Lesson 8: Peer-to-Peer Architectures

Phillip J. Windley, Ph.D.

CS462 – Large-Scale Distributed Systems

Contents

Introduction

Peer-to-peer (P2P) architectures are networked systems where each node in the network is on equal footing with every other node.

The largest and most successful P2P network is the Internet itself. Every machine connected to the network gets an IP address and can interact with any other machine on the Internet so long as it knows that machine's IP address.

The Web is not a P2P architecture because it sees some machines as clients and some as servers. They are not peers.

Special purpose P2P systems are frequently built on top of the Internet. Such a network is called an *overlay network*.

Designing a functional P2P system can be difficult because humans think in stories—linearly. And we love centralized systems because they're easy to understand.

This lesson introduces concepts and algorithms for building practical decentralized peer-to-peer systems.

Lesson Objectives

After completing this lesson, you should be able to:

1. Explain the eight fallacies of distributed computing
2. Give examples of the CAP theorem and explain tradeoffs
3. Show how distributed file sharing works and relate it to consensus
4. Describe distributed discovery and show how a distributed hash table works

Reading

Read the following:

- Fallacies of Distributed Computing Explained (PDF) by Arnon Rotem-Gal-Oz (http://www.rgoarchitects.com/Files/fallacies.pdf)
- A plain English introduction to CAP Theorem by Kaushik Sathupadi (http://ksat.me/a-plain-english-introduction-to-cap-theorem/)
- Perspectives on the CAP Theorem (PDF) by Seth Gilbert and Nancy A. Lynch (http://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf)
- The Scalability of Swarming Peer-to-Peer Content Delivery by Daniel Stutzbach, Daniel Zappala, and Reza Rejaie (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.1599&rep=rep1&type=pdf)
- Bittorrent, how it works? (https://www.youtube.com/watch?v=6PWUCFmOQwQ)
- Chord: a scalable peer-to-peer lookup protocol for internet applications (PDF) by Ion Stoica, *et. al.* (in LMS)
- Intro to Distributed Hash Tables by real (https://www.freedomlayer.org/intro-to-distributed-hash-tables-dhts.html)

Additional Resources

Additional Resources:

- Availability and Consistency by Werner Vogels (http://www.infoq.com/presentations/availability-consistency)
- Eventually Consistent – Revisited by Werner Vogels (http://www.allthingsdistributed.com/2008/12/eventually_consistent.html)
- The Short History of Napster 1.0 by Alex Winter (http://www.wired.com/2013/04/napster/)

Fallacies of Distributed Computing

Developers often reason poorly about distributed systems because of misconceptions they hold

Some Reasons Distributed Systems Are Hard to Get Right

Fallacies are false beliefs based on unsound arguments. Fallacies are often unconscious.

Developers frequently run into problems when they scale up a prototype. One key difference between prototypes and production code is taking into account fallacies.

While the number of mistaken assumptions one can make is large, a few are so widespread that they rise to the level of being a common fallacy.

By understanding these fallacies and the arguments that lead us to believe them, we can avoid the problems they cause in our architectures. Or, at least, more accurately pinpoint the root cause of a failure after the fact.

The Eight Fallacies of Distributed Computing

In 1994, Peter Deutsch published a list of seven poor assumptions that distributed system architects often make. In 1997, James Gosling added an eighth.

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

The Network Is Reliable

We've all heard that the Internet routes around problems. And generally speaking, that statement is true—until it isn't.

While the Internet in general is quite reliable, that doesn't mean that Comcast built redundant fiber links to your neighborhood or the data center engineers didn't make a mistake when they configured their routers.

Networks fail all the time. Distributed systems have to be designed to be mostly (and sometimes just occasionally) connected and still function.

Latency Is Zero

A bundle of nanoseconds

Latency in distributed systems is the amount of time it takes to get a response to a message sent to a remote process.

Latency can be caused by propagation delay, network congestion, data serialization, data transmission, and application processing and delay.

Network delay is usually so small that it disappears for humans, but to machines, it's a very long time.

For example, hard disk access times are measured in milliseconds while RAM access times are measured in microseconds or even nanoseconds.

Sending a packet over a network from the US to Europe can take 10's of microseconds. So, to a program, communicating with remote machines appears *very slow*.

Bandwidth Is Infinite

Bandwidth is the carrying capacity of the network. In distributed systems, you can think of it as the total throughput of the remote process.

Bandwidth is an emergent feature of the architectural choices and can often be increased with additional time and money. But it is not infinite and it is certainly not free.

A *scalable* architecture has the ability to increase its throughput to meet the needs of it's clients. Designing for scalability is a tough problem that good distributed system architects keep top of mind.

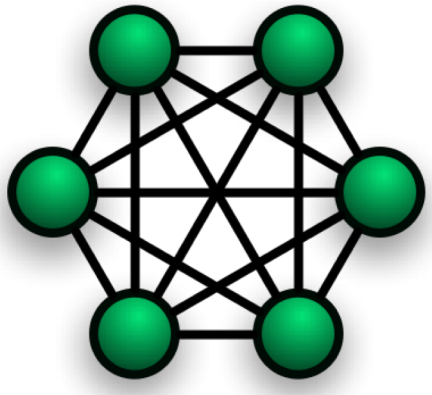
The Network Is Secure

The only completely secure system is one that's not connected to the Internet.

Of course, everyone knows this. But many systems are developed with security as a second order concern. Sometimes the system is a prototype that grew up. And pressure to deliver always gets in the way.

Distributed systems are especially vulnerable because, by definition, they're using networks to communicate. Network security is a complicated subject, but many security problems can be mitigated by well-known design principles.

Topology Doesn't Change



Network topology can change when new network nodes are added or a node is deleted. Topology can also change because of *spanning tree* changes in the network itself, making previously close processes appear more distant.

When the topology changes, assumptions about where remote processes are and how to reach them can change as well. Configurations that rely on fixed numbers of components, hand-built configuration files (instead of discovery), or poor expectations about latency can fail when faced with a topology change.

