SCC.311: Peer to Peer and Cloud Systems

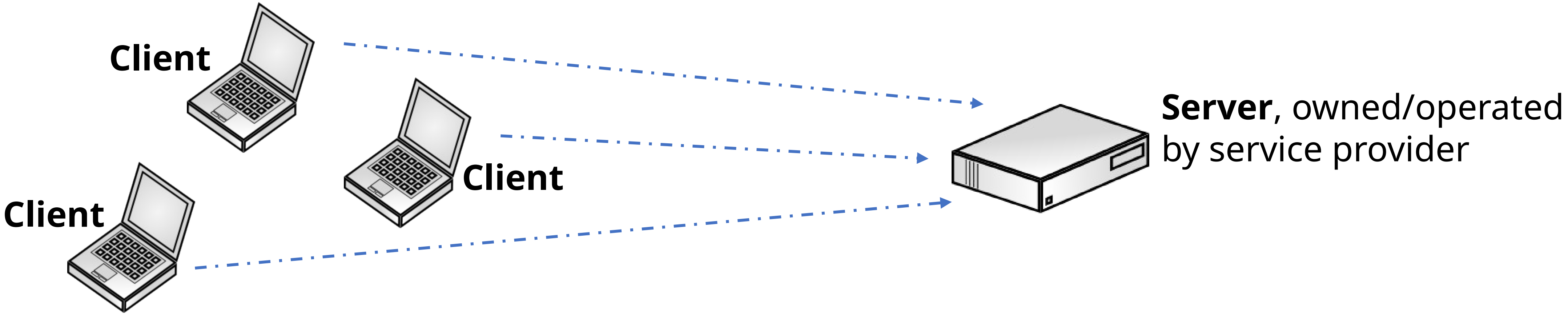
Prof. Barry Porter

Coursework reminders

- The first coursework stage is due this Friday
- The coursework test page **is not the submission site**
  - Please submit your work via Moodle when you’re happy with it
- You must submit all of your source code, including your client

Overview and history

- Much of what we've studied so far relates to what we call **client-server** architectures for distributed systems
  - Here we use one or more dedicated, always-on *servers* to implement and provide a service to many *clients*
  - Servers only *provide* a service, and clients only *consume* a service

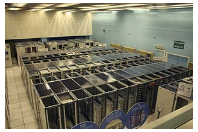


In these architectures, clients have different code to servers – they don't need any of the service implementation logic

Overview and history

- But client-server architectures haven't always been dominant, and still aren't universal today...

Mainframes + dumb terminals, operated by large companies able to afford room-sized computers



Personal computer becomes ubiquitous; World Wide Web launched; distributed systems are operated by companies able to afford to run servers

Peer to peer systems become dominant approach for end-users to operate/use shared distributed systems *without the need for central servers*

Cloud computing allows anyone to cheaply rent "infinite" servers & create their own server –based distributed systems, becomes new standard approach

1970/80's **1990's** **2000's** **2010's**

Overview and history

- Client-server architectures are easy to build, but have their own shortcomings as distributed systems…
  - You still need someone to pay for the always-on (physical or virtual) servers
  - The more users (clients) you have, the more you need to pay for your servers to distribute load sufficiently
  - For many users, the nearest data centre is geographically distant, adding latency to requests and requiring a large network infrastructure
  - Users are forced to trust server operators with their personal data, and often are forced to consent to complex data privacy and sharing policies
  - Data centres now consume a significant amount of global energy for power and cooling, a cost which continues to increase…

Overview and history

- The modern distributed systems ecosystem is now a hybrid of client-server and peer-to-peer architectures
  - Both approaches are used where it most makes sense
  - The Internet network backbone itself is a peer-to-peer network
  - Content distribution networks form peer-to-peer networks among their data centres
  - Media streaming services often take advantage of peer-to-peer content sharing to reduce the load on central servers and provide lower latency
  - Social, anti-centralisation movements embrace peer-to-peer concepts like Bitcoin

Peer to Peer Systems

- In a peer-to-peer system, every member of the system is **both** a client and a server at the same time
- Every member therefore both provides **and** consumes part of the shared service
- The total compute, storage, and network resource available is determined by the current number of members of the service

Peer to Peer Systems

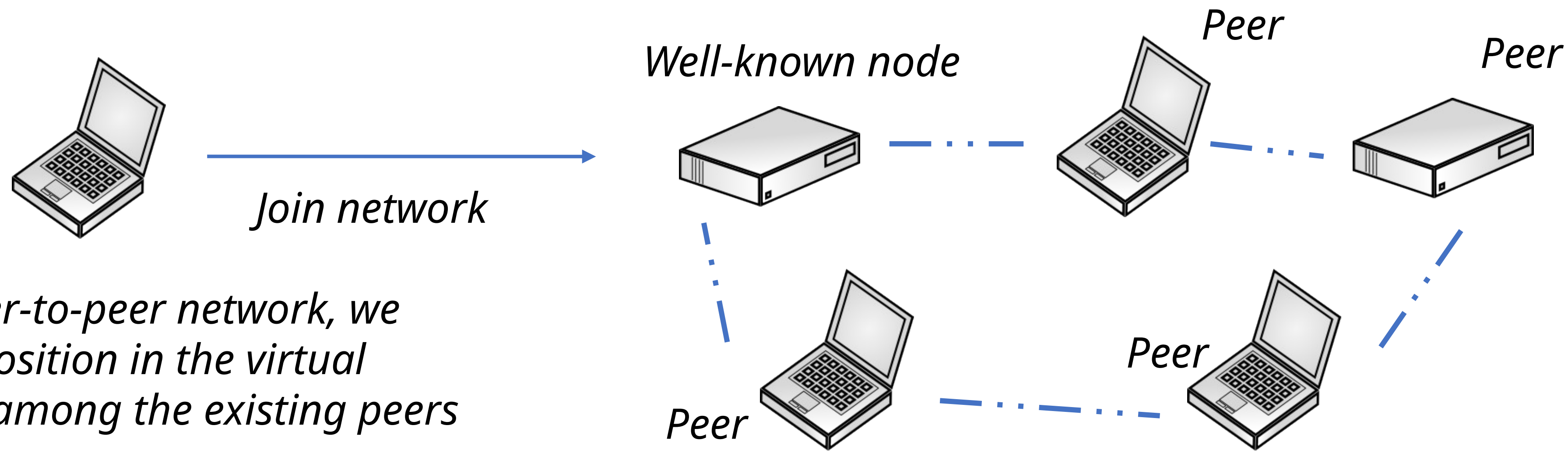
- Each node (participant/user) in the peer-to-peer system has the same code, which has both consumer and provider elements

- The different nodes in a peer-to-peer system are therefore interacting with different instances of exactly the same program

- This is a little bit like programming with recursion, where a function calls itself; peer-to-peer protocols can be challenging to build in the same way that recursive procedures can be hard to understand

Peer to Peer Systems

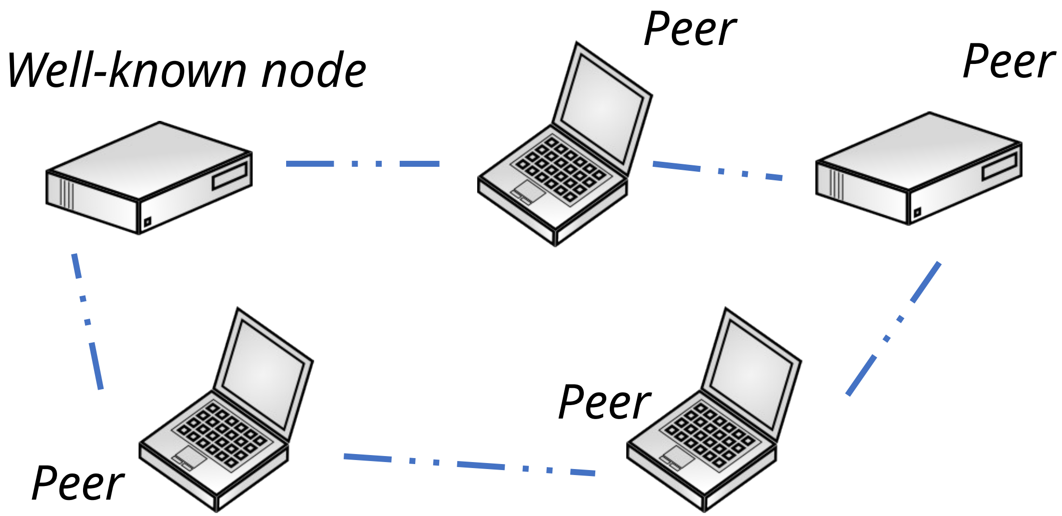
- All peer-to-peer systems need to have a well-known node which acts as an entry-point to the system for other nodes to join
  - This is often implemented by using a single fixed server / web site to store a list of the IP addresses of well-known nodes which are currently available

