September 9

Tuesday, September 9, 2025 11:14 AM

Resource Discovery 1

Recall: We can use DHCP (Dynamic Host Configuration Protocol) to identify all the resources on the network.
-> everyone broadcasts who they are

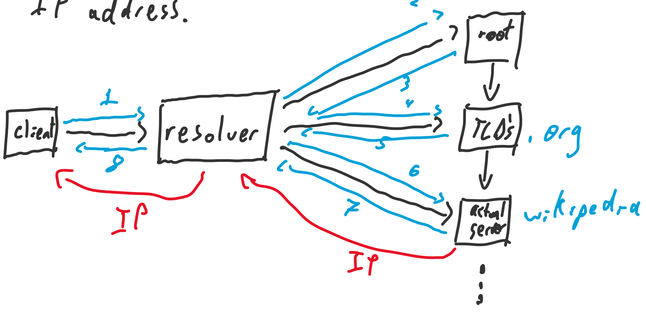
What about the internet? Can we use DHCP internet-wide?
-> No: - scale, too many devices
- there isn't a list of every device connected to the internet (foreshadowing: we can find every web server, but we probably don't want to)

To find stuff on the internet we use DNS (Domain Name System).
-> resolves a location in the form of a URL

eg http://en.wikipedia.org/

-> root server — directs you to org
-> org directs you to wikipedia
-> wikipedia directs you to en

A DNS Resolver is a server (computer) that will go through the recursive process of resolving a domain name to an IP address.



Suppose you want to start up a new DNS resolver. How would you do it? How do you start?
-> hard-code the IP addresses of the 13 root servers

DNS gives us location transparency - why is this great?
-> easier to use
-> can move hosts without the client needing to know
-> can hide complexity (are we talking to 1 machine or a cluster? We don't care!)

Is the location 100% hidden?
-> No, at some point it must resolve to an IP address

Are DNS entries globally unique?
-> Yes! DNS distributes this

How can we make DNS faster?
-> Use a cache! Once a resolver finds an IP address, it can store it.
-> What if a host moves?
-> Time To Live (TTL): each cache entry is only valid for a fixed amount of time.
-> Is this secure? No! There are cache poisoning / DNS stealing attacks possible.

September 11

Thursday, September 11, 2025 11:09 AM

Resource Discovery (continued)

Web servers: we can just do it!

-> there is a web server daemon running on the university machines (silicon.cs.umanitoba.ca?), we just have to fill in the details

(1) Create a directory called public_html
(2) Run chmod a+x public_html
(3) Put something in public_html
(4) Run chmod a+r resource_name
(5) Check www-test.cs.umanitoba.ca/~umnetID/resource_name

We use URI's (Uniform Resource Identifiers) to refer to network resources. They have a standard format:

protocol://user:password@host:port/dir?query#fragment

ex: https://example.com/path/resource.txt?GET=query&Key=value#fragment

dir

no port number since https has a standardized port

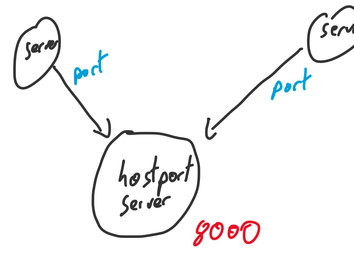
we used "/" - is this a path?
-> Yes! This is just a path on the host's computer

A port is a logical construct on a machine that specifies where some communication should be received.

Ports are good!
-> things are standardized (ish)
-> we can communicate with multiple things at the same time

Free ports: 1024 - 65535

But: what happens if a host changes ports?
-> it's gone!
-> how might we fix this?
  -> set up a server to track the hosts that could move
  -> could list them or act as an intermediary or something else



Recall that DNS requires that domains are globally unique. How do you ensure this?
-> what if 2 people register the same domain at the same time, or 2 DNS servers announce changes that are different?

This is a concurrent system - things are happening at the same time.

We need to establish consensus - everyone needs to agree on the state of the system. This is not an easy problem to solve!

September 16

Tuesday, September 16, 2025 11:30 AM

Message Passing

What is message passing?
-> Sending any message
-> Sending a message is sharing state

What issues do we see in message passing?
-> latency
-> what if a message doesn't arrive?
-> what if a host drops?
-> what if a message is garbled or incomplete?
-> what if messages arrive out of order?

What is state? Any data stored by a host, client, or server.

