Distributed Systems CS6421

Scaling the Web

Prof. Tim Wood

Practice / Projects

You will learn more by trying to build something real!

If you want to get involved in research, this is your chance!

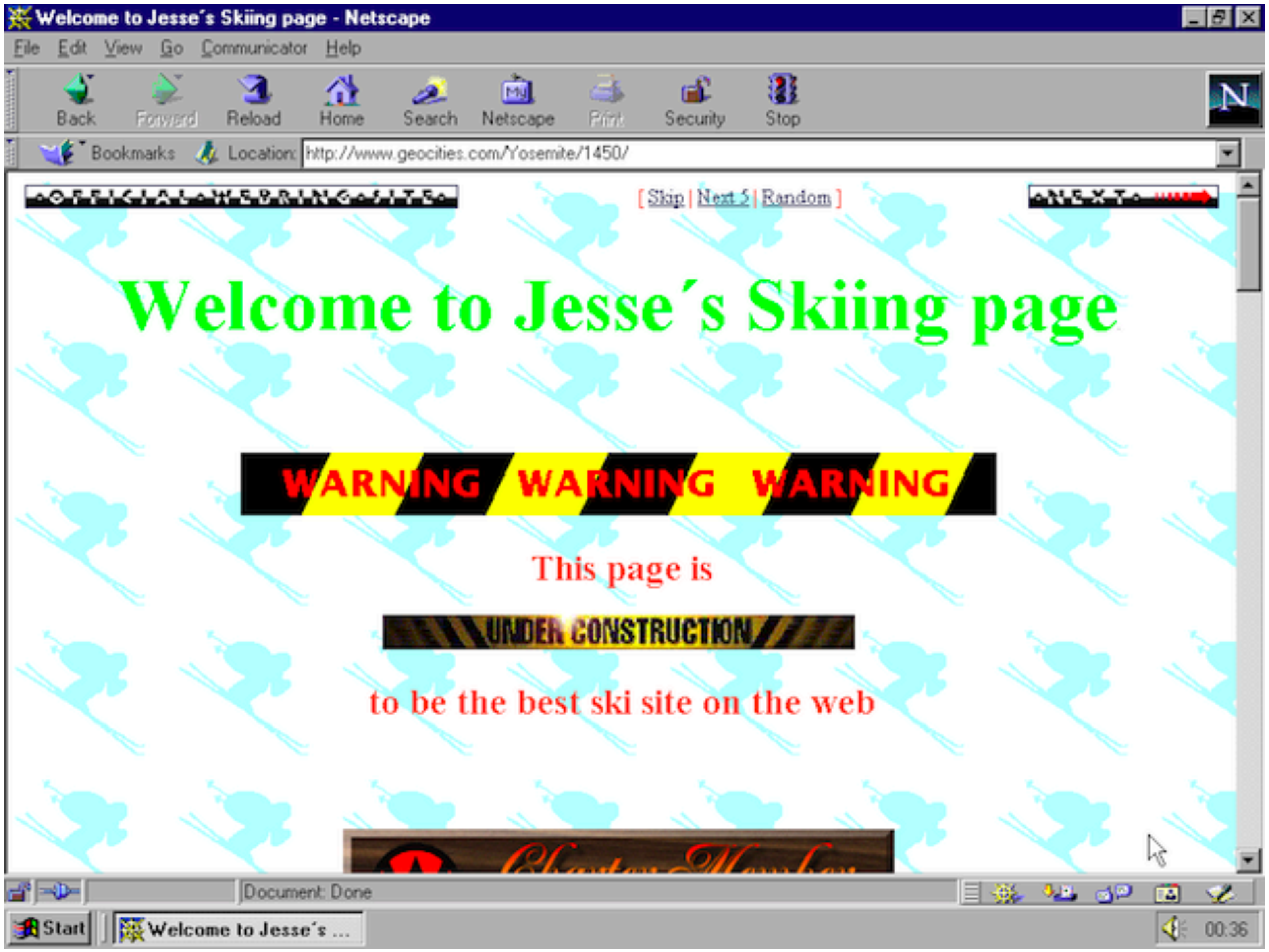
- I will be accepting students into a 3 credit Research course for the spring… but you need to do a cloud/NFV project and it needs to be done well! Impress me!

If you don't do a project, you need to write a technical "blog" post explaining a cloud technology

Antique Web Servers

Serve static content

- Read a file from disk and send it back to the client
- images, HTML



Dynamic Content

- CGI Bin
- executes a program
- Not very safe or convenient for development…

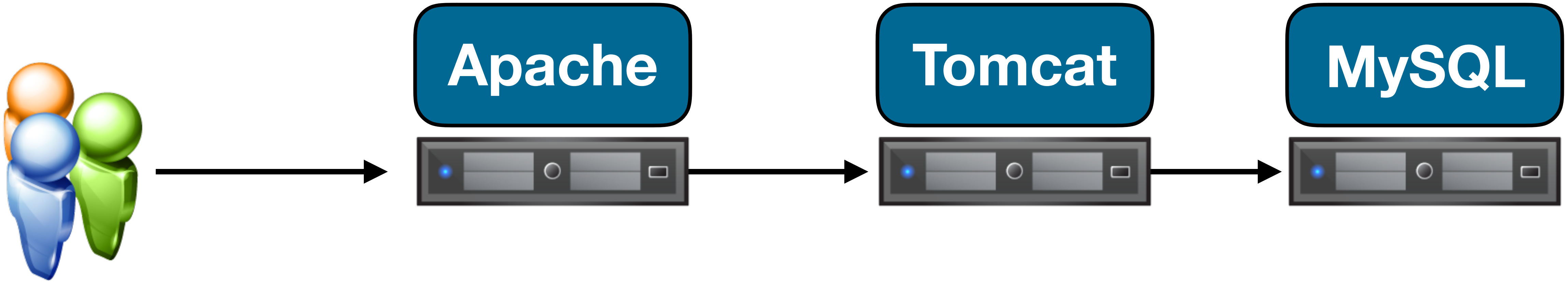
3-tier Web Applications

LAMP = Linux, Apache, MySQL, PHP

Separation of duties:

- Front-end web server for static content (Apache, lighttpd, nginx)
- Application tier for dynamic logic (PHP, Tomcat, node.js)
- Database back-end holds state (MySQL, MongoDB, Postgres)

Why divide up in this way?



Stateful vs Stateless

The multi-tier architecture is based largely around whether a tier needs to worry about state

Front-end - totally **stateless**

- There is no data that must be maintained by the server to handle subsequent requests

Application tier - maintains **per-connection state**

- There is some temporary data related to each user, e.g., my shopping cart
- May not be critical for reliability - might just store in memory

Database tier - global state

- Maintains the global data that application tier might need
- Persists state and ensures it is consistent

N-Tier Web Applications

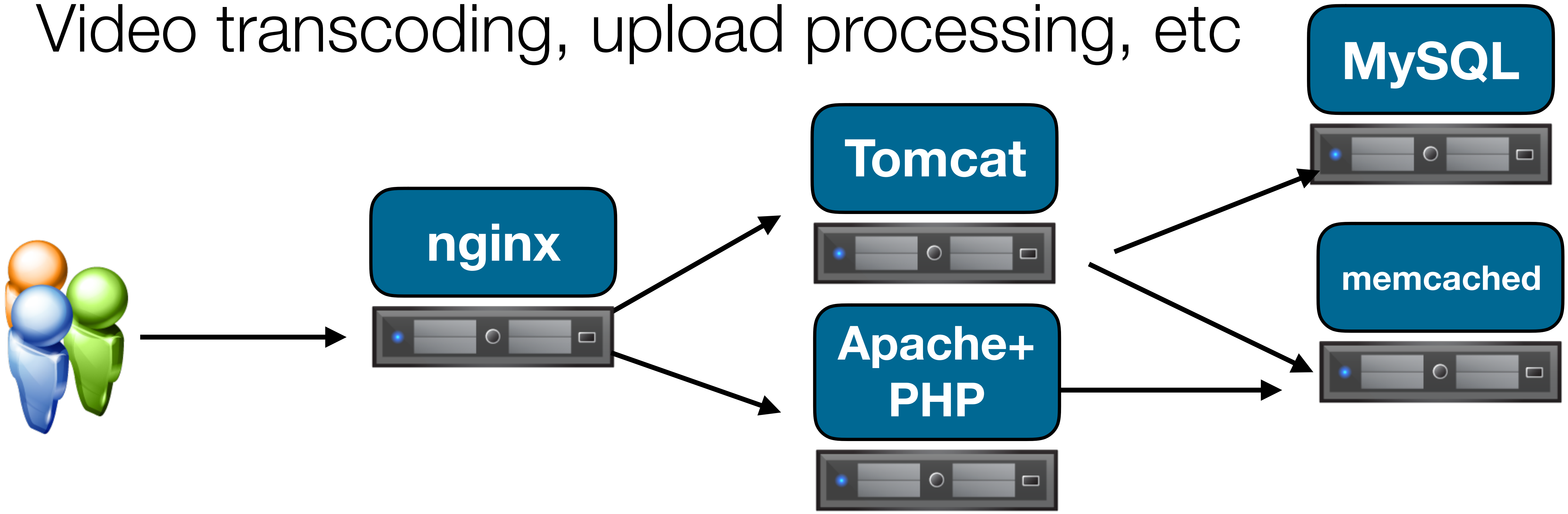
Sometimes 3 tiers isn't quite right

Database is often a bottleneck

- Add a cache! (stateful, but not persistent)

Authentication or other security services could be another tier

Video transcoding, upload processing, etc

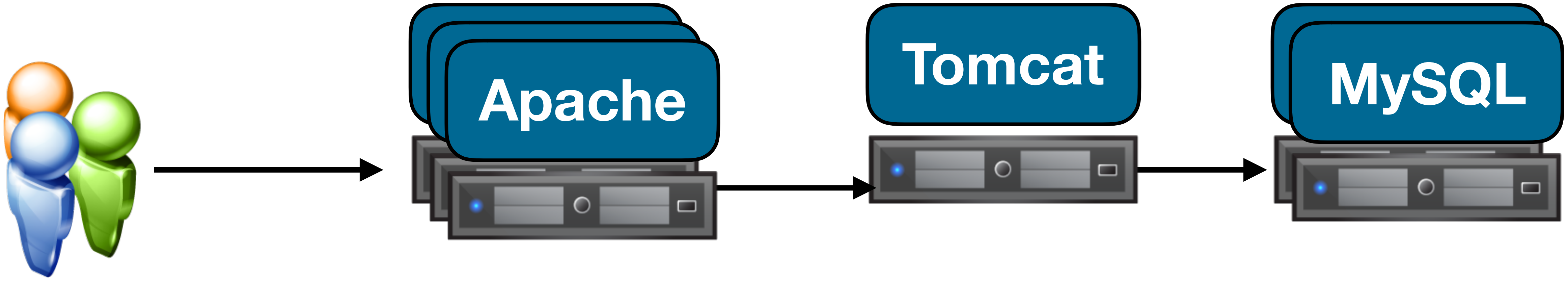


Replicated N-Tier

Replicate the portions of the system that are likely to become overloaded

How easy to scale…?

- Apache serving static content
- Tomcat Java application managing user shopping carts
- MySQL cluster storing products and completed orders



Tune number of replicas based on demand at each tier

Wikipedia: Big scale, cheap

5th busiest site in the world (according to alexa.com)

Runs on about **~ 1000** servers? (700 in 2012)

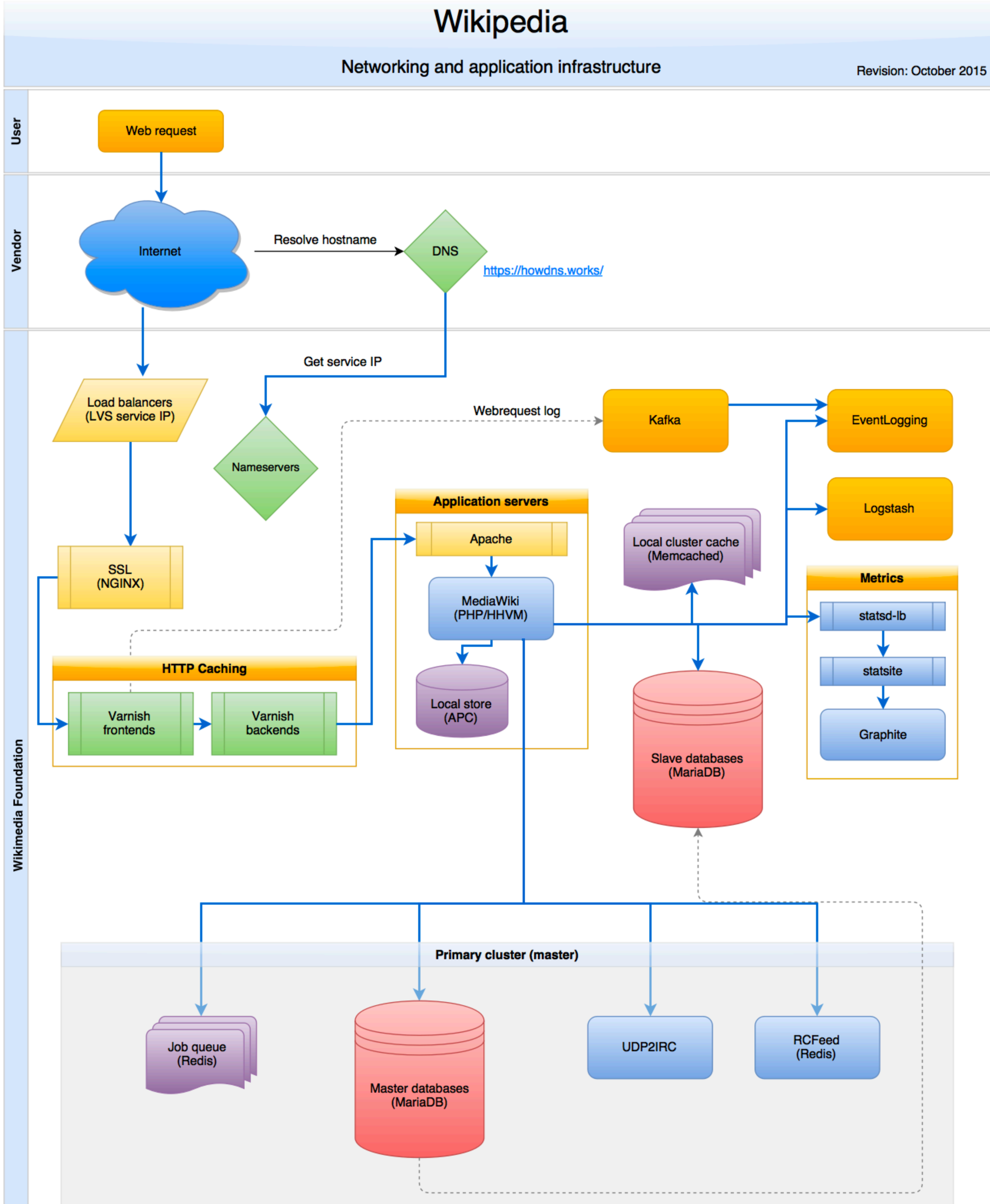
All open source software:

- PHP, MariaDB, Squid proxy, memcached, Ubuntu

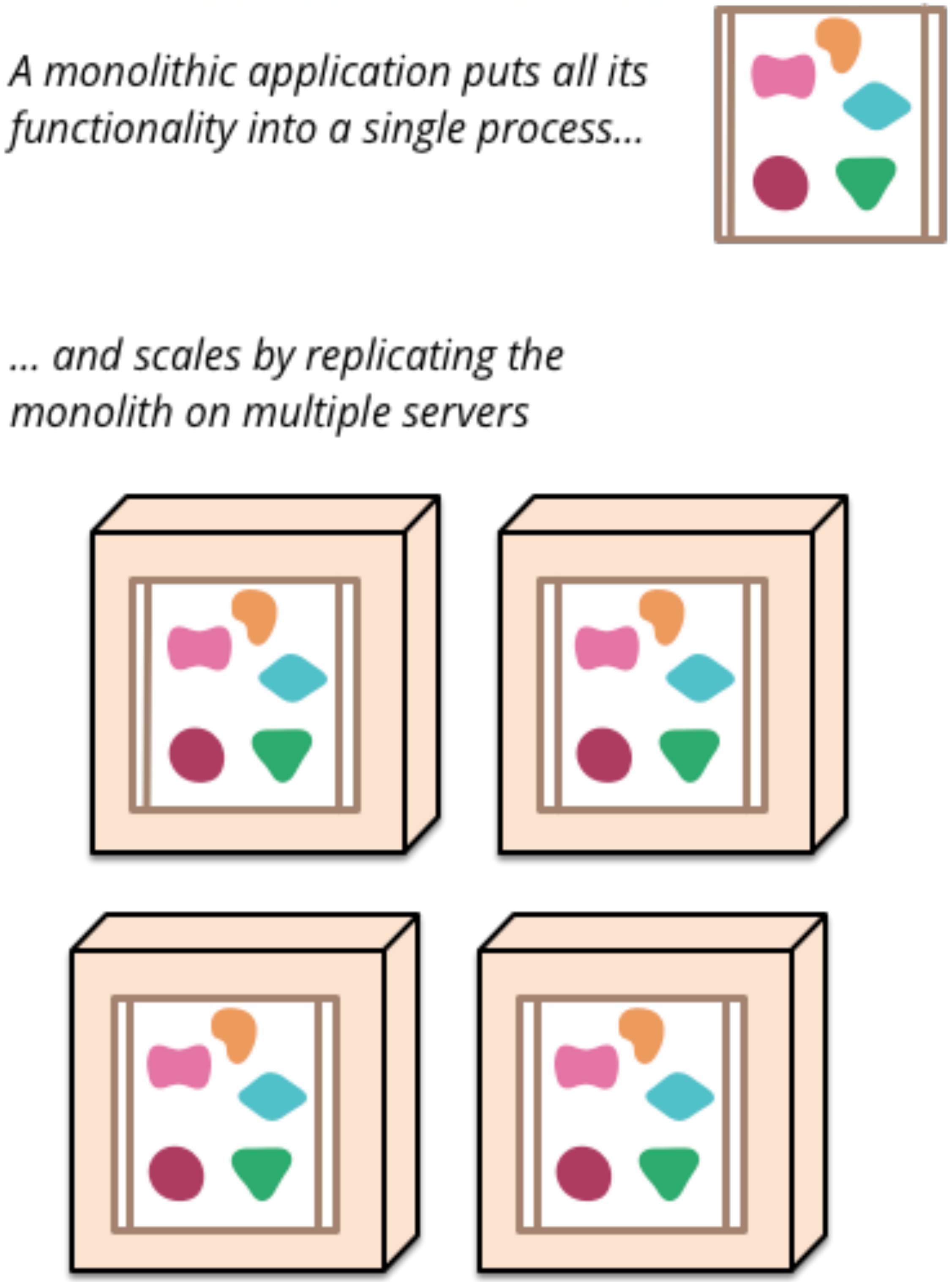
Goals:

- Store lots of content (6TB of text data as of 2018)
- Make available worldwide
- Do this as cheaply as possible
- Relatively weak consistency guarantees

Stats: https://grafana.wikimedia.org


Wikipedia
Networking and application infrastructure
Revision: October 2015
User
Vendor
Wikimedia Foundation
Web request
Internet
Resolve hostname
DNS
https://howdns.works/
Get service IP
Load balancers
(LVS service IP)
Nameservers
Webrequest log
Kafka
EventLogging
Logstash
Application servers
Apache
MediaWiki
(PHP/HHVM)
Local store
(APC)
Local cluster cache
(Memcached)
SSL
(NGINX)
HTTP Caching
Varnish
frontends
Varnish
backends
Slave databases
(MariaDB)
Metrics
statsd-lb
statsite
Graphite
Primary cluster (master)
Job queue
(Redis)
Master databases
(MariaDB)
UDP2IRC
RCFeed
(Redis)

Application Tier



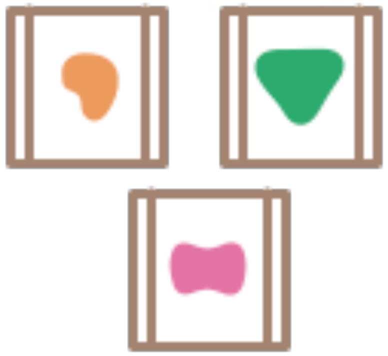
http://martinfowler.com/articles/microservices.html

Problems with Monolithic approach?

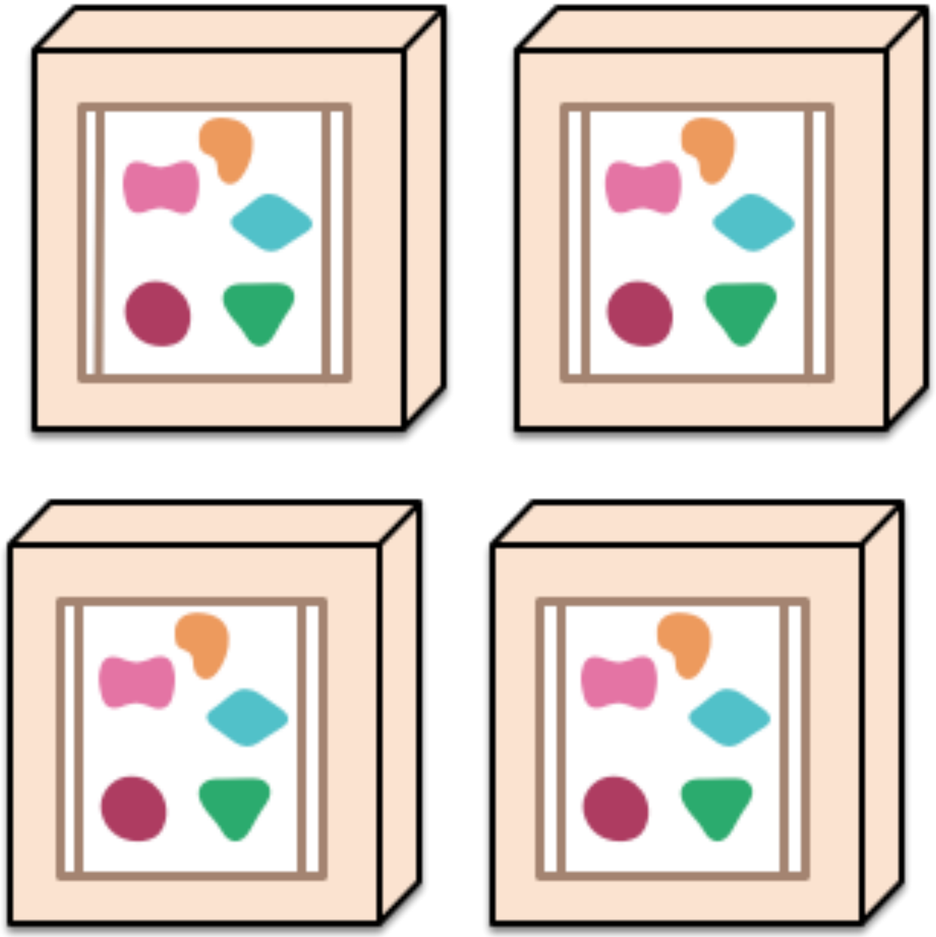
Microservices

A monolithic application puts all its functionality into a single process...

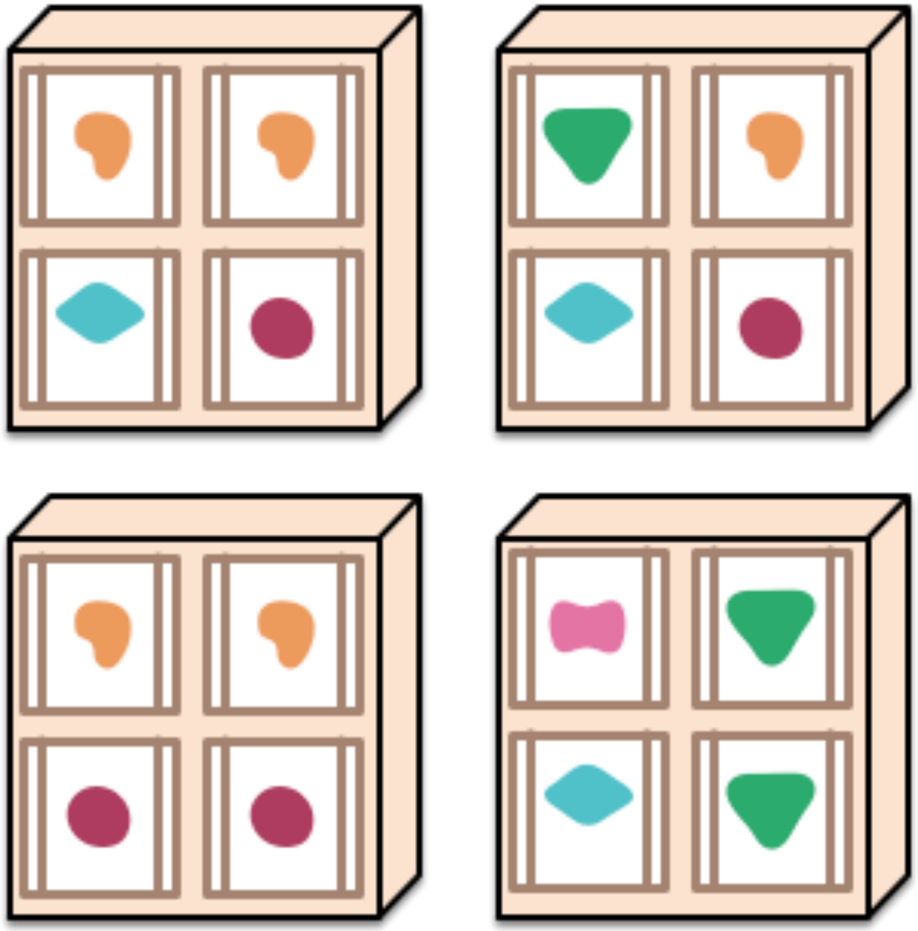
A microservices architecture puts each element of functionality into a separate service...



... and scales by replicating the monolith on multiple servers



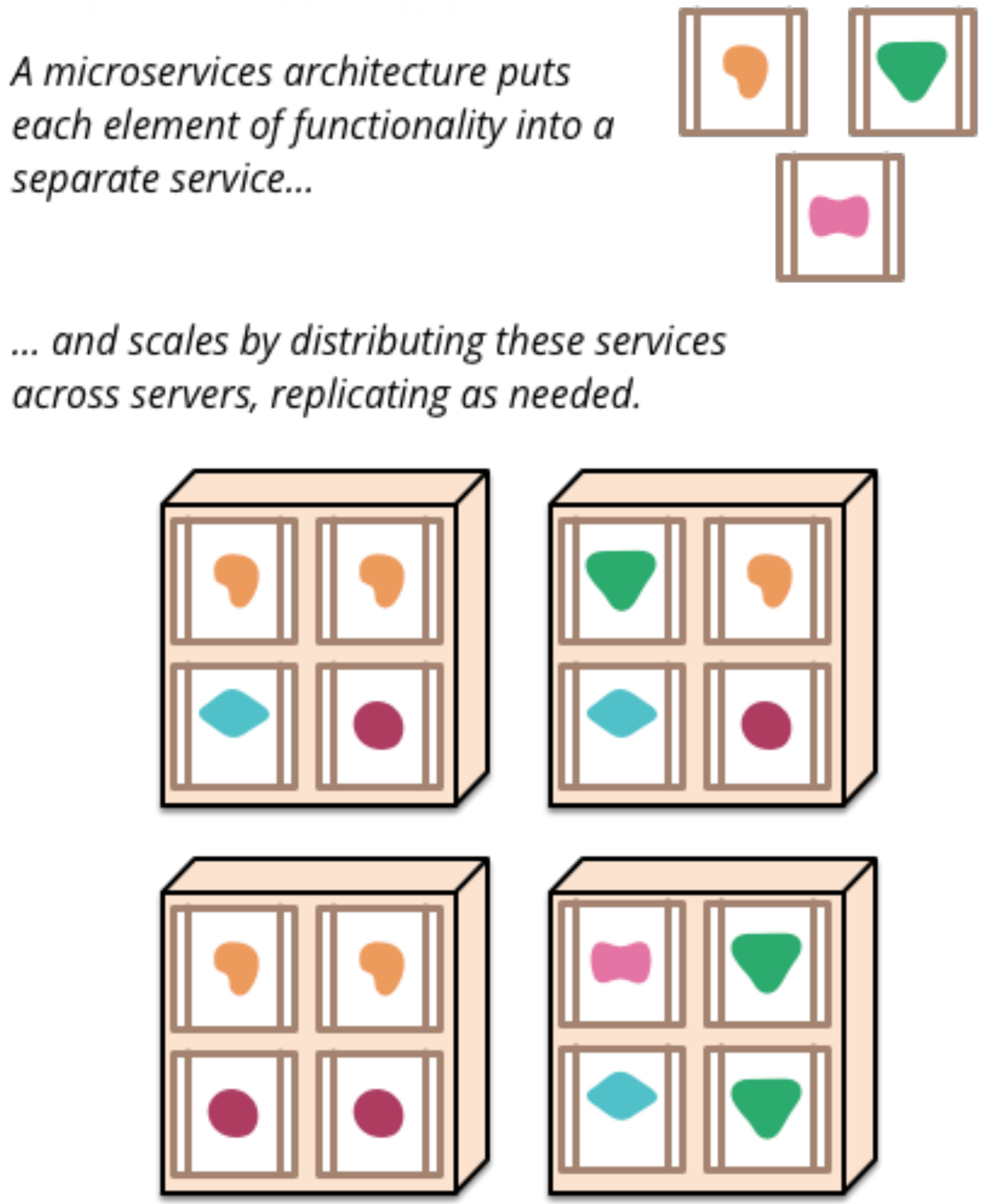
... and scales by distributing these services across servers, replicating as needed.



Read more: https://martinfowler.com/articles/microservices.html

Microservices

Challenges with Microservices approach?



Microservices Challenges

Discovery: how to find a service you want?

Scalability: how to replicate services for speed?

Openness: how to agree on a message protocol?

Fault tolerance: how to handle failed services?

All distributed systems face these challenges, microservices just increases the scale and diversity…

Netflix

26th most popular website according to Alexa

Zero of their own servers

- All infrastructure is on AWS (2016-2018)
- Recently starting to build out their own Content Delivery Network

Netflix

One of the first to really push microservices

- Known for their DevOps
- Fast paced, frequent updates, must always be available

700+ microservices

Deployed across 10,000s of VMs and containers

Netflix ecosystem

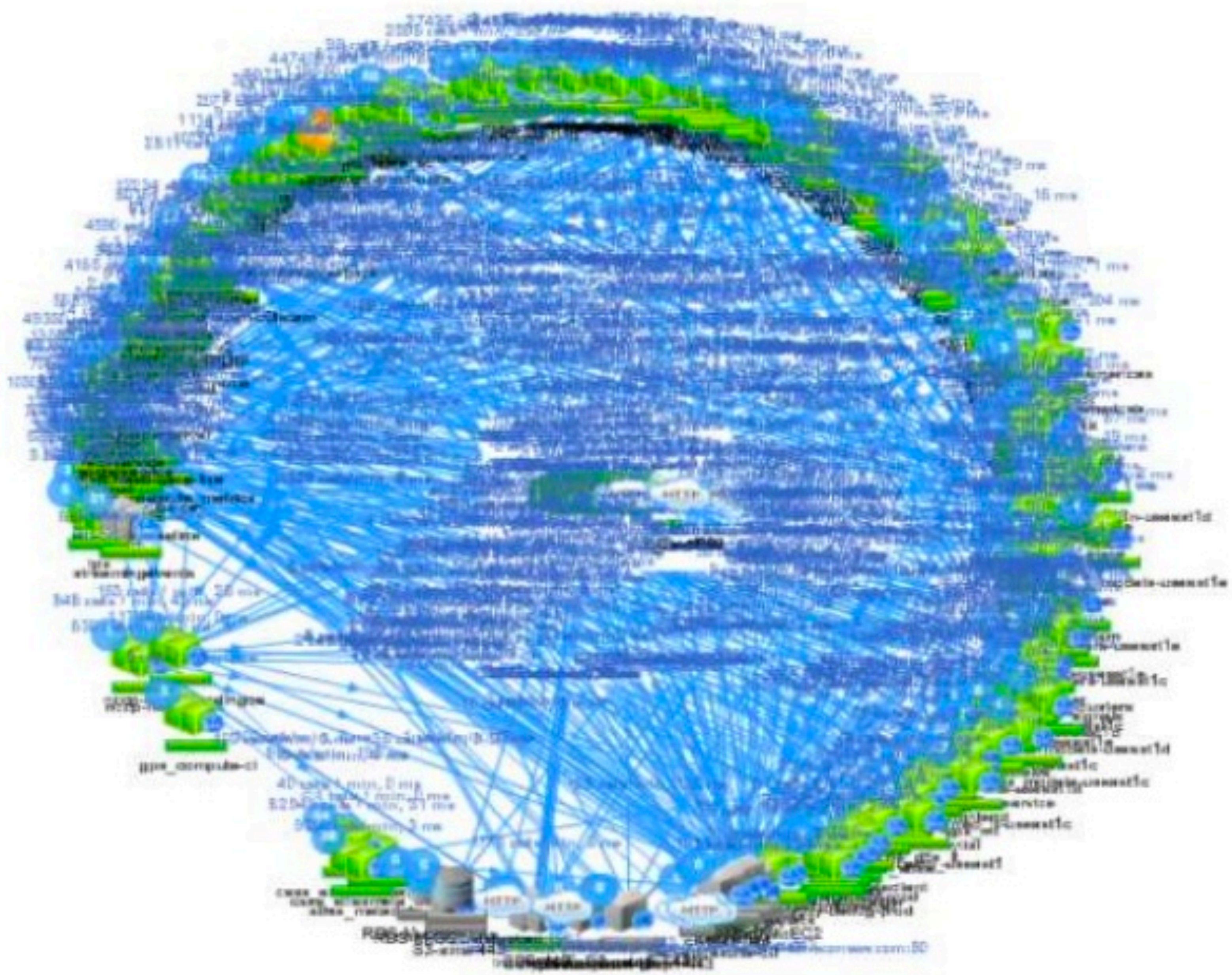
100s of microservices
1000s of daily production changes
10,000s of instances
100,000s of customer interactions per minute
1,000,000s of customers
1,000,000,000s of metrics
10,000,000,000 hours of streamed
10s of operations engineers

Netflix tech talk: https://www.youtube.com/watch?v=CZ3wIuvmHeM

Netflix "Deathstar"

Microservice architecture results in a extremely distributed application

- Can be very difficult to manage and understand how it is working at scale



How to know if everything is working correctly?