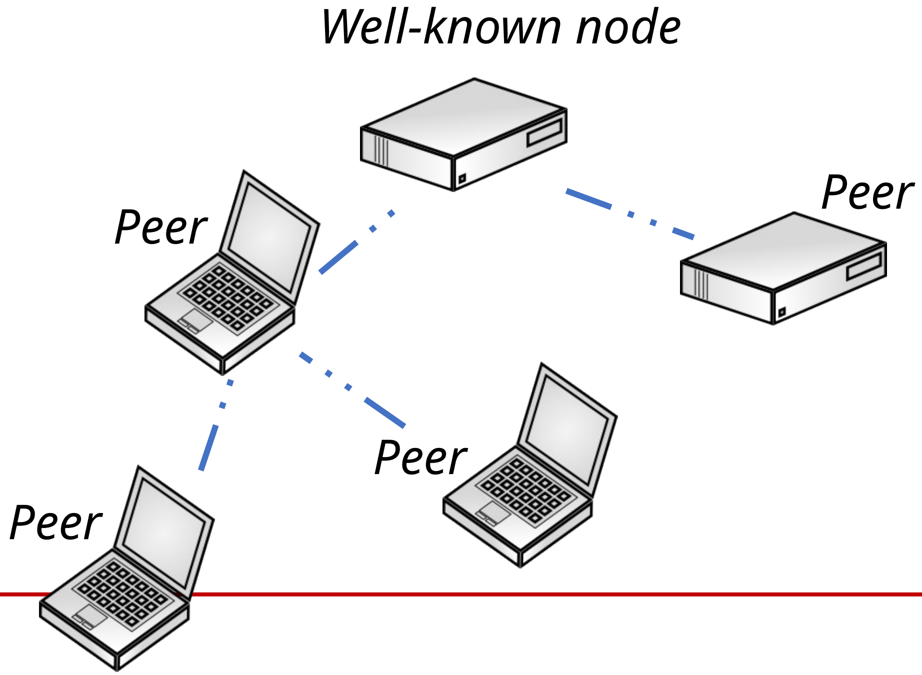


Once we join a peer-to-peer network, we are directed to a position in the virtual network topology among the existing peers

Peer to Peer Systems

- The task of a peer-to-peer network is to (i) manage the *virtual network topology* between peers to match the system's objective; and (ii) manage the content/service within the system to uphold some policy (e.g. mining bitcoins)





Peer to Peer Systems

- As well as performing their key roles, peer-to-peer networks must deal with a set of difficult challenges
  - **Churn**: the effect of nodes joining and leaving the network frequently, so that the virtual network topology must constantly adjust to the current members
  - **Network divergence**: all peer-to-peer networks build a virtual network topology between their peers; if this topology differs too far from the physical network between those nodes, the peer-to-peer system may suffer very poor performance
  - **Selfish users**: if very few users actually provide a service (e.g. sharing files), and the majority only consume, the viability of the system breaks down

Coming up…

- We examine peer-to-peer and cloud technology in common use today

  - Distributed Hash Tables (used e.g. in BitTorrent, Dynamo, Cassandra)

  - Cloud Computing and Things as a Service

Distributed Hash Tables

- The aim here is to support a *put(key, value)* and *get(key)* function which work in the same way as a local hash table / hash map
  - In the distributed version (**D**istributed **H**ash **T**able – DHT), keys and values are stored at different nodes in the peer-to-peer network

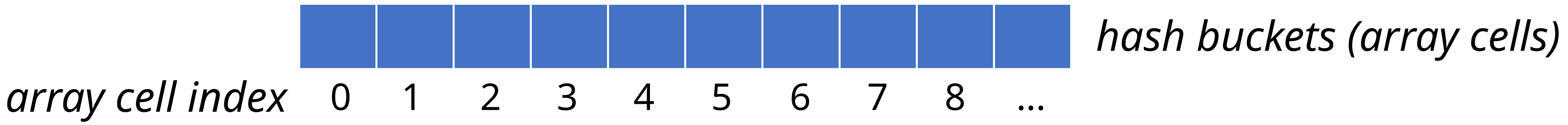

```
void put(String key, byte
value[])
byte[] get(String key)
void joinNetwork(String
atNodeIP)
```

Distributed Hash Tables

- Refresher on hash tables…
  - A hash table works by allocating a fixed-length array, where each array cell represents a bucket where things can be stored
  - Each bucket has a dynamic list (e.g. linked list) of keys stored at that index

void put(key, value)

We pass the key into a hash function which converts the string key into an integer index; the maximum index value generated by the hash function is the allocated array length - 1



Distributed Hash Tables

- Refresher on hash tables…
  - Every *key* placed into the hash table must be unique, but there are no constraints on what the associated *value* can be

void put(key, value)

We pass the key into a hash function which converts the string key into an integer index; the maximum index value generated by the hash function is the allocated array length - 1

hash buckets (array cells)

array cell index 0 1 2 3 4 5 6 7 8 …

Key: jun / value: [binary data]
Key: mohammed / value: [binary data
Key: sarah / value: [binary data]

To retrieve an item from a hash table, we supply the key that we want, which is hashed to an index; then walk through all the keys stored at that index until we find the one we wanted.

Distributed Hash Tables

- DHTs convert this concept into a distributed peer-to-peer network
  - In the distributed version, nodes represent buckets, where the index of the bucket represented by a node is derived by e.g. hashing its IP address
  - We need to decide how to deal with the fact that nodes can freely join and leave the network, so the list of buckets present may not be continuous

  - One of the seminal works on DHTs is the **Chord** protocol (designed at MIT)
    - This has inspired the design of many newer DHT systems

Distributed Hash Tables // Chord

Assuming we have one "well-known" node at which others join...

32

IP address: 54.90.12.233 hash function result = **32**

DHTNode

Each node maintains a network connection to a **successor** (s) node, and a **predecessor** (p) node in the DHT key space; the Chord topology management protocol takes care of correctly placing nodes

Distributed Hash Tables // Chord

Assuming we have one "well-known" node at which others join...

32

IP address: 54.90.12.233 hash function result = **32**

DHTNode

s:

p:

join network

98

IP address: 129.14.40.5 hash function result = **98**

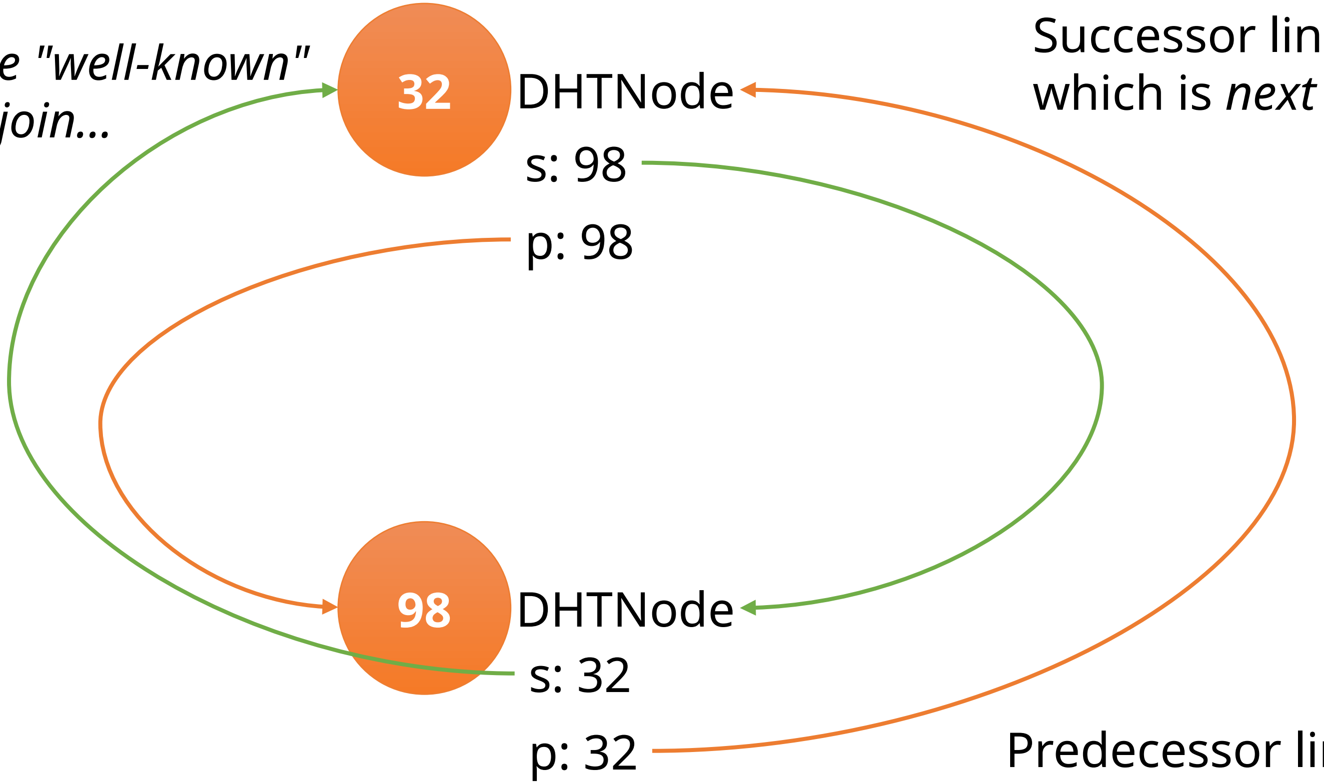
DHTNode

s:

p:

Distributed Hash Tables // Chord

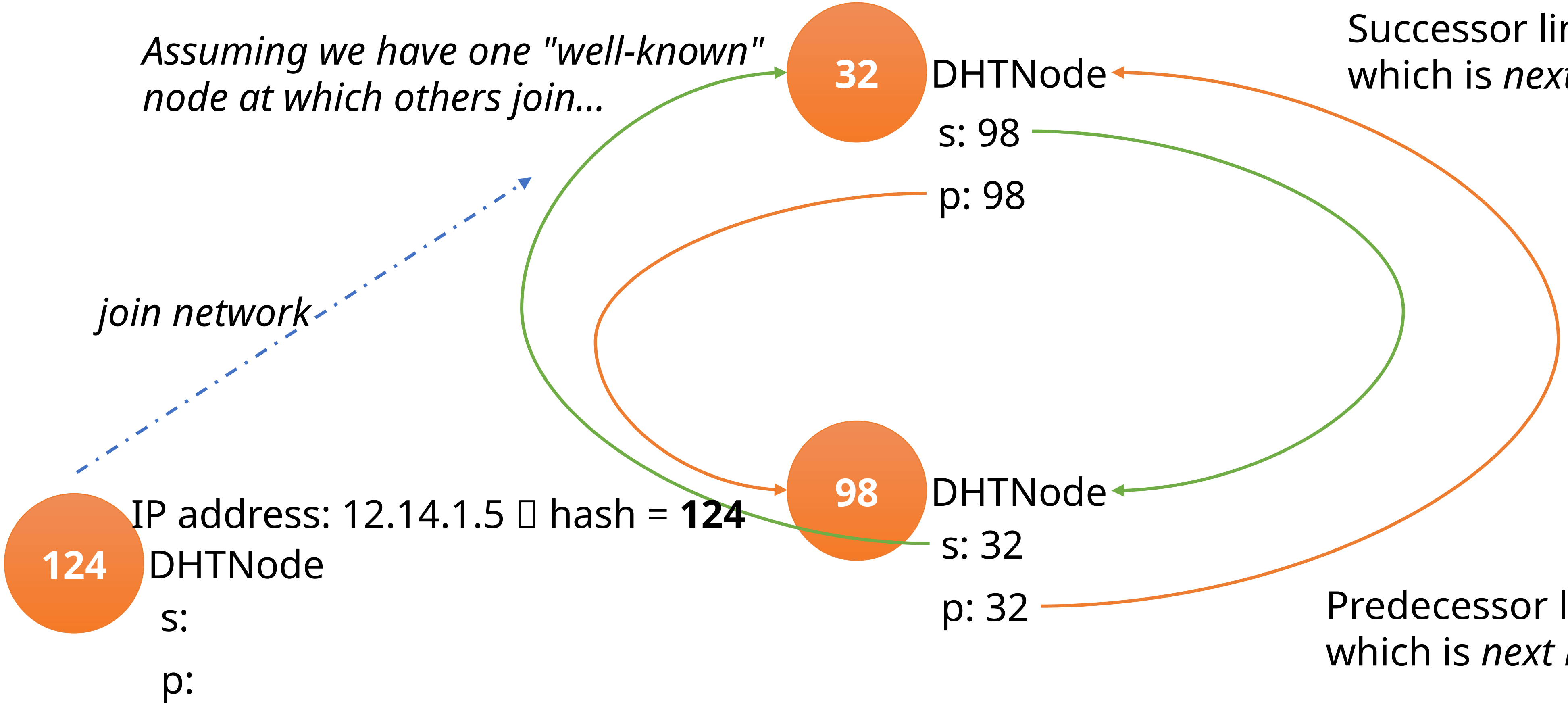
Assuming we have one "well-known" node at which others join...



Successor links point to the node which is *next highest* in the key space

Predecessor links point to the node which is *next lowest* in the key space

Distributed Hash Tables // Chord



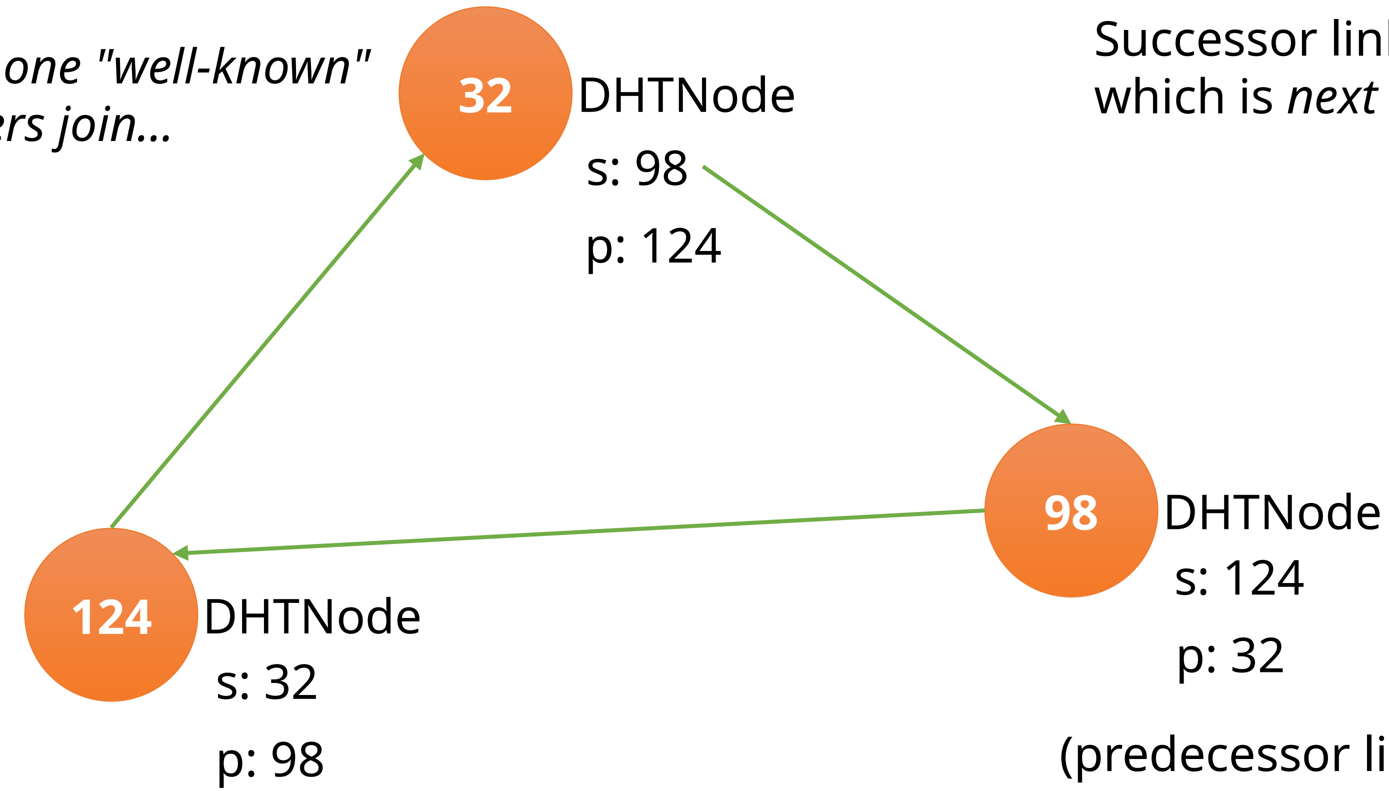
Successor links point to the node which is *next highest* in the key space

Predecessor links point to the node which is *next lowest* in the key space

Distributed Hash Tables // Chord

Assuming we have one "well-known" node at which others join...