There Is One Administrator

Distributed systems cannot be coordinated by fiat.

Most distributed systems make use of resources that are controlled by multiple administrators. For a decentralized system where the remote processes are under the control of separate entities, the is no central entity who can force compliance or cooperation.

Distributed system designers have to take system ownership and administration into account when considering how their system will deploy, how to handle failures, user accounts and provisioning, and a host of other concerns.

Transport Cost Is Zero

Networks cost money. Bandwidth costs money. If your messages are small, the cost may be insignificant, but many payloads are not.

As we saw in Lesson 01, overcoming latency problems is even more expensive because you're in a race with physics. When latency matters, reducing it can cost millions of dollars.

Underestimating transport costs can turn a project or business into a good idea that failed when funding becomes unfeasible.

The Network Is Homogenous

Distributed systems run on networks with different physical layers, different topologies, and different protocols.

Moreover, distributed applications may deal with different APIs that use differing serialization formats, authentication protocols, and design philosophies.

Successful distributed applications flexibly handle these differences. Successful distributed system architects use dynamic applications and discovery rather than static configuration in their design wherever possible.

The CAP Theorem

Distributed system architects are frequently forced to make trade-offs between consistency, availability, and network partitioning

The Cap Theorem

Eric Brewer originally formulated what has come to be known as the CAP Theorem in 1998. The CAP Theorem states that it is impossible for a distributed system to simultaneously guarantee:

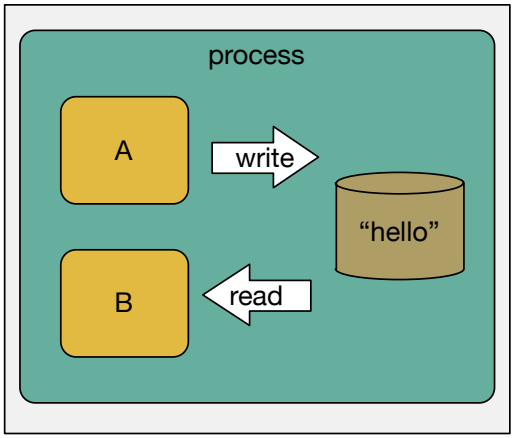
- Consistency—all nodes see the same data at the same time
- Availability—every request received by a *non-failing node* in the system must result in a response
- Partition tolerance—the system continues to operate despite arbitrary partitioning due to network failures

Let's explore these further.

The Availability Problem

One way to get consistency is to make sure everything is looking at the same data.

The figure shows two algorithms, A and B, in the same process and sharing a database. They see the same data. When A writes to the database, B sees the new value.



As we saw in Lesson 07, this configuration can only achieve consistency by carefully controlling when A and B access the data. B may not always be allowed to read A's changes due to locking, isolation, congestion, and other factors.

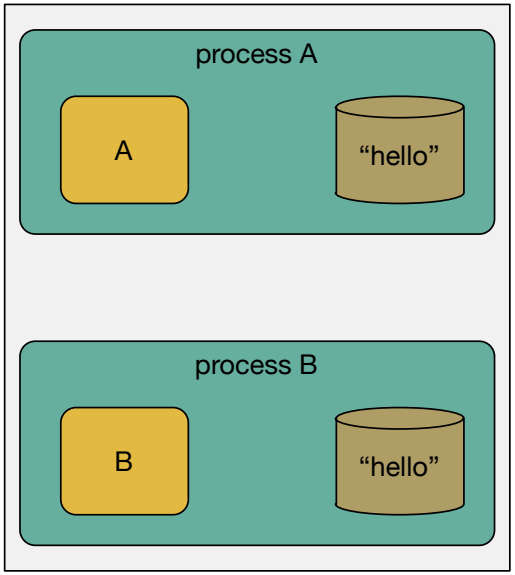
Put another way, the data isn't consistently *available* to B.

Problem Solved! Duplicate the Data

We can solve the availability problem by duplicating the data and creating two processes, A and B—one for each of the two algorithms.

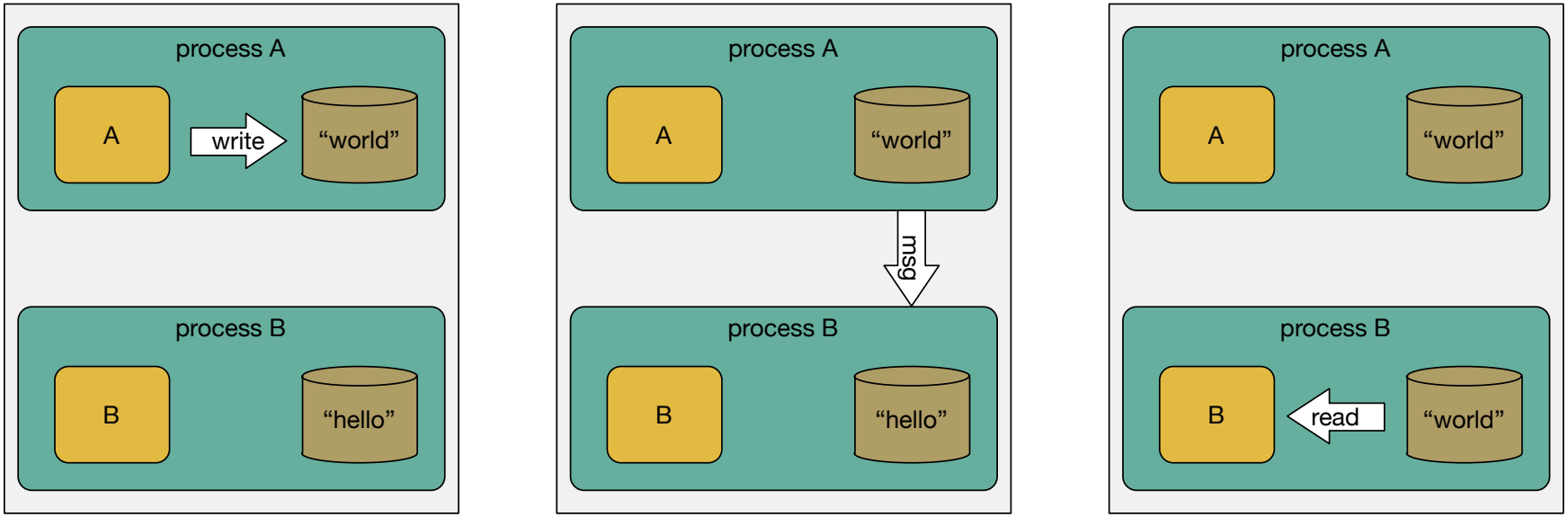
A and B both have a separate data store.