Netflix Chaos Monkey

Idea: If my system can handle failures, then I don't need to know exactly how all the pieces themselves interact!

Chaos Monkey:

- Randomly terminate VMs and containers in the production environment
- Ensure that the overall system keeps operating
- Run this 24/7

Make failures the common case, not an unknown!

http://principlesofchaos.org/

Distributed Systems CS6421

Scaling the Web (Part 2)

Prof. Tim Wood

Serverless Computing

Trendy architecture that improves the agility of microservices

What does “serverless” mean?

APACHE OpenWhisk™

AWS Lambda

Azure Functions

Google Cloud Functions

Serverless Computing

Trendy architecture that improves the agility of microservices

What does “serverless” mean?

You still need a server!

BUT, your services will not always be running

Key idea: only instantiate a service when a user makes a request for that functionality

How will this work for stateful vs stateless services?

Serverless Startup

AWS Lambda

- Define a stateless "function" to execute for each request
- A container will be instantiated to handle the first request
- The same container will be used until it times out or is killed

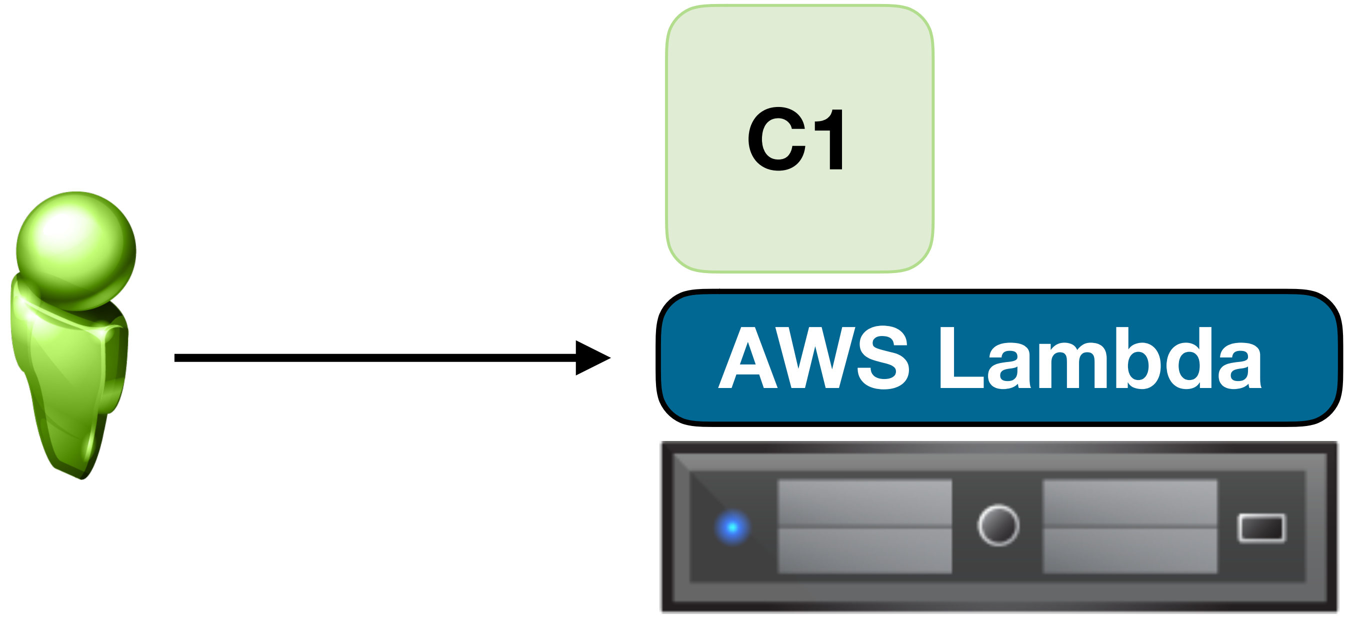
No workload means no resources being used!

Serverless Startup

AWS Lambda

- Define a stateless “function” to execute for each request
- A container will be instantiated to handle the first request
- The same container will be used until it times out or is killed

Request arrives, start green container

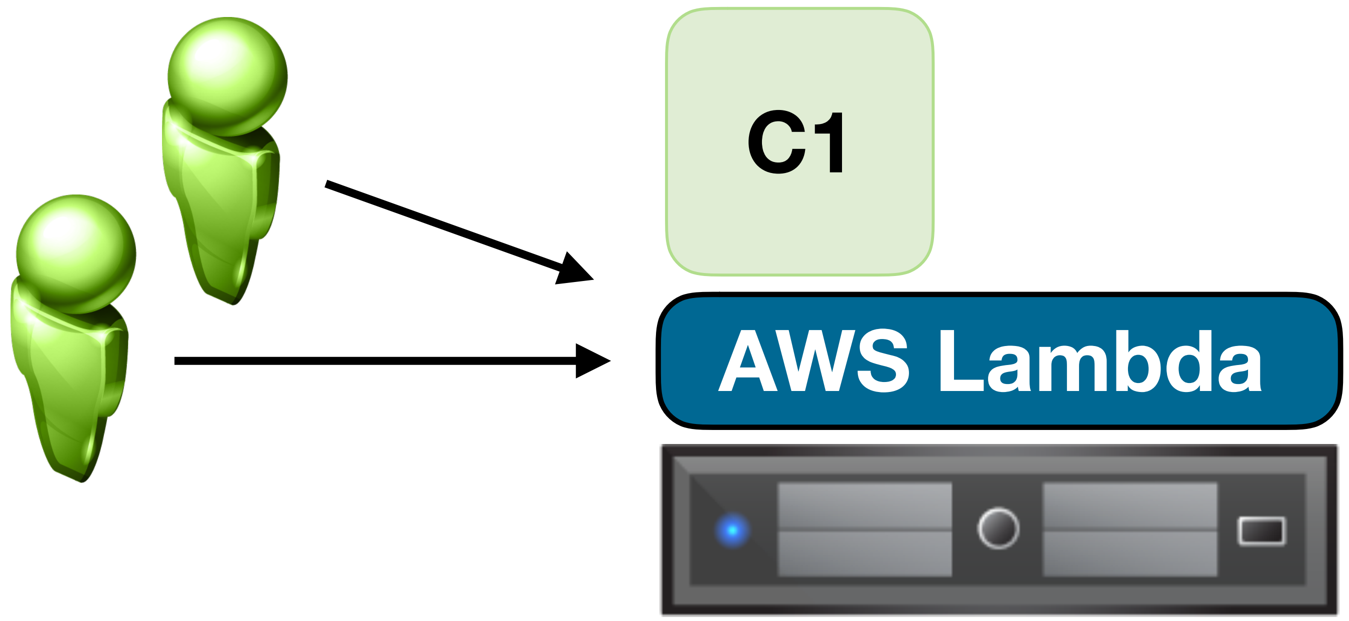


Serverless Startup

AWS Lambda

- Define a stateless "function" to execute for each request
- A container will be instantiated to handle the first request
- The same container will be used until it times out or is killed

Reuse that container for subsequent requests

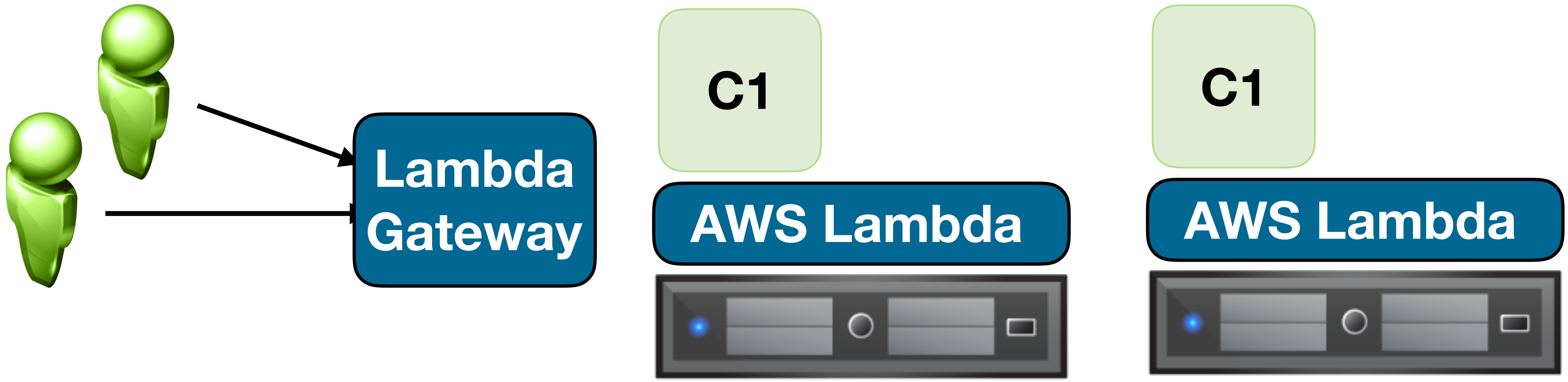


Serverless Startup

AWS Lambda

- Define a stateless “function” to execute for each request
- A container will be instantiated to handle the first request
- The same container will be used until it times out or is killed

Reuse that container for subsequent requests

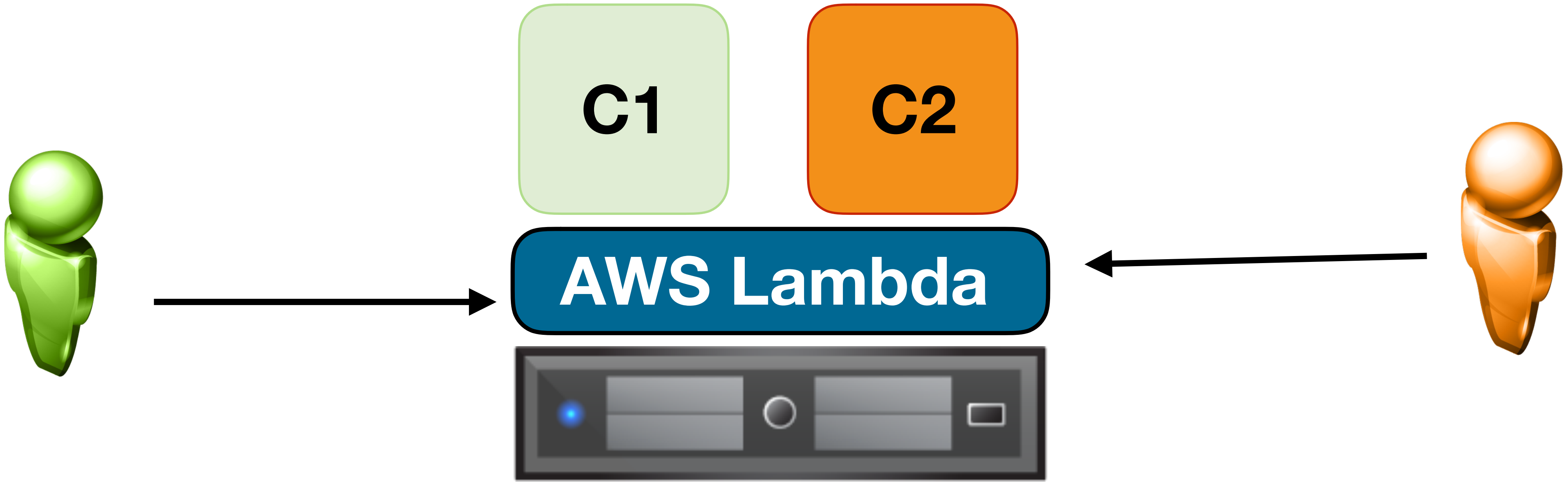


Serverless Startup

AWS Lambda

- Define a stateless “function” to execute for each request
- A container will be instantiated to handle the first request
- The same container will be used until it times out or is killed

Start new container if user needs a different function

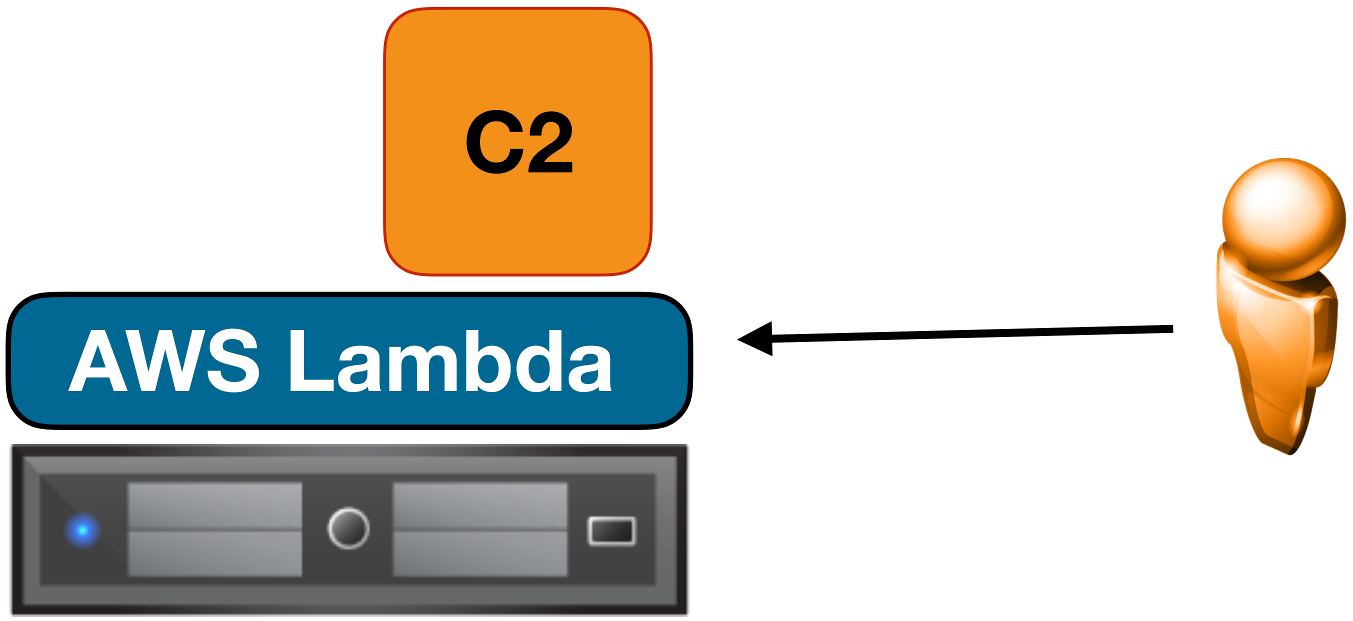


Serverless Startup

AWS Lambda

- Define a stateless “function” to execute for each request
- A container will be instantiated to handle the first request
- The same container will be used until it times out or is killed

Clean up old containers once not in use



Serverless Pros/Cons

Benefits:

- Simple for developer when auto scaling up
- Pay for exactly what we use (at second granularity)
- Efficient use of resources (auto scale up and down based on requests)
- don’t worry about reliability/server management at all

Drawbacks:

- Limited functionality (stateless, limited programming model)
- High latency for first request to each container
- Some container layer overheads plus the lambda gateway and routing overheads
- Potentially higher and unpredictable costs
- Difficult to debug / monitor behavior
- Security

Serverless Pros/Cons

Benefits:

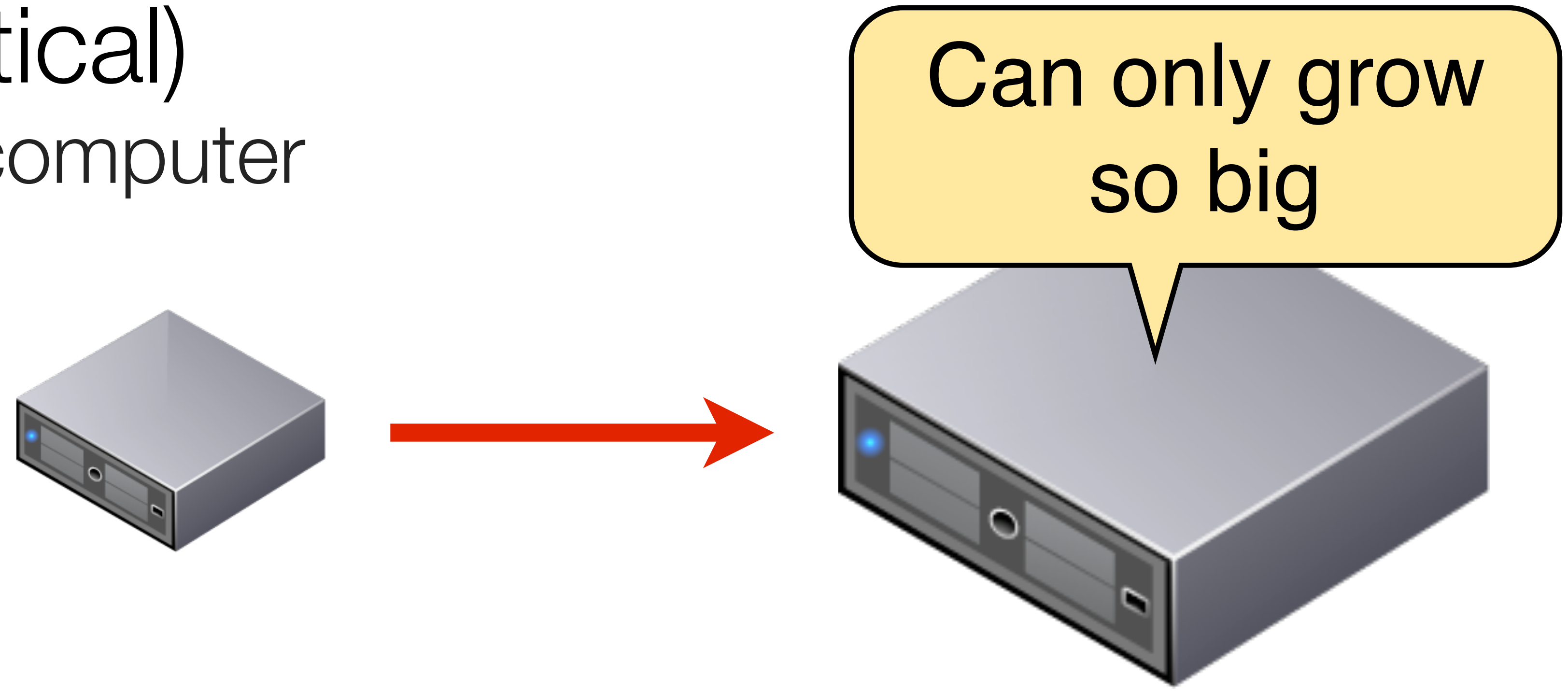
Drawbacks:

Scaling

Two ways to scale

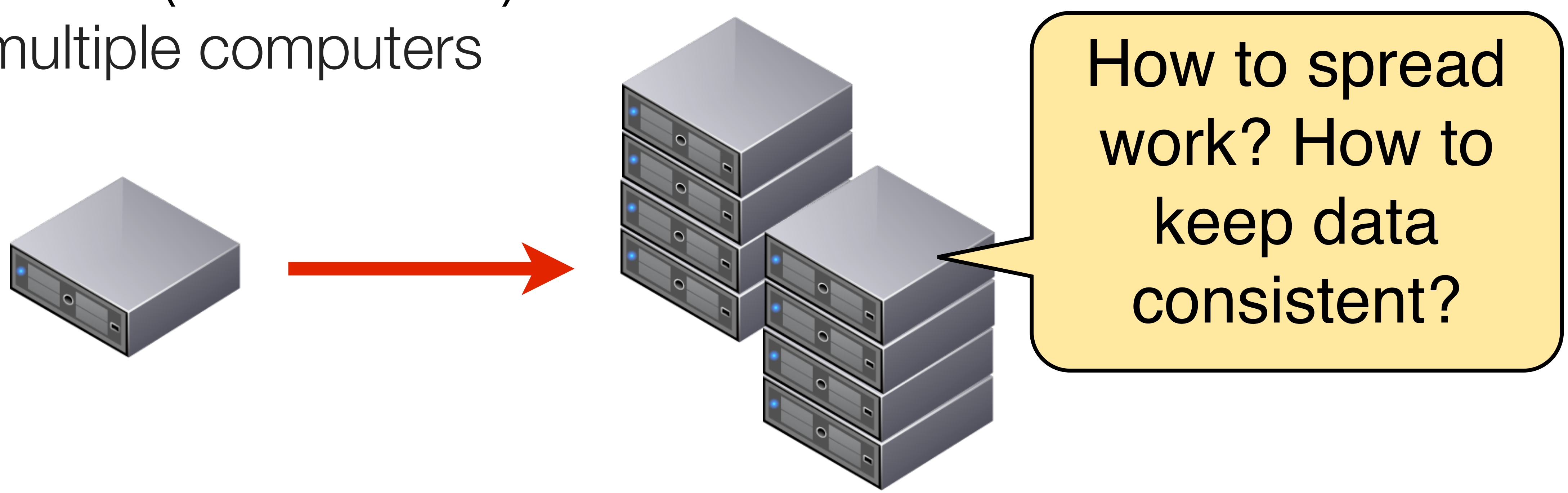
Scale UP (vertical)

- Buy a bigger computer

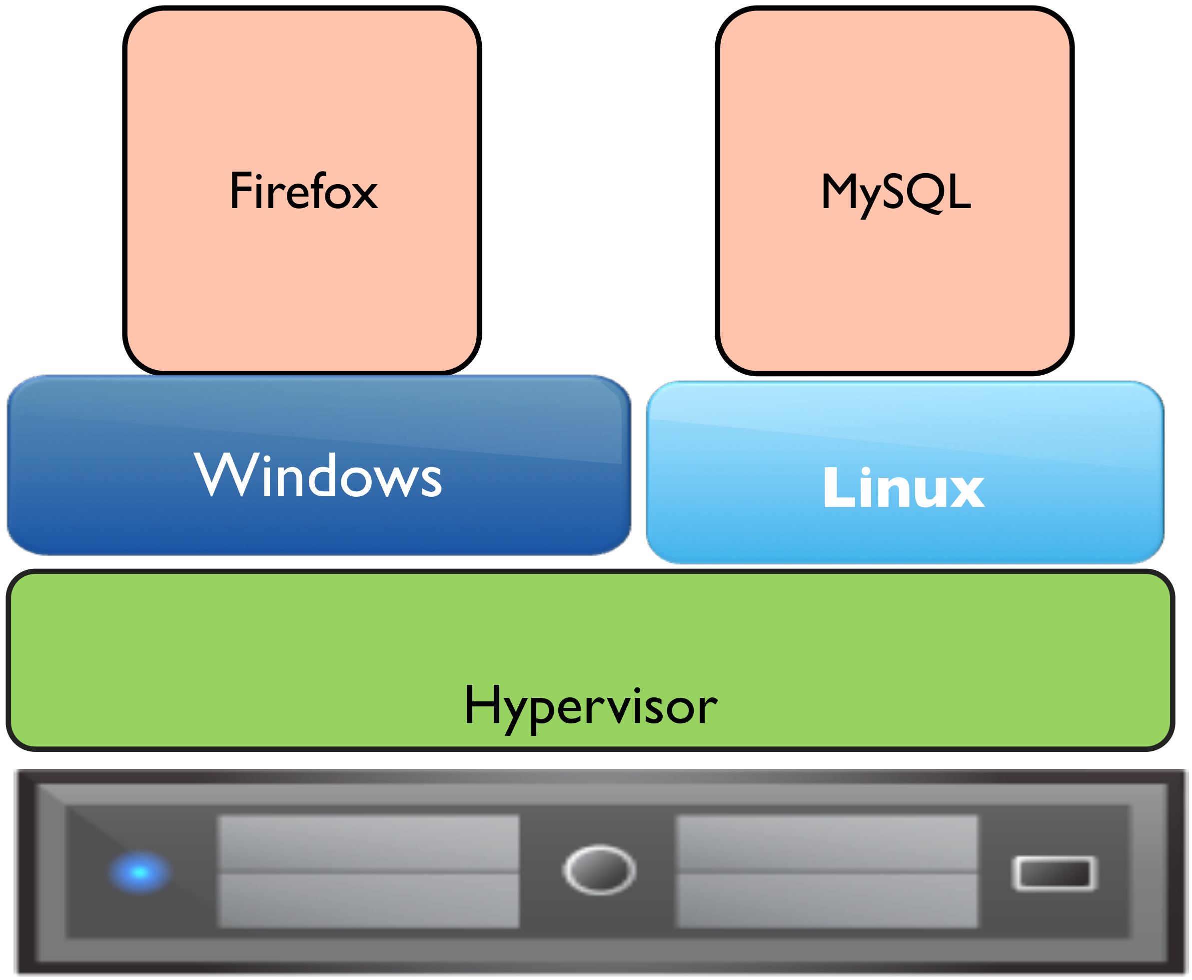


Scale OUT (horizontal)

- Buy multiple computers



Does virtualization help?



Does virtualization help?

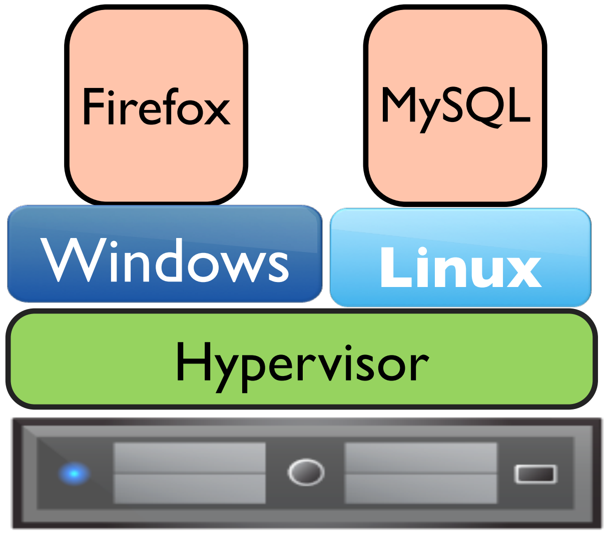
Not exactly...

Virtualization divides something big into smaller pieces

but still has features which can assist with scalability:

- Easy replication of VM images
- Dynamic resource management

Simplifies scale OUT, but has limits on how much you can scale UP

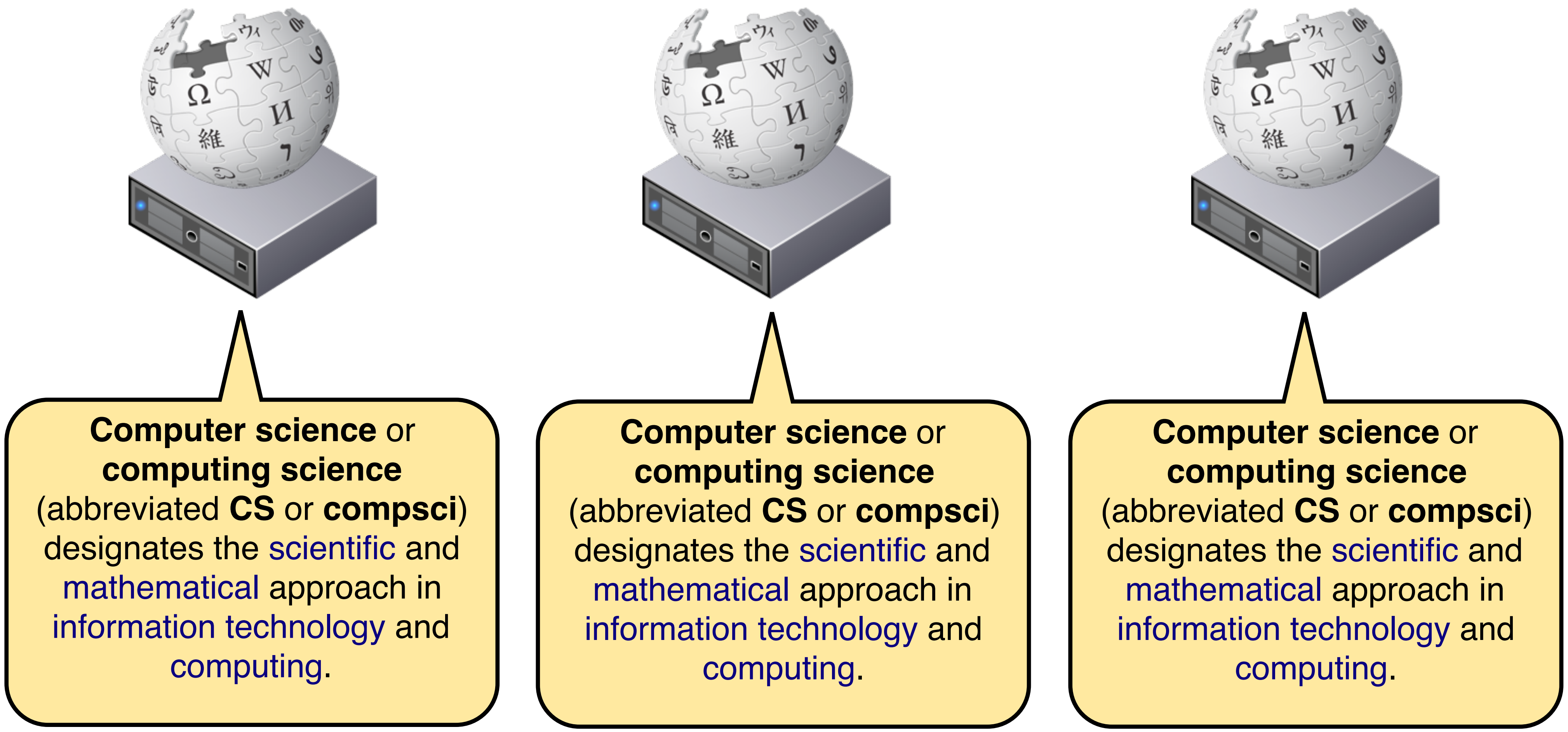


Replication

Scale Out v1

Biggest Challenge: Consistency

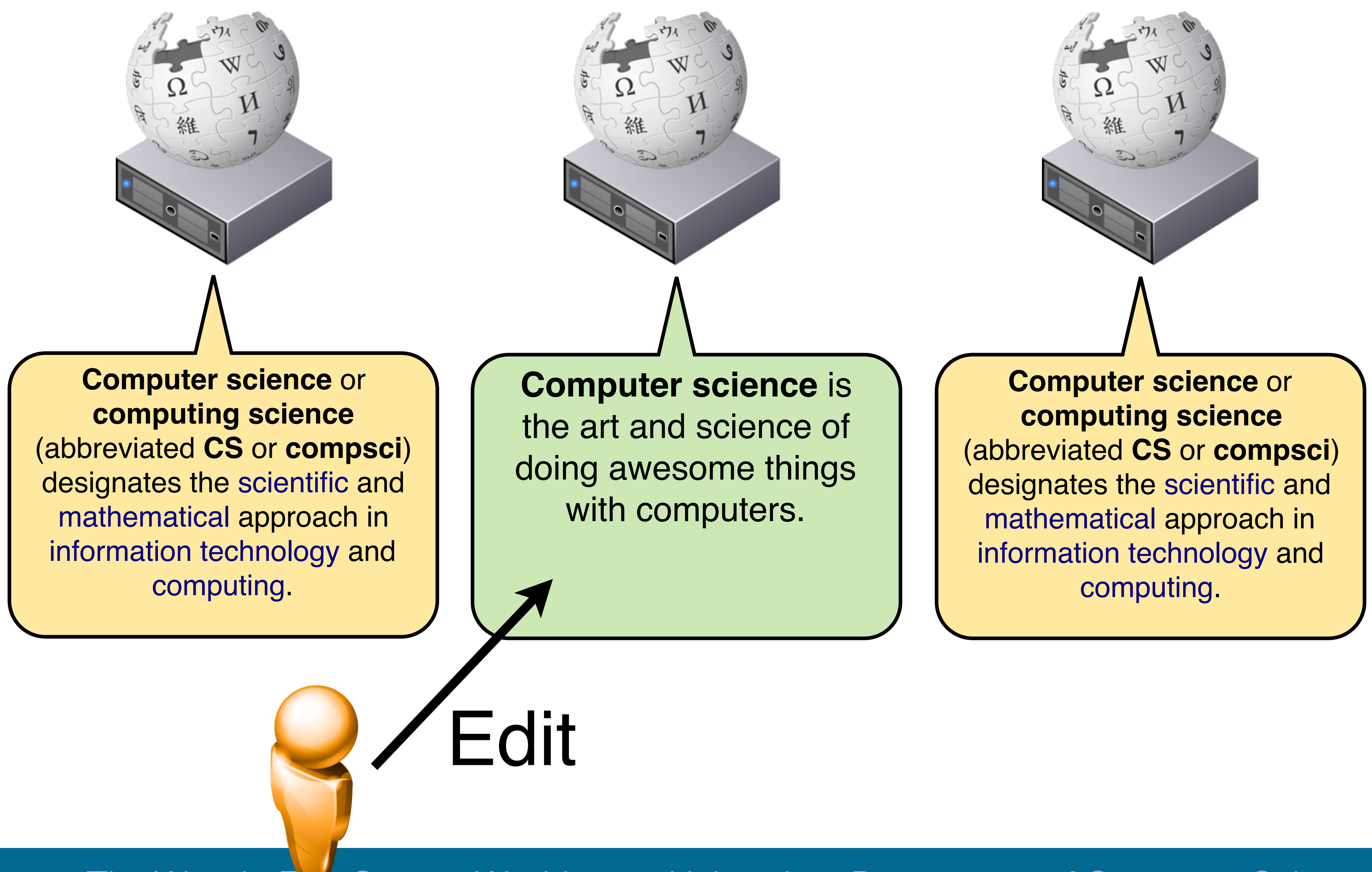
Replicating data makes it faster to access