Successor links point to the node which is *next highest* in the key space



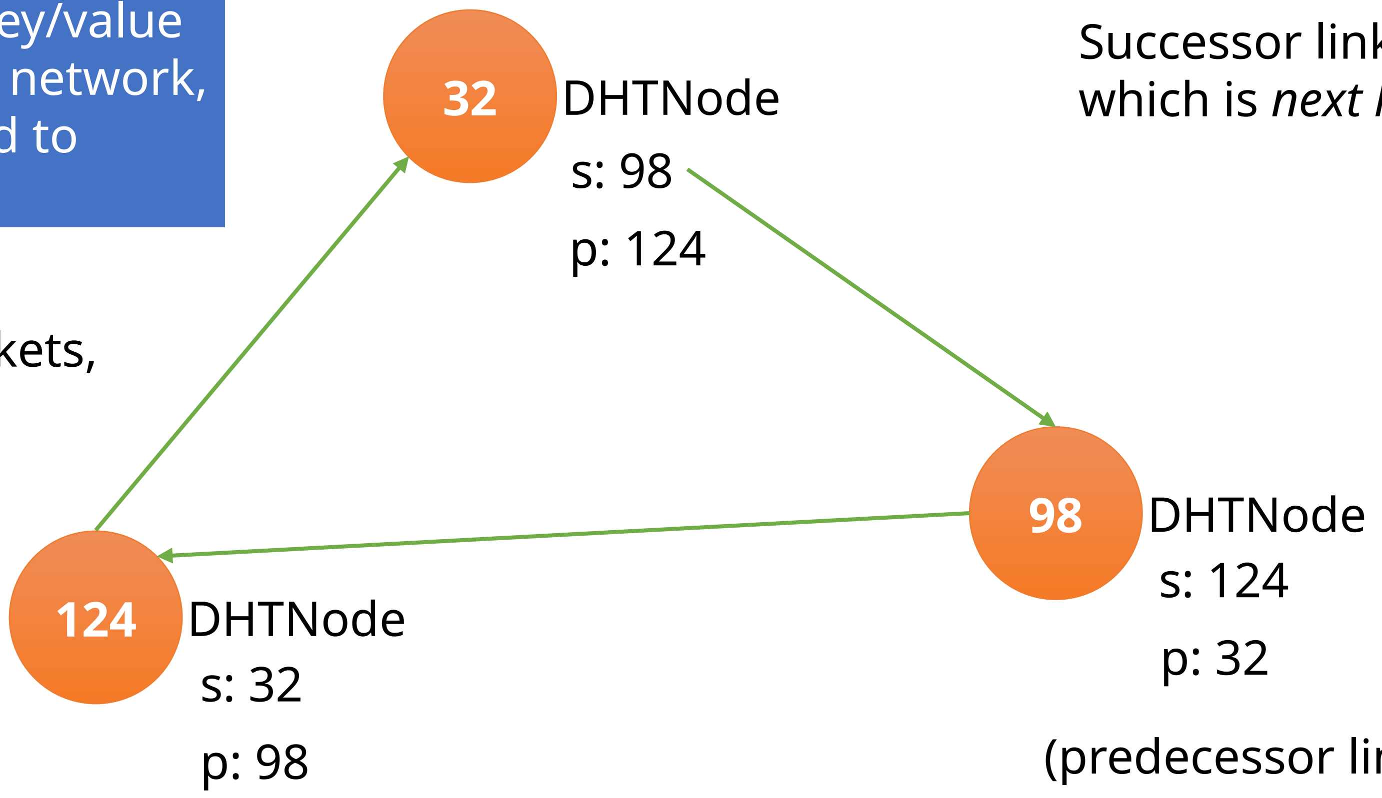
(predecessor links not shown here...)

Distributed Hash Tables // Chord

We can *put(key, value)* a key/value pair into **any** node in the network, and the key will be routed to the correct location

Successor links point to the node which is *next highest* in the key space

To deal with missing buckets, Chord stores keys at the *next highest* available bucket/node: here, keys 33…97 are all stored at node 98



(predecessor links not shown here…)

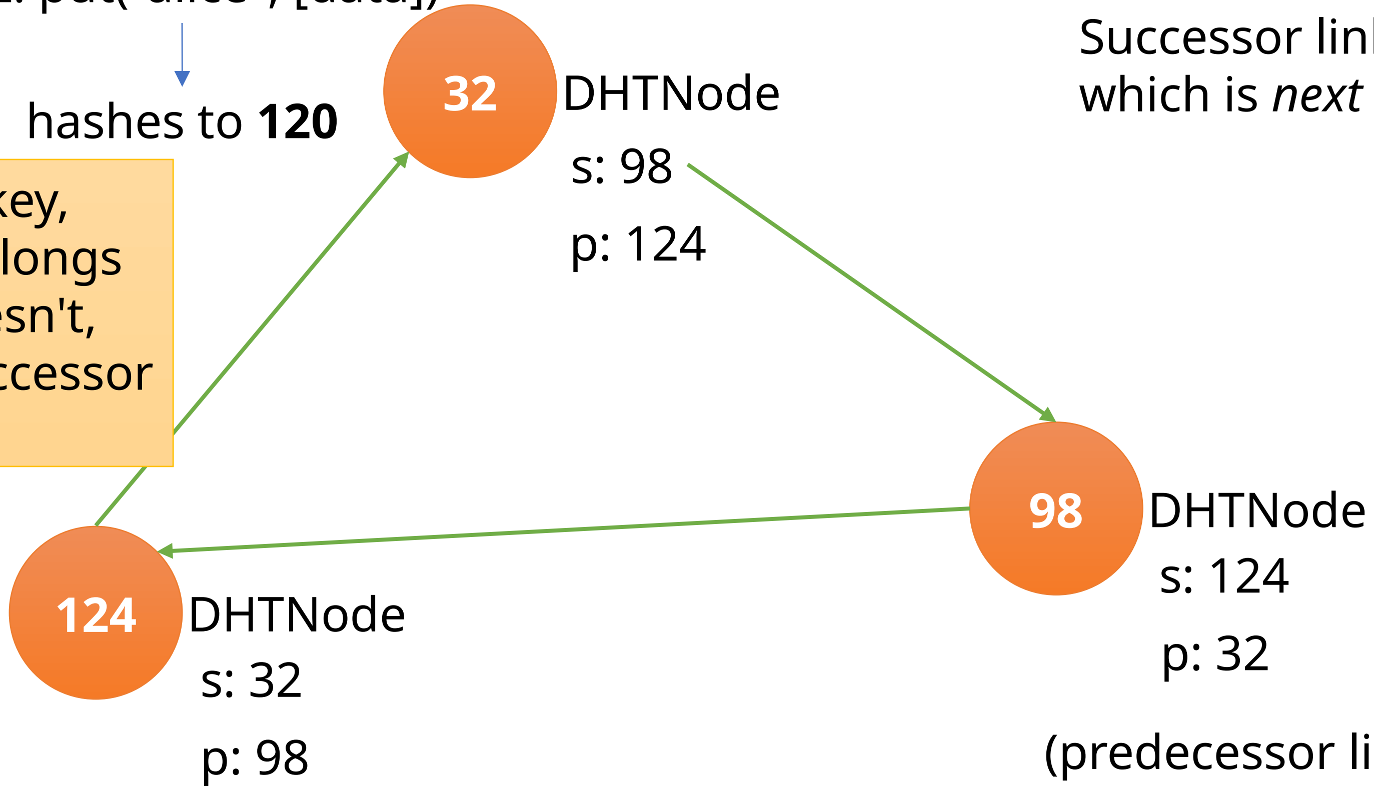
Distributed Hash Tables // Chord

@ node 32: put("alice", [data])

hashes to **120**

To decide where to store a key, a node checks if that key belongs to its own ID range. If it doesn't, it forwards the key to its successor to check there, etc.