Of course, without further work, an update by A doesn't result in an update to B's data. To make this work, we need a method for synchronizing the data in A and B's data stores.



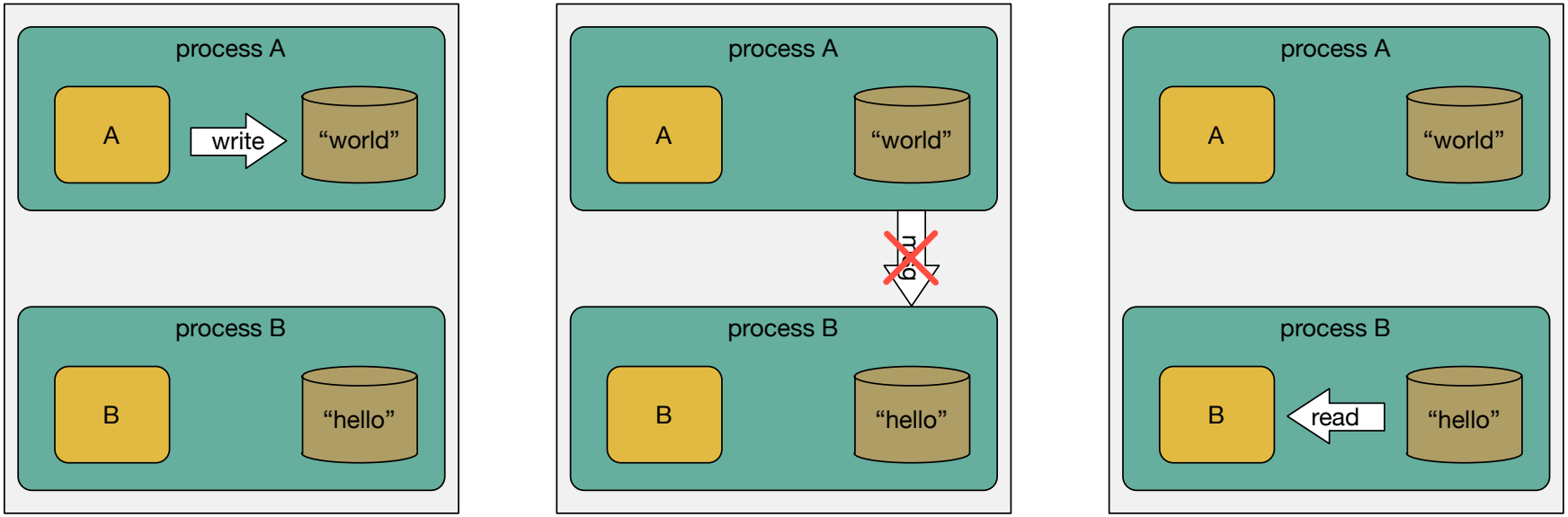
Achieving Consistency

Suppose A writes the value “world” into the data store. A and B are inconsistent. A message from A to B can restore consistency so that a read by B returns the new value. We’ve now solved the *consistency* problem too! We’re on a roll.

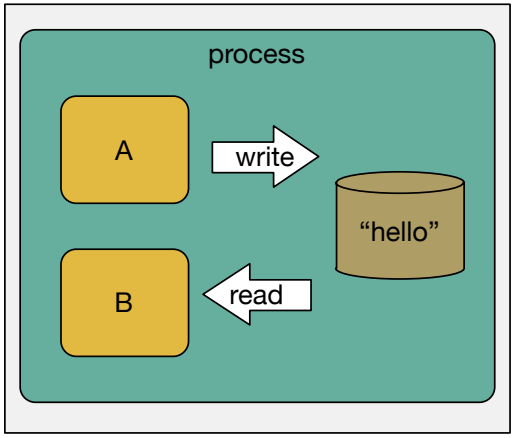


Partitioning the Network

But there's a problem. Suppose A writes the value "world" into the data store leaving A and B inconsistent. This time, however, the synchronization message from A to B is lost. This is a simple example of a *partition*. When B reads, it gets the old value.



Getting Rid of the Partition



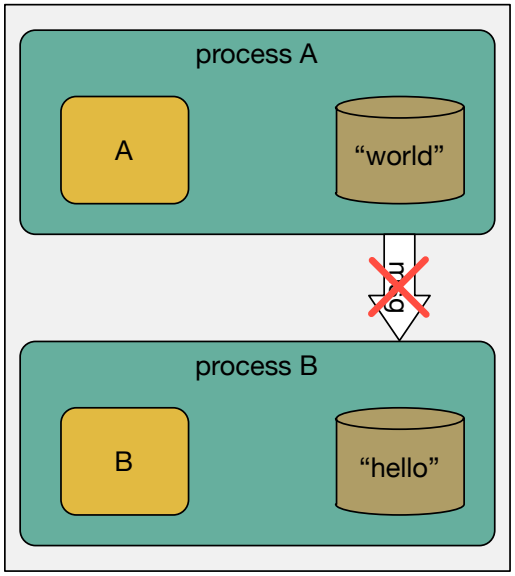
Unfortunately, the way to solve the potential for partitioning the network is to get rid of it and then we're back where we started.

There are many things we can do to mitigate the problems of partitioning. We learned about many of them in the *Failure and Distributed Systems* section of Lesson 07. Even so, we still have to recognize that despite our best efforts, our two processes may fail to achieve consensus and spoil consistency.

