Cloud Computing and Big Data

Cloud Computing Background and Theory

Oxford University
Software Engineering
Programme
July 2020

Contents

- Distributed Computing
- Scalability
- Virtualization
- Multi-tenancy
- Amdahl's Law and Gustavson's Law
- Karp-Flatt Metric
- Shared Nothing Architectures
- CAP Theorem
- Eventual Consistency

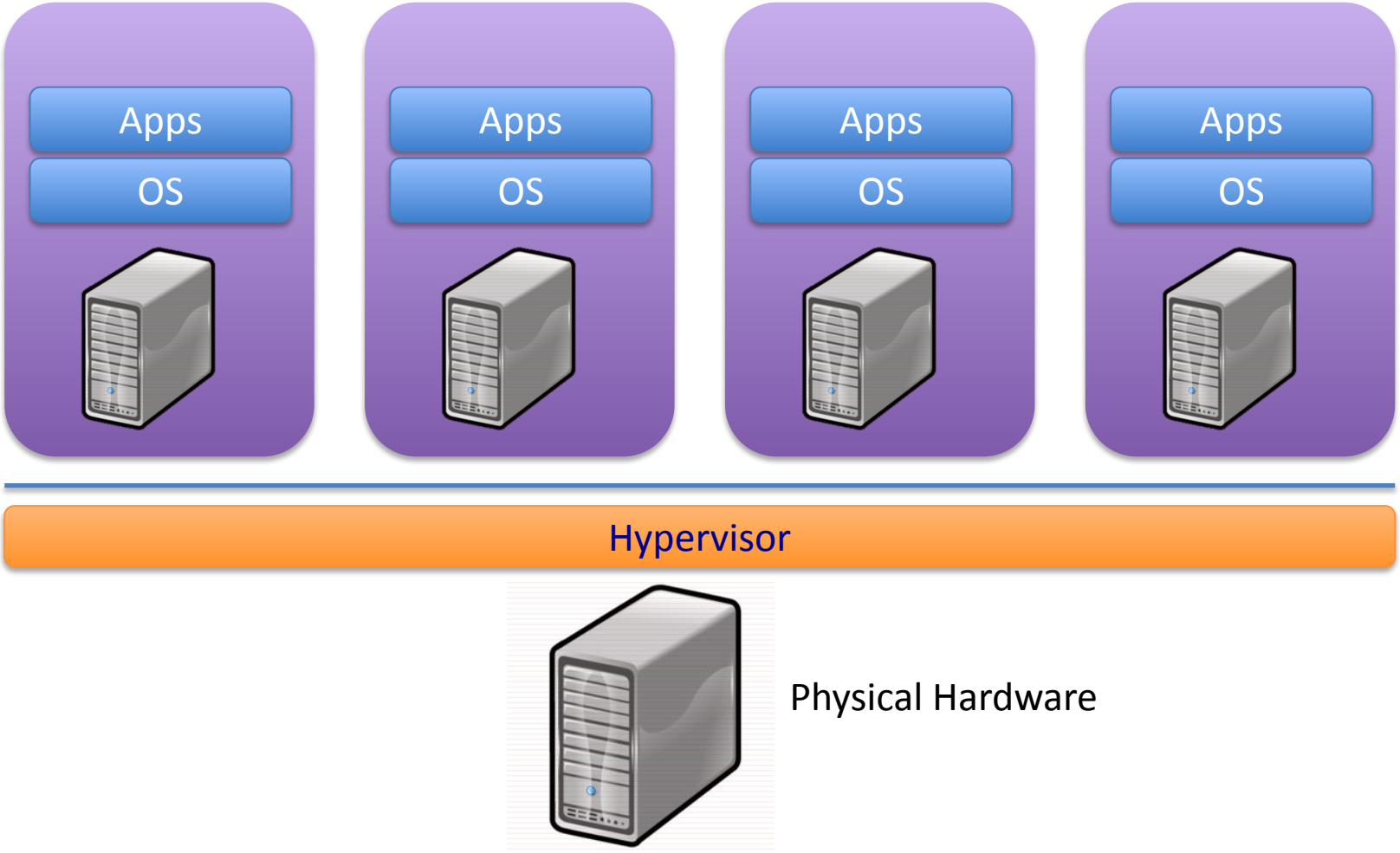
Distributed Computing

- Cloud would not be possible without RPC/Services/APIs
  - e.g. Call a service to instantiate a machine image for us
- Grid and Cloud both emerged from distributed computing concepts

Fundamental problems in Distributed Computing

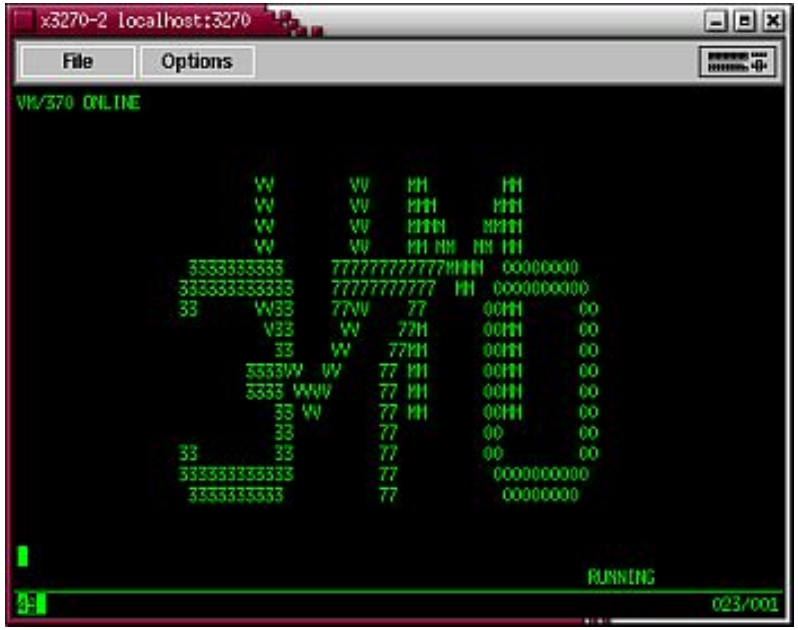
- Efficient distribution of work
  - combating *serialization*
- Consensus
  - combating *failure*

Virtualization



Virtualization

- Dates back to 1972 with IBM VM/370
- Each user had a “virtual mainframe”
  - Including a virtual punch card reader and writer!



Commodity Hardware Virtualization

1985

Late 2005 / Early 2006

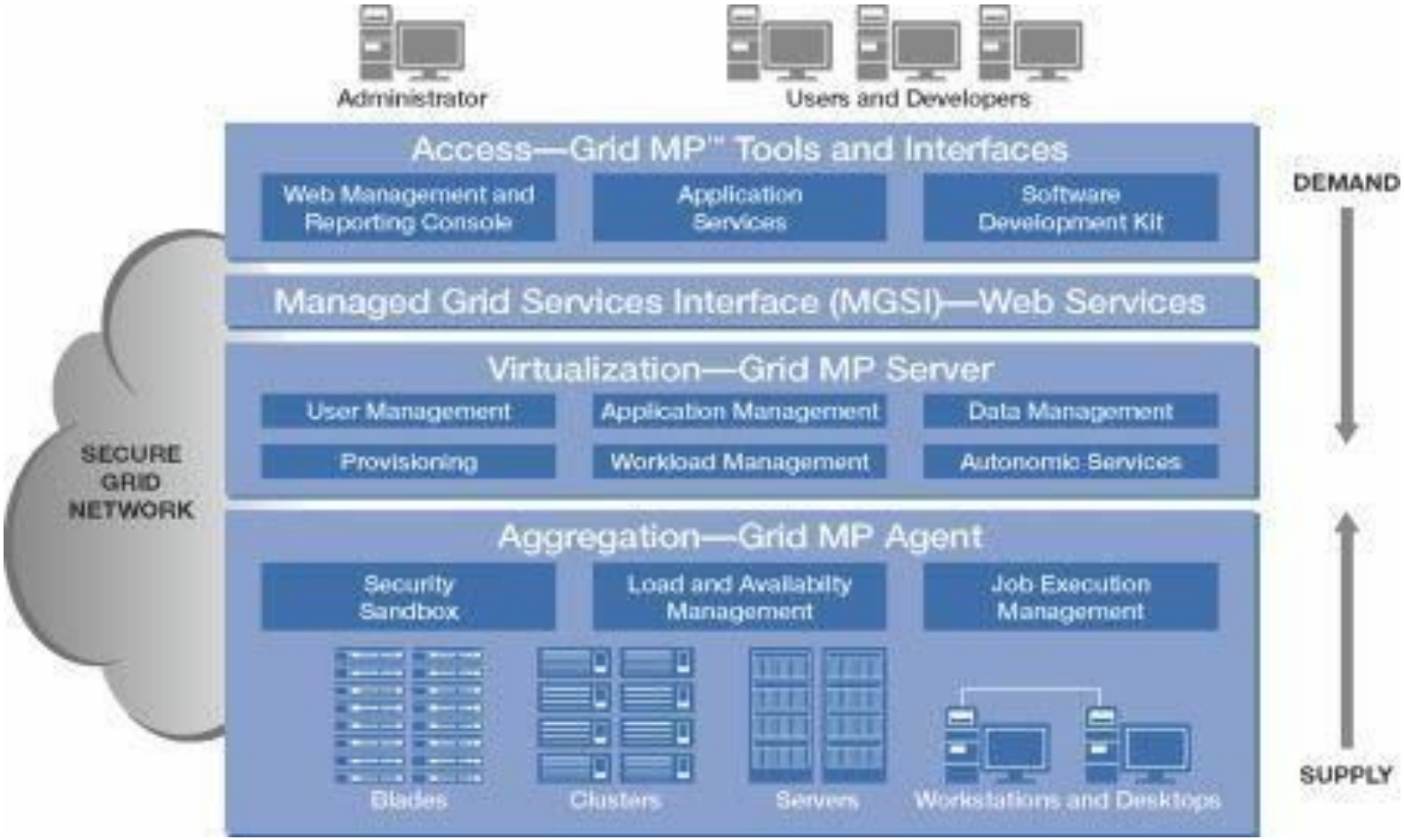
VT-X

AMD-V

Benefits of Virtualization

- Replicability
  - Machines become repeatable images
  - Can be migrated, snapshotted, version controlled
- Better resource utilization
  - Most servers run at about 6-12% utilization
- Flexibility
  - New instances don't necessarily require new hardware