Biggest Challenge: Consistency

Replicating data makes it faster to access

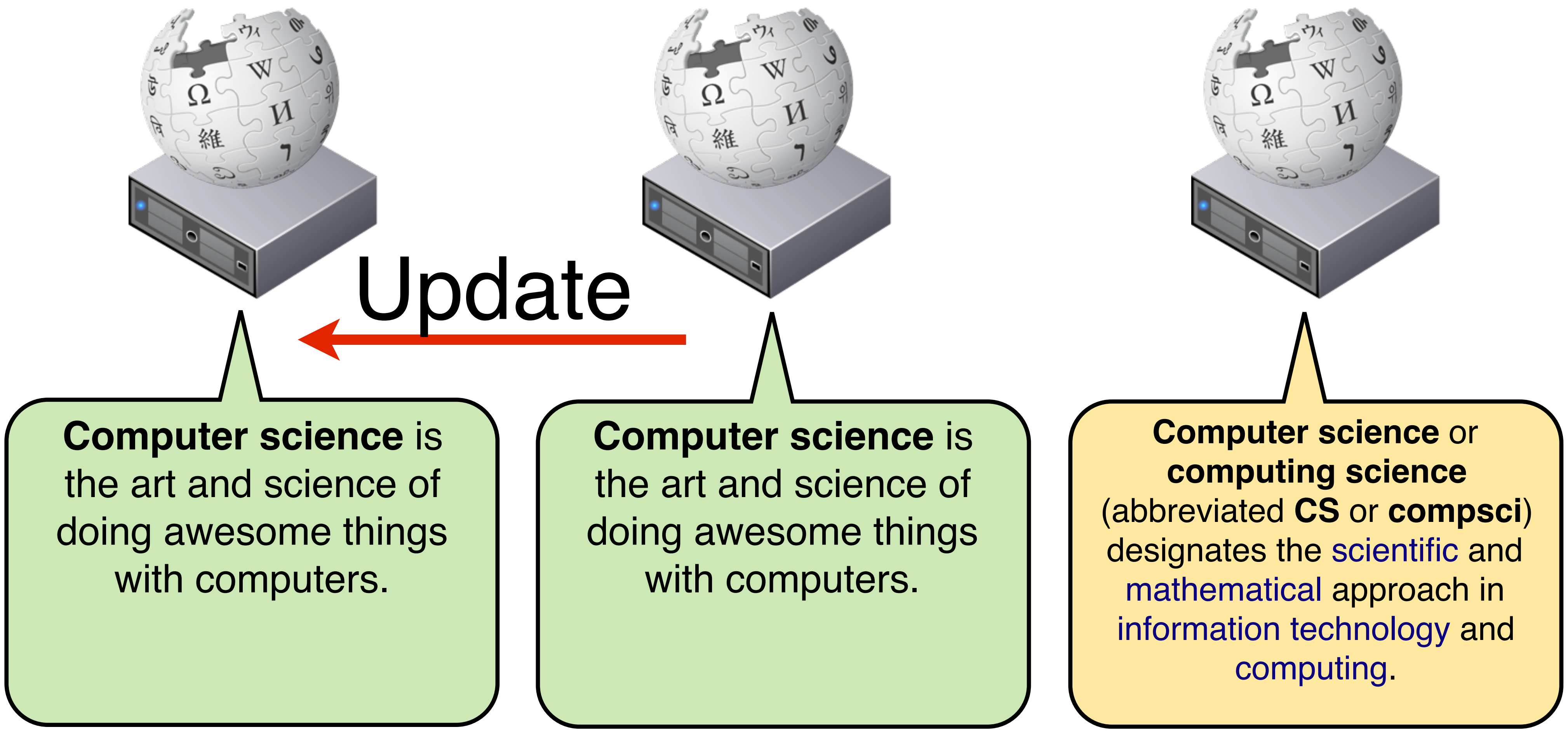
- But how to keep all copies of data consistent?



Biggest Challenge: Consistency

Replicating data makes it faster to access

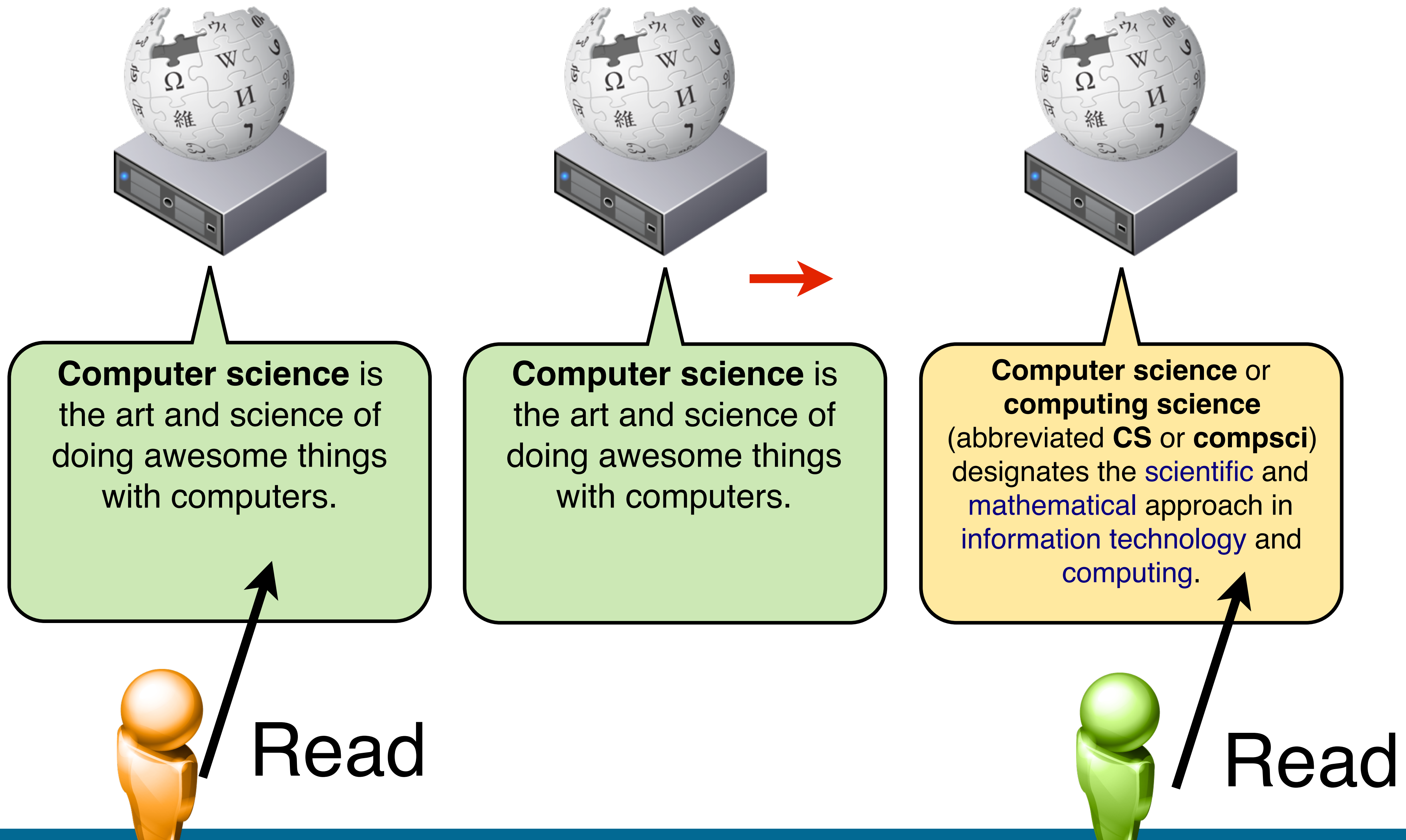
- But how to keep all copies of data consistent?



Biggest Challenge: Consistency

Replicating data makes it faster to access

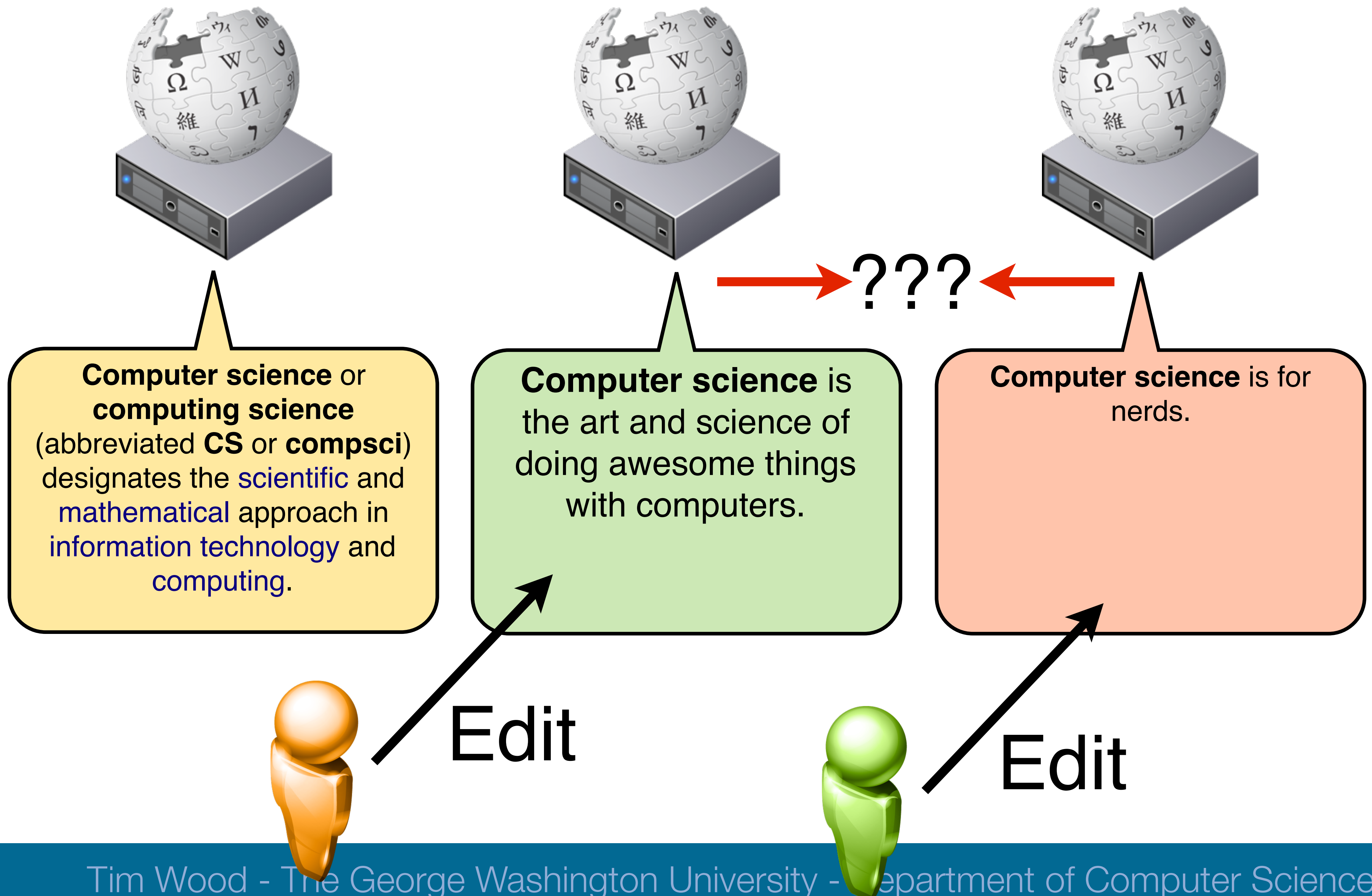
- But how to keep all copies of data consistent?



Biggest Challenge: Consistency

Writes are even harder

- Would need time stamps or a consistent ordering
- Or, if writes are rare, just have a master coordinate



Does it Matter?

A slightly out of date wikipedia page?

A post to your facebook profile?

1. Remove boss from friends list
2. Post "My boss is a moron, I want a new job!"

A change to a stock price in the NASDAQ exchange?

Providing Consistency

We have already seen techniques that will help:

- Version vectors
- Distributed locking based on Lamport Clocks
- Election-based systems with a master/slave setup

There are many different types of **consistency**

- **Strict** - updates immediately available after a write
- **Sequential** - result of parallel updates needs to have the same effect as if they had been done sequentially
- **Causal** - updates that are casually related (e.g., where vector clocks can prove the -> relationship) are ordered sequentially, but others may not be

  … (several more) …
- **Eventual** - updates will converge so at some point reads to any replica will get the same result

End of Semester

Practice 2 / Projects — Sunday 12/9 (extended)

Exam — Friday 11/30

- Concepts from lecture
- 8.5x11 page (two sided) of hand written notes
- I will post some practice questions this weekend

Partitioning

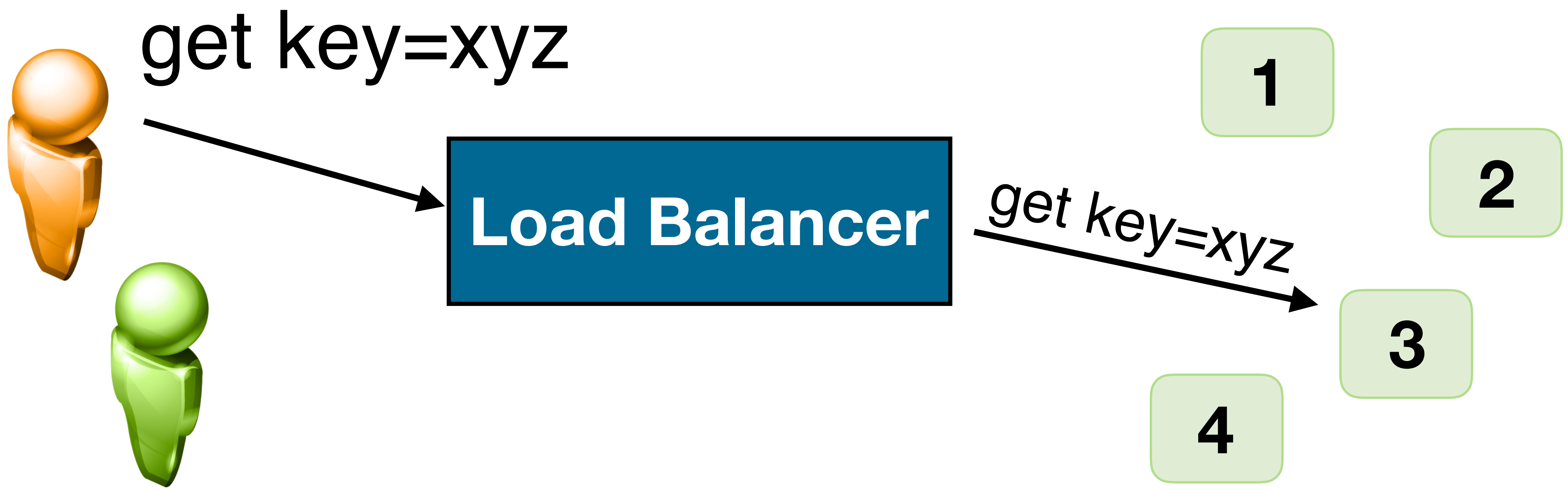
Scale Out v2

Spread data across servers

Useful if all data does not fit on one server

Let's consider a Key Value store like Memcached

- Lots of data to store
- Consistency is not that important
- Might need to add or remove nodes to the cluster
- **How should we partition the keys across the nodes?**



DHTs

A **Distributed Hash Table** is a key-value store that can be implemented in a **Peer-2-Peer** fashion.