There are good ways to guarantee any two of the CAP properties, but for all three, the best we can do is mitigate the problems.

Distributed systems must deal with partitions as a fact of life. Consequently, distributed systems are usually trading off consistency and availability.

Eventually Consistent



Even when partitions happen, strategies like retry and undo can usually fix the problem given some time. The watchword for distributed data consistency is "eventually."

You may not think that eventual consistency is acceptable, but for a surprisingly large set of problems, it is.

Suppose for example, that a network partition at Amazon results in you and someone else buying the same last copy of *The Live Web.* The partition prevented the process you were using from seeing that the last copy had already been purchased.

Amazon has several options: they can refund your money or they can delay your order and send you a copy when one becomes available. Both of these are better, from Amazon's perspective than making you wait and potentially losing your business.

A bank, on the other hand, usually favors consistency over availability for obvious reasons.

The New pH of Databases Transactions: BASE vs ACID

We've seen that ACID is used to describe constraints in traditional databases. ACID systems use complex mechanisms to assure strong consistency.

Distributed databases use the (too cute by half) acronym BASE to describe constraints:

- **B**asically **A**vailable—there will always be a response to a request (although it might be "failed").
- **S**oft state—assume the state will change over time, even when there's no input.
- **E**ventually consistent—once a system stops receiving input it will eventually reach consistency.

Availability and scalability are the highest priorities of a BASE-oriented system.

Swarm File Sharing

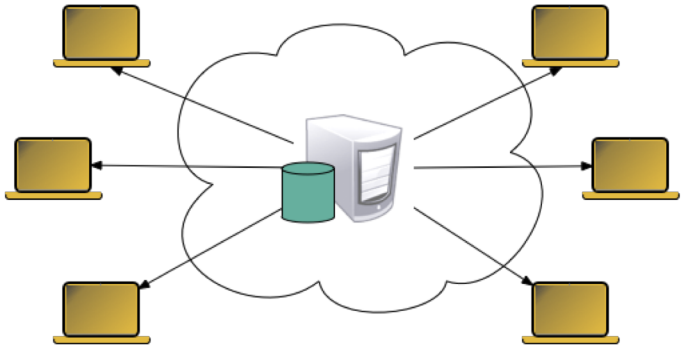
One of the most popular uses of peer-to-peer systems is efficiently sharing large files

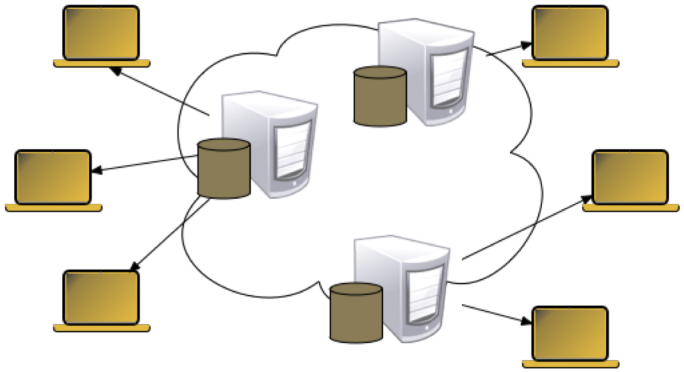
Content Delivery Networks

Transferring large files requires bandwidth—lots of it. Companies that do this for a living like Netflix, Microsoft, Apple and so on, use content delivery networks (CDN) to save money and provide a better user experience.

A CDN works by pre-positioning large files that are anticipated to be in high demand, like an OS upgrade, on proxy servers spread around the Internet.

These proxy servers effectively cache the large files and ensure that demand is spread out to multiple servers in many locations.





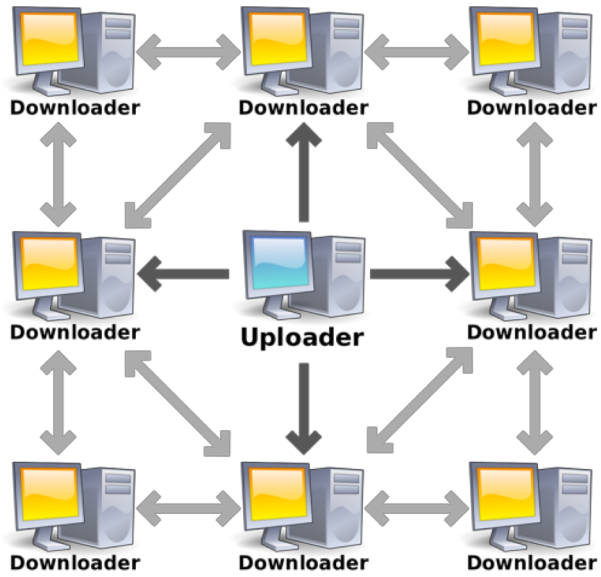
Swarming File Transfers

Swarming file transfer is a viable option when CDNs don't solve the problem.