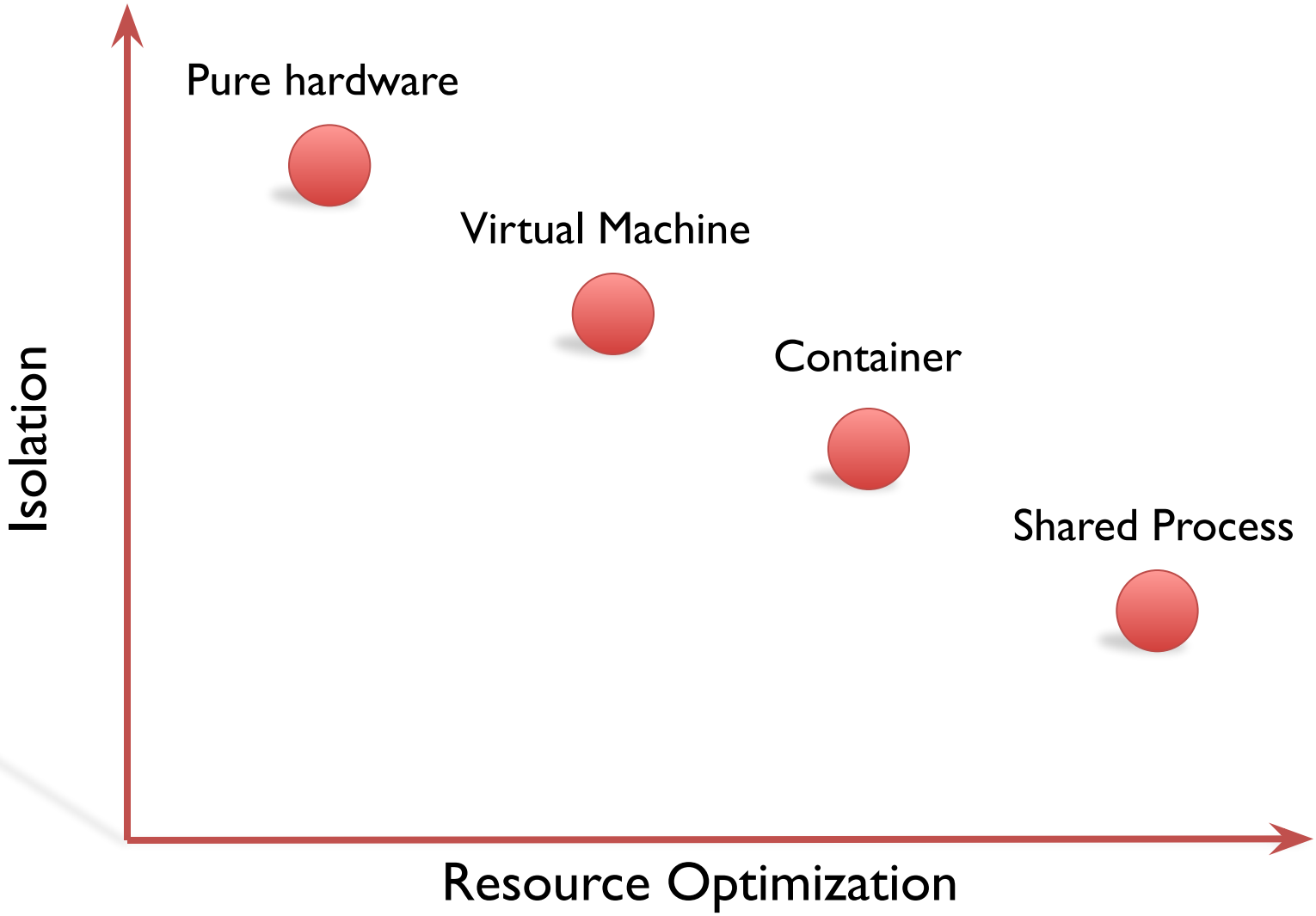
Grid Computing



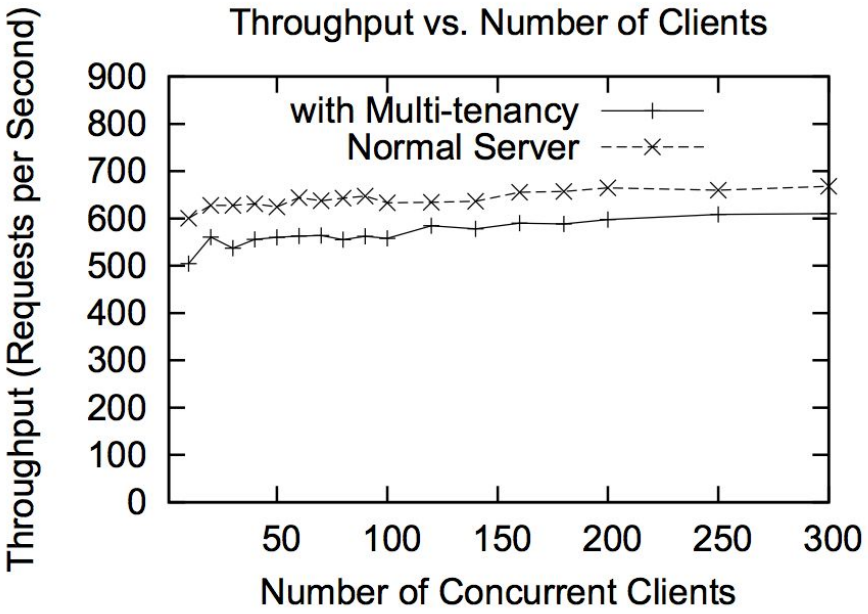
What is Multi-tenancy ?

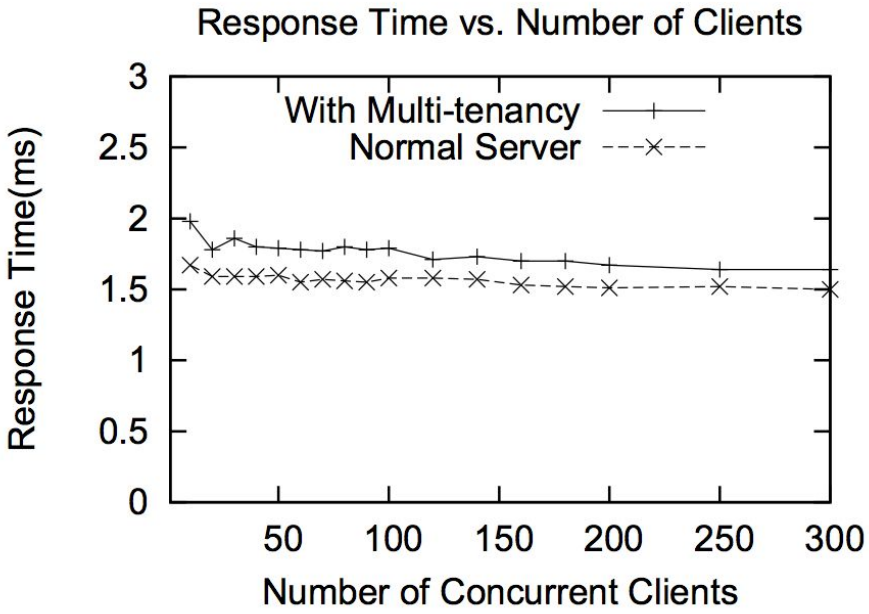
- Many parties sharing the same set of resources, while giving each their own space

Multi-tenancy models



Performance Overhead of Multi-Tenancy in WSO2 Carbon platform





Azeez, Afkham, Srinath Perera, Dimuthu Gamage, Ruwan Linton, Prabath Siriwardana, Dimuthu Leelaratne, Sanjiva Weerawarana, and Paul Fremantle. *"Multi-tenant SOA middleware for cloud computing."* In Cloud computing (cloud), 2010 ieee 3rd international conference on, pp. 458-465. IEEE, 2010.

scalability

/ˌskeɪləˈbɪlɪtɪ/

noun

1. the ability of something, esp a computer system, to adapt to increased demands

Collins English Dictionary - Complete & Unabridged 2012 Digital Edition

Speedup

- The **speedup** is defined as the performance of new / performance of old
  - e.g. move from 1 -> 2 servers
  - New system is 1.8 x faster than the old
    - In terms of transactions/sec (throughput)
  - Speedup = 1.8

What inhibits speedup?

