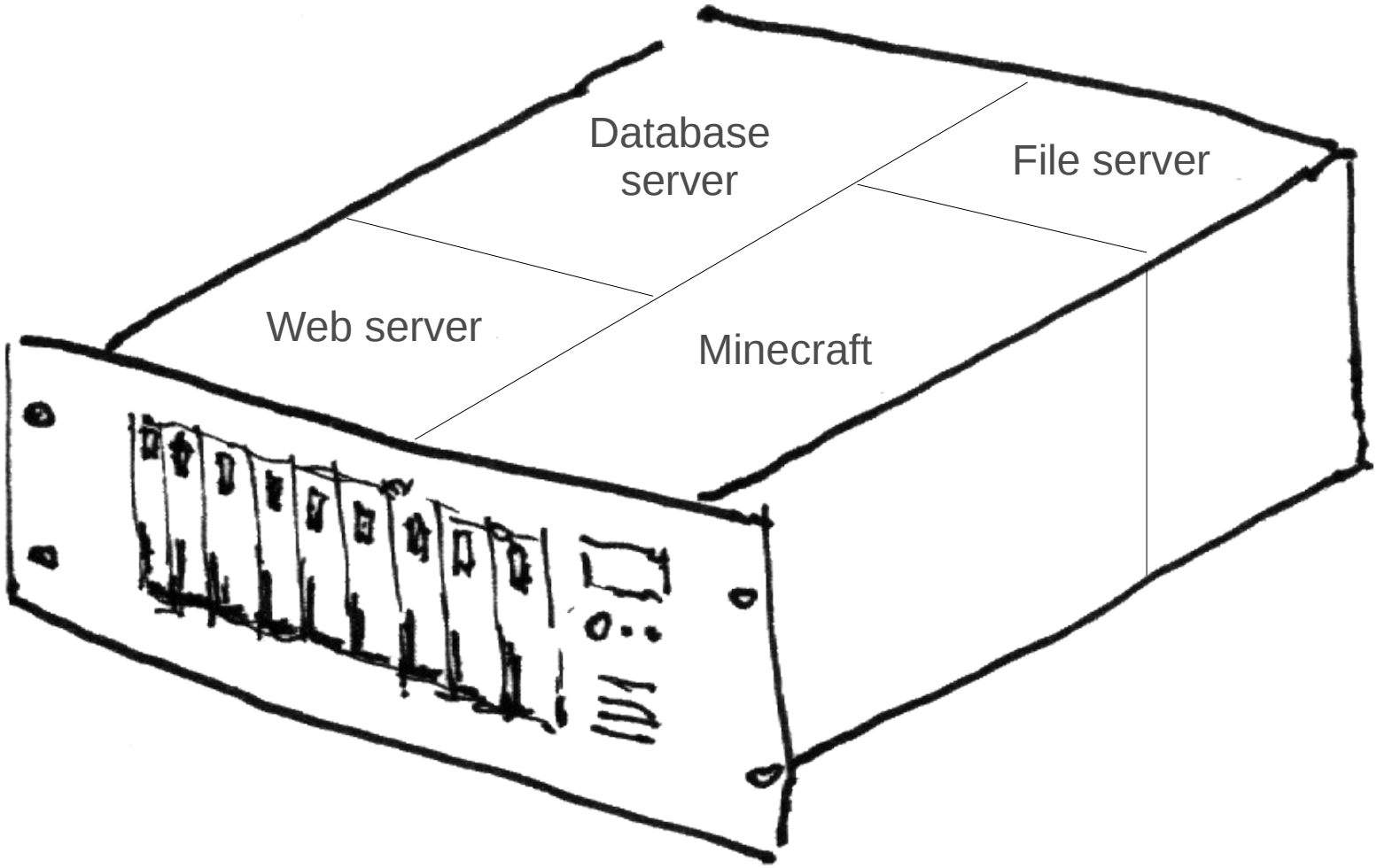


Servers



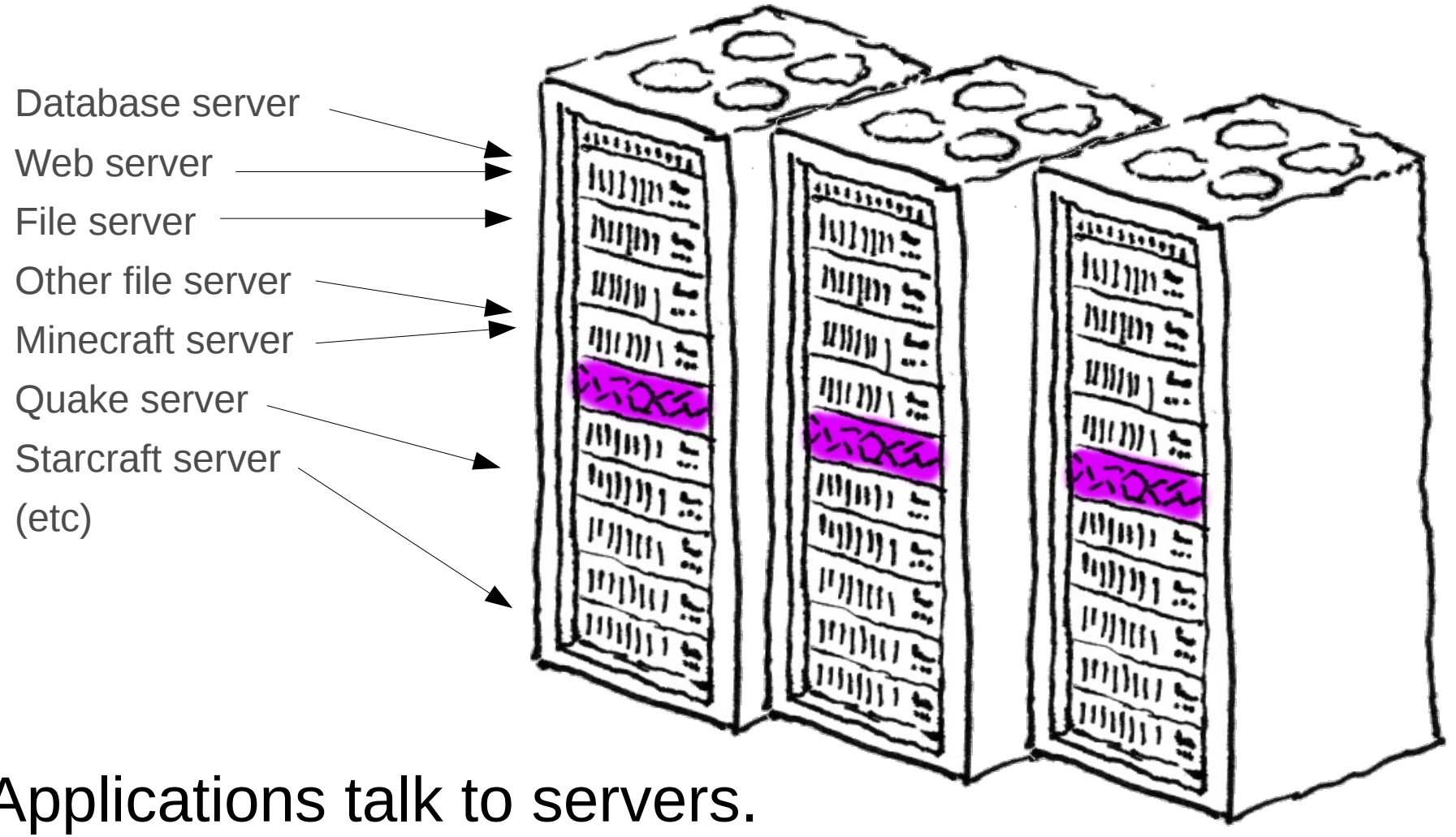
- One machine per job.
- Manual management.
- Adding a job requires buying a server.

Virtualization



- One server does many jobs.
- Still manual management.

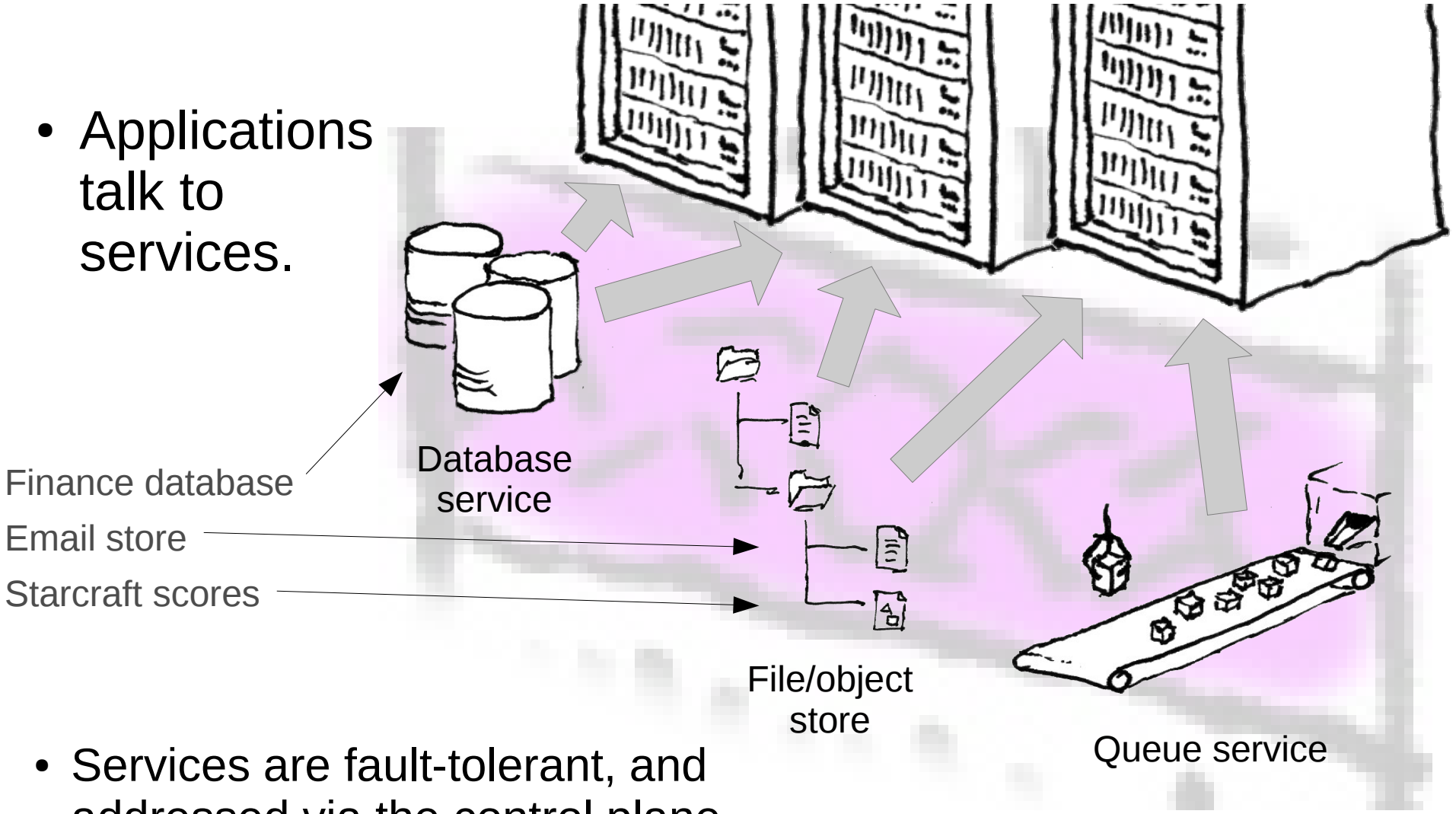
Infrastructure as a Service



- Applications talk to servers.
- Management is an automated control plane.