Successor links point to the node which is *next highest* in the key space



(predecessor links not shown here...)

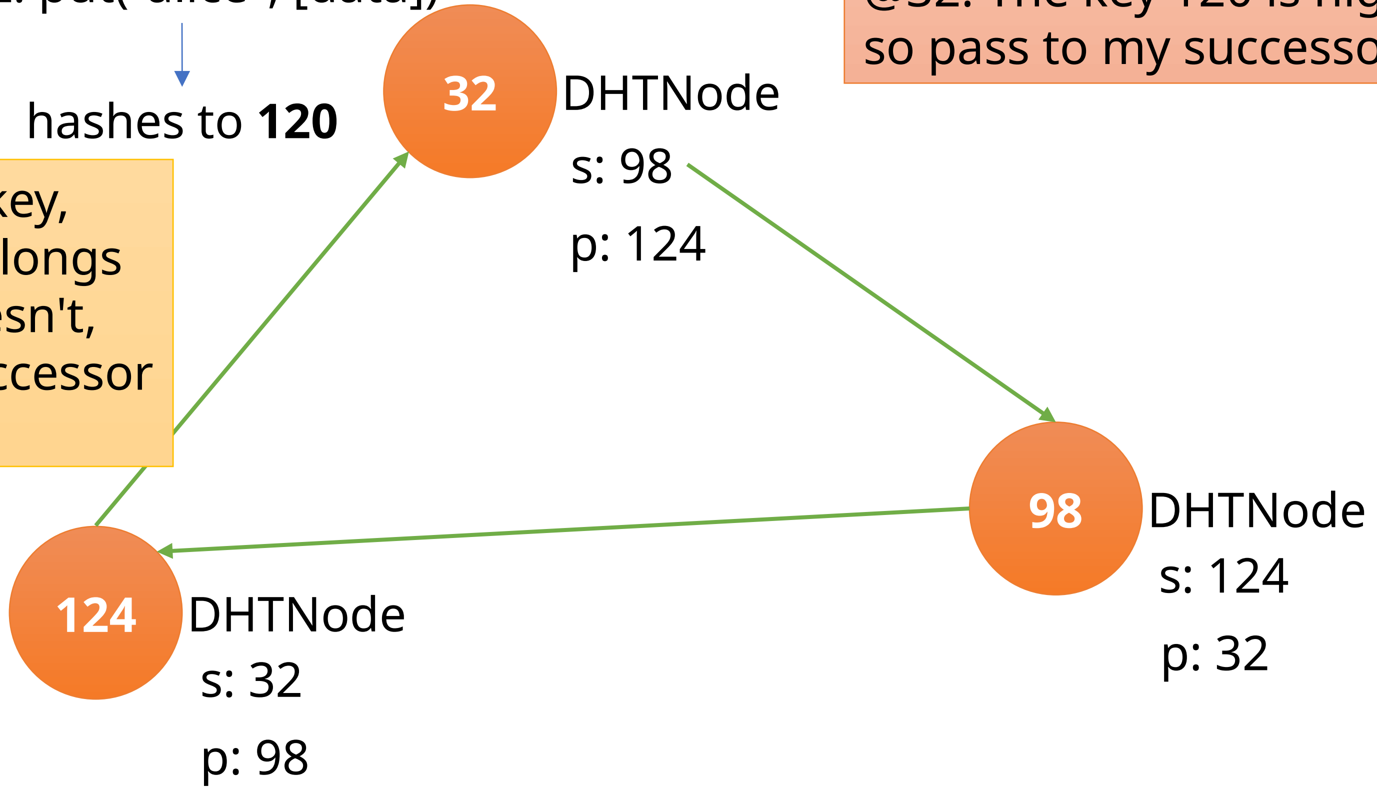
Distributed Hash Tables // Chord

@ node 32: put("alice", [data])

hashes to **120**

@32: The key 120 is higher than my own ID (32), so pass to my successor...

To decide where to store a key, a node checks if that key belongs to its own ID range. If it doesn't, it forwards the key to its successor to check there, etc.



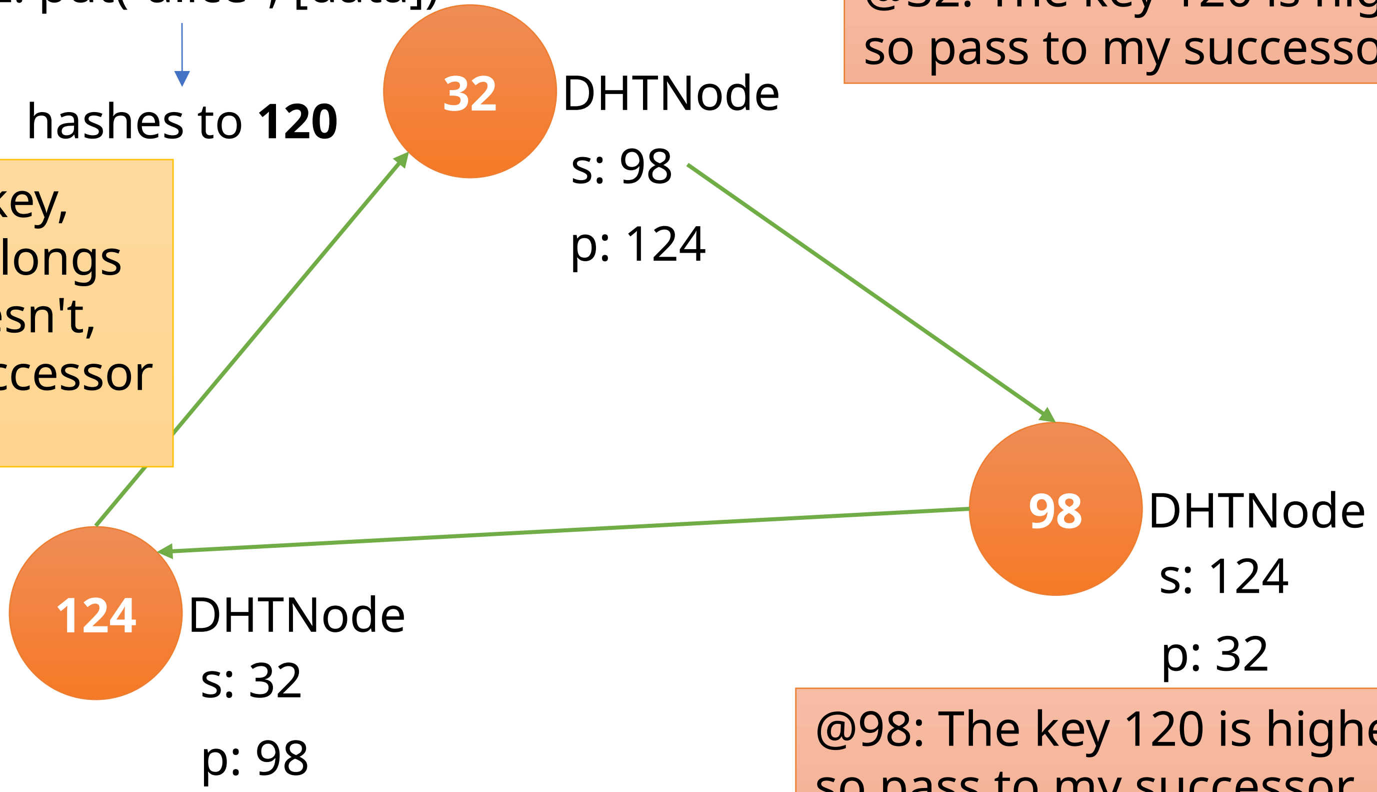
Distributed Hash Tables // Chord

@ node 32: put("alice", [data])

hashes to **120**

@32: The key 120 is higher than my own ID (32), so pass to my successor...

To decide where to store a key, a node checks if that key belongs to its own ID range. If it doesn't, it forwards the key to its successor to check there, etc.



@98: The key 120 is higher than my own ID (98), so pass to my successor...