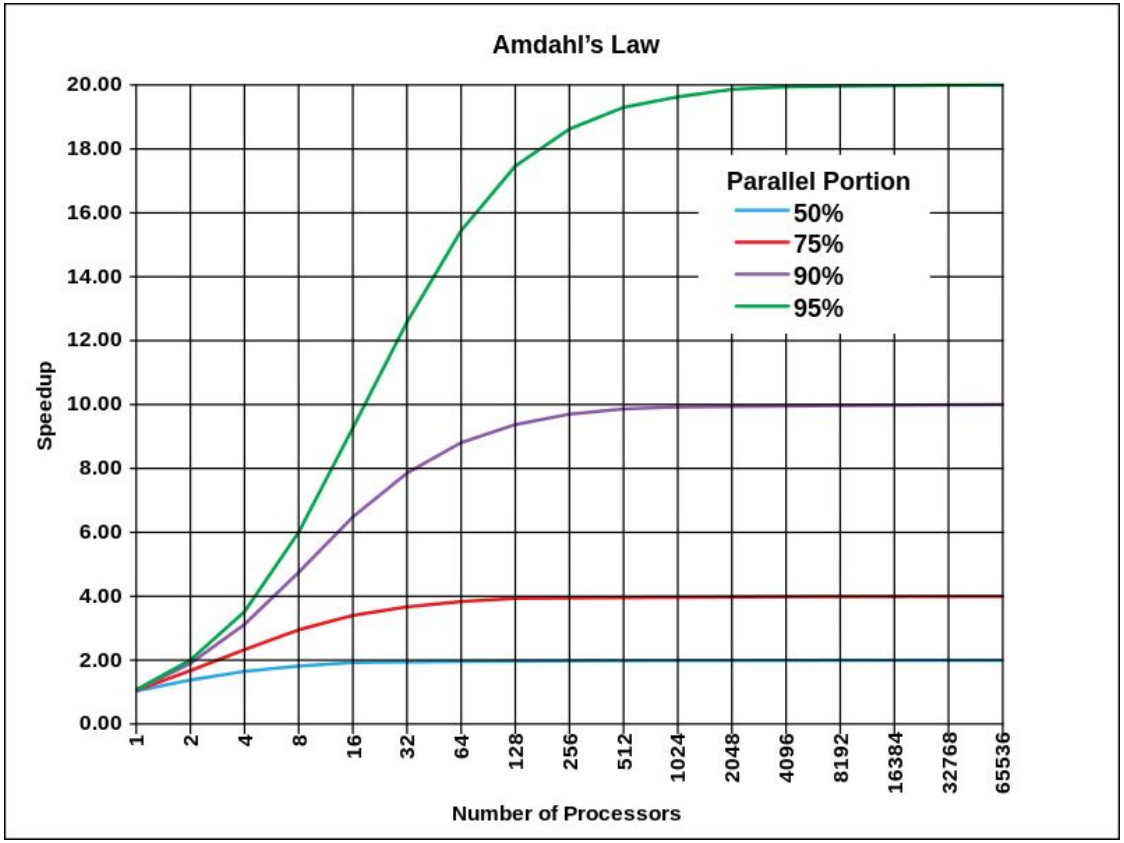
- In general you can split work into
  - Parallelizable and
  - Serial

    parts
- The serial parts stop you from scaling

Amdahl's Law

Theoretical speedup given a fixed data size

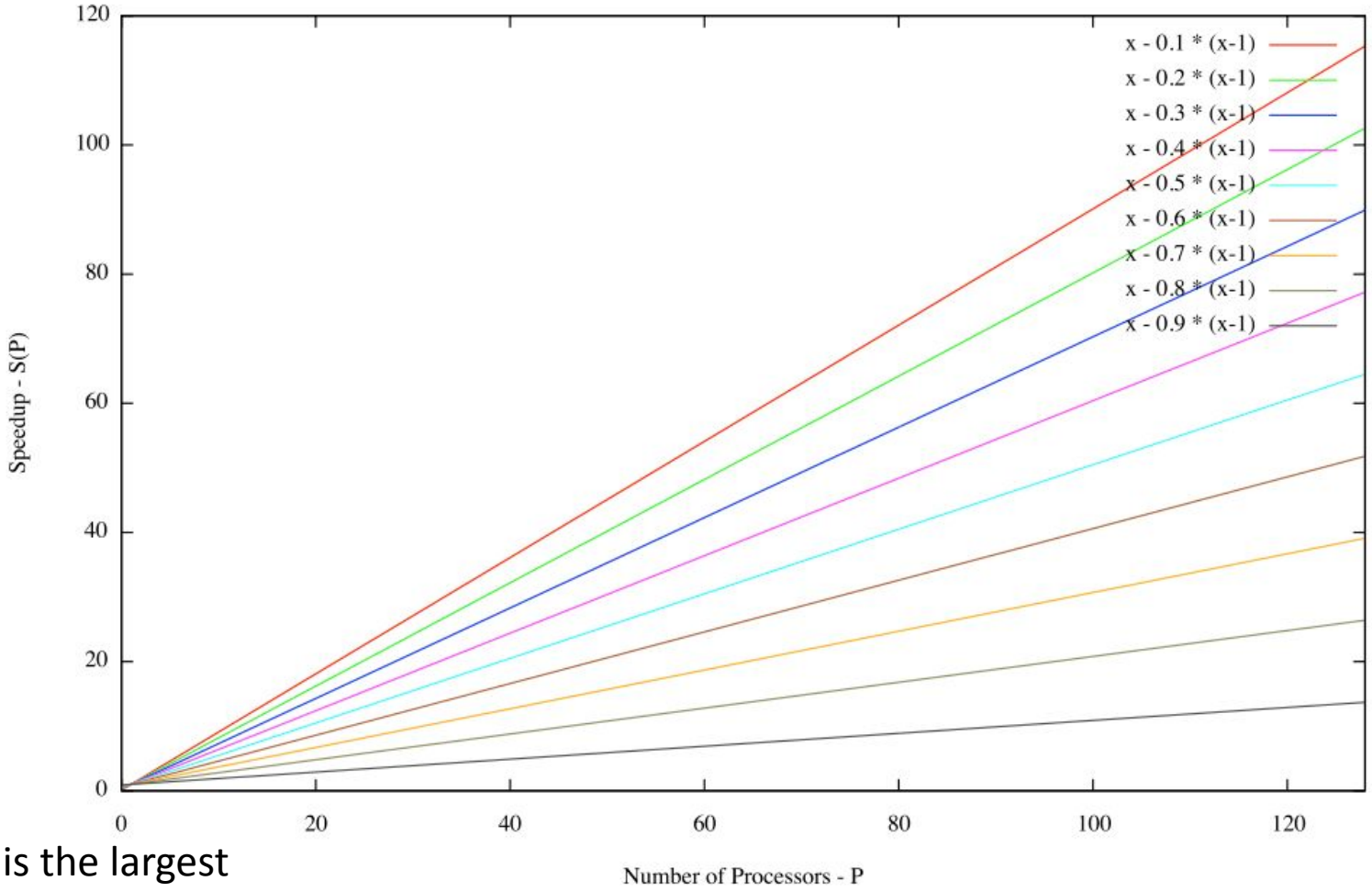
The speedup of a program using multiple processors in parallel computing is limited by the time needed for the serial fraction of the program, given a fixed size of data



Gustafson's Law

What if the data increases too?

$$S(P) = P - \alpha \cdot (P - 1)$$



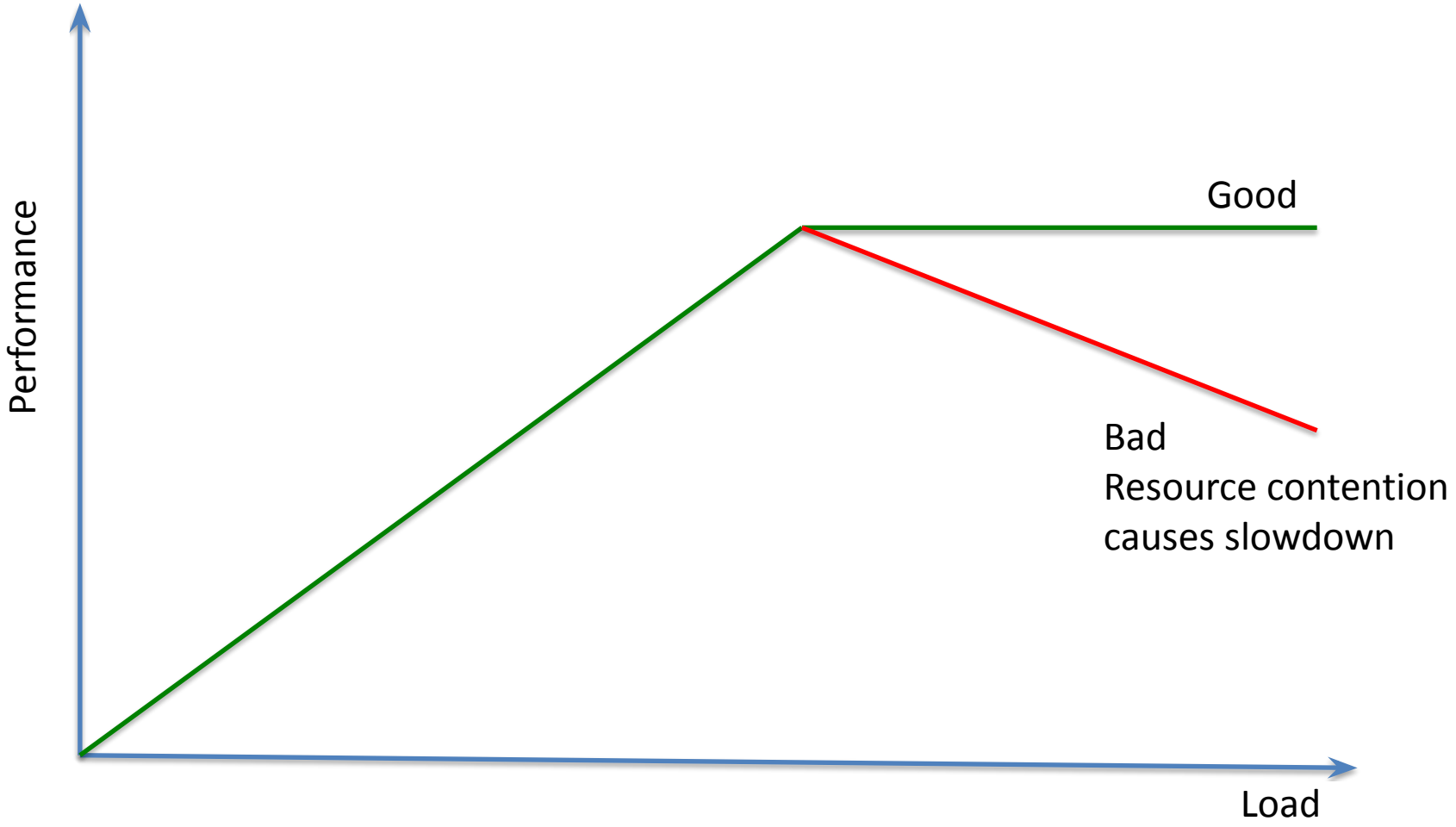
α is the largest non-parallelizable fraction