Platform as a Service

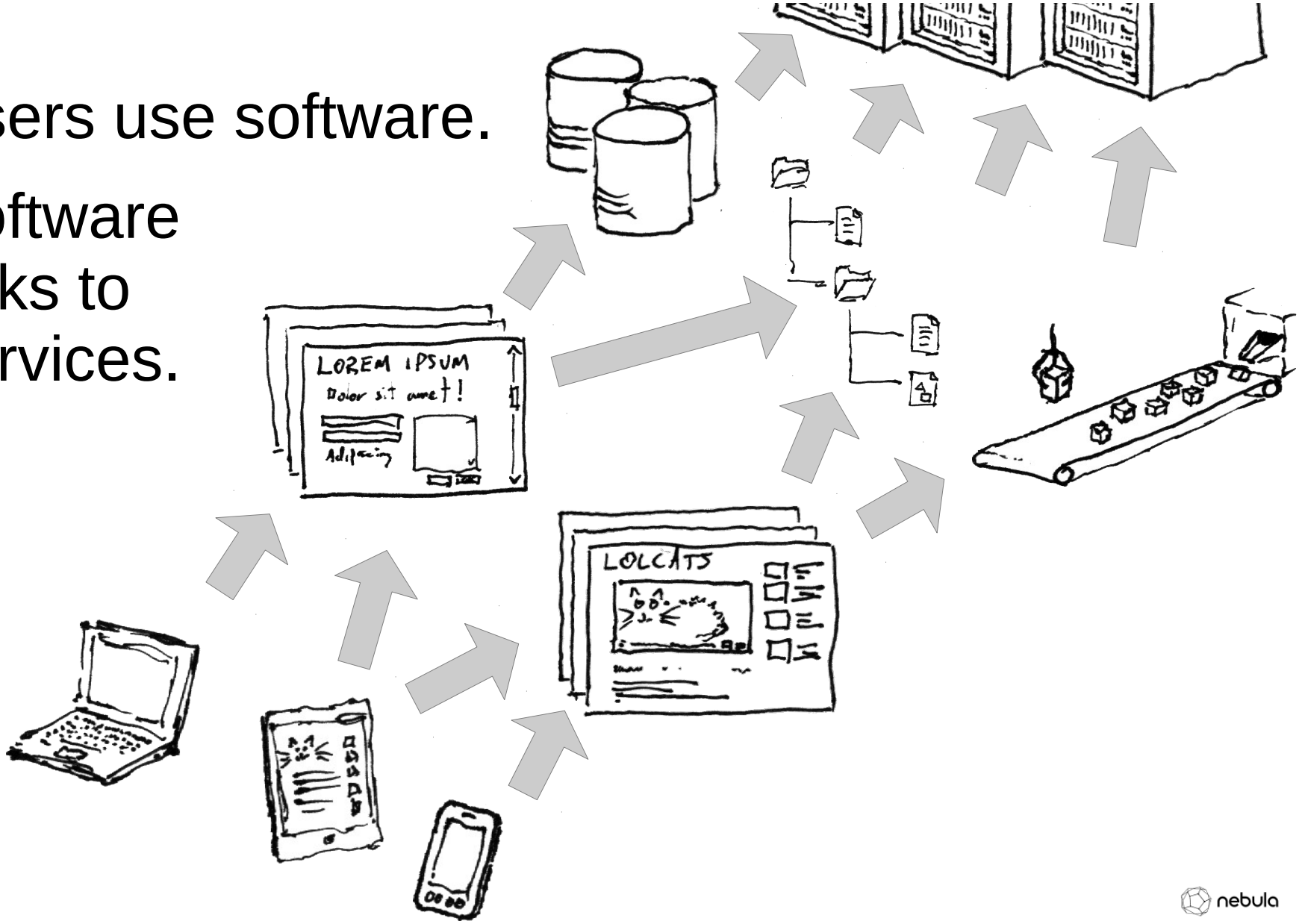
- Applications talk to services.



- Services are fault-tolerant, and addressed via the control plane.
- The control plane hides the mapping to hardware.

Software as a Service

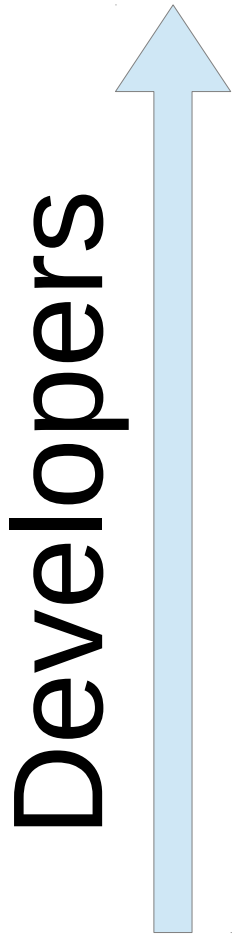
- Users use software.
- Software talks to services.



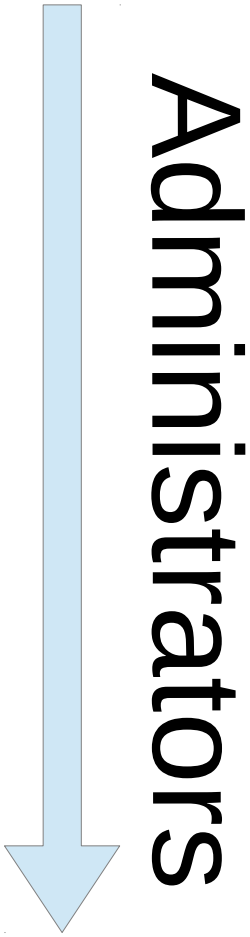
Aggregation and Disaggregation

- 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.

Why did XaaS Change Business?



- Uniformity of administration.
- Lower overheads for running the system.
  - Disaggregating a large object allows right-sized shards to be created.
  - Cost to provision scales with usage, not number of services.
- Cost scales with the success of the customer.
  - If the pricing model is good.
  - The cost of an account on a shared system is low.
- Lower barrier to entry for end users.
  - Caused by disaggregation.



Where does “Cloud” come from?

- What we now call “cloud” is the sweet spot of XaaS, with 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 avoiding the unavoidable failure rapidly exceeds the benefits.
- Solution: Don't try to avoid failures.

- Failures are easier to handle at the SaaS layer than at the PaaS layer.
  - The SaaS layer can decide how much of the failure to expose to the user.
- The resulting simplicity of the stack can create a more reliable service overall.

- The cloud isn't fundamentally less reliable than an individual system.
  - Perhaps we are just facing reality better.
  - We also allow sysadmins to manage hardware without notifying applications, by advertising a particular reliability contract of the application.

Look-ahead: Handling Failures

- Replicated/restartable computation.
  - (MapReduce, Google)
- Fallback behaviour on error.
  - (Hystrix, Netflix)
- Deliberate introduction of failures
  - (ChaosMonkey, Netflix)

We'll see more of these later.

What Do We Lose Over Scale-Up?

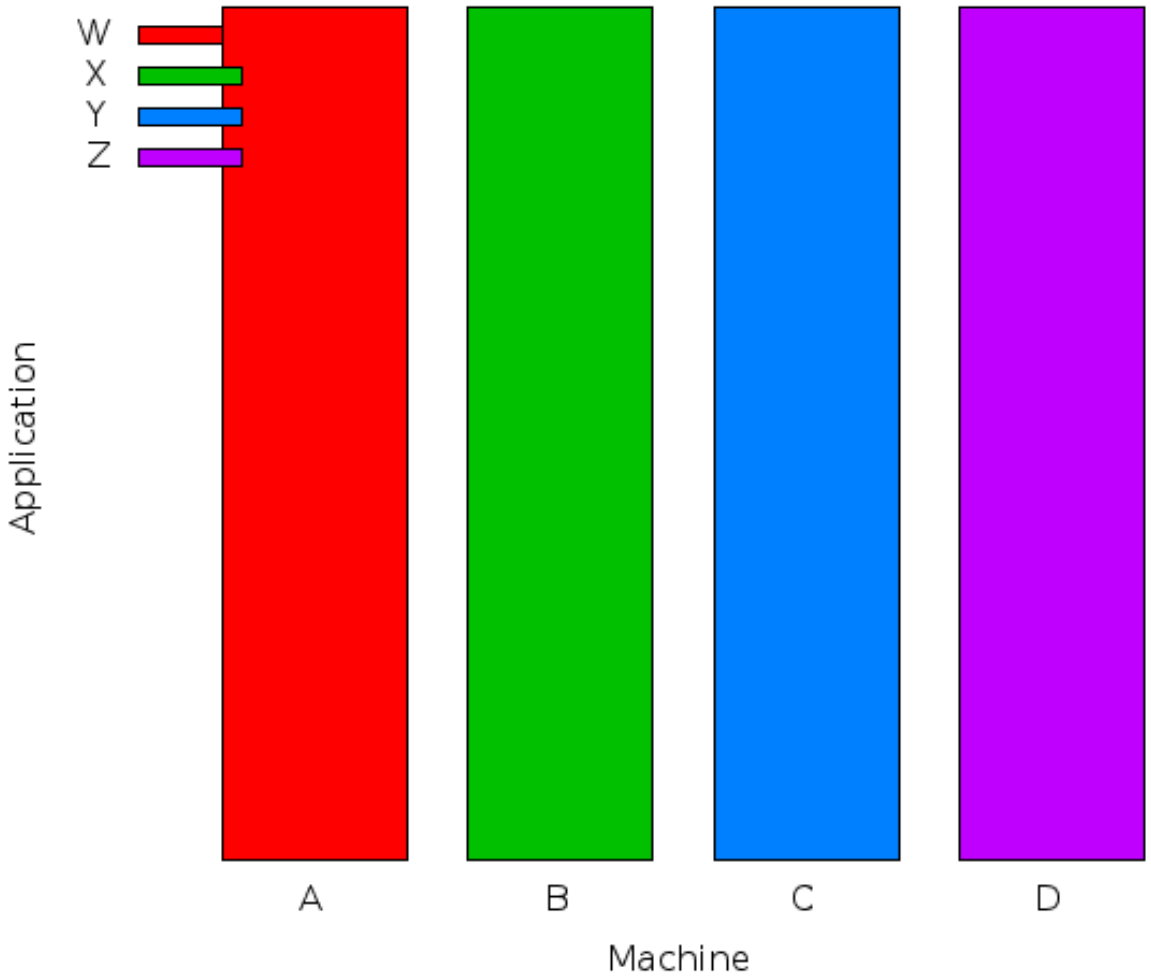
- Lazy algorithms with bad access patterns.
  - Many of these are bad in any case.
  - See Mechanical Sympathy.
- Shared-anything vs shared-nothing.
  - Do we need to go as far as shared-nothing?
  - Remote memory / RDMA.
  - MapReduce vs Bloom-Filter feedback.
- The same as NUMA, but more so.
  - Distances 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.

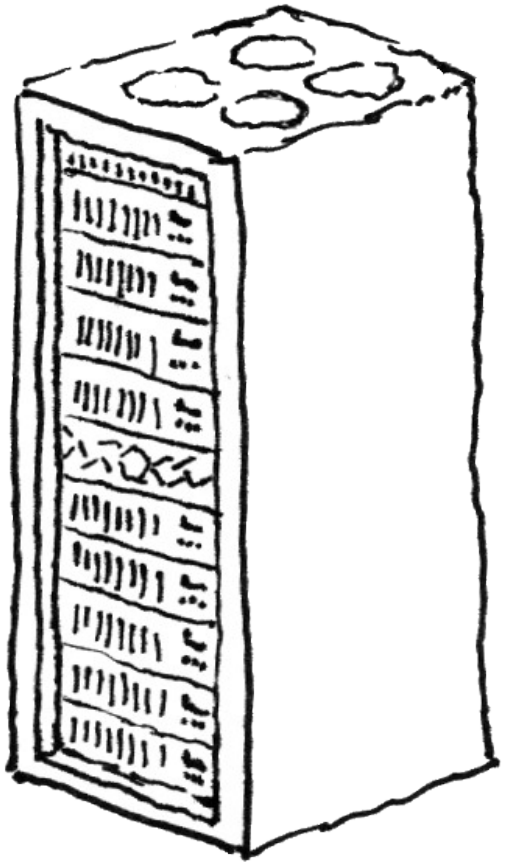
What Do We Lose Over …?