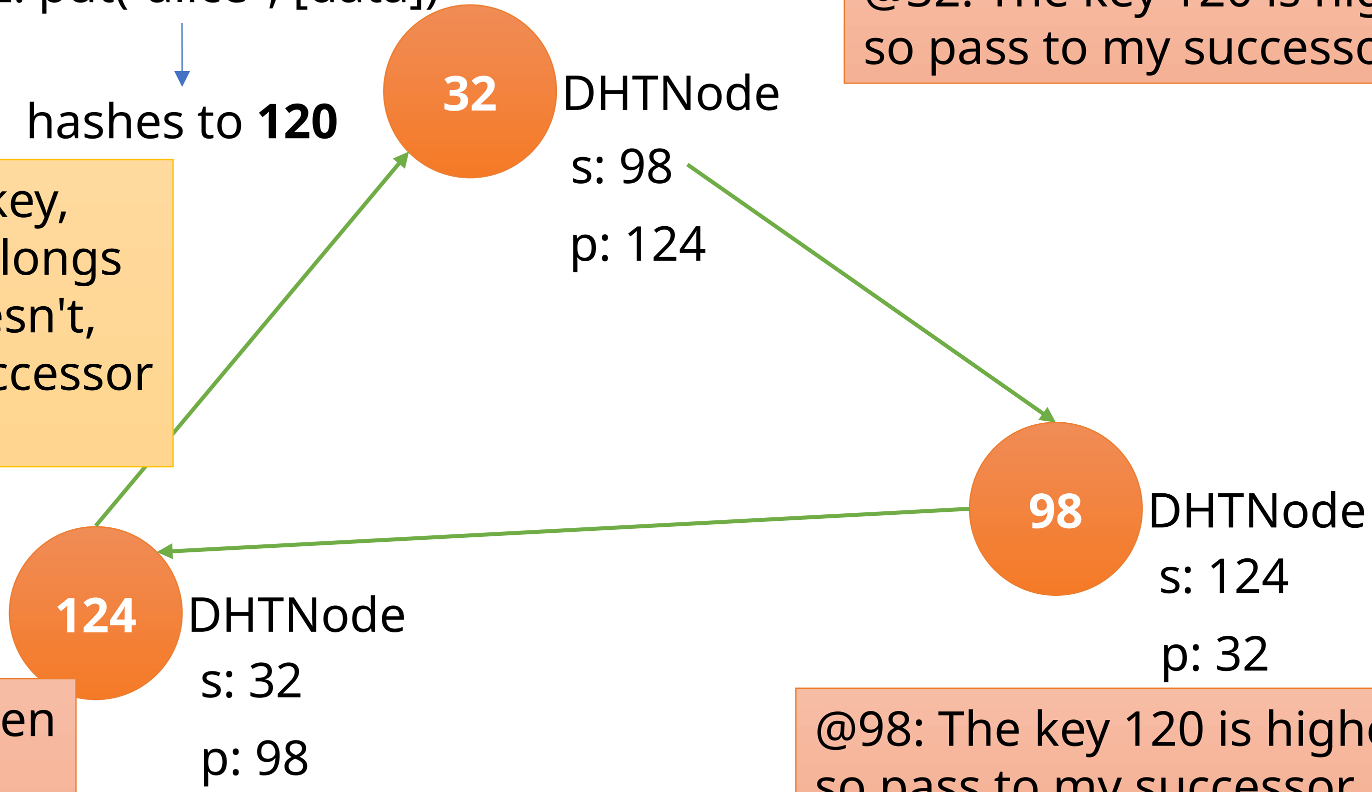
Distributed Hash Tables // Chord

@ node 32: put("alice", [data])

hashes to **120**

@32: The key 120 is higher than my own ID (32), so pass to my successor...

To decide where to store a key, a node checks if that key belongs to its own ID range. If it doesn't, it forwards the key to its successor to check there, etc.

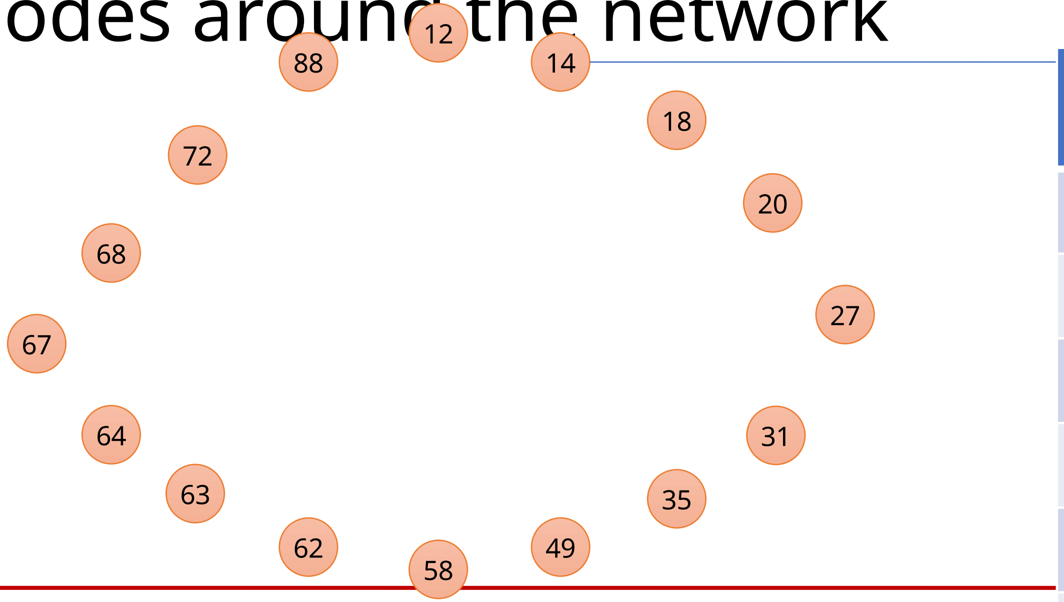


@98: The key 120 is higher than my own ID (98), so pass to my successor...

@124: The key 120 is between my predecessor and me, so store it here.

Distributed Hash Tables // Chord

- This basic approach scales poorly to large networks with thousands of nodes; Chord therefore also uses a **finger table** at each node to store a fixed number of increasingly long-range nodes around the network



@ node 14

Finger index	Node ID
0	15
1	16
2	18
3	22
4	38
5	46

$2^0 + 14 = 15$

$2^1 + 14 = 16$

$2^2 + 14 = 18$

$2^3 + 14 = 22$

$2^4 + 14 = 38$

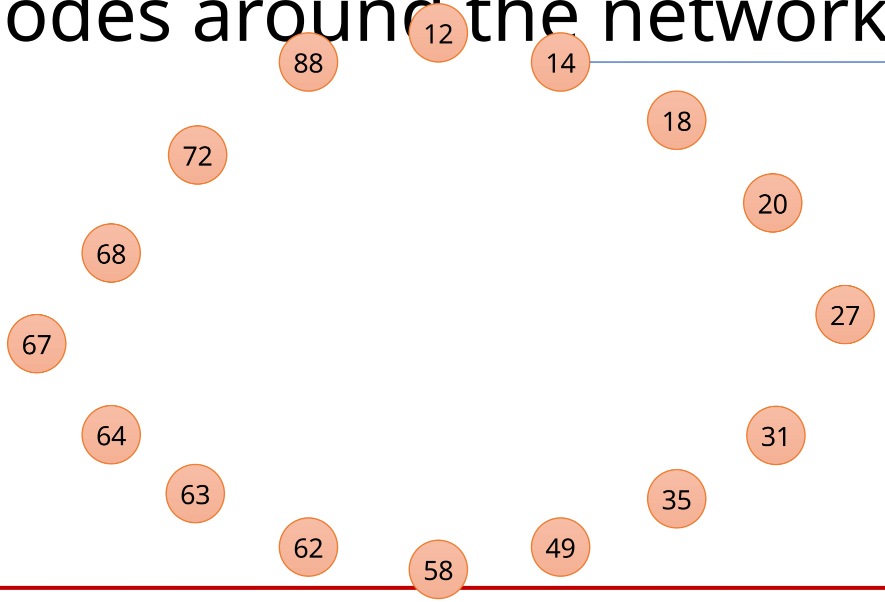
$2^5 + 14 = 46$

Just like keys, if a node ID doesn't exist, we point to the *next highest node* instead

Distributed Hash Tables // Chord

- This basic approach scales poorly to large networks with thousands of nodes; Chord therefore also uses a **finger table** at each node to store a fixed number of increasingly long-range nodes around the network

Now, when looking up a key, we can consult our finger table to make a large jump across the network to the area closest to that key (instead of traversing each node in order).