A driving metaphor

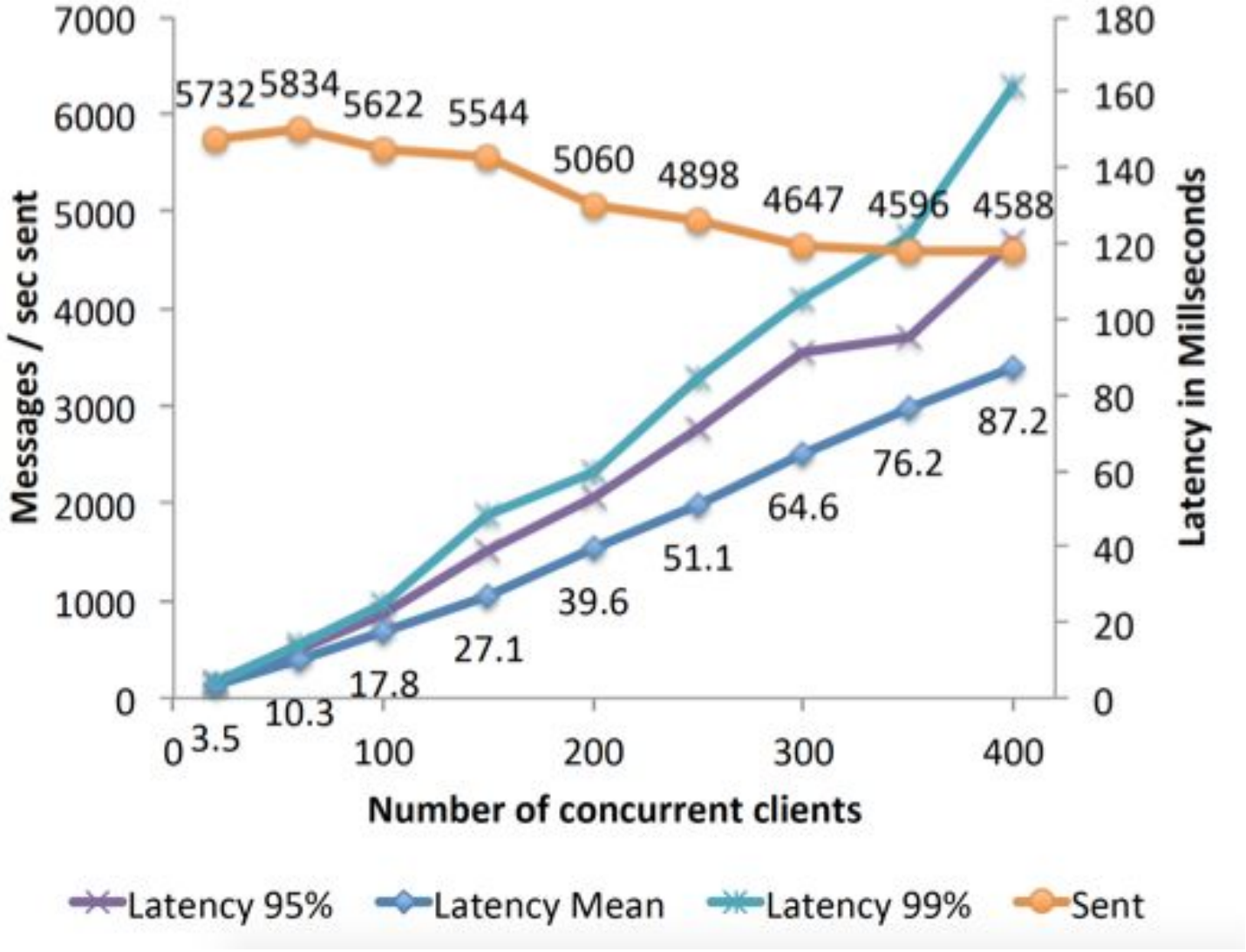
- **Amdahl's Law**
  - You are travelling to London (60 miles)
  - 30 miles in you have spent one hour
  - You can never average > 60 mph
- **Gustafson's Law**
  - You are travelling across the US
  - You've spent an hour at 30 mph
  - You can achieve any average speed given enough time and distance

Performance

Single system under increasing load

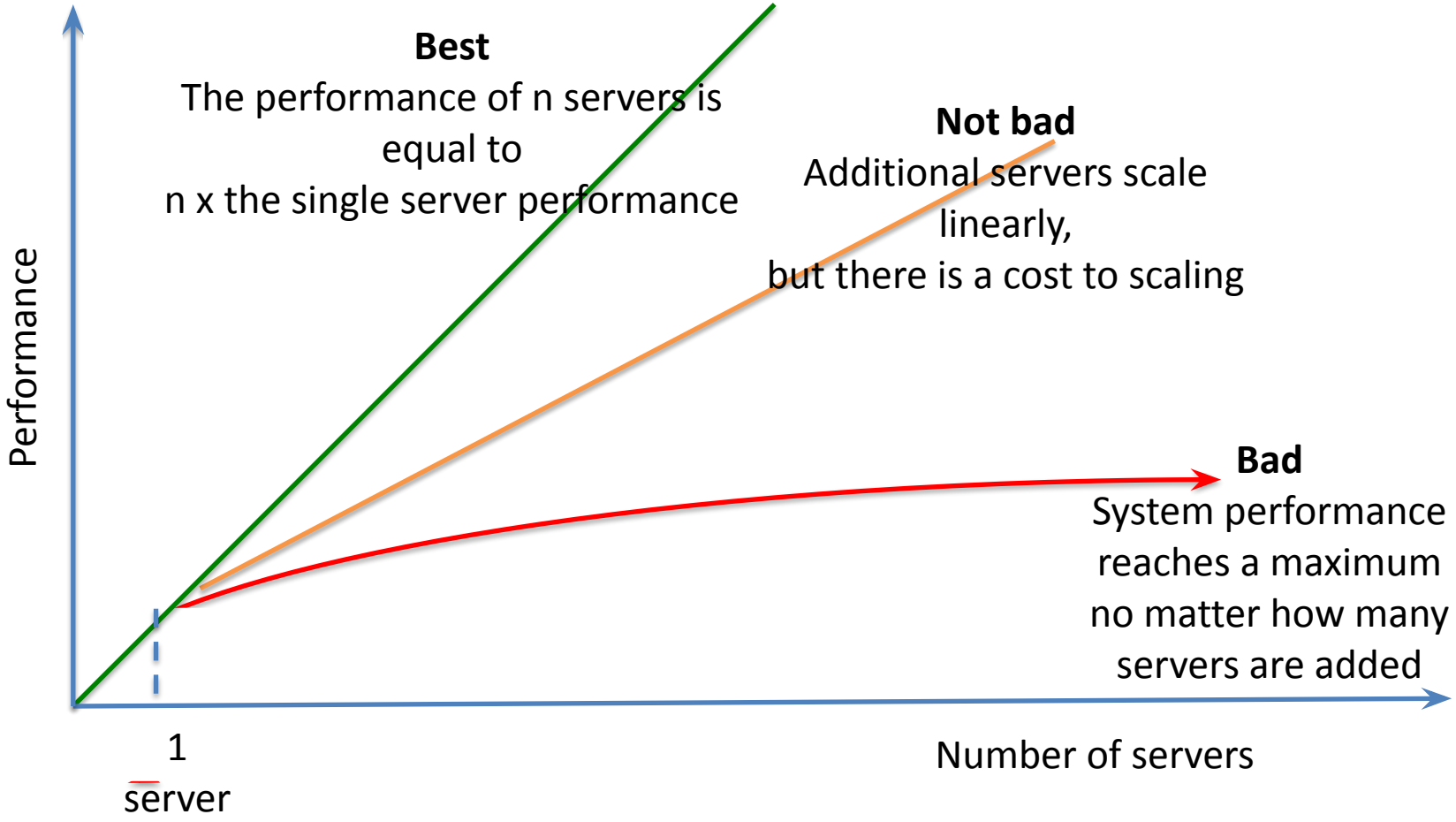


A real example (OAuthing)