- Loss of explicit control vs automatic recovery.
  - Explicit management of a single system.
- Loss of fixed location vs dynamic allocation.
  - Ease of debugging vs overall reliability.
- Loss of permanence vs uniform management.
  - No system is allowed to be a unicorn.

But what else do we gain?

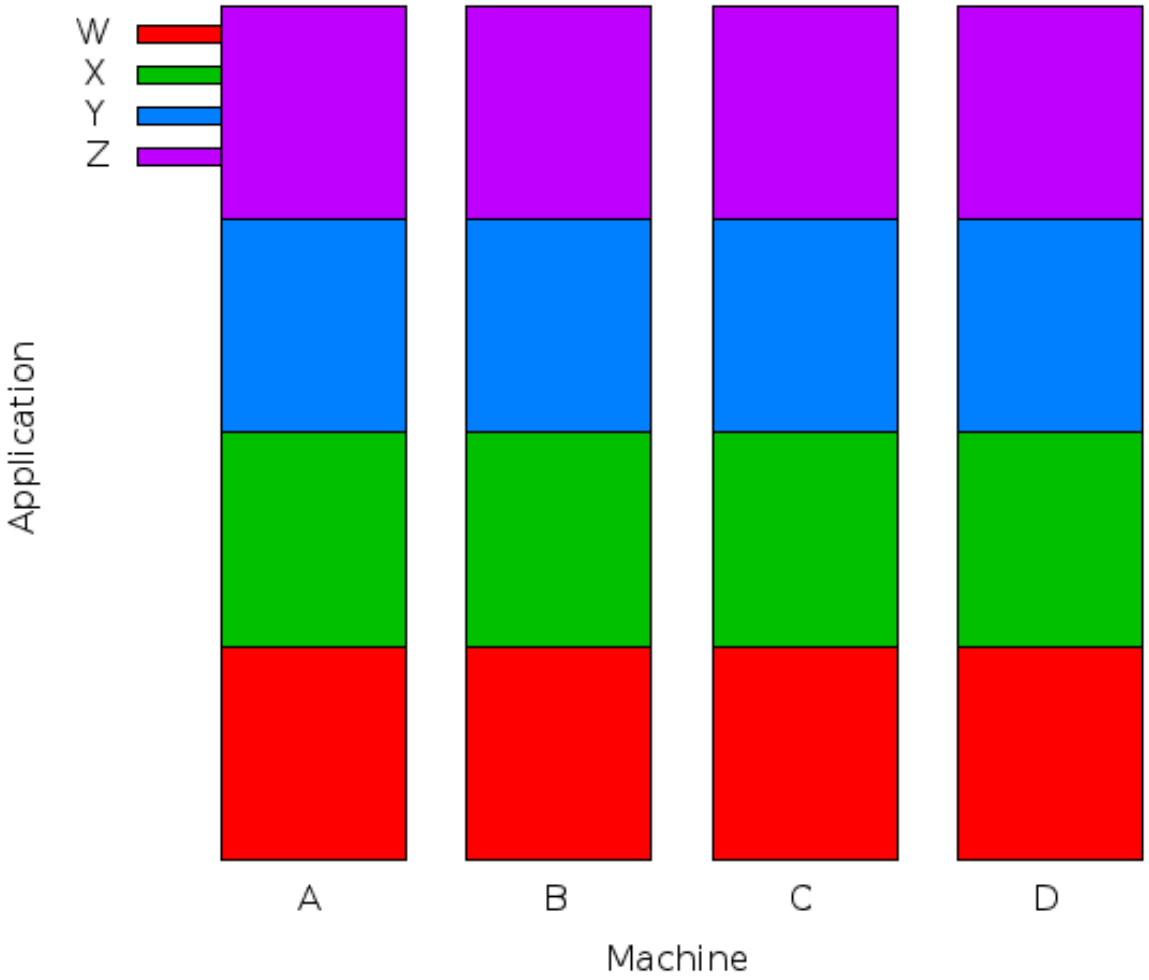
Consequences of Virtualization

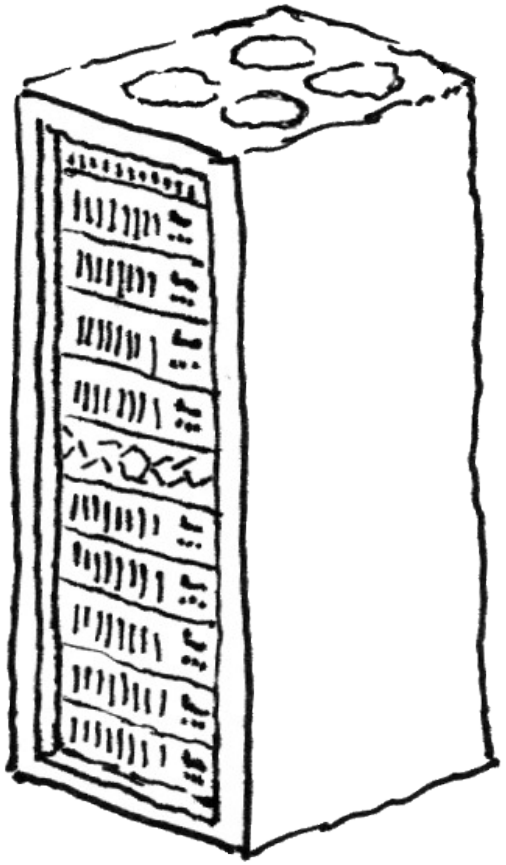




… and a machine fails.

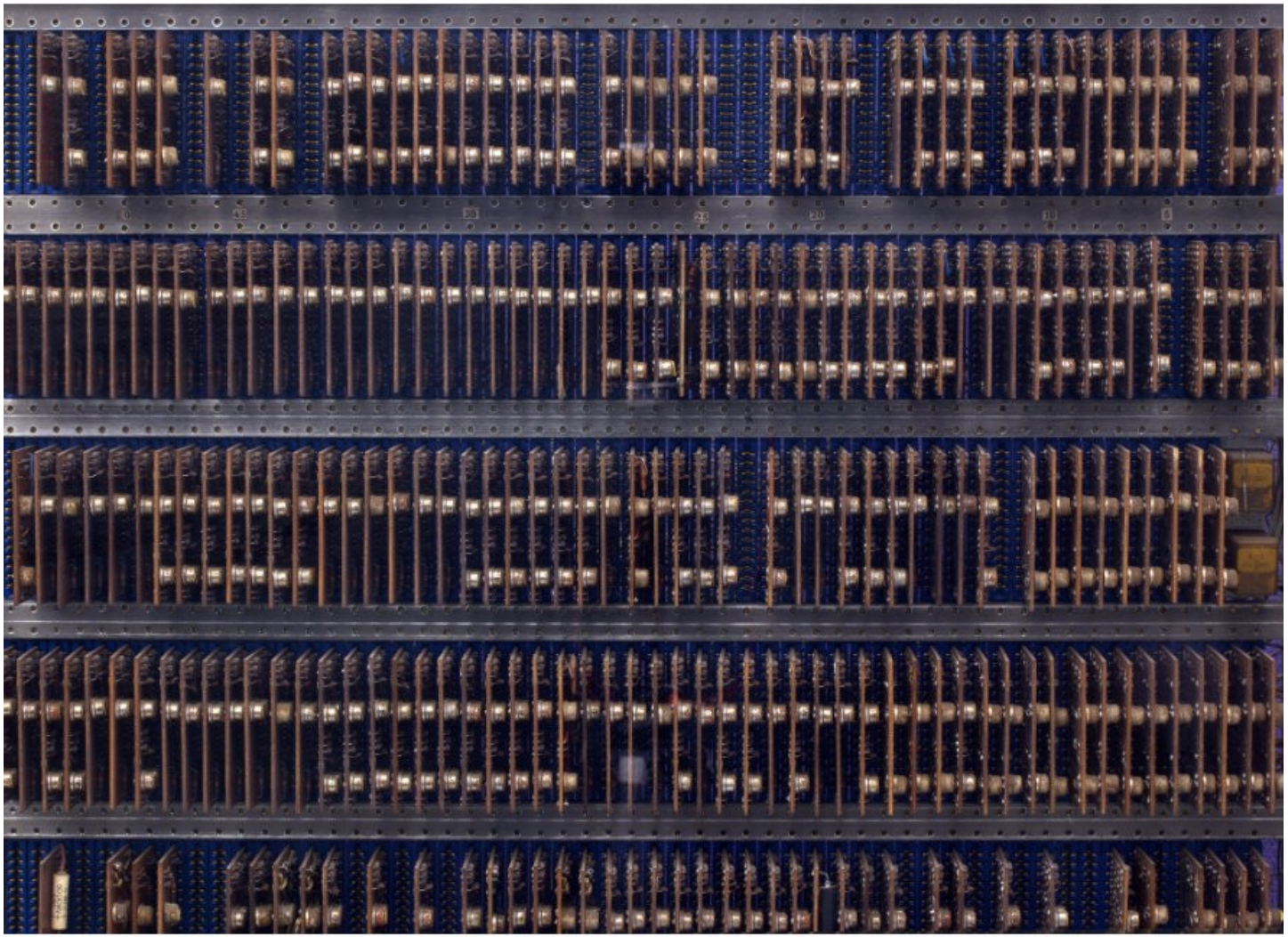
Consequences of Virtualization





Now we have only two cases!

Cloud Isn't New!



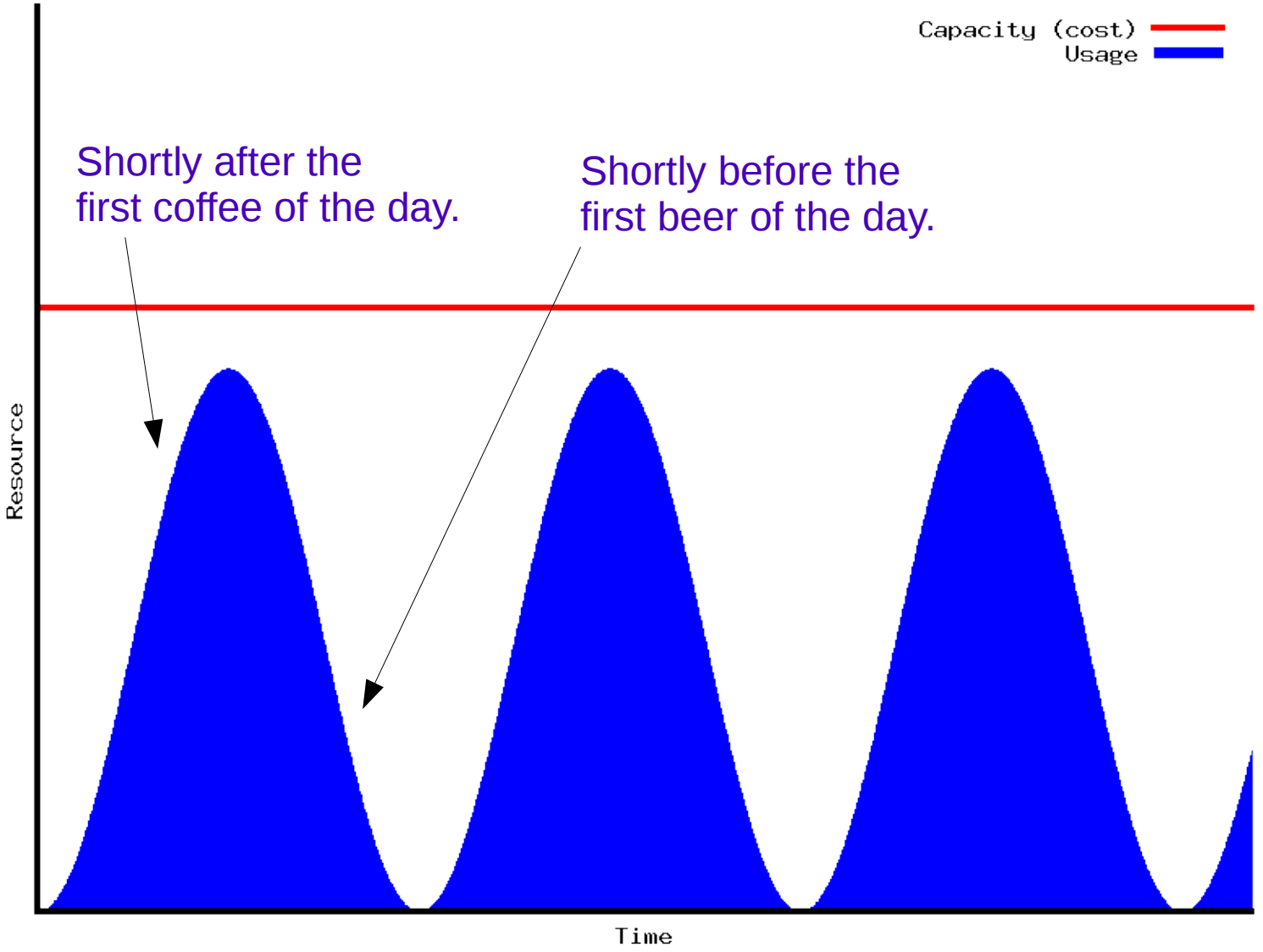
“Little Character”, Control Data / Seymour Cray

Why Buy Cloud?