@ node 14

Finger index	Node ID
0	15
1	16
2	18
3	22
4	38
5	46
6	(etc)

$2^0 + 14 = 15$

$2^1 + 14 = 16$

$2^2 + 14 = 18$

$2^3 + 14 = 22$

$2^4 + 14 = 38$

$2^5 + 14 = 46$

Distributed Hash Tables // Chord

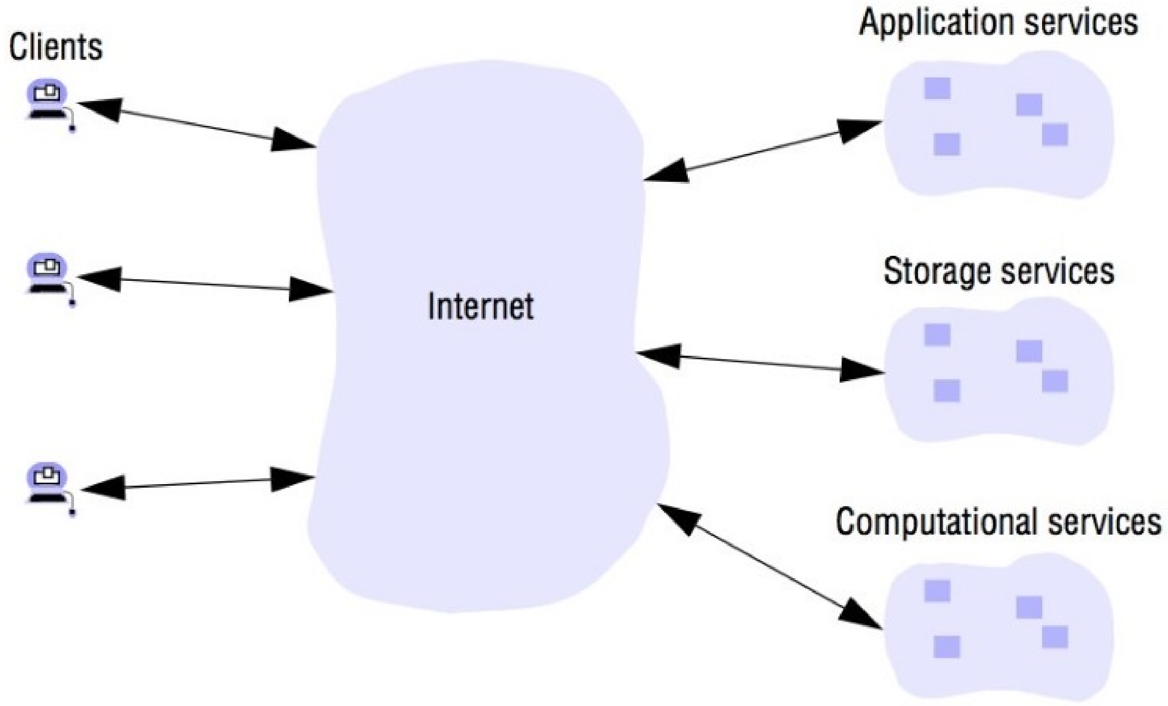
- This basic approach scales poorly to large networks with thousands of nodes; Chord therefore also uses a **finger table** at each node to store a fixed number of increasingly long-range nodes around the network
  - Using this approach, Chord is able to guarantee a maximum of O(log N) hops to resolve a key lookup to the correct node
  - DHTs like Chord allow us to store a set of key/value pairs across a large number of peer nodes in the network, where each key maps to exactly one node

Cloud Computing

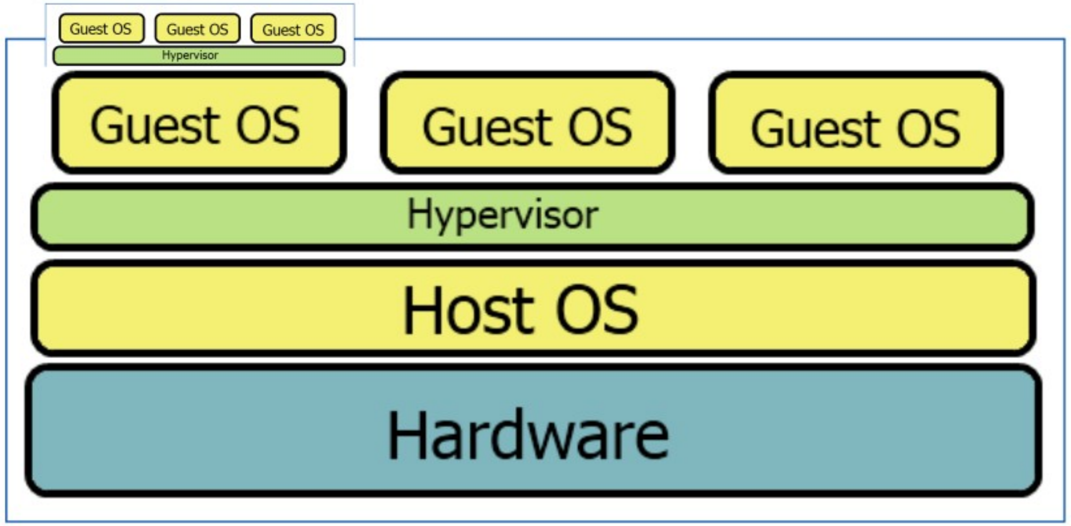
- Cloud computing aims to make datacentre-based compute resource into a public utility
- This allows many users (e.g. businesses) to share the same physical resources, reducing costs for everyone
- It also allows businesses to rapidly *scale up* their compute capability by adding more nodes (pay-as-you-use)

Cloud Computing

- A cloud is a set of Internet-based application, storage and computing services sufficient to support most users' needs, thus enabling them to largely or totally dispense with local data storage and application software



Cloud Computing – How?



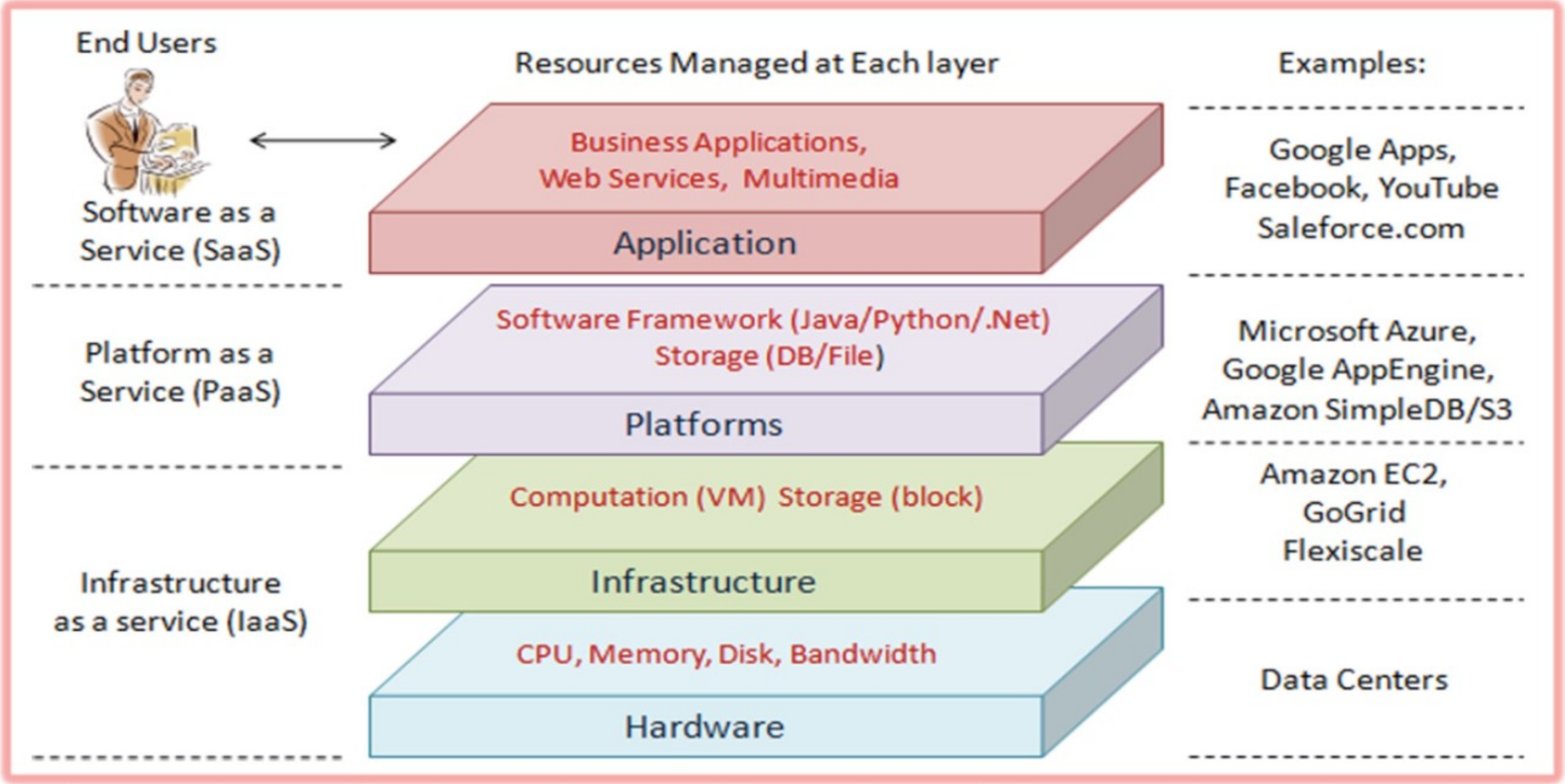
- Virtualisation
  - Host + guest(VM)
  - *Close* to physical performance
  - **Multi-tenancy**
  - Nested virtualisation
  - Tech: Xen, KVM, VMWare, …
- Different economic model
  - By-product of the rapid expansion of online giants
- XaaS: everything has an API (accessible)
  - Potential to build complex interconnected systems