Performance

Scaling servers when fully loaded



Karp-Flatt Metric

e is the Karp-Flatt Metric

ψ is the speedup

p is the number of processors

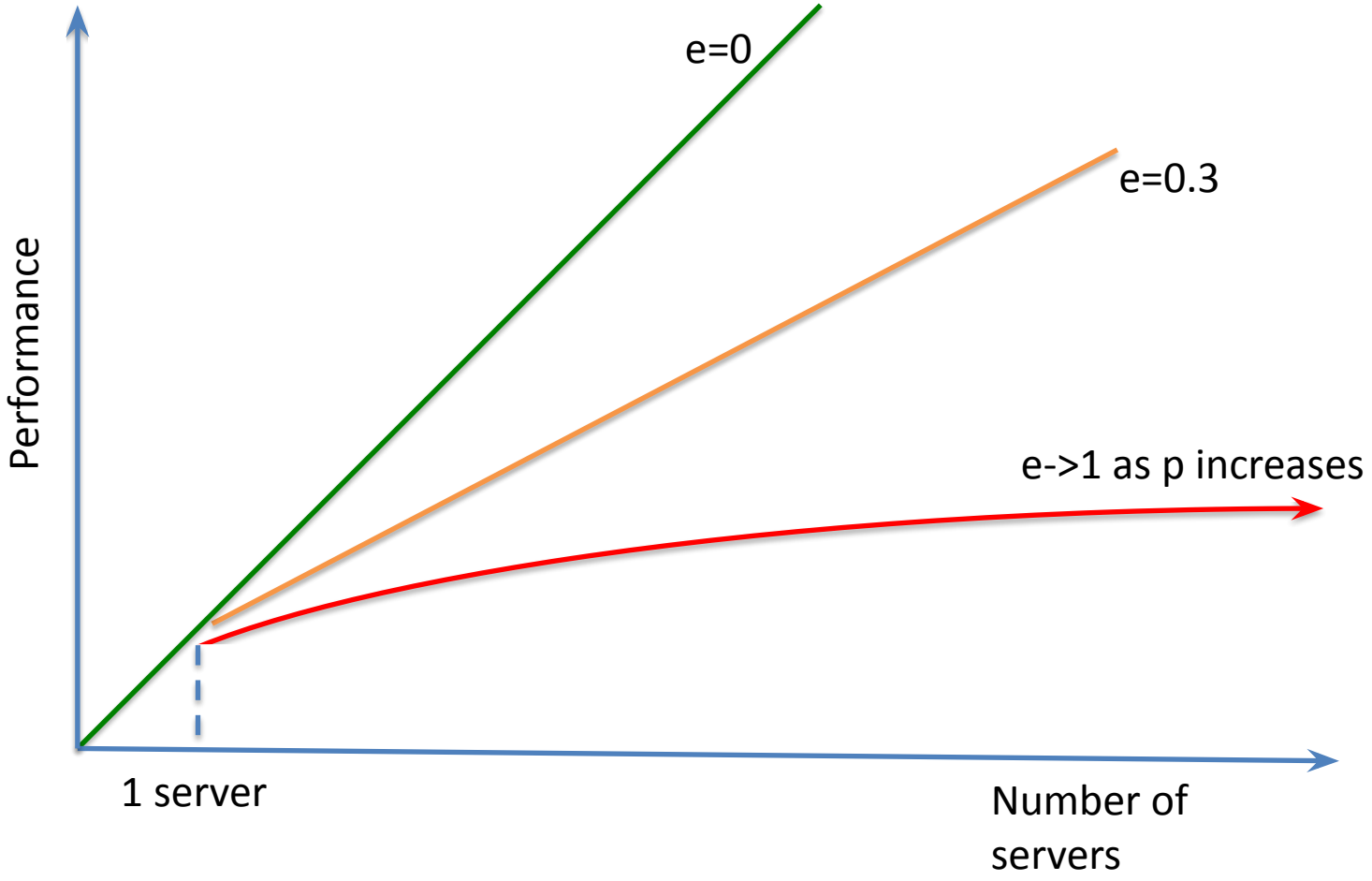
$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$

e = 0 is the best

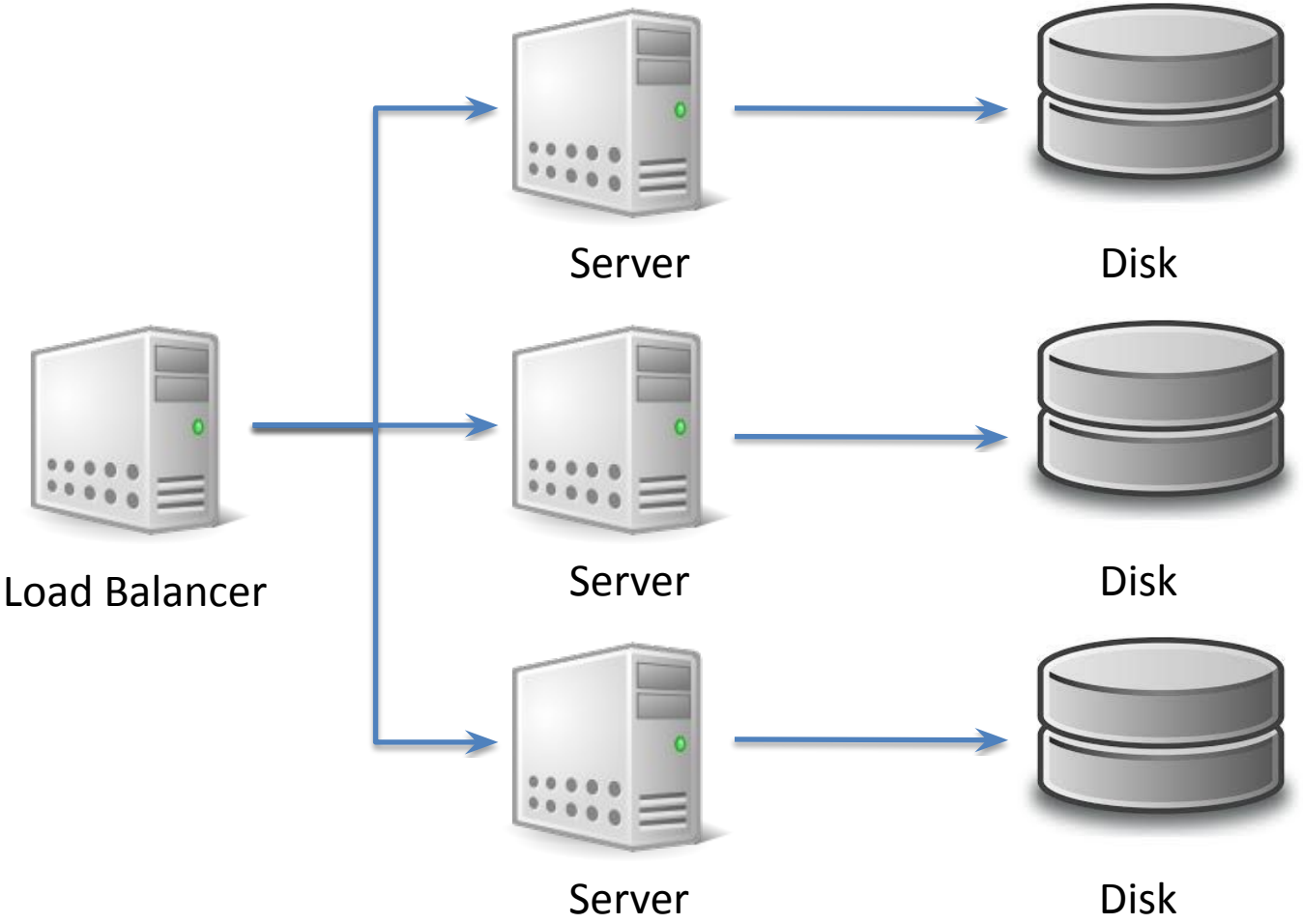
e = 1 indicates no speedup

e > 1 indicates adding processors slows down the system!!!