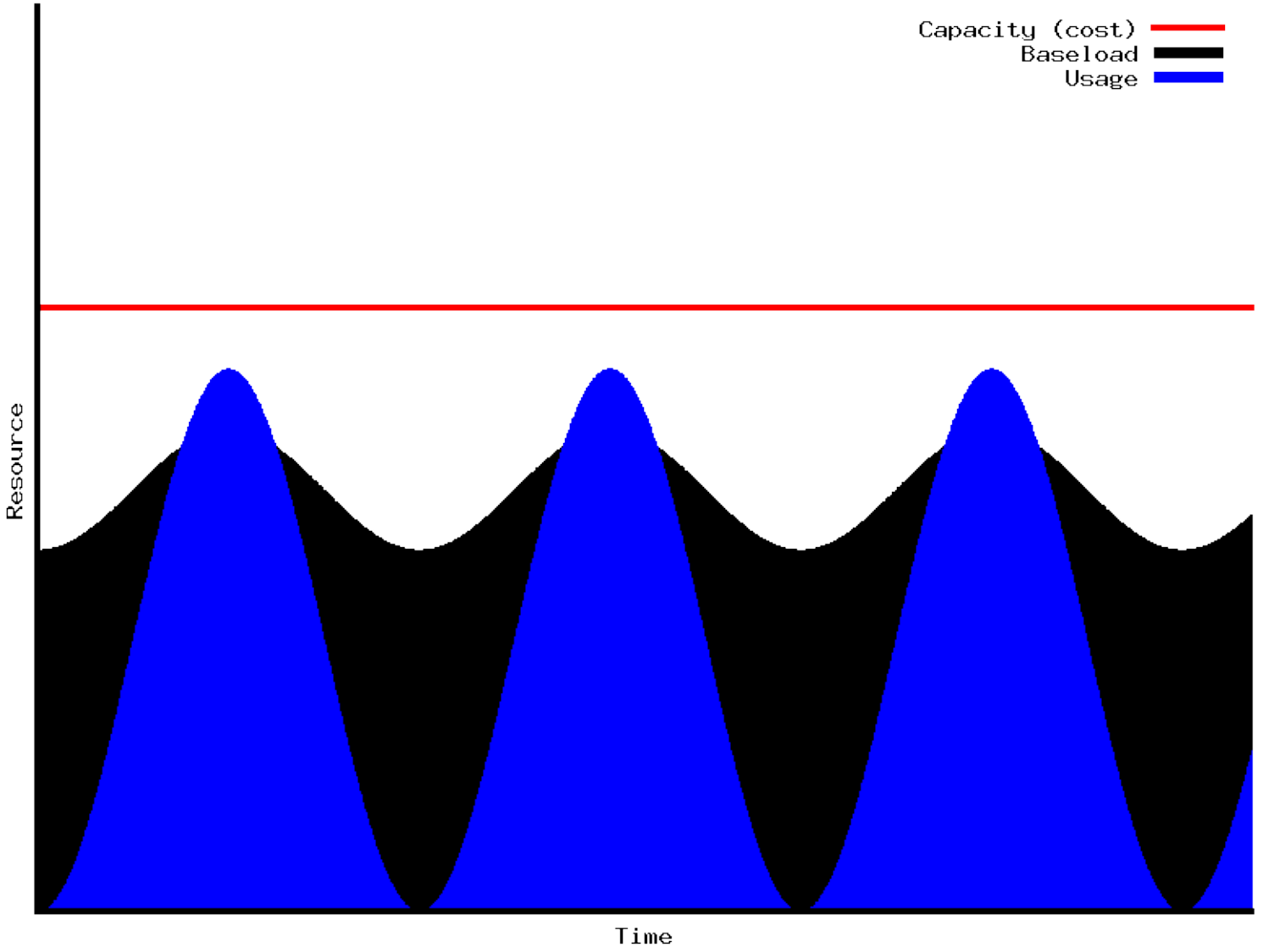
- Not Invented Here?
  - If you're going to scale out, you have to build a cloud anyway.
  - A lot of companies did just that before Amazon made it a public commodity.
  - Most of python is just reinventing Java.
  - But python has not yet reinvented most of Java.