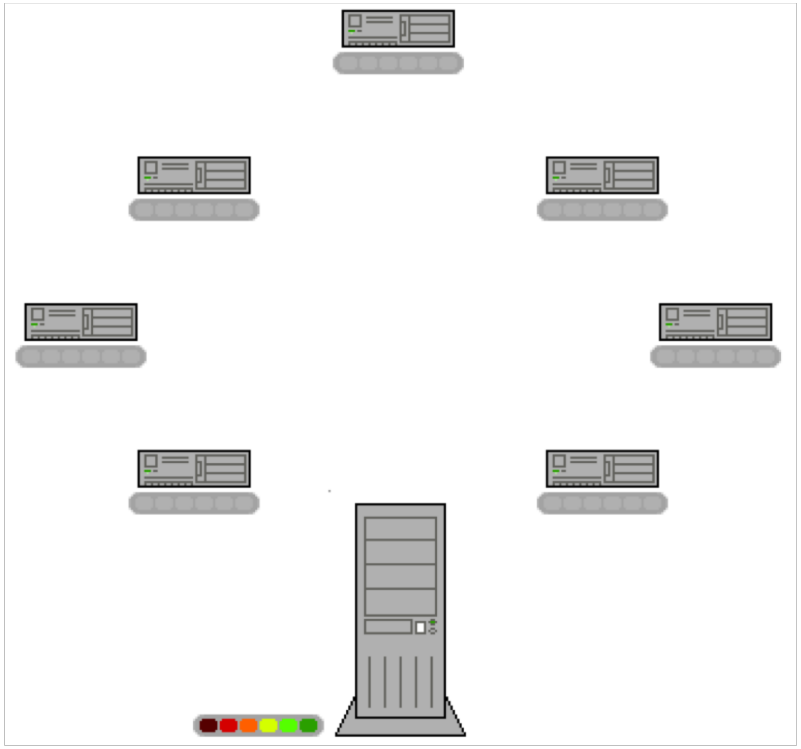
CDNs work fine for companies who can afford to buy their services and when demand can be anticipated. But plenty of small companies, and even individuals, need to share large files.

Peer-to-peer file distribution, sometimes called swarming file distribution, solves this problem by exploiting the bandwidth and server capacity of a community of computers rather than relying on one server.

In swarming file transfer, an uploader starts everything off, but then clients download files from each other.



Segmented File Transfer



Segmented file transfer extends swarming by breaking the file into segments that clients can exchange as smaller chunks to assemble the complete file.

In this animation, the colored bars beneath the clients represent individual segments of the file.

The seed (bottom) transfers segments to different machines in the network. After the initial segment's transfer, the segments are transferred directly from client to client.

The original seeder only needs to send one copy of the file for all the clients to receive a copy. The load is spread out among all the clients.

File Sharing Protocols

There are numerous file sharing protocols of varying sophistication. Perhaps the most used is Bittorrent.

The most famous is probably Napster, since it created a significant controversy because of its use to transfer music and video files, often against the copyright owner's wishes.

There are numerous Bittorrent clients available and services like Bittorrent Sync, a peer-to-peer replacement for Dropbox, are built on top of the protocol.

Explain file sharing protocols to a friend. After they understand what they are and how they can be used, explore the history of Napster. Then discuss this question: *Is peer-to-peer file sharing wrong?*

Discussion Exercise

Distributed Discovery

A key problem for peer-to-peer systems is discovery—determining which peer can services a given need

Napster's Downfall

Napster was a peer-to-peer file sharing service that was shutdown because it was being used to share materials without copyright owner's permission. Napster's directory was a single point of failure. Taking it out killed Napster.

In addition, Napster was found culpable because while the sharing happened peer-to-peer, Napster ran a central directory that allowed users to find which peers had which files.

This was also Napster's technical undoing. Music companies killed the network by killing the discovery service.

Bittorrent traditionally uses *trackers*—directories--to serve torrents. But trackers can run anywhere and a torrent can be on multiple trackers. Consequently, shutting down one tracker doesn't take down the entire network.

People find trackers using Google, but that's just another central directory. How can we distribute the directory?

Directories are Everywhere

Directories map keys to values.

The world is full of directories:

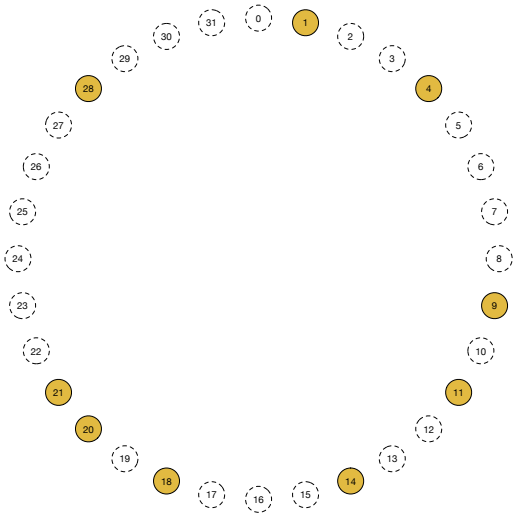
- A phone book maps names to numbers.
- Google maps search terms to search results.
- DNS maps domain names to IP addresses.
- The blockchain is a ledger but people sometimes use it as a directory.

Directory Exercise

Complete the table to the right by placing check marks in the cells that show the properties of each directory. Be prepared to justify your answers.

	Distributed	Decentralized	Heterarchical
Phone book			
Google			
DNS			
Blockchain			

Distributed Hash Tables



A distributed hash table (DHT) is a directory with no central service. DHTs automatically heal when nodes fail. New nodes can also join at any time.

DHTs form a ring in the address space with each node knowing about its successor and predecessor nodes.

A simple lookup algorithm is to linearly ask down the chain until the key is found, but that's slower than it needs to be. Instead with an address space of *N*, a table of *log(N)* entries can give us sufficient data so that *log(N)* hops will get us the value associated with a particular key.

DHTs provide a fast and efficient decentralized directory system.

There are many DHT algorithms including Kademlia and Chord. The examples on the following pages use the Chord algorithm.

Example: A Small DHT

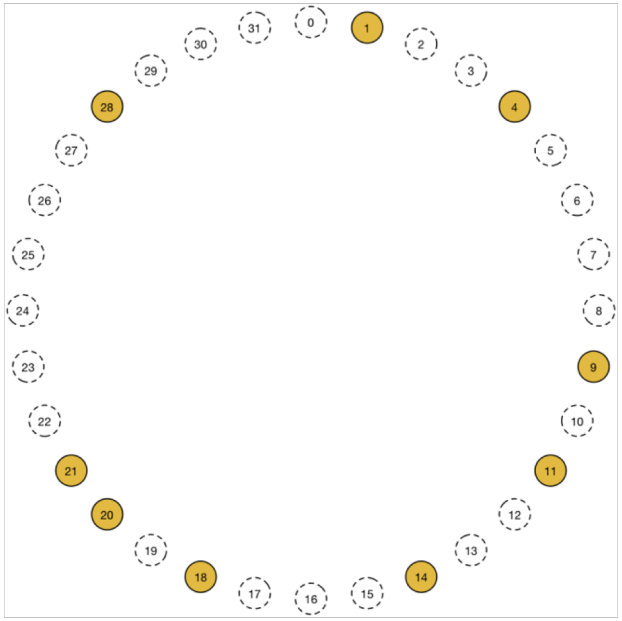
Let's see how Chord manages lookup.