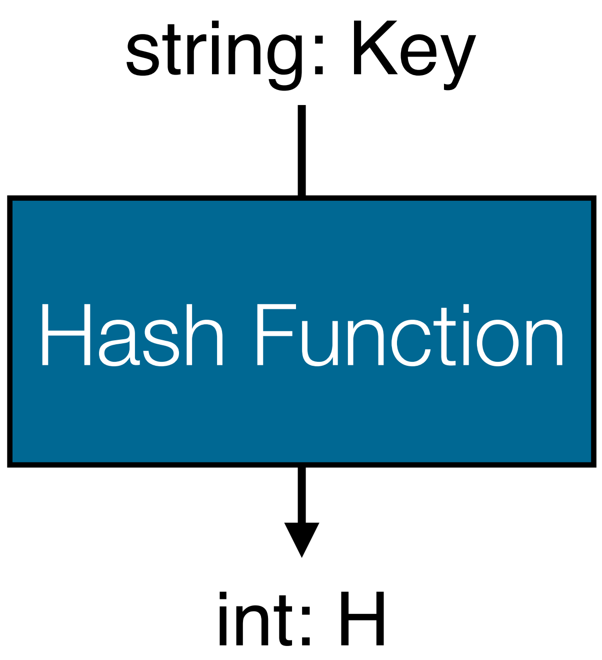
Goals:

- Evenly **partition** data across the nodes
- Efficient lookups
- Gracefully handle nodes leaving and joining

DHTs

A **Distributed Hash Table** is a key-value store that can be implemented in a P2P fashion.

Simple Hash Table



Value is stored in
`array[H % S]`

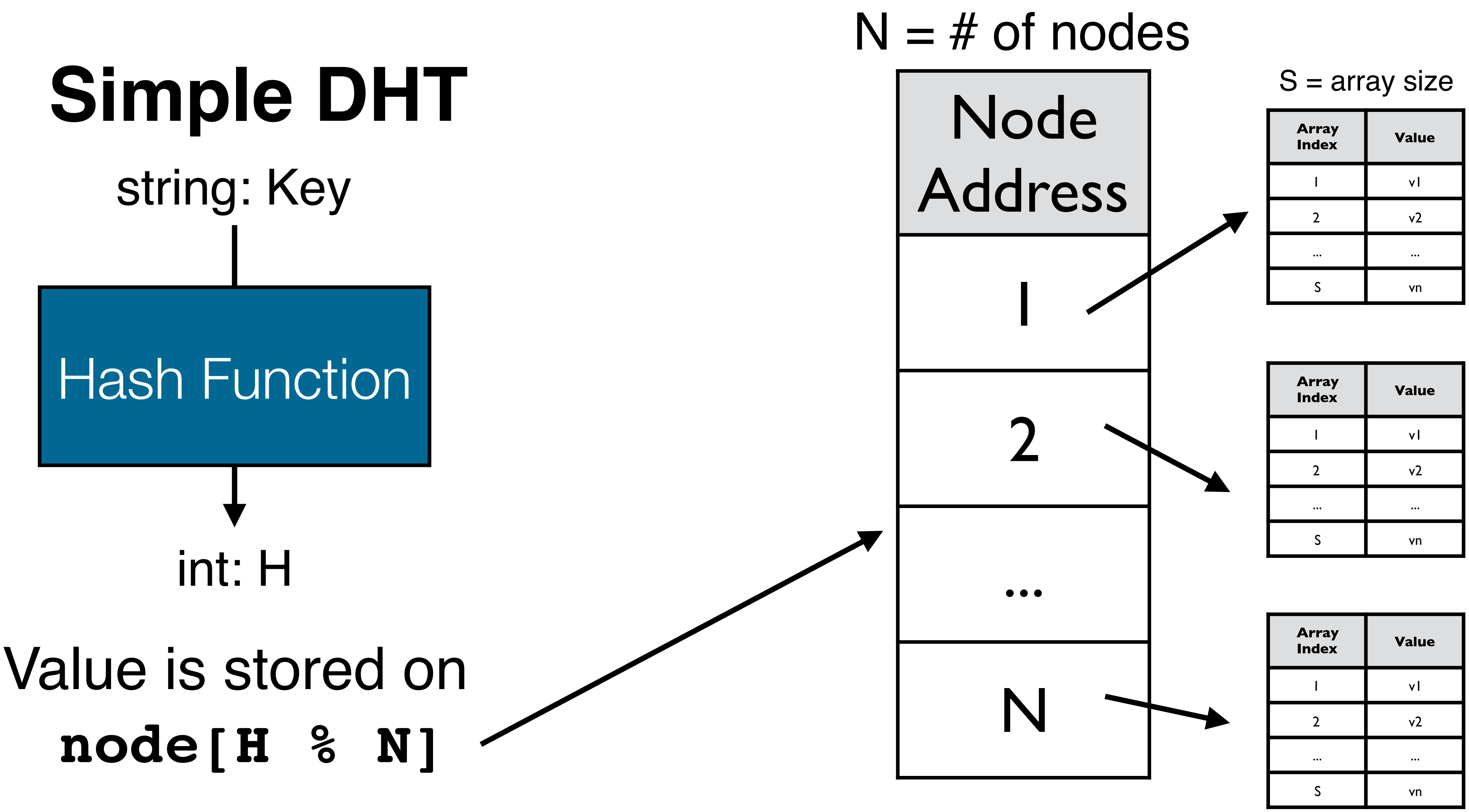
S = array size

Array Index	Value
1	v1
2	v2
…	…
S	vs

DHTs

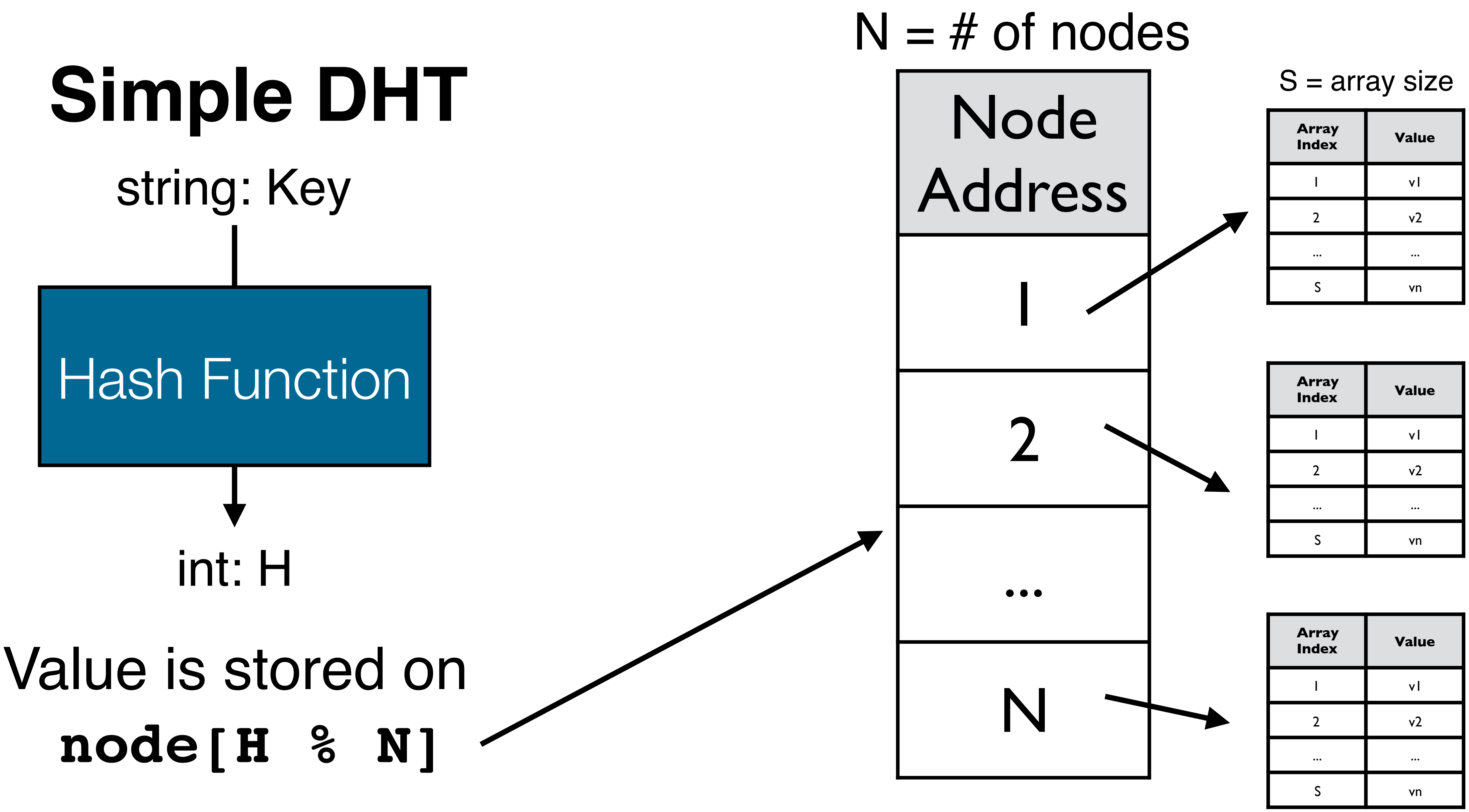
What if one node can't fit all the data?

Do two hash lookups!



DHTs

When will this perform poorly?



Churn

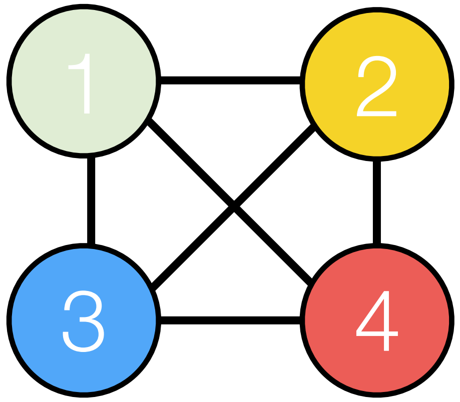
Churn is when nodes are frequently joining or leaving

- In a DHT it is OK to lose data when a node leaves, but it shouldn't cause all other nodes to reshuffle their data!

Simple DHT

Value is stored on

`node[H % 4]`



Hash Space
0
...
...
...
...
...
...
MAX

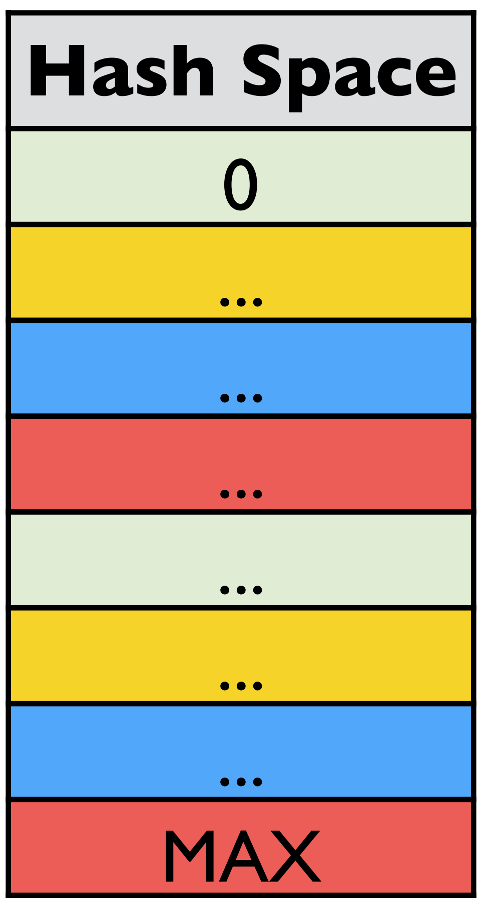
Divides hash space into 4 equal partitions for 4 servers

Churn

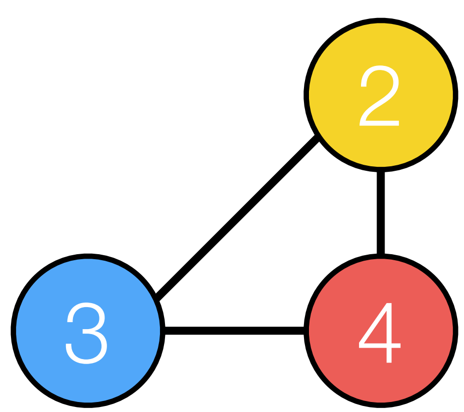
Churn is when nodes are frequently joining or leaving

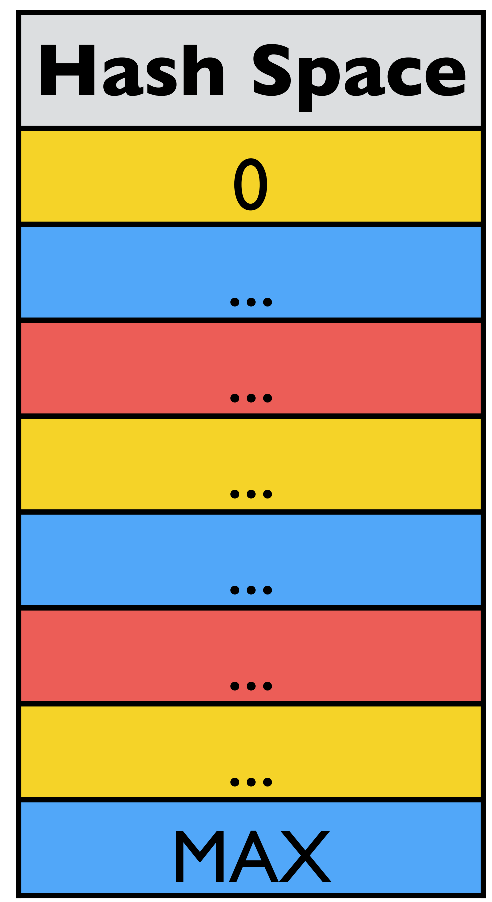
- In a DHT it is OK to lose data when a node leaves, but it shouldn't cause all other nodes to reshuffle their data!

Simple DHT



Oops! Green node failed!





All nodes needs to be reorganized!

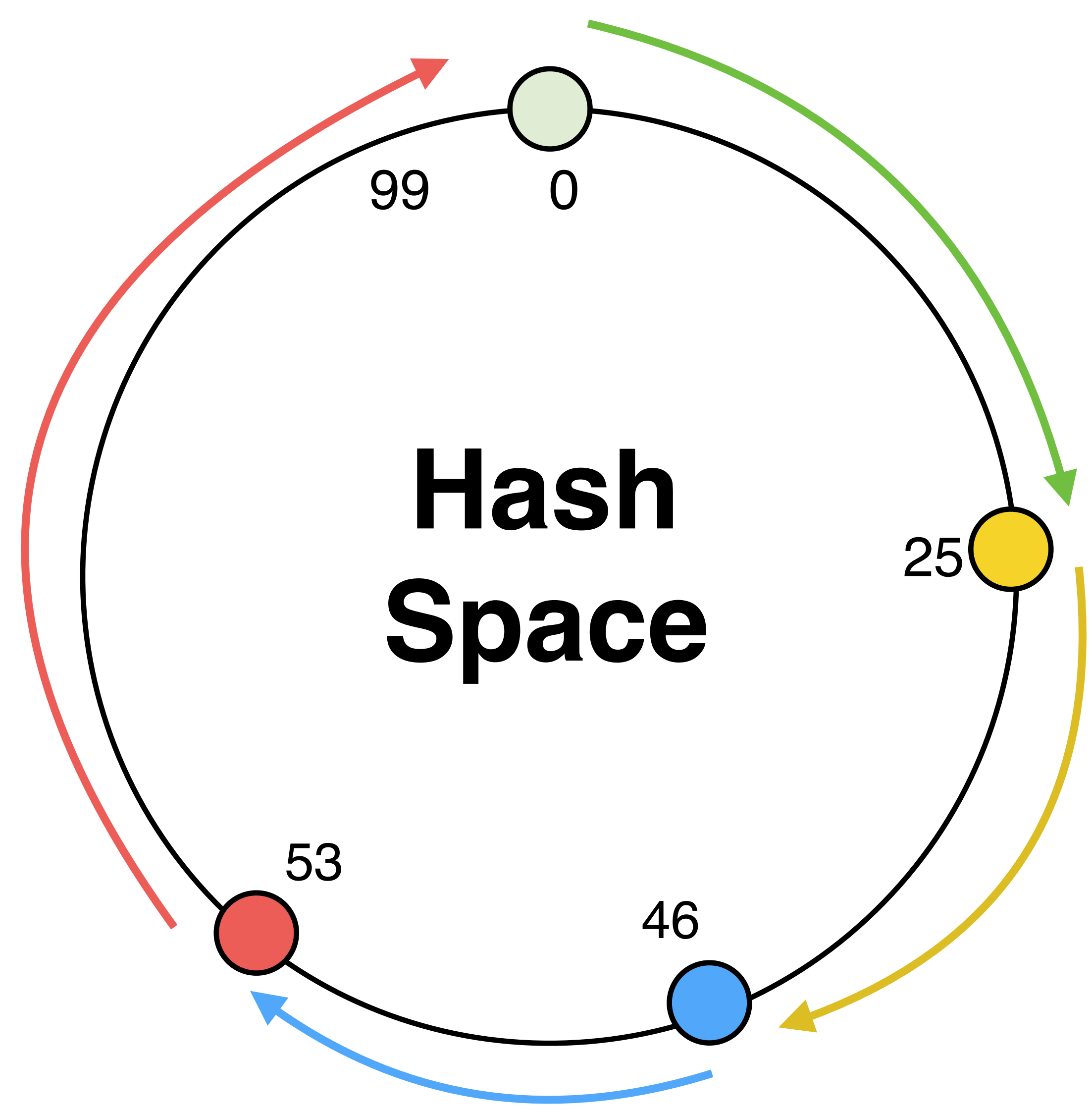
Chord DHT Architecture

Think of hash space as a ring

Nodes pick a random ID when they join: 0 to MAX-1

Nodes are assigned contiguous portions of the ring starting at their ID until they reach the subsequent node

Will this evenly divide up the hash space?