Economics of the Cloud

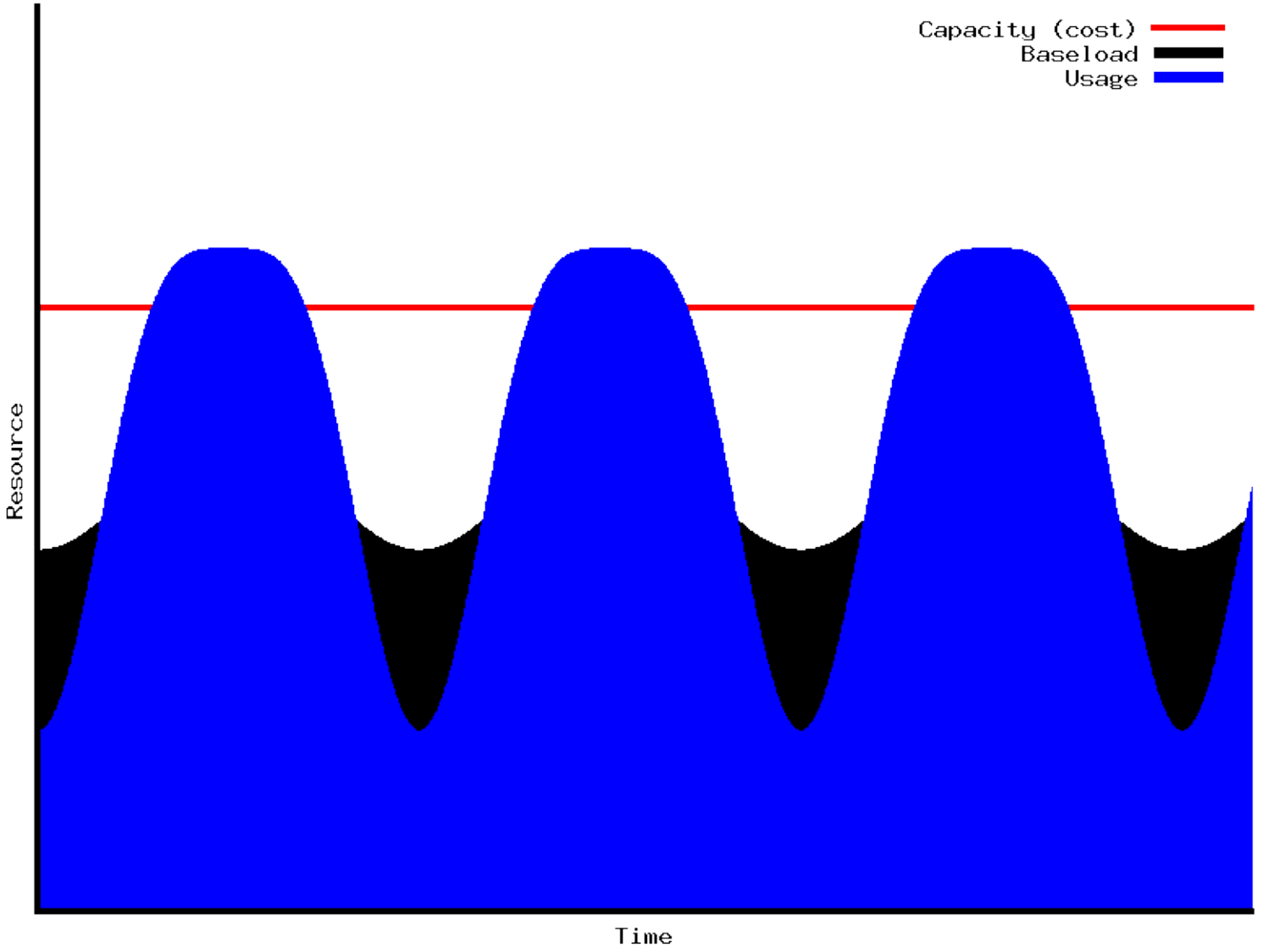
Basic Usage Pattern



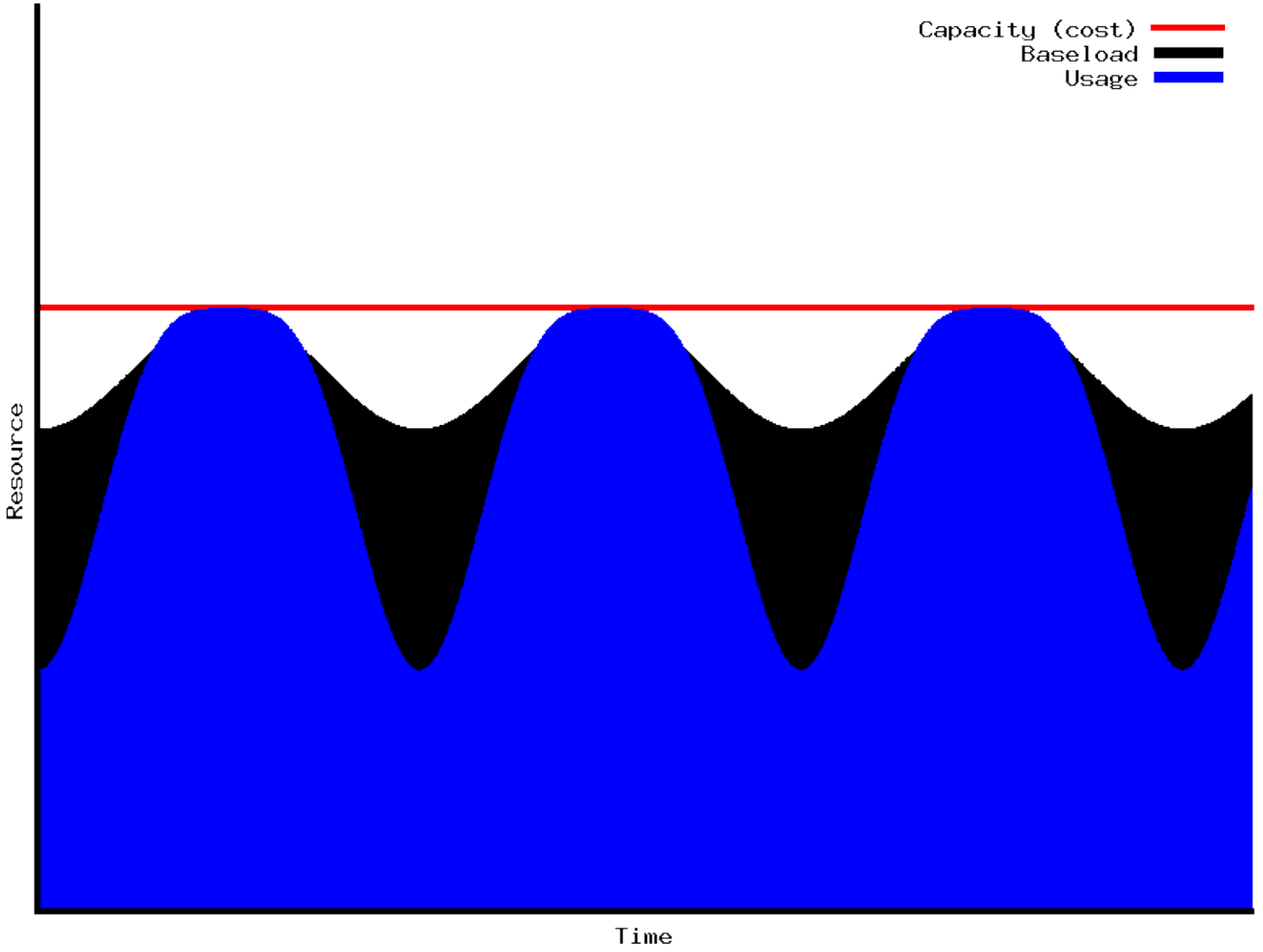
Baseload: The Movable Work



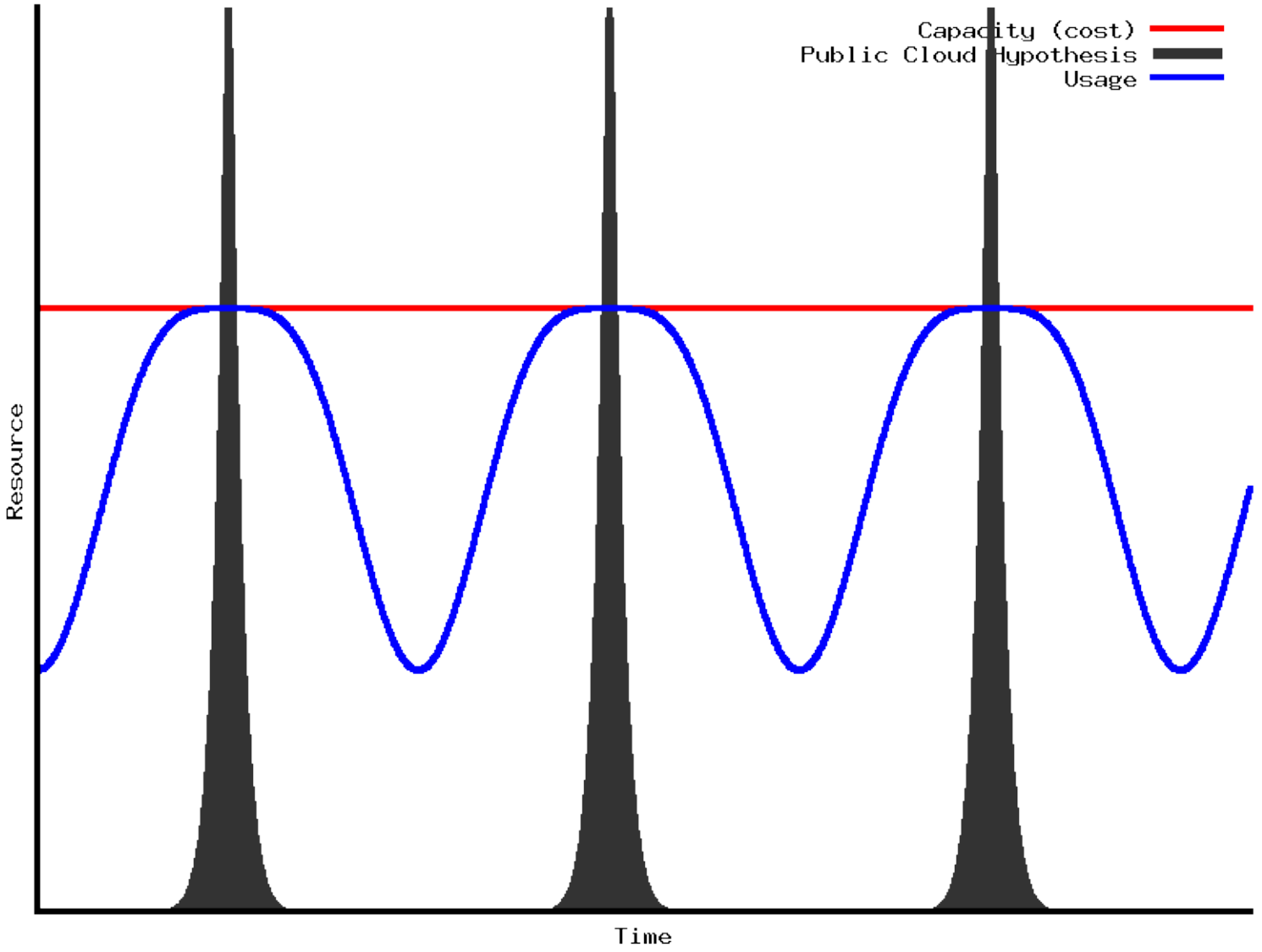
Managing Overload



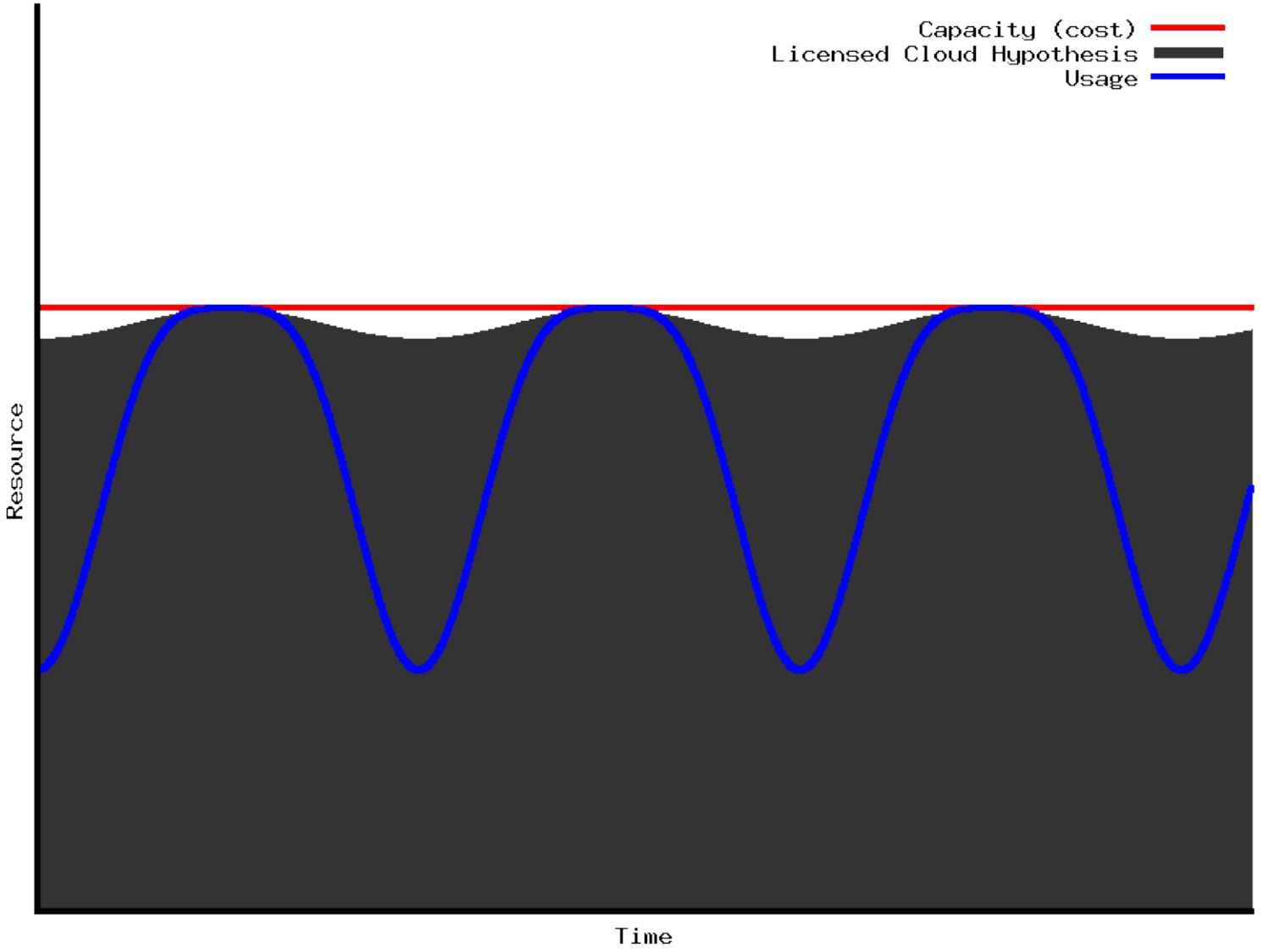
The Ideal Usage Pattern



Dubious Claims

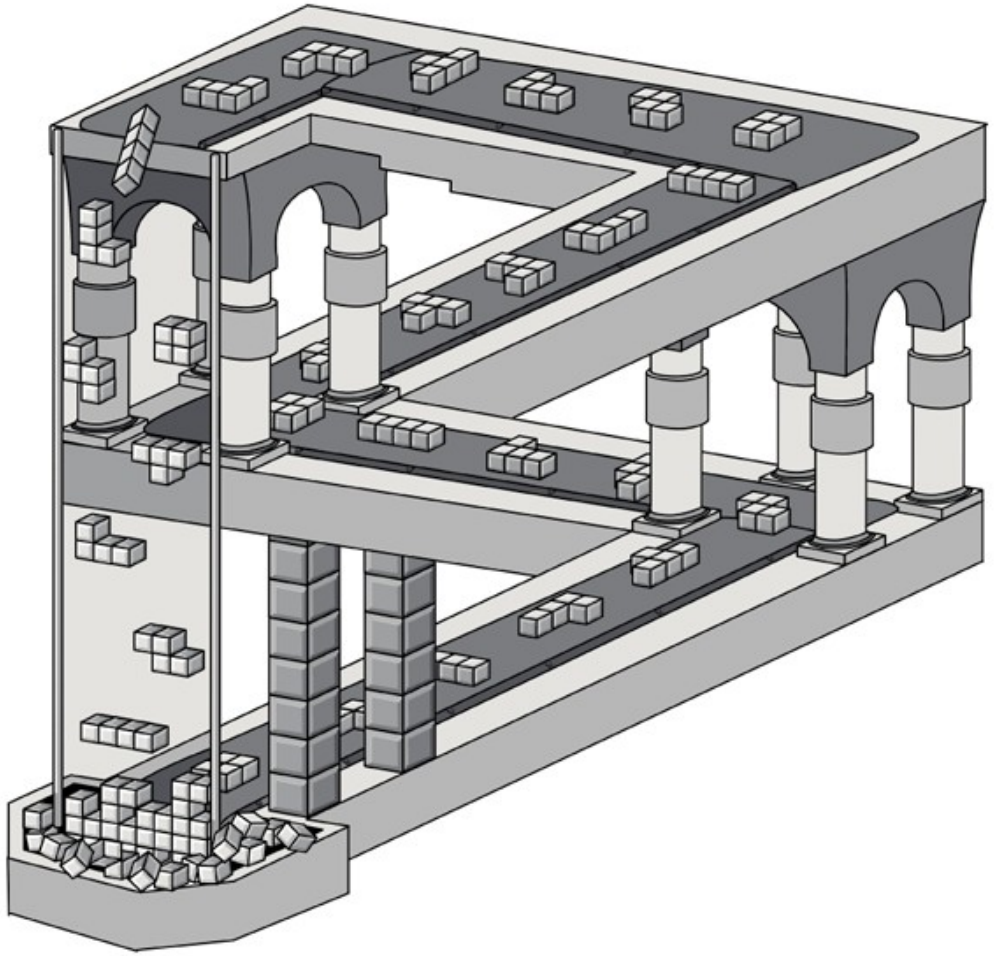


More Dubious Claims



nebula

Using the Cloud



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

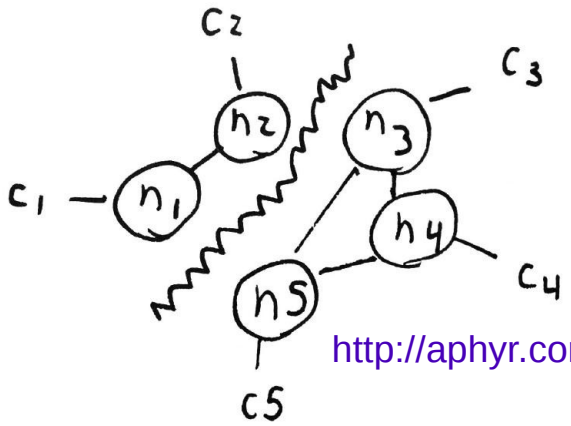
Moving to the Cloud

- If your application or subdivides trivially:
  - Cloud gives you management.
  - You still need to deal with failures.
  - You had to do this anyway.
- If your application does not subdivide trivially:
  - You should still scale out, not scale up.
  - How you behave when you reach this threshold will determine your success.
  - Getting this wrong is painful and costly.
  - Some SaaS companies are just hanging on, using larger instances. Amazon will keep taking their money. They also own private HPC machines.
- So how do we scale out?

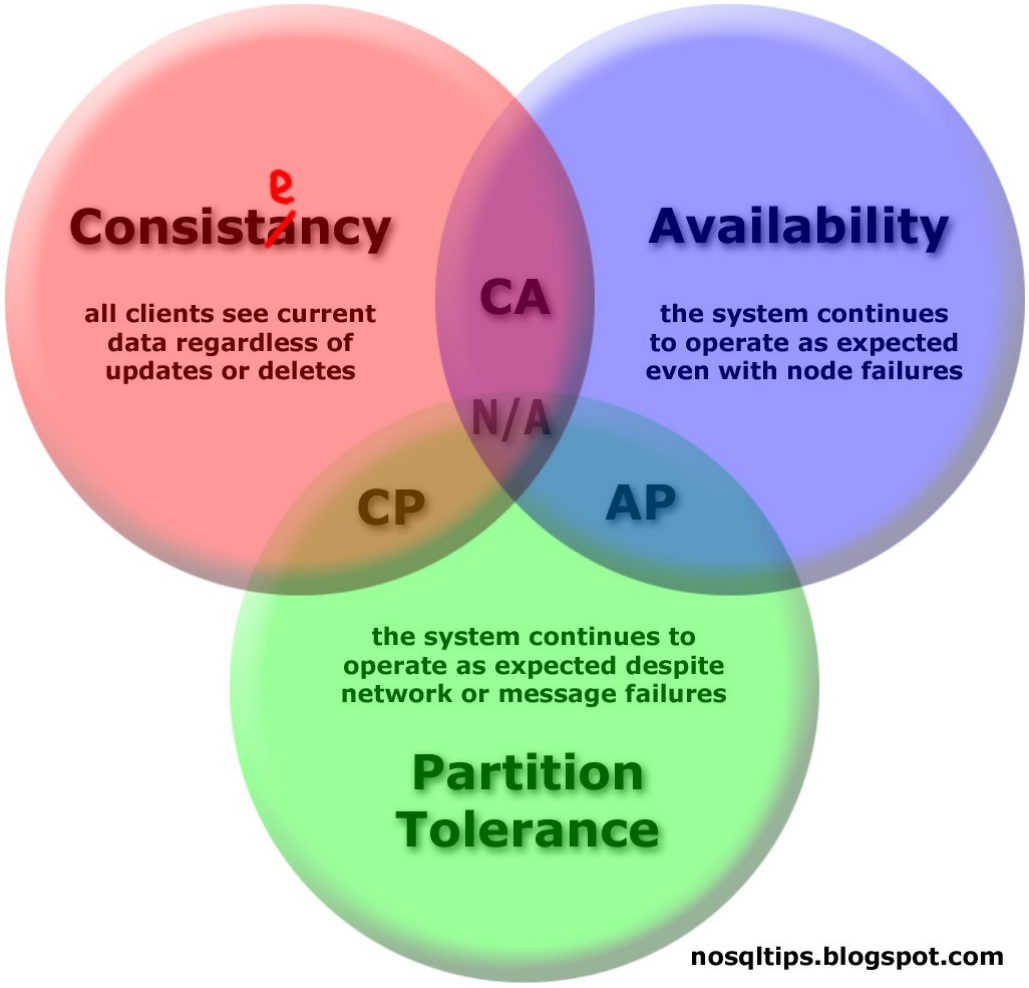
Brewer's CAP Theorem

Any distributed system must either

- fail, or
- give the wrong answer.



http://aphyr.com/



Yes, I fixed the tyop in the image.

Moving to the Cloud: HowTo

- 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.