Karp-Flatt metric



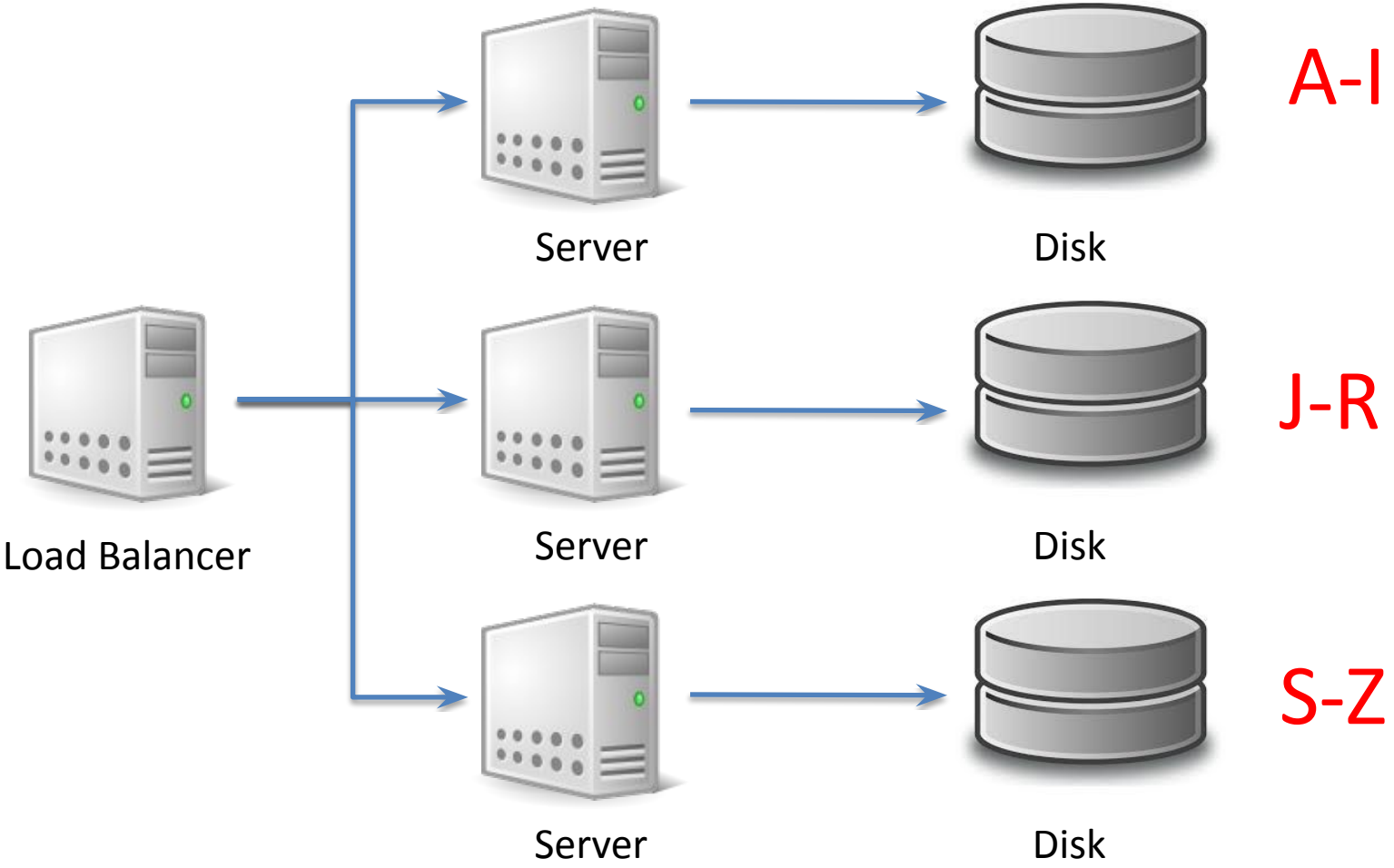
Shared Nothing Architecture



Shared Nothing Architecture

- Implies there is no serial part to the computation
- Karp-Flatt Metric of 0
  - Assuming 100% efficient load balancing
- In practice, this is difficult!

Partitioning / Sharding



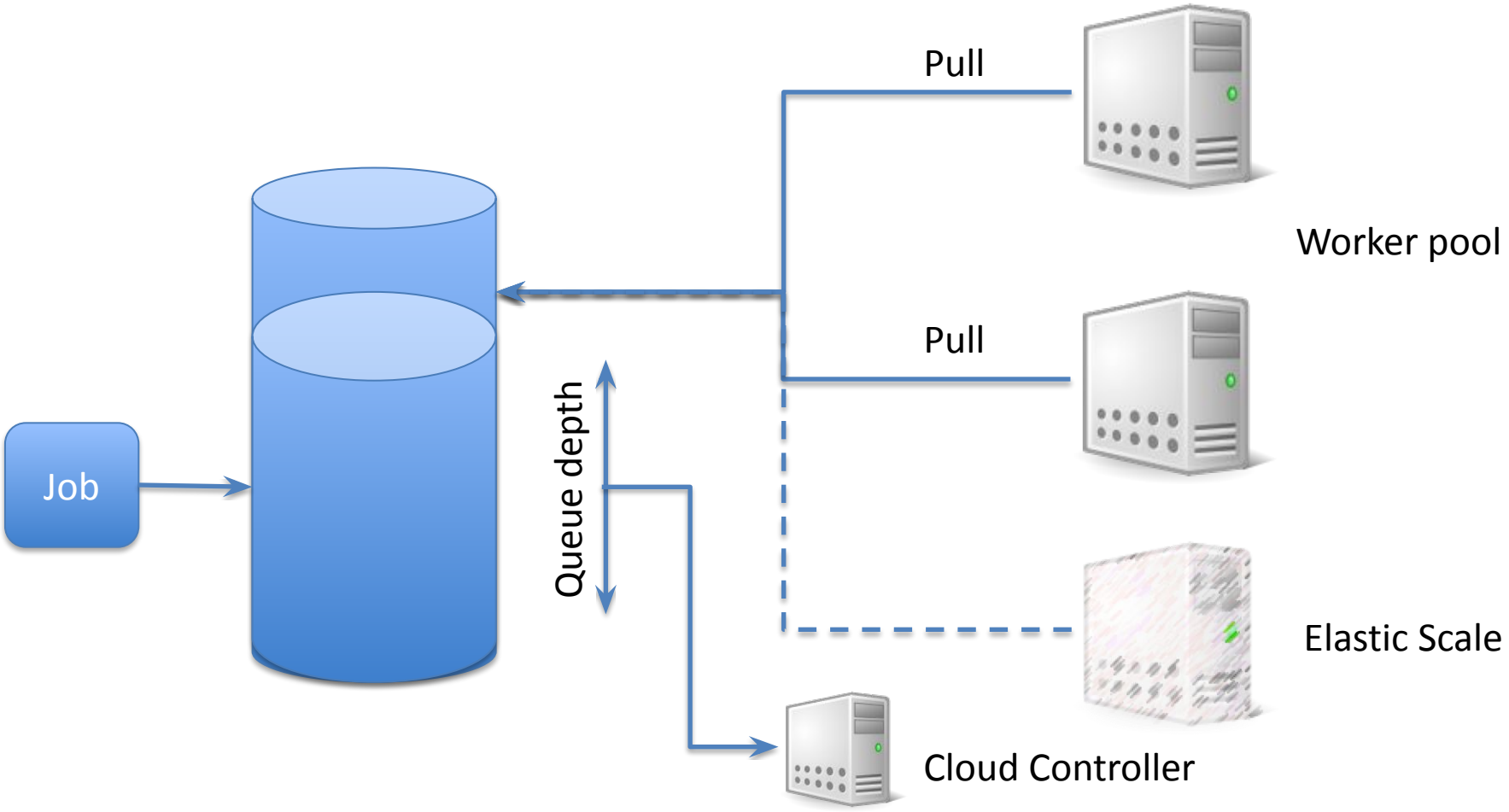
Problems with Sharding

- Imbalance
  - Fewer S-Z's than A-I's
- Failover
- Adding new servers requires a re-balance
  - Is this automatic or manual?!

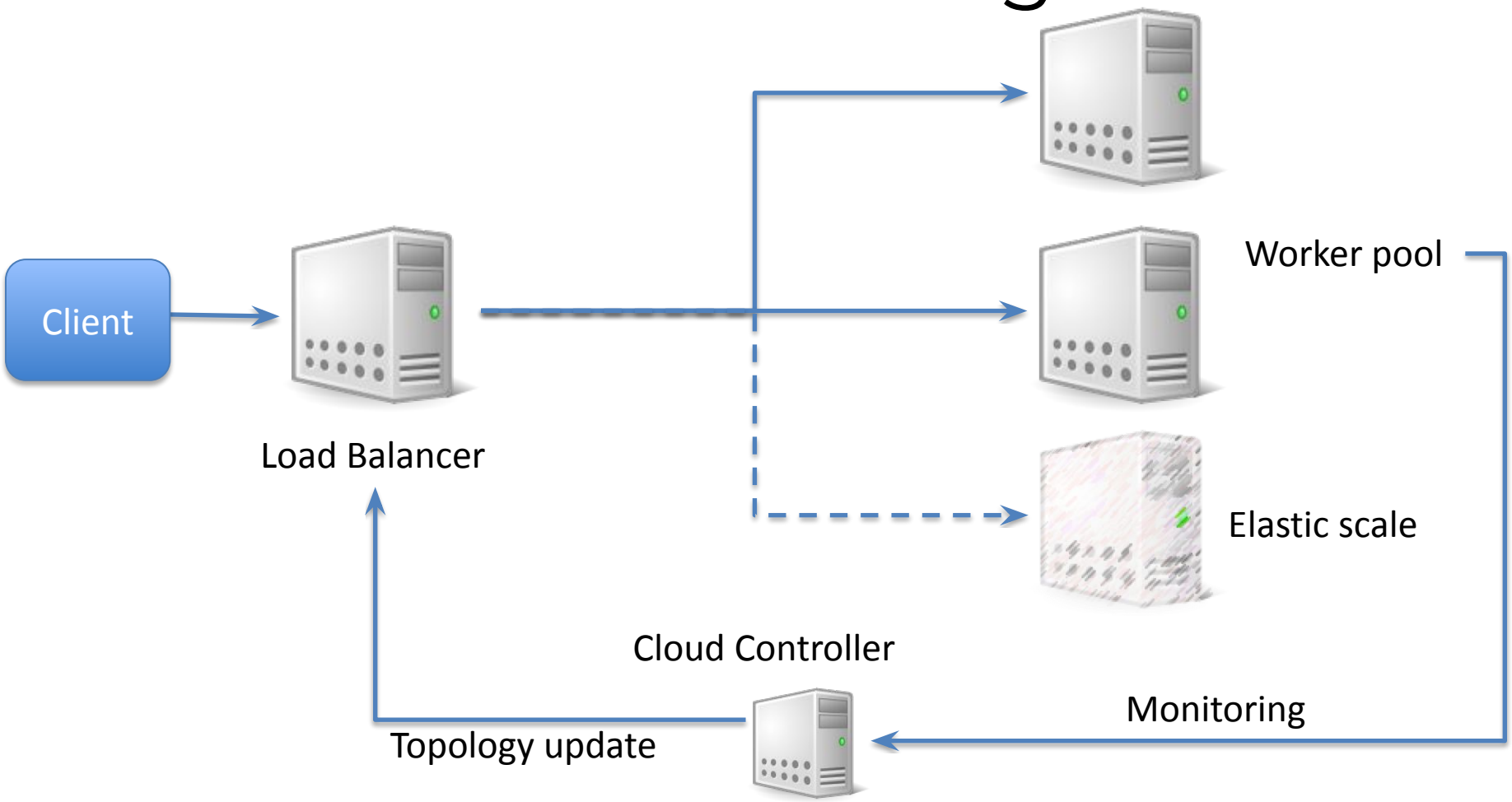
Elastic Scaling

- Dynamically adjusting the number of nodes in a cloud
  - Both up and down
  - Based on input load
  - Aiming to meet a specific SLA