Cloud Computing

- Every cloud comes in different forms
  - Provisioning level
  - Deployment model

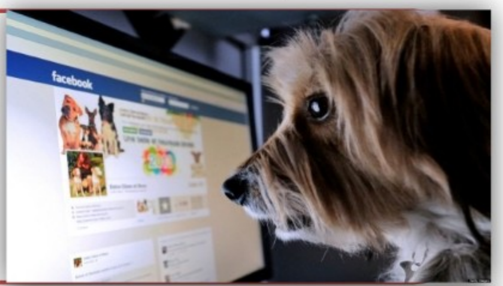
- With different costs attached to each

Cloud Provisioning Levels

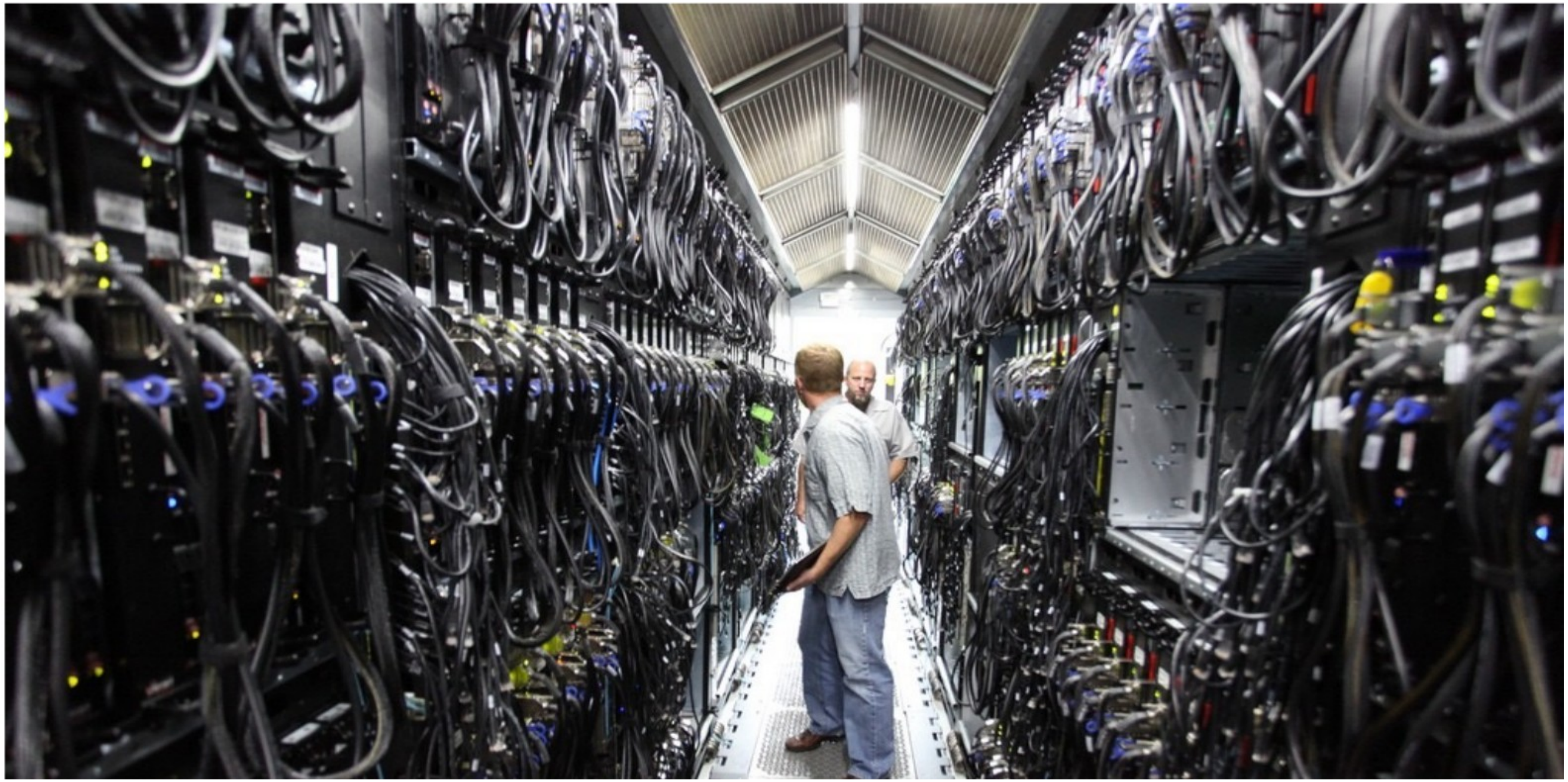


From: Zhang, Cheng and Boutaba. “Cloud computing: state-of-the-art and research challenges”. Journal of Internet Services and Applications 1 (1). May 2010. Springer.

Cloud Provisioning Levels

	What is Provided	Usage	Example
SaaS	Turn-key Application	Application transactions: run jobs, manipulate data, etc.	
PaaS	Runtime environment: software packages and storage support	Develop applications	
IaaS	Barebones hardware resources (CPU, memory, disk, networking) + OS	Customise runtime environments	

Public Deployments



Public Deployments

Provider	SaaS	PaaS	IaaS
Google	Google Docs, Cloud Dataflow	App Engine, Firebase	Compute Engine
Microsoft	Office 365, Outlook.com	Azure Functions, CosmoDB	Azure IaaS, Azure Batch
Amazon	KMS, Amplify, QuickSight	S3, Aurora, EMR, DynamoDB, Beanstalk	EC2, Spot, ELB
Oracle			OCI
SalesForce	SalesCloud	VMforce	*SalesForce*
Yahoo	YQL	Hadoop clusters	
IBM	LotusLive	WebSphere CA	CloudBurst
RackSpace		Mosso	
Joyent		Accelerator	SmartDataCenter
DigitalOcean			*DigitalOcean*

Private Deployments

- Outsourced private clouds:
  - On-site clouds managed by 3rd parties, e.g. IBM,HP, Oracle.
- Open-source virtual infrastructure managers:
  - OpenStack: A collaboration project between RackSpace, NASA and several commercial companies.
  - OpenNebula: Advanced virtual environmentadministration.
  - Eucalyptus: API compatible with Amazon EC2 & S3.
  - CloudStack Community : By Cloud.com
  - Nimbus: Based on GlobusToolkit 4.

Deployment Models

	Upfront Cost	Time to Build	Security Risk
Public	Low	Low	High
Private Outsourced	High	Medium	Low
Private Internally Managed	High	High	Medium

Summary

- Peer-to-peer network technologies have grown and shrunk in overall popularity over time, but remain a firm fixture in distributed systems today

- Cloud Computing offers things-as-a-service, changing our relationship with physical compute; cloud and peer-to-peer are commonly used together

Further reading

- Section 13.2.3 of Tanenbaum & van Steen; Sections 16 & 17 of Coulouris & al

- Chapter 10 of Coulouris & et. al