I can't believe I even tried to write a slide like this.

Attributes of Cloudy Applications

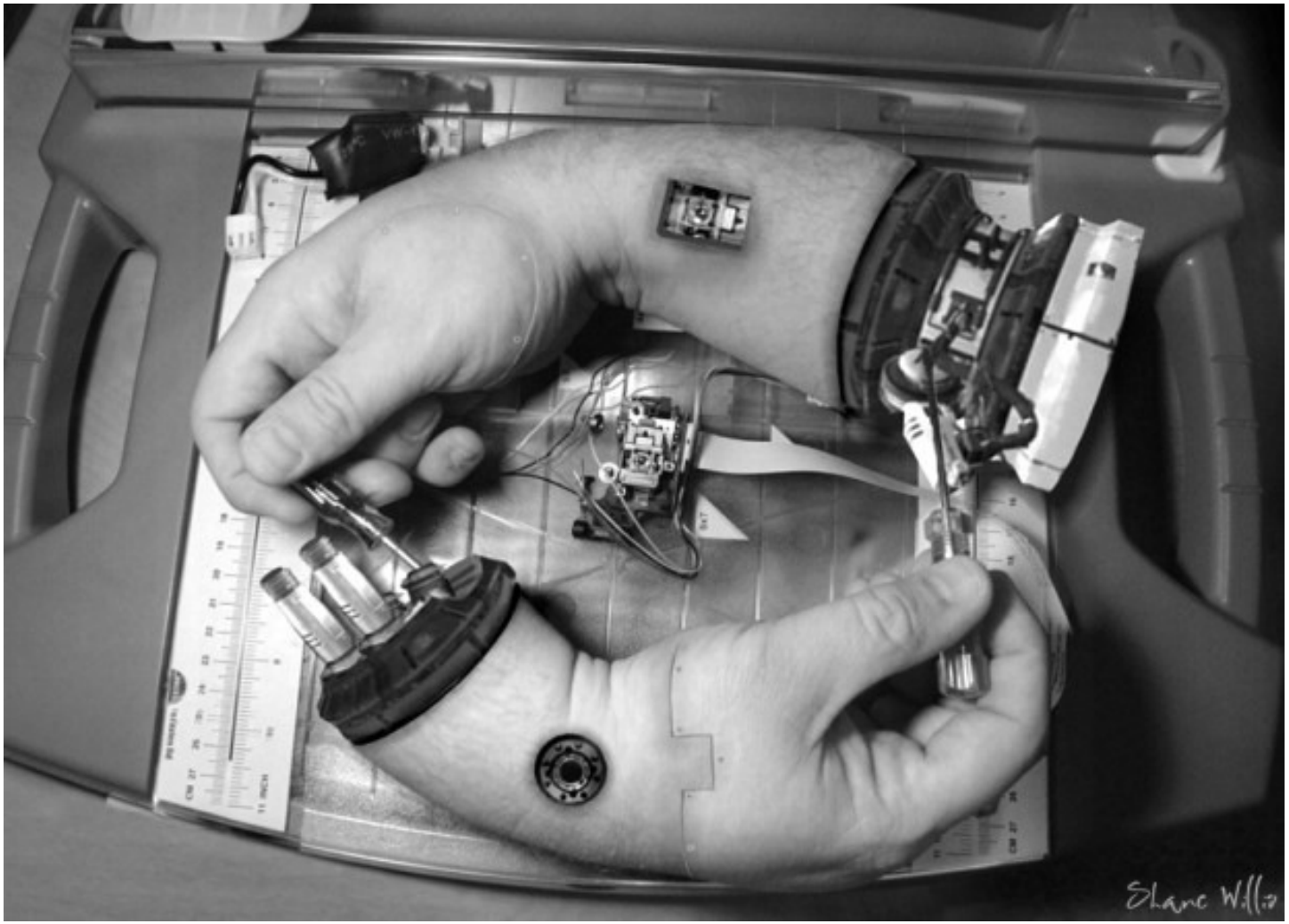
- Of systems:
  - Stateless components
  - Failure tolerance, failover, circuit-breakers
  - Replication
  - Independent, loosely coupled components
- Of processes:
  - Independence of datasets
  - Repeatability

Consequences of Cloudiness

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

Let's talk an example, while we're here.

Implementations and Examples



Implementations of Algorithms

- Cassandra
- ZooKeeper
- MapReduce/Hadoop
- Hystrix
- ChaosMonkey
- JGroups
- Counterexamples: MPI, MySQL, Mosix, DRBD

Implementations of Algorithms

- Cassandra
- ZooKeeper
- MapReduce/Hadoop
- Dapper
- Hystrix
- Scribe
- ChaosMonkey
- JGroups
- Counterexamples: MPI, MySQL, Mosix, DRBD

Implementations: Cassandra

- 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, ...

Implementations: ZooKeeper

- 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

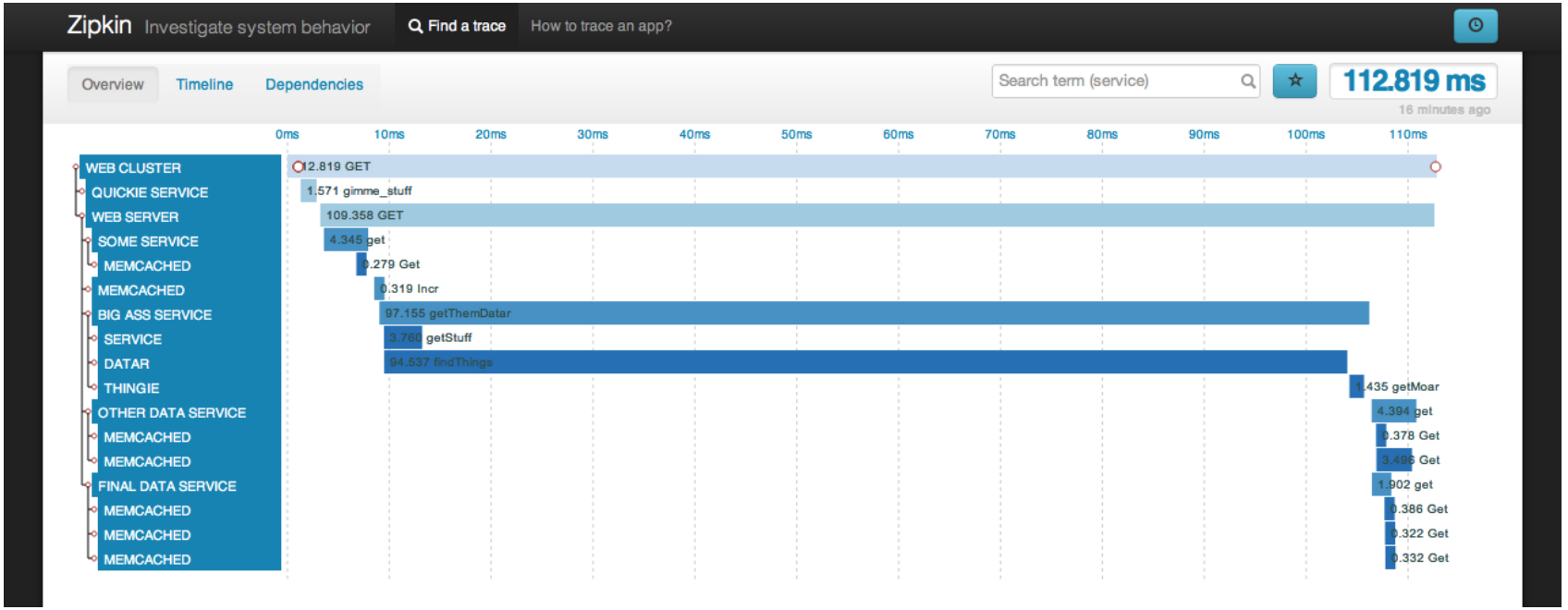
- HDFS:
  - Distributed filesystem, reasonably robust.
  - Very restricted API.
- MapReduce:
  - Restartable and repeatable computation.
- Hive:
  - A MapReduce-based SQL engine.
- HBase:
  - A distributed hash table.
- Other projects:
  - Varying levels of maturity and reliability.