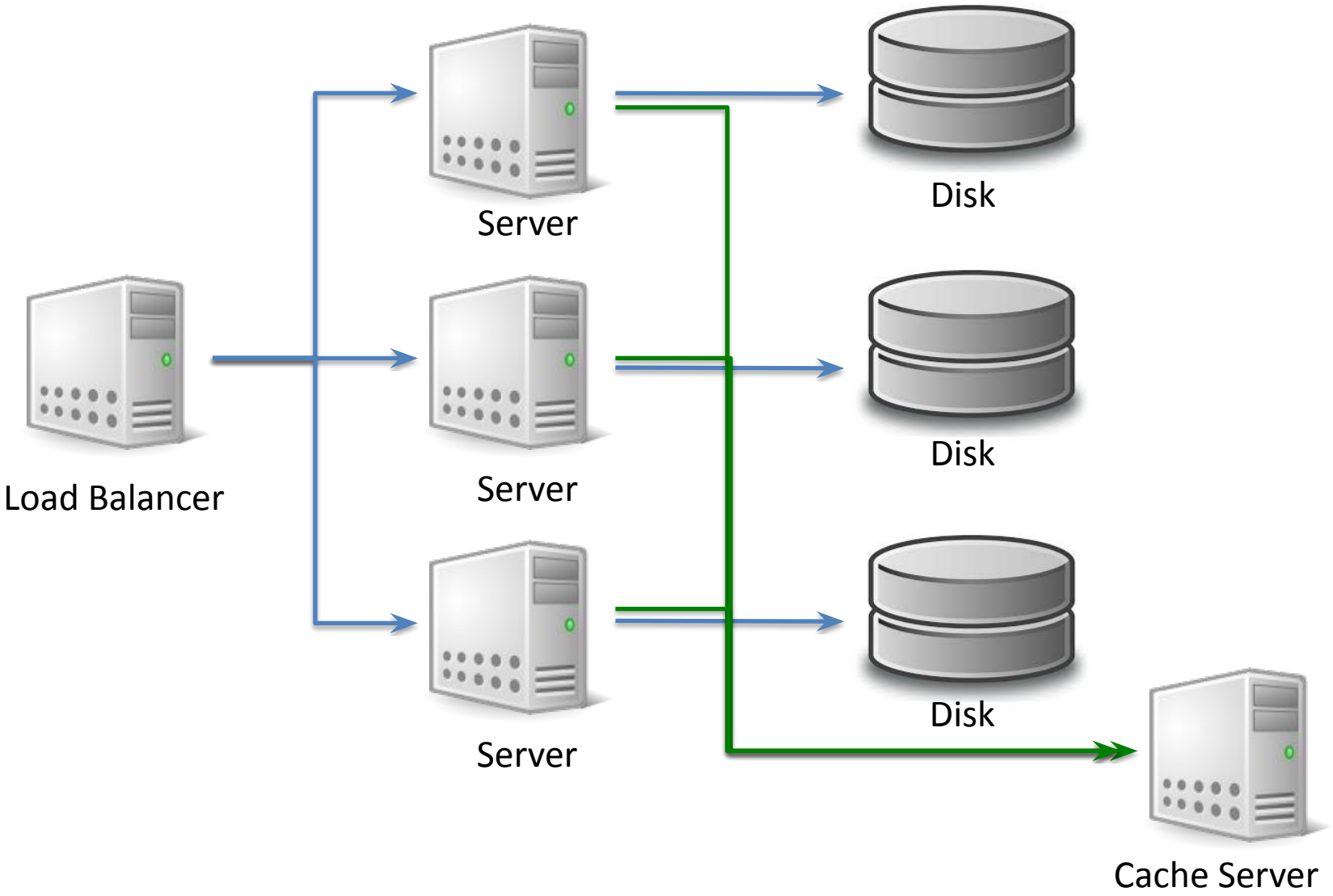
Elastic Queue Consumers



Load Balancer-based elastic scaling



Cache


Server
Disk
Load Balancer
Server
Disk
Server
Disk
Cache Server

Consensus

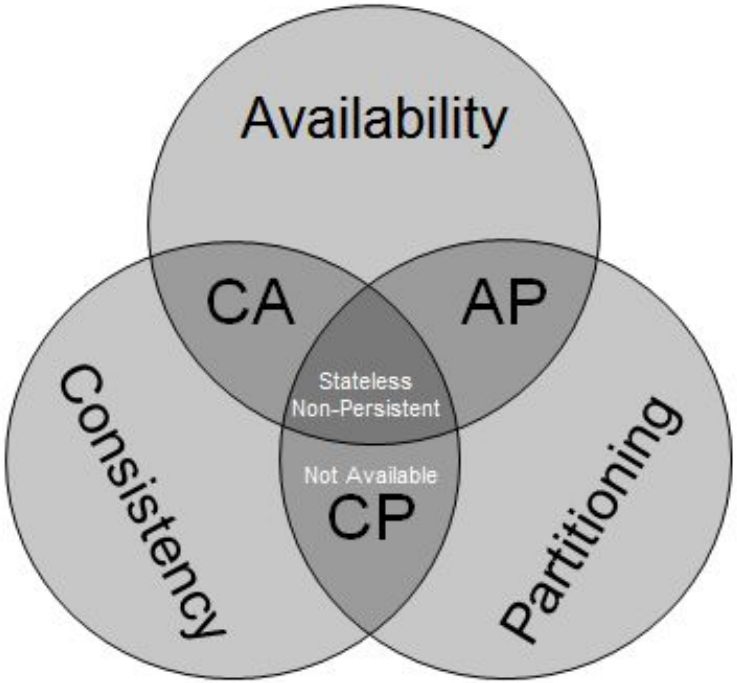
- Leadership
  - How to elect a leader from a group of nodes
- Failure detection
  - How to spot a failed leader

ACID

- *atomicity*
  - all-or-nothing
- *consistency*
  - integrity-preserving: invariants satisfied
- *isolation*
  - hidden intermediate results: multi-user behaviour consistent with single-user mode
- *durability*
  - permanent committed results

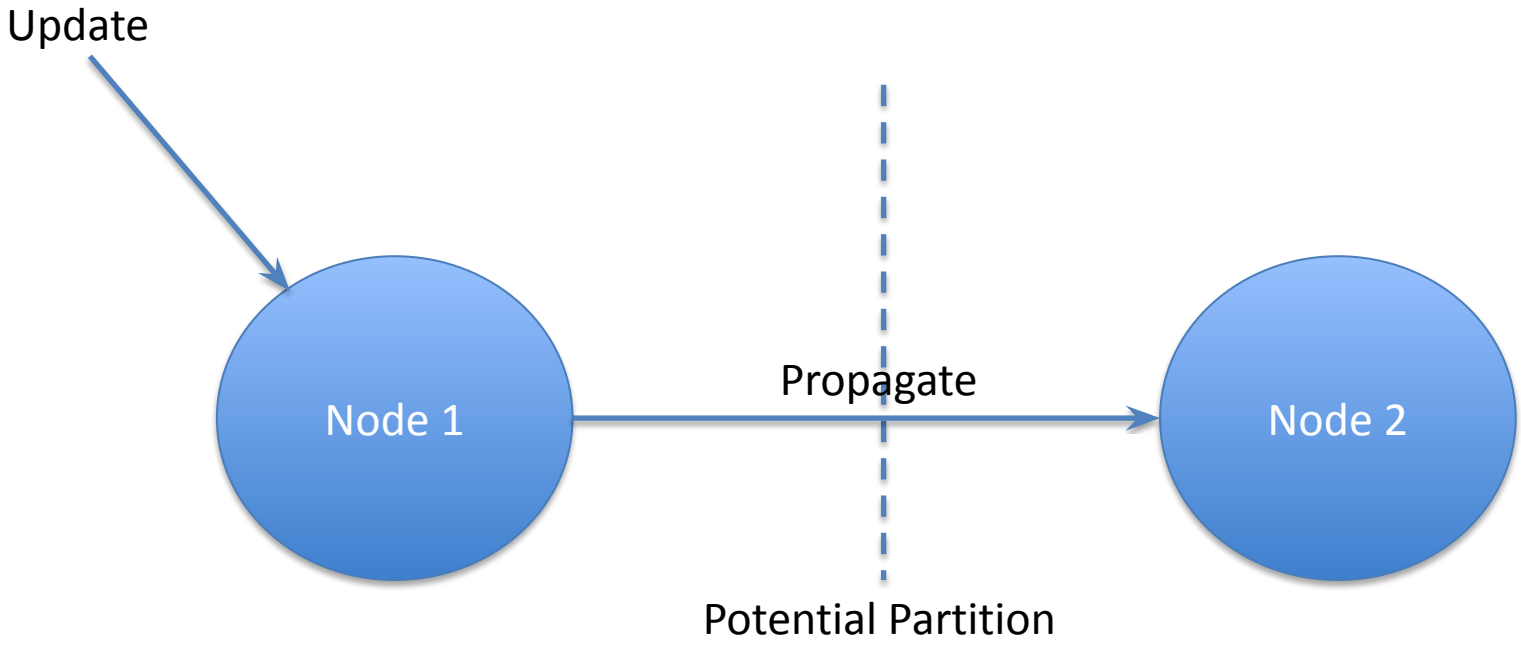
CAP Theorem

- Originally proposed by Eric Brewer
  - Inktomi and Berkeley
- Proved in 2002 by Gilbert and Lynch
- You can have 2 out of three:
  - Consistent
    - ACID
  - Available
  - Partitioned
    - Survive network down between nodes