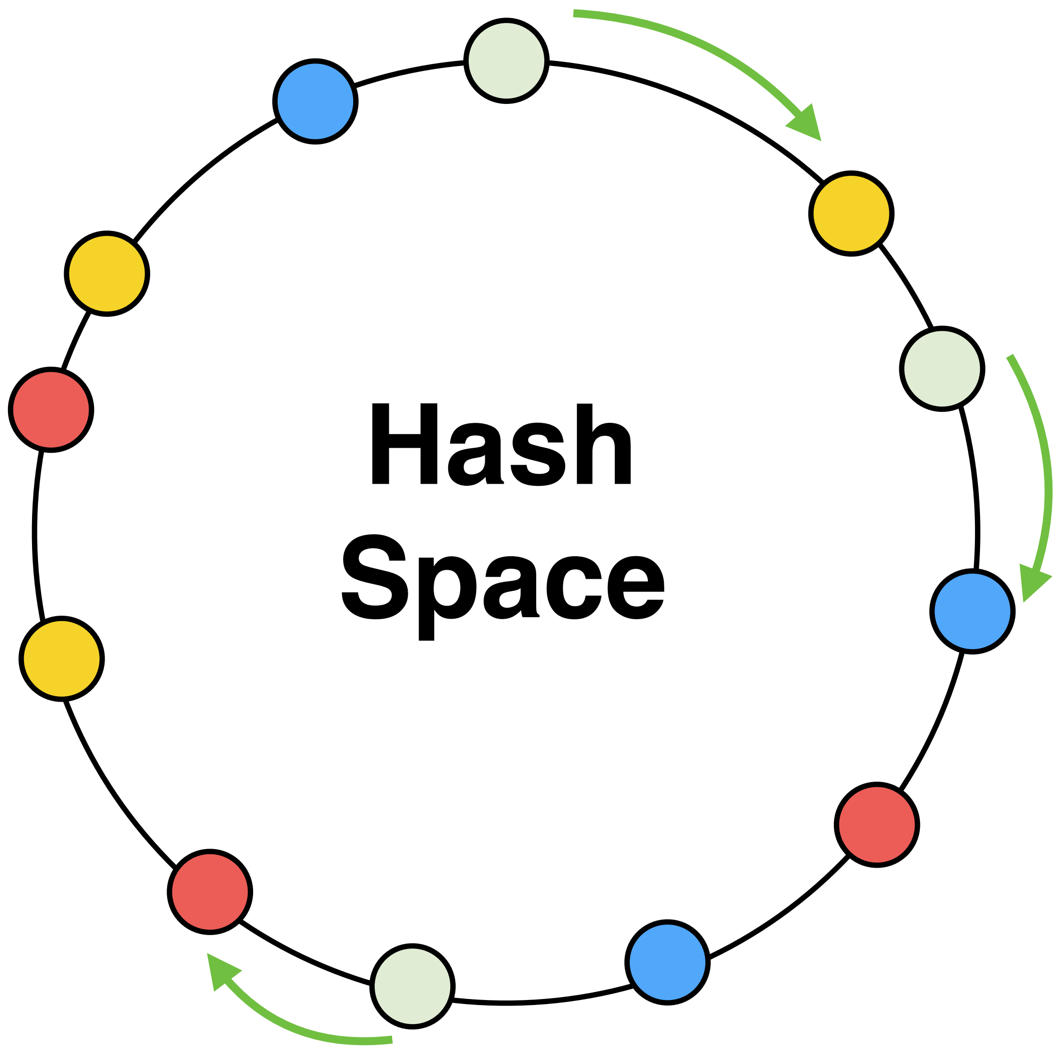


Chord DHT Architecture

Will this evenly divide up the hash space?

If we have a lot of nodes, **probably** yes!

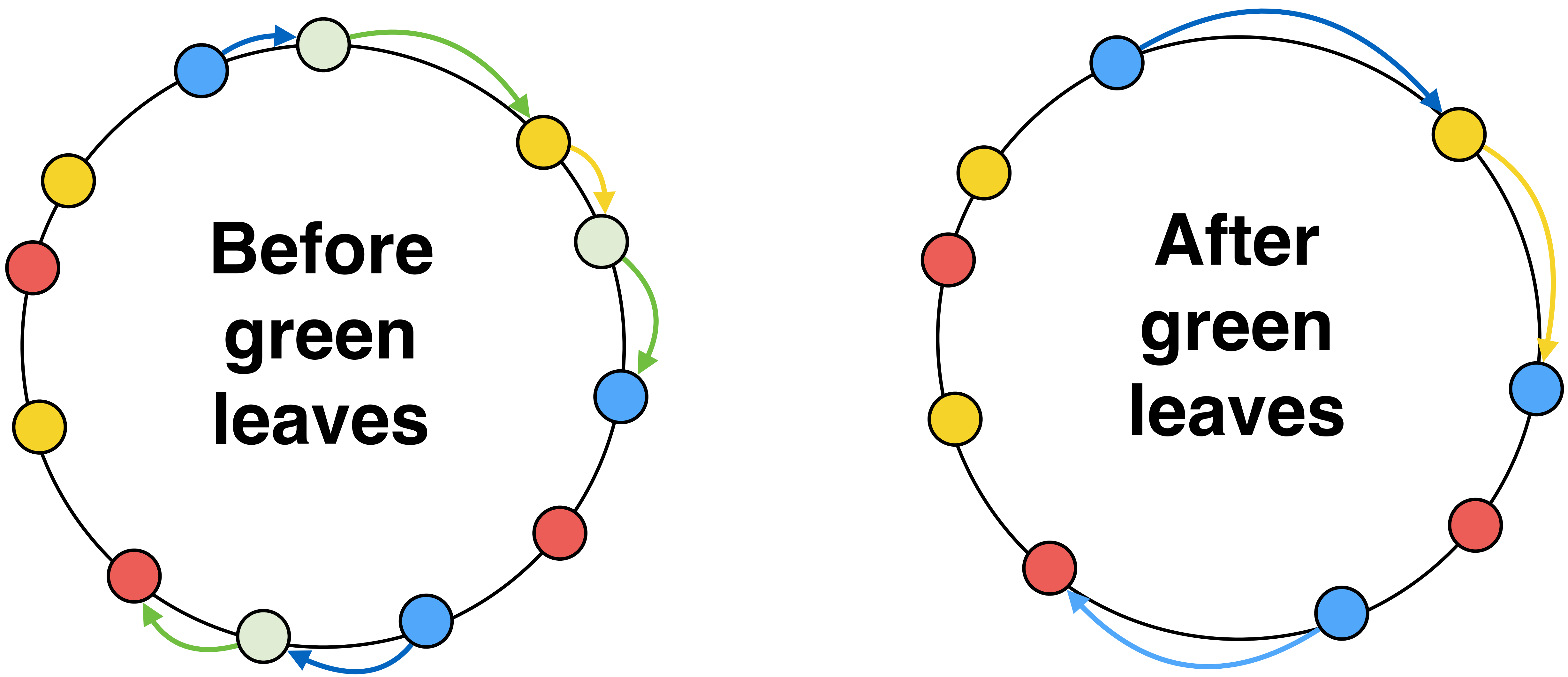
Or, each node can claim multiple IDs (virtual nodes)



Chord Churn

What happens when a node is removed?

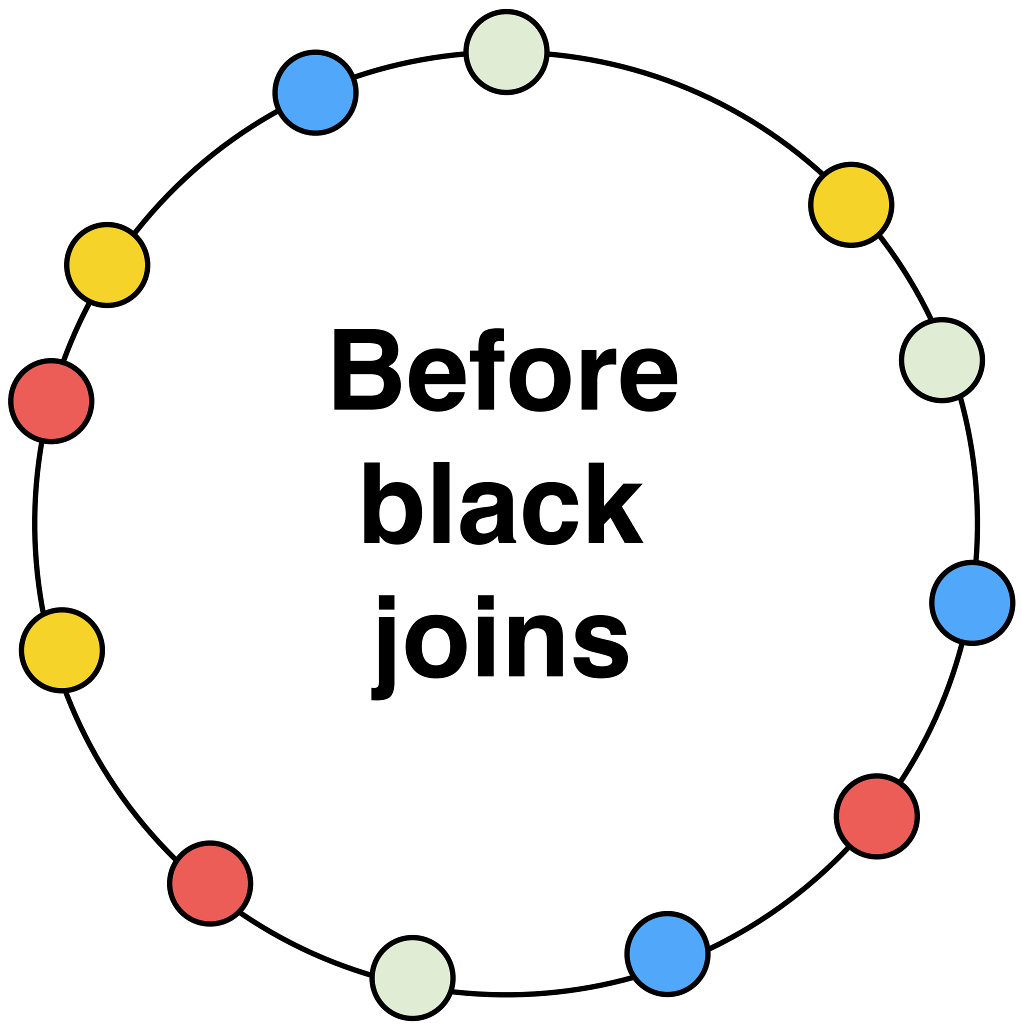
- How many nodes were affected?

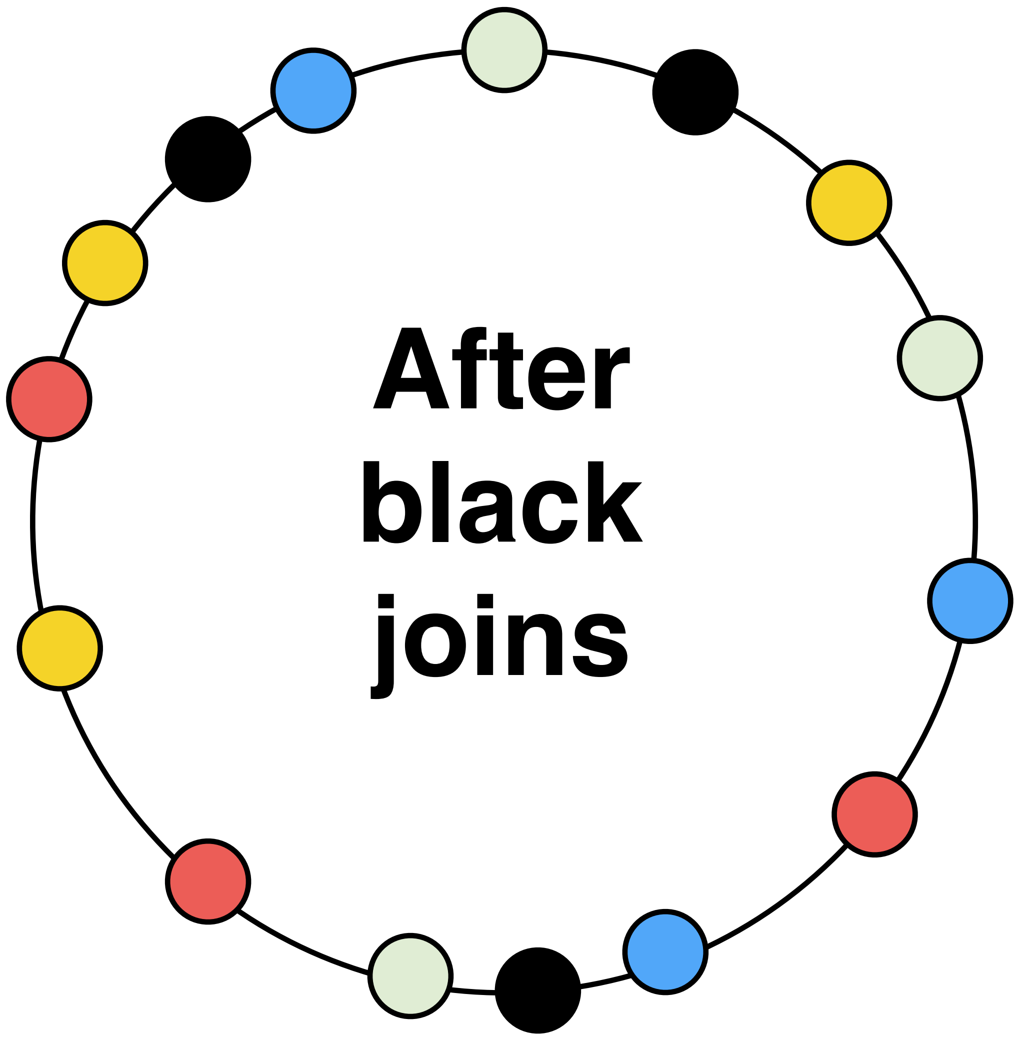


Chord Churn

What happens when a node is added?

- How many nodes were affected?