Implementations of Tools

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

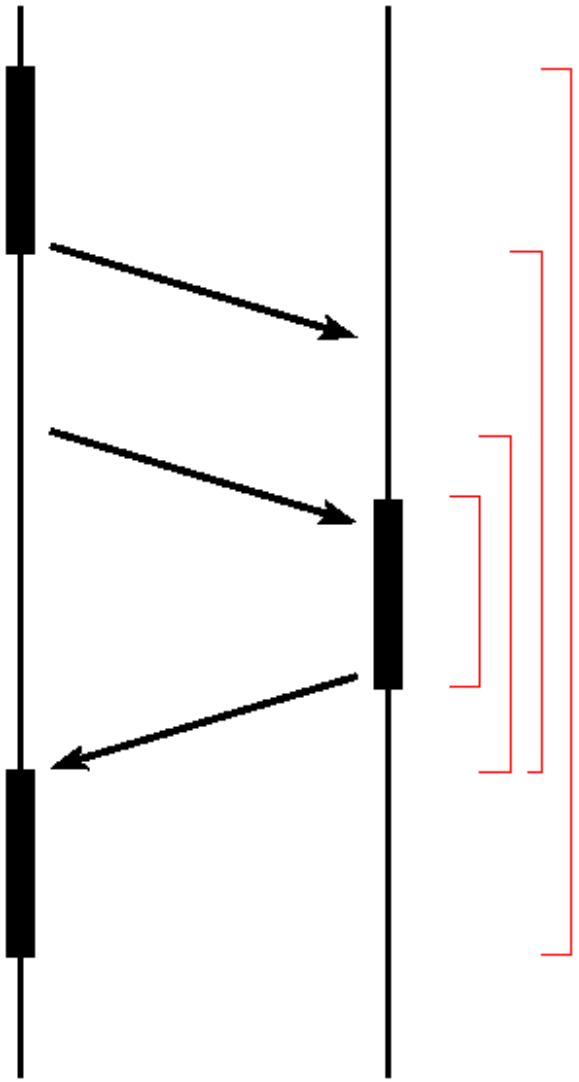
Tools: Zipkin (Twitter)

- 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

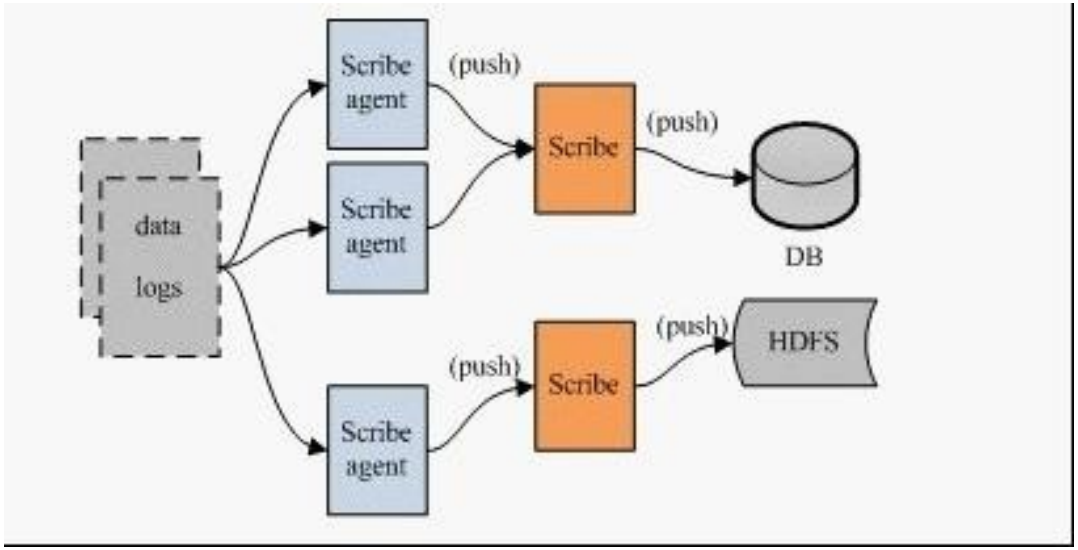


- In a large system, overheads matter.
- We must account for:
  - Setup time.
  - Failed calls.
  - Network delay.
  - Tear-down time.

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

Tools: Scribe (Facebook)

- A fault tolerant log-routing framework.

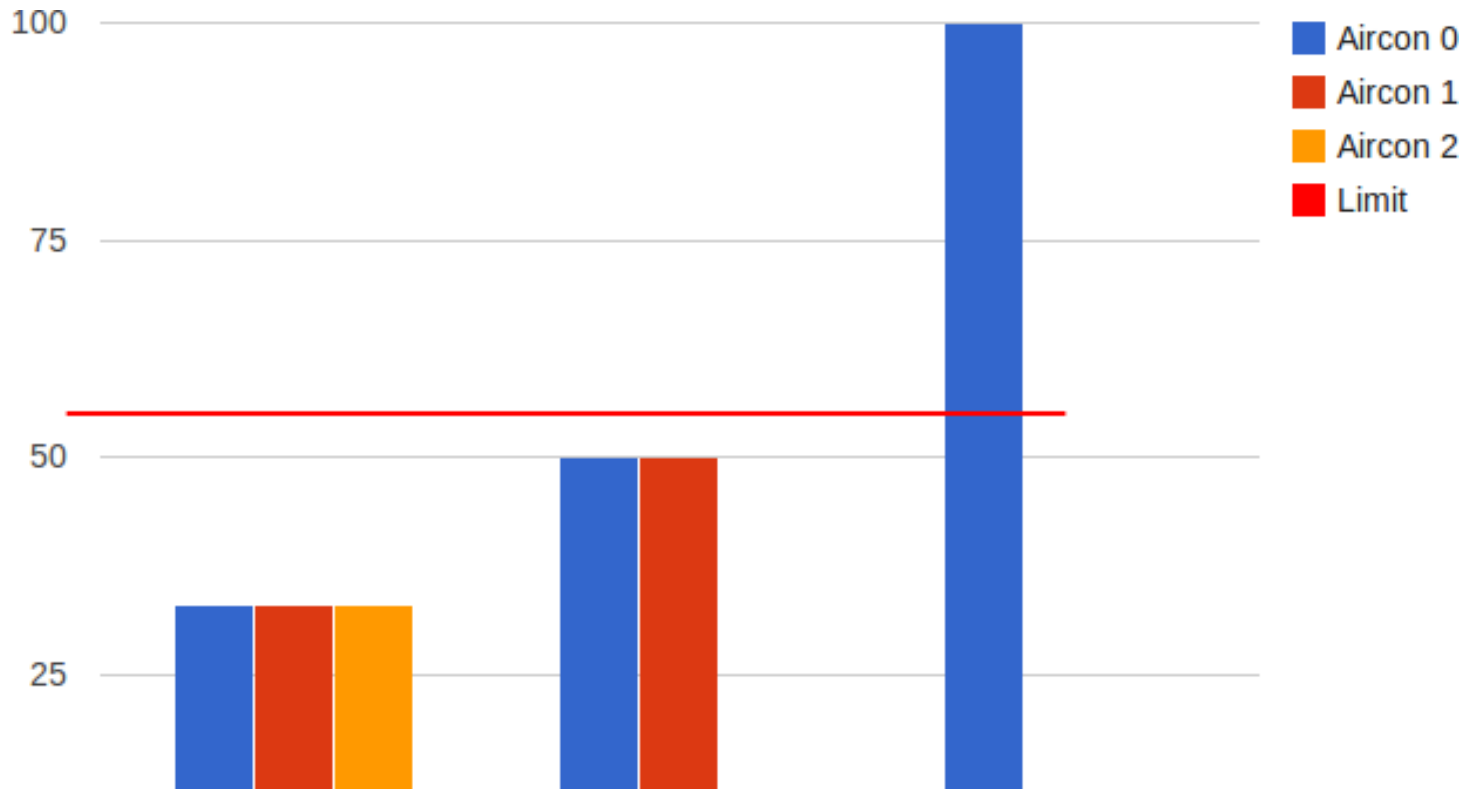


The median latency for trace data collection is less than 15 seconds. The 98th percentile latency is itself bimodal over time; approximately 75% of the time, 98th percentile collection latency is less than two minutes, but the other approximately 25% of the time it can grow to be many hours. – Sigelman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

See: http://www.facebook.com/note.php?note_id=32008268919

Failures Cascade

- Failure of one mirror transfers load to others.

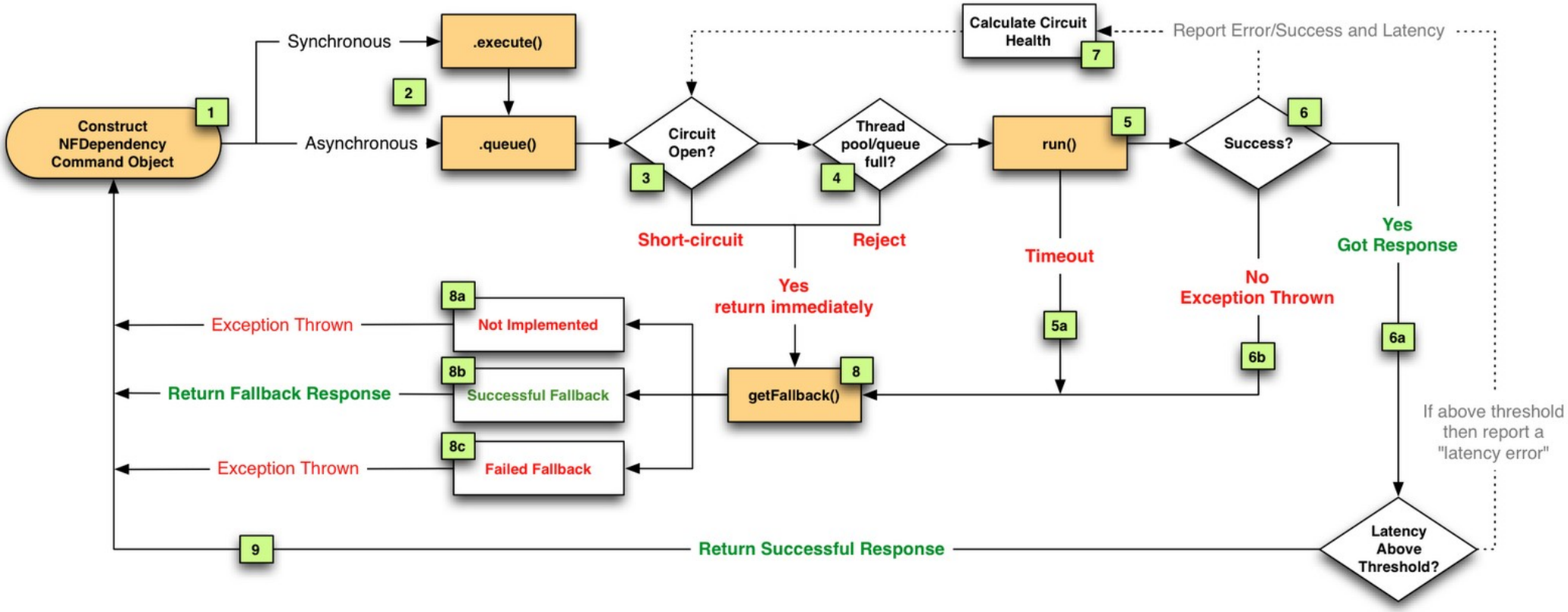


See: http://upalc.com/google-amazon.php

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)