The variable *m* represents the logarithmic size of the address space. For simplicity, we'll let *m* = 5.

With *m* = 5, we can support as many as *32* nodes in the system.

DHTs are sparsely populated. In the diagram to the right, only *9* out of *32* nodes exist. This is even more densely populated than most DHTs since with *m=128* or *m=160* we'd have lots of space (2^{128} or 2^{160}) and even millions of nodes would be a small fraction.

Each node knows its predecessor and successor in the ring. Thus, node *14* knows that it comes after *11* and before *18*.



Example: A Small DHT: Finger Tables

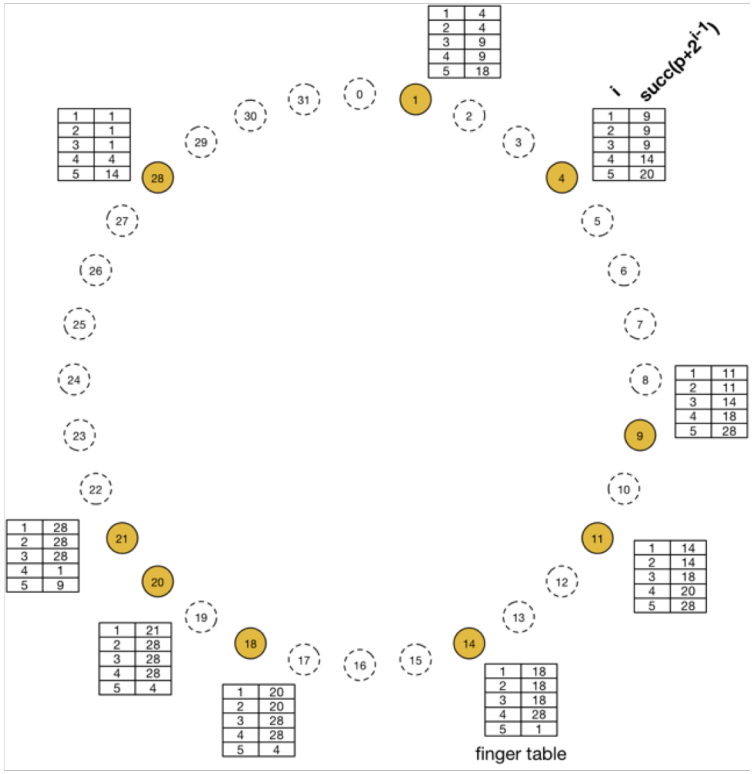
Chord uses a data structure called a finger table to hold information used for looking up keys.

Finger tables have *m* rows.

So, in the example at the right, each node keeps information about 5 other nodes. Even in a system with an address space of 2^{160} possible nodes, each node only needs a table with 160 entries.

Line *i* in node *p's* table shows the next node a logarithmic number of hops from *p* using the formula $succ(p+2^{i-1})$. Note that *succ()* here is the next node, not increment.

Study the finger tables for each node at the right and understand why they are correct.



Example: A Small DHT: *lookup(26)*

Looking up key 26 from node 1 involves forwarding the request to a series of nodes that are progressively exponentially closer to the node responsible for that key.

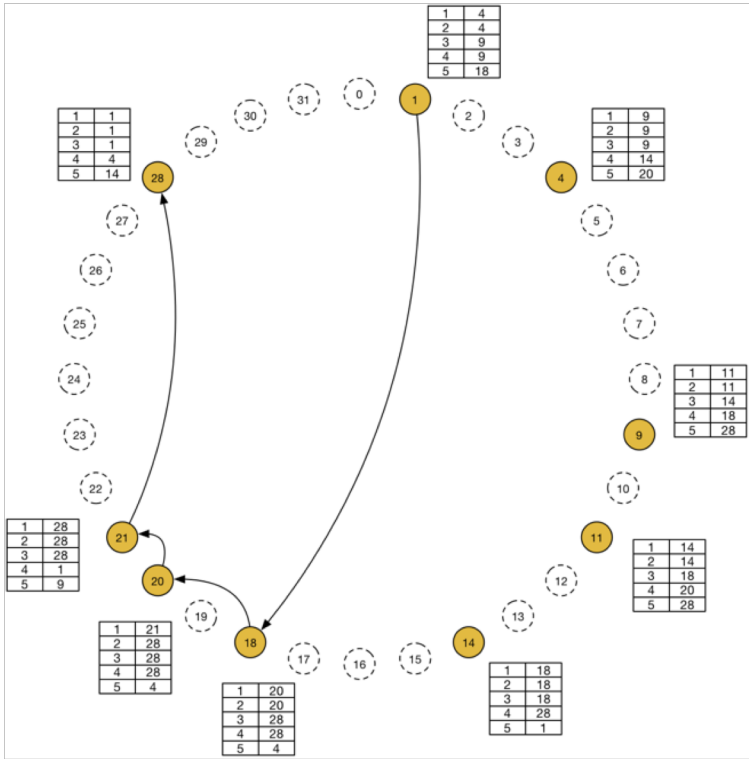
In the figure, no value in node *1*'s finger table is greater than *26*, so we hop to the highest value, *18*, taking the request halfway around the ring.

Since *20* < *26* < *28*, node *18* forwards to *20*.

Similarly, since *21* < *26* < *28*, *20* forwards to *21*.

Finally *26* is less than any entry in *21*'s finger table, so the value in row *1*, namely *28*, is the node responsible for key *26*.