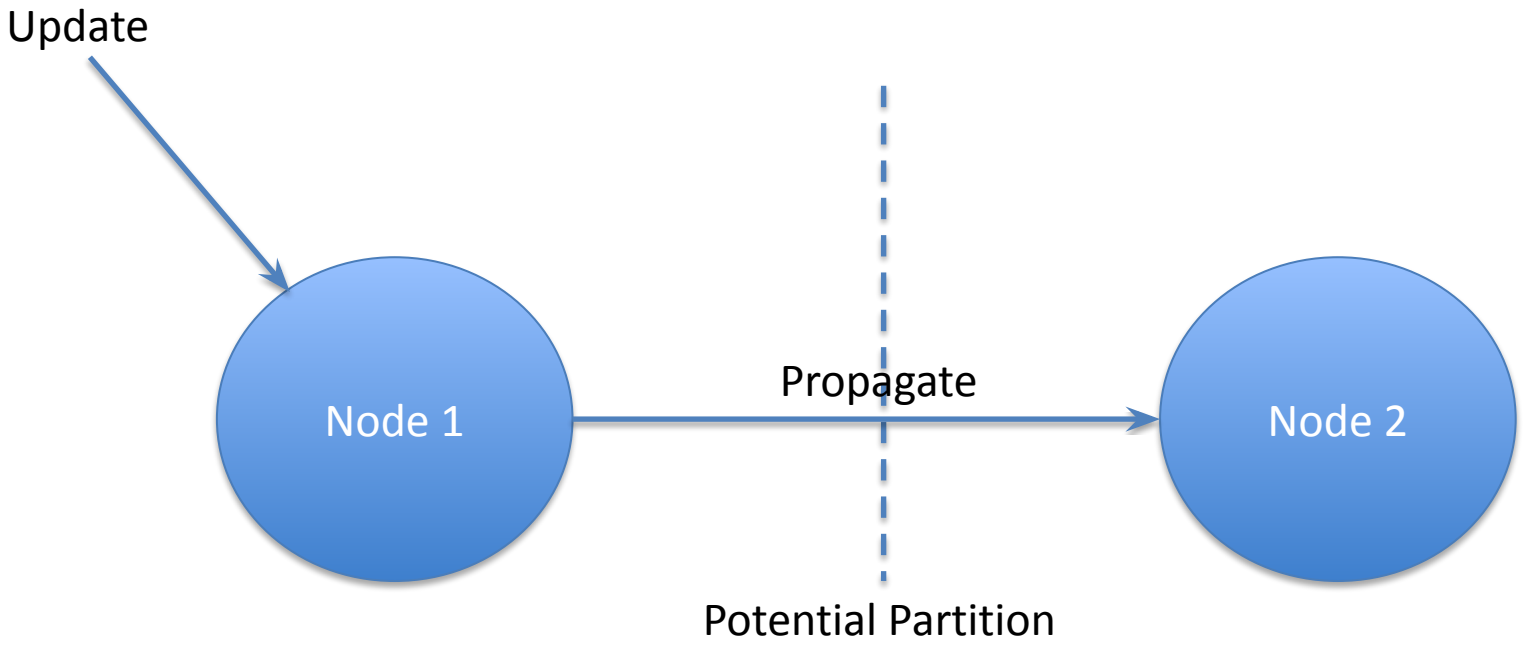


http://blog.mccrory.me/2010/11/03/cap-theorem-and-the-clouds/

Imagine two nodes



Imagine two nodes



If there is a partition, then you can **either** update one node (give up on C), **or** make one node unavailable (give up on A).

If you want C and A you can't allow a Partition.

CAP options

- CA
  - Traditional databases
  - Cannot be scaled multi-datacentre or work in cases of high-latency
- AP
  - Multi-master NoSQL databases
    - Dynamo, Cassandra, CouchDB
    - Not consistent but work across datacentres in a highly available model
- CP
  - Not a good idea, as not available!

CAP Theorem

- However, the details are important
  - The proof requires some complex definitions of C, A and P
- I recommend reading Brewer's update:
  - http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed
  - "The 2 of 3 formulation was always misleading"
  - "CAP prohibits only a tiny part of the design space"

In real life

- Partitions are rare
- So we can implement a strategy:
  - Detect a partition
  - Enter “partition mode”
  - Carry on with inconsistency
  - Recover when partition vanishes
- Known as “eventually consistent”

What does recovery mean?

- Depends on your database and requirements
  - E.g. Amazon's shopping cart is made consistent by creating the union of the inconsistent carts
  - Deleted items may re-appear
- Another option is to forbid certain operations during partition mode
  - To make it easier to recover consistency
- A simplistic approach would be to go read-only

What does that mean in real-life?

- Databases like Cassandra let you "tune" consistency and availability
  - Define the quorum you need for a response
  - Trades off latency vs consistency
    - Choose an "easy quorum" for guaranteed low latency
    - Choose a "hard quorum" for higher potential latency

PACELC

(pr. pass-elk)

- **P**artition: **A**vailability vs **C**onsistency, **E**lse **L**atency vs **C**onsistency
  - *"For data replication over a WAN, there is no way around the consistency/latency tradeoff"*
  - Usually a combination of sync/async
    - Synchronous writes to *n* systems followed by asynchronous writes to *m* systems

http://cs-www.cs.yale.edu/homes/dna/papers/abadi-pacelc.pdf

Cassandra Quorum Levels (Write)

Write Consistency Levels

Level	Description	Usage
ALL	A write must be written to the commit log and memtable on all replica nodes in the cluster for that partition.	Provides the highest consistency and the lowest availability of any other level.
EACH_QUORUM	Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in *all* data centers.	Used in multiple data center clusters to strictly maintain consistency at the same level in each data center. For example, choose this level if you want a read to fail when a data center is down and the QUORUM cannot be reached on that data center.
QUORUM	A write must be written to the commit log and memtable on a quorum of replica nodes.	Provides strong consistency if you can tolerate some level of failure.
LOCAL_QUORUM	Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in the same data center as the coordinator node. Avoids latency of inter-data center communication.	Used in multiple data center clusters with a rack-aware replica placement strategy, such as NetworkTopologyStrategy, and a properly configured snitch. Use to maintain consistency locally (within the single data center). Can be used with SimpleStrategy.
ONE	A write must be written to the commit log and memtable of at least one replica node.	Satisfies the needs of most users because consistency requirements are not stringent.
TWO	A write must be written to the commit log and memtable of at least two replica nodes.	Similar to ONE.
THREE	A write must be written to the commit log and memtable of at least three replica nodes.	Similar to TWO.
LOCAL_ONE	A write must be sent to, and successfully acknowledged by, at least one replica node in the local data center.	In a multiple data center clusters, a consistency level of ONE is often desirable, but cross-DC traffic is not. LOCAL_ONE accomplishes this. For security and quality reasons, you can use this consistency level in an offline datacenter to prevent automatic connection to online nodes in other data centers if an offline node goes down.
ANY	A write must be written to at least one node. If all replica nodes for the given partition key are down, the write can still succeed after a hinted handoff has been written. If all replica nodes are down at write time, an ANY write is not readable until the replica nodes for that partition have recovered.	Provides low latency and a guarantee that a write never fails. Delivers the lowest consistency and highest availability.
SERIAL	Achieves linearizable consistency for lightweight transactions by preventing unconditional updates.	You cannot configure this level as a normal consistency level, configured at the driver level using the consistency level field. You configure this level using the serial consistency field as part of the native protocol operation. See failure scenarios.
LOCAL_SERIAL	Same as SERIAL but confined to the data center. A write must be written conditionally to the commit log and memtable on a quorum of replica nodes in the same data center.	Same as SERIAL. Used for disaster recovery. See failure scenarios.

One more surprising thing

(from 1985!)

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

FLP Result

(Fischer, Lynch, Paterson)

- In a truly async system, consensus cannot be achieved if even one part fails
  - We cannot distinguish between failure and delay
- Distributed consensus systems use non-deterministic timeouts to prevent infinite leader election processes
  - E.g. Paxos, Raft

Summary

- We have looked at the challenges to scaling on multiple servers
  - Serial vs Parallel
  - Fixed data vs growing
  - CAP
  - Eventually Consistent

Questions?