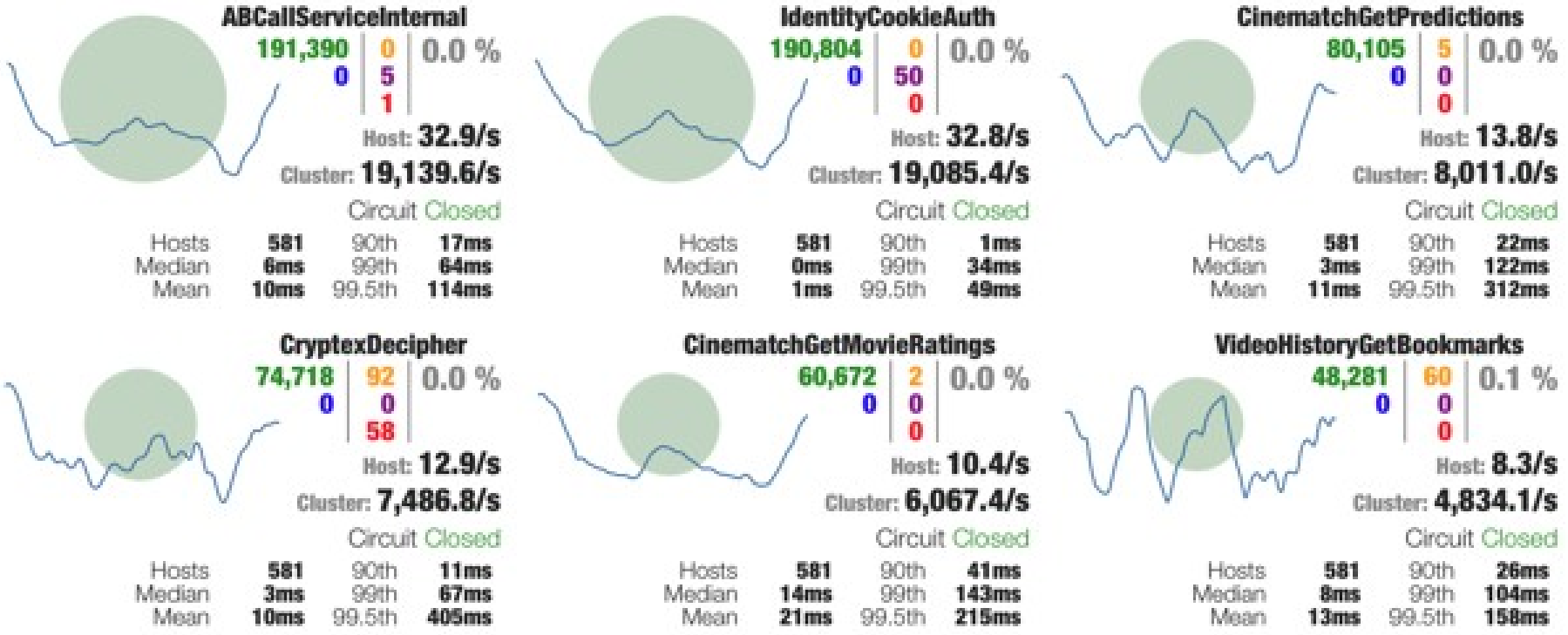


Aside: Steve Yegge's Google Rant

- 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

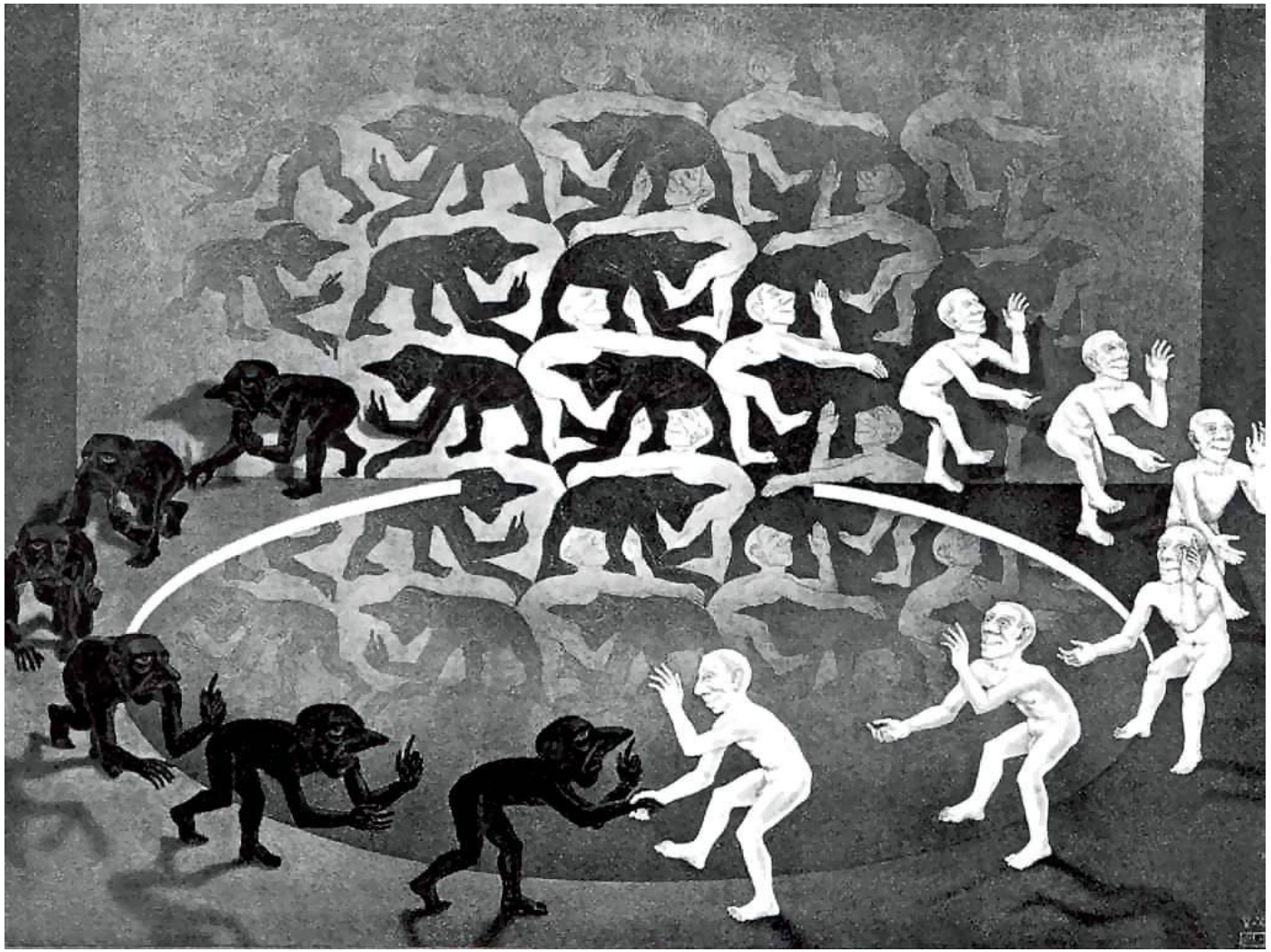
Monitoring Failure (Netflix)



- 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.

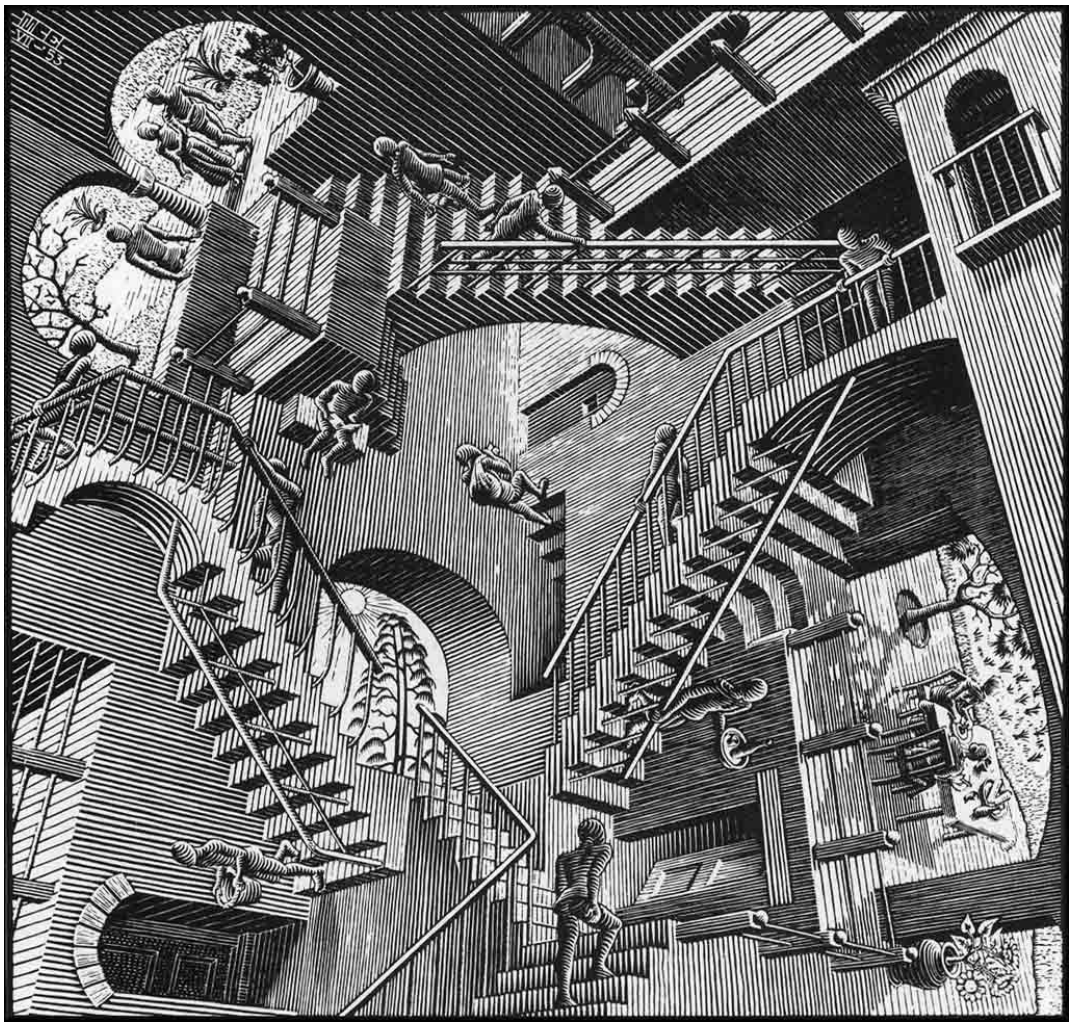
Fabric Services

- 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.

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...

Conclusions

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