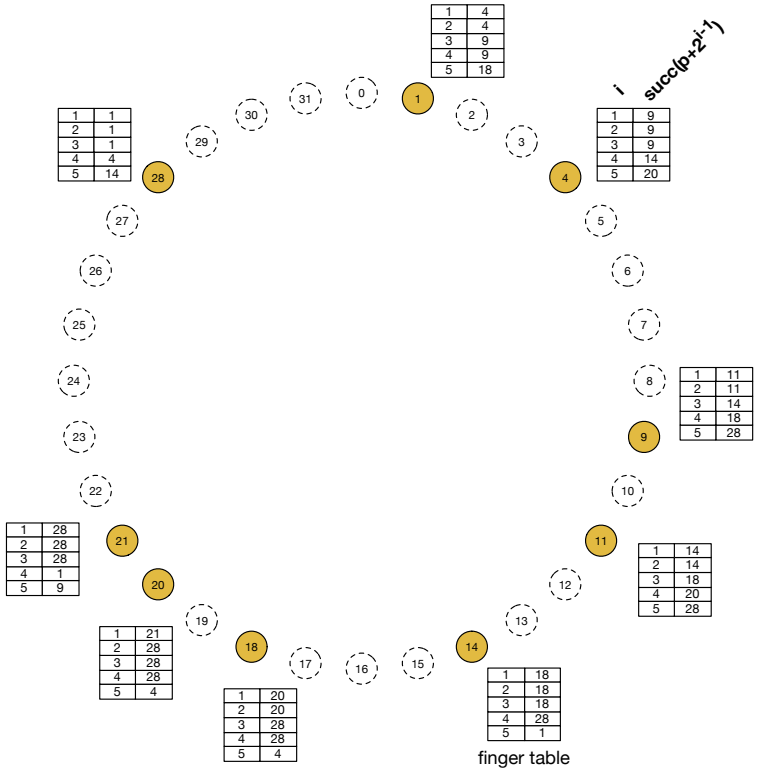
Note that some lookups loop across the top. Tables use modulo arithmetic to stay within the address space.



Example: A Small DHT: *lookup(12)*

Using the figure to the right, see if you can write down the sequence of hops that would be involved in looking up key 12 from node 28.

Turn to the next page when you have your answer.



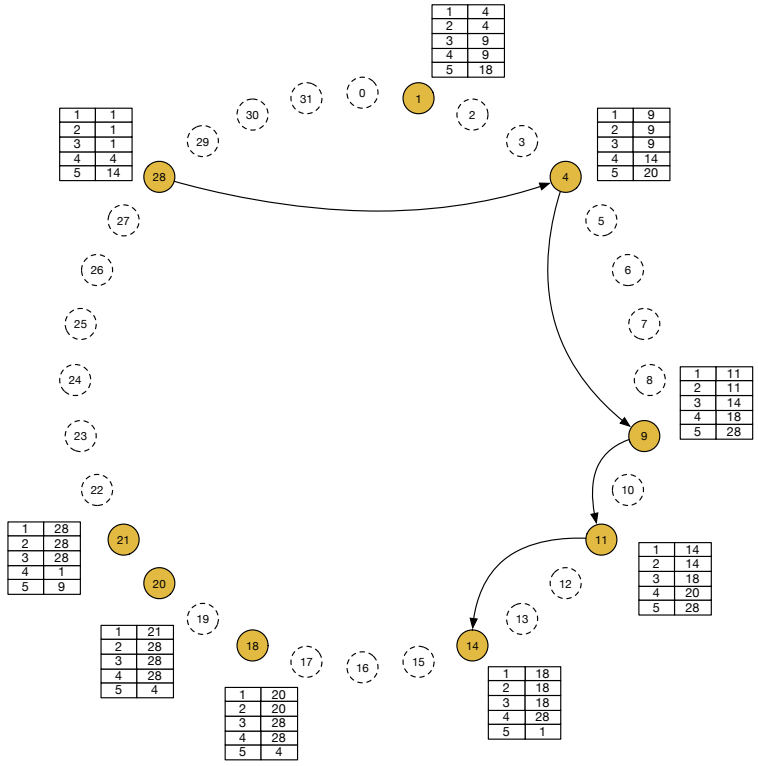
Example: A Small DHT: *lookup(12)*

The figure to the right shows the sequence of hops that would be involved in looking up key 12 from node 28.

The lookup follows the sequence

28, 4, 9, 11, 14

Node 14 is responsible for key 12.



Example: A Small DHT: *insert(7)*

When a node, *p*, wants to join, it looks up the *succ(p+1)* and calculates its own finger table.

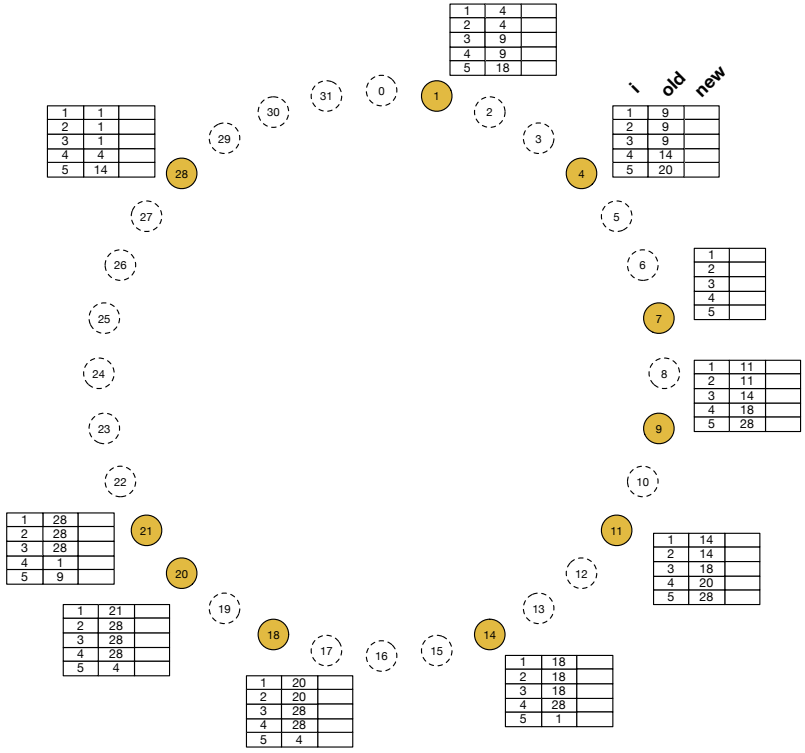
Potentially, each node also has to update its finger table.