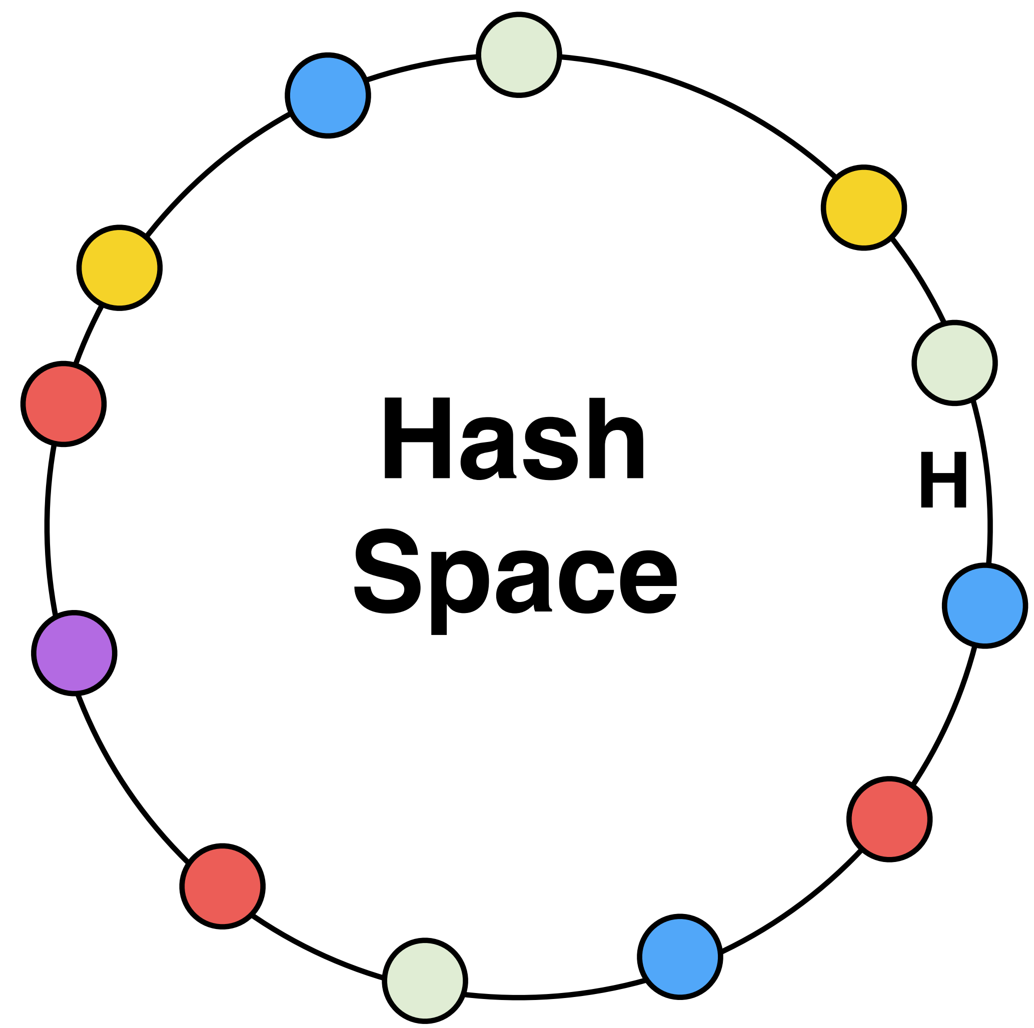




Chord Lookups

Where can we find the key with hash H?

How can the purple node get the data for H?



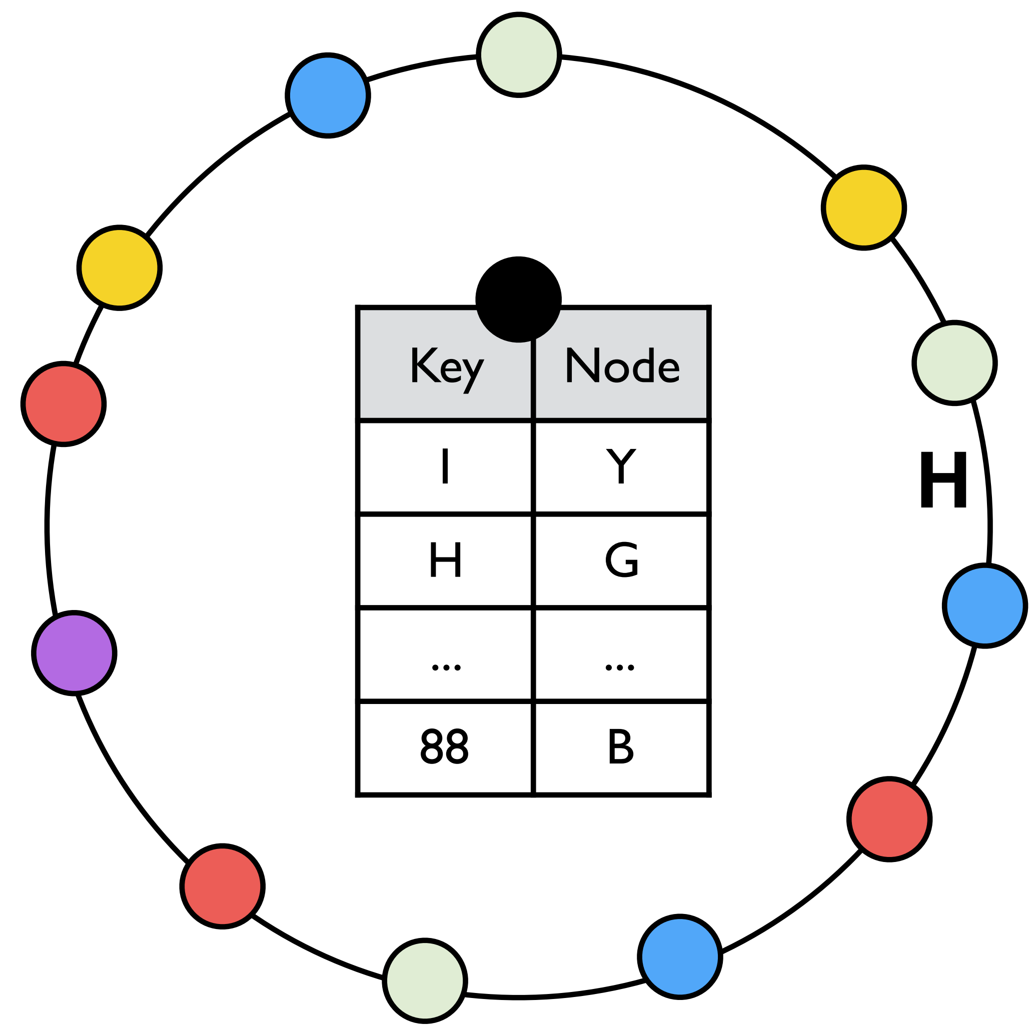
Chord Lookups

Where can we find the key with hash H?

How can the purple node get the data for H?

Options 0: Key Index Table

- Store the node holding each keys in a central server
- Directly access the node!
- If we have millions of keys this table will be really big!
- The node that manages the index table will be a centralized bottleneck!



Key	Node
I	Y
H	G
…	…
88	B

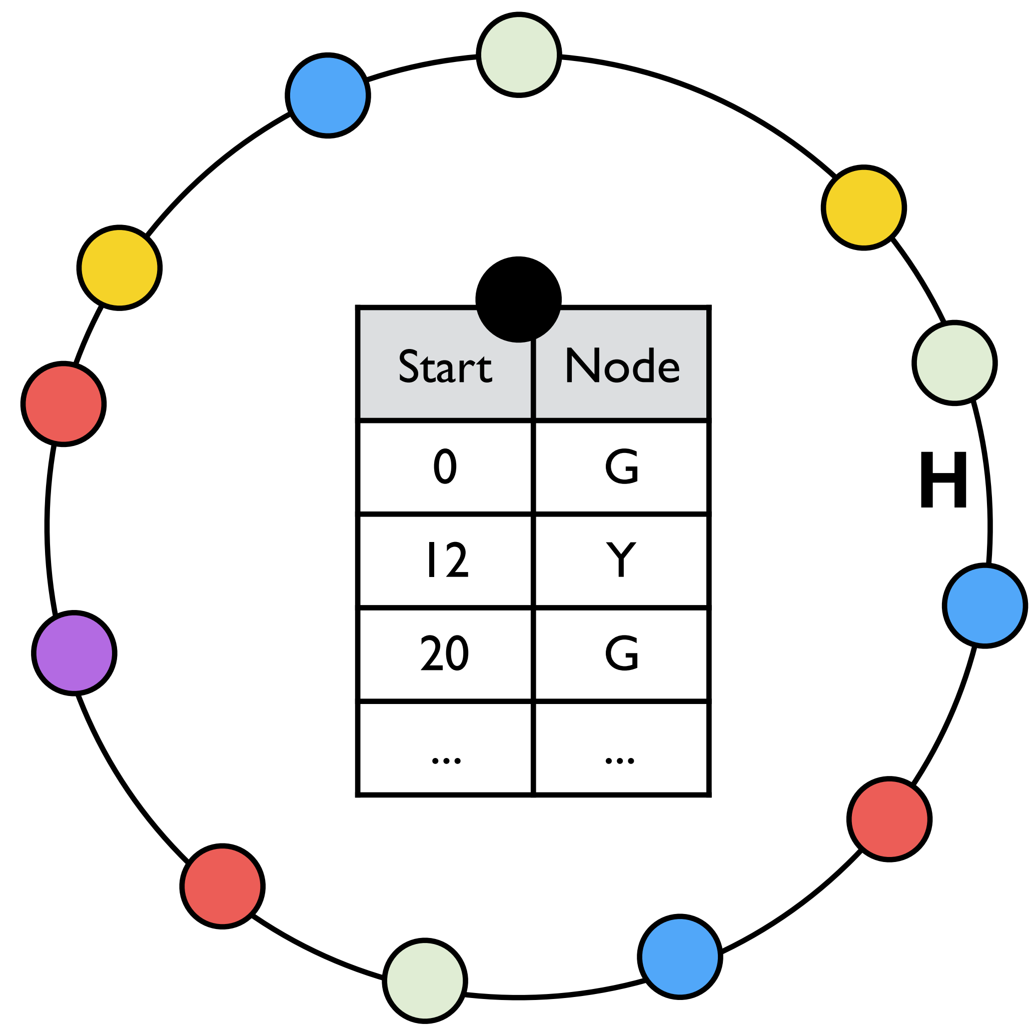
Chord Lookups

Where can we find the key with hash H?

How can the purple node get the data for H?

Options 1: Node Index Table

- Store the indices of all node IDs
- Find which ID is closest to H
- Table is still very large and may be bottleneck!
- Also need to worry about consistently updating the table!



Start	Node
0	G
12	Y
20	G
...	...

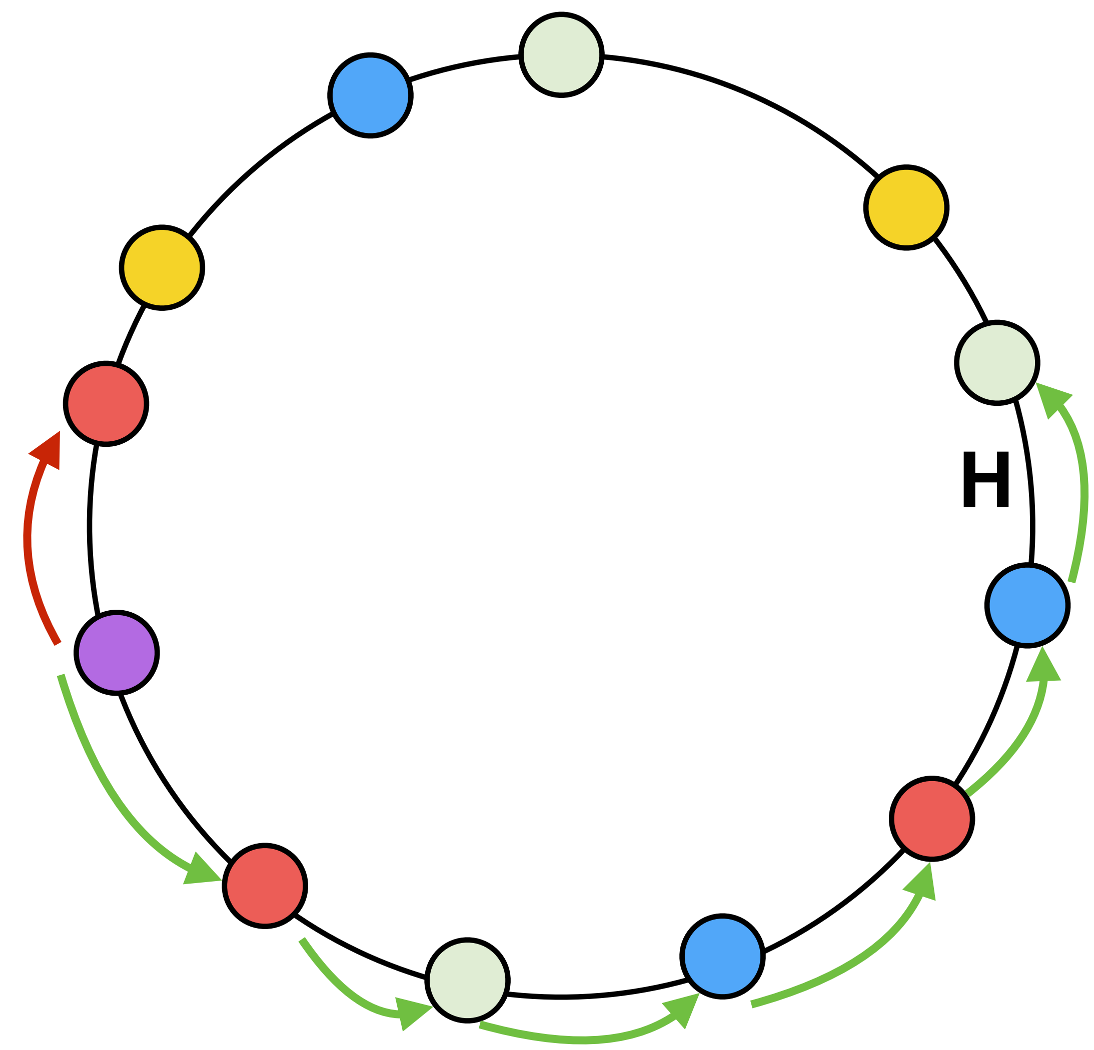
Chord Lookups

Where can we find the key with hash H?

How can the purple node get the data for H?

Options 2: Neighbors

- Each node tracks its **successor** and **predecessor**
- If H > ID, ask successor else ask predecessor
- Requires minimal state
- Can take a long time to traverse the ring! O(N)



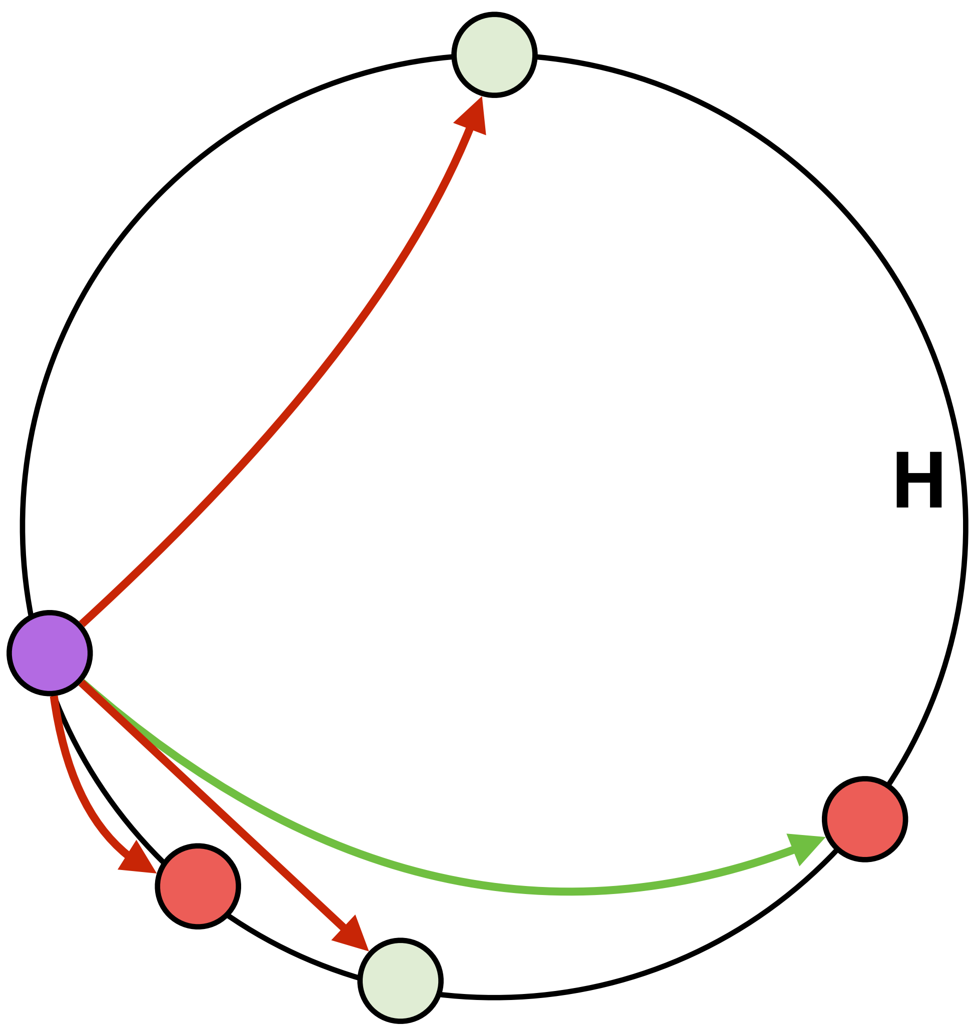
Chord Lookups

Where can we find the key with hash H?

How can the purple node get the data for H?

Options 3: Finger Tables

- Track m additional neighbors: successor 2^0, 2^1, 2^2, ... 2^m
- Jump to closest successor to find H, then jump again
- Requires minimal state
- Can find item in log(N) steps



Chord Lookups

Where can we find the key with hash H?

How can the purple node get the data for H?

Options 3: Finger Tables

- Track m additional neighbors: successor $2^0, 2^1, 2^2, \ldots 2^m$
- Jump to closest successor to find H, then jump again
- Requires minimal state
- Can find item in log(N) steps