Suppose we insert a node at 7. Calculate the new finger tables for each node.

Hint: not every finger table needs to be updated. Can you determine a rule for deciding if a finger table should be checked for updating?

In a Chord-based DHT, each node is running a background process that continually checks to see if it's finger table is up to date. These check messages are overhead on the DHT network.

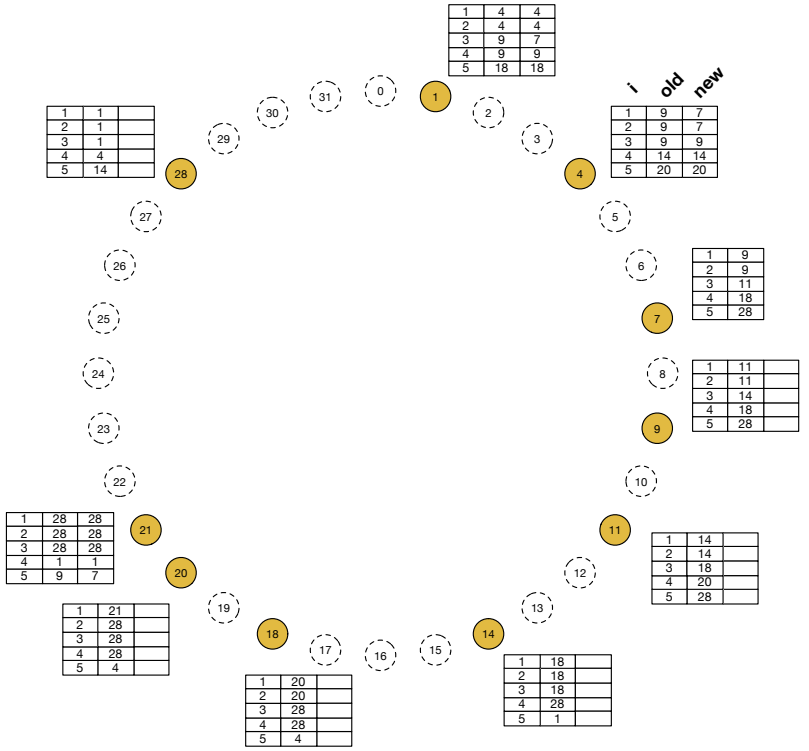
Turn the page for the answer.



Example: A Small DHT: *insert(7)*

The finger tables at the right have the old (before 7 is inserted) and new (after 7 is inserted) values for each row.

Note that tables in nodes with a value in the last row greater than the node number we're inserting *might* need to be updated.



Conclusion

Summary & Review

Credits

Summary and Review

In this lesson we've explores common problems that peer-to-peer architectures face.

There are a number of common fallacies. Distributed system architects often assume things that aren't true when designing peer-to-peer systems.

The CAP theorem is a expression of one set of trade-offs that peer-to-peer architects must make. Since distributed systems must live with network partitioning, the choice comes down to one between consistency and availability.

One big use for peer-to-peer architectures is sharing large files. Swarm file sharing uses peers, cryptographic signatures, and segmented files to efficiently overcome availability problems caused by congestion at a single file server.

Distributed discovery makes use of clever algorithms to efficiently lookup values that are distributed among a set of peers. DHTs overcome the need for centralization while making lookup efficient.

Credits

Photos and Diagrams:

- US Navy Equipment Operator (https://commons.wikimedia.org/wiki/File:US_Navy_060127-N-4215N-005_Equipment_Operator_3rd_Class_Tyler_Randall_assigned_to_Naval_Mobile_Construction_Battalion_Four_(NMCB-4),_Det_San_Clemente_Island_operates_an_excavator_during_a_rock_crushing_evolution.jpg), Public domain
- Steel pipes (https://commons.wikimedia.org/wiki/File:Metal_tubes_stored_in_a_yard.jpg), CC BY-SA 2.0
- Anonymous mask (https://pixabay.com/en/anonymous-mask-protest-people-615157/), Public domain
- Fully-Connected Topology (https://commons.wikimedia.org/wiki/File:NetworkTopology-FullyConnected.png), Public Domain
- Three-headed dog (https://commons.wikimedia.org/wiki/File:Three-headed_dog_at_Steampunk_HQ.jpg), Public domain
- USI Router (https://commons.wikimedia.org/wiki/File:USI_router.jpg), CC BY-4.0

Credits (cont.)

Photos and Diagrams:

- Eric Brewer (https://commons.wikimedia.org/wiki/File:TNW_Con_EU15_-_Eric_Brewer_(scientist)-2.jpg), CC BY-SA 4.0
- Two small test tubes (https://commons.wikimedia.org/wiki/File:Two_small_test_tubes_held_in_spring_clamps.jpg), CC BY-SA 3.0
- Torrentcomp small by Wikiadd (https://commons.wikimedia.org/wiki/File:Torrentcomp_small.gif), CC BY-SA 3.0
- Bittorrent (https://commons.wikimedia.org/wiki/File:BitTorrent_network.svg), CC BY-SA 3.0
- HK CityU Directory (https://commons.wikimedia.org/wiki/File:HK_CityU_To_Yuen_Building_directory_yellow_sign_Shek_Kip_Mei_Estate_Sept-2012.JPG), CC BY-SA 3.0
- Bundle of nanoseconds (https://commons.wikimedia.org/wiki/File:Grace_Murray_Hopper_visualizing_nanoseconds.png), Public domain

All other photos and diagrams are either commonly available logos or property of the author.