How does an SSH server know who you are?
-> link state - the status of each connection
-> the server stores who is connected on each port
-> TCP holds a connection, UDP does not

What are some examples of stateless servers?
-> Basic HTTP site
-> FTP (sometimes)
-> Relays
-> IRC only holds link state (in theory)
-> root DNS servers (mostly)

What are some examples of stateful servers?
-> any dynamic website (storing information about users)
-> ssh

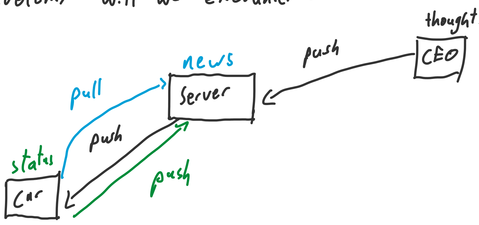
Client state: how much state does the client need to get appropriate data?
-> It depends, but usually as little as possible (some drawbacks to this)

Case study: suppose we are building a news application. How can we ensure that the user only sees news stories they haven't already seen? What is the thick client solution? Thin client?
-> thick client gets all articles from server, only displays new ones
-> thin client only gets new articles from the server

The thin client uses less communication! (in this example)

Message Passing (continued)

Case study: your new car displays news feeds, does software updates, and sends the CEO's thoughts directly to the dash. It also sends information about the car's status to the manufacturer. How do we design this system?
-> what state is where?
-> what problems will we encounter?

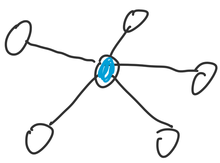


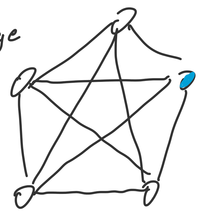
Problems?
-> lots of load on server
-> the car moves? What if the connection drops?

How do we establish connections?
-> server should have DNS or well-known IP address
=> cars & brain can establish connections to server
-> car & brain can periodically check if connection is still up, and re-establish if necessary

How do we handle "offline" with thick clients?
-> what do you do when the client comes back online?
-> What does dropbox do? Git?
-> There is state synced between client and server. What if the state has changed?
↳ take more recent?
↳ have fixed priority over who to trust?
↳ record all changes and do them in order?
↳ let the user choose?

Consider a <u>two-phase commit</u> (2PC) database:
-> Distributed database (all hosts have a copy or all hosts have part of database)
-> For each update, all participants must vote if the update gets carried out or not. If all vote yes, it happens, otherwise it doesn't.
-> where is the state stored?
↳ central server
↳ whoever initiated the change





When would we use a relay vs a buffer?
• Buffer
-> we need the receiver to actually get the message
-> sending messages to a host that may be busy (maybe a server with many clients)

• Relay
-> sending messages from a server to many clients (too many to store at a server)
-> if there are many messages and you don't need all of them

To summarize?
-> Thin clients (usually) don't hold state. Thick clients do.
-> Stateful servers hold state.
-> Syncing between thick clients and stateful servers is complicated.
-> General rule: if state is stored somewhere, it should probably be processed there.
-> Consider who needs to push or pull what state.
-> KISS - Keep it simple, stupid!

September 23

Tuesday, September 23, 2025 11:10 AM

Practical Communication

Encoding - what do sockets send?
- binary
- text -> an intermediary encoding, encode to text and then to binary. Why?
- Why would we encode directly to binary?
-> some things are hard to translate to text
-> can be more compressed
- What are some disadvantages of binary?
-> can be... problematic to decode (endianess!)
- Text has standardized encodings - ASCII, UTF-8

How do we send an object on a socket?
- Flatten the object using JSON
- Python also has encodings directly into binary - "pickle" and "marshall" (pickle is preferred)

With flattening, what can we not encode?
- Circular linked list
- File-like objects (sockets, for example)
- How can we fix this?
-> make pointers again
-> each object has an "encode" method?

UDP can be sent to multiple locations - multicast
- when would this be useful?
- what are the limitations of this?

-> announcing you're on a network (DHCP?)
-> chatrooms/ group calls/ group chats?
-> News/ Post feeds
Is everyone is on the same frame/block of bytes

-> Lossy because of UDP
-> what can you receive back?
-> hard to roll back

TCP vs UDP - which protocols use which? Why are most TCP?
-> can get reliable responses, guarantee message delivery
- Not NTP?
-> UDP is faster, it needs to be precise

UDP does not have a connection => we can write any return address!
- Why does this not work with TCP to the same degree?
-> need acknowledgement to send messages

Blocking or not?
- What are the pros/ cons of blocking?
- What are the pros/ cons of polling?

- Blocking:
-> will respond to/ process the message as soon as it's available
-> Synchronizes states
-> know for sure message was delivered before we continue
-> Good for listening on a port
-> what if the message never arrives?
-> what if we need to monitor multiple ports? - multithreading?

- Polling:
-> Can do other tasks while we wait
-> What if we need a message to continue? - possible, but annoying

- What about a small device? Might not have blocking capabilities
-> Real-time system problems

Is there a third option?
-> block with a timeout

September 25

Thursday, September 25, 2025 11:22 AM

Practical Communication (continued)

Coding Exercise: I have a server running on goose.cs.umanitoba.ca on port 9001. This server will receive 1 TCP message, then respond with 2 TCP messages. Try to connect to it!
-> send whatever (nice) message you like

This server has a timeout - why?
-> what if a user/client opens a connection and never sends anything! -> the whole server gets bricked

Also, we can only process 1 client at a time - what if they take a while?
-> multithreading?
-> Python's select?

select is passed non-blocking file-like objects, and it will block until something happens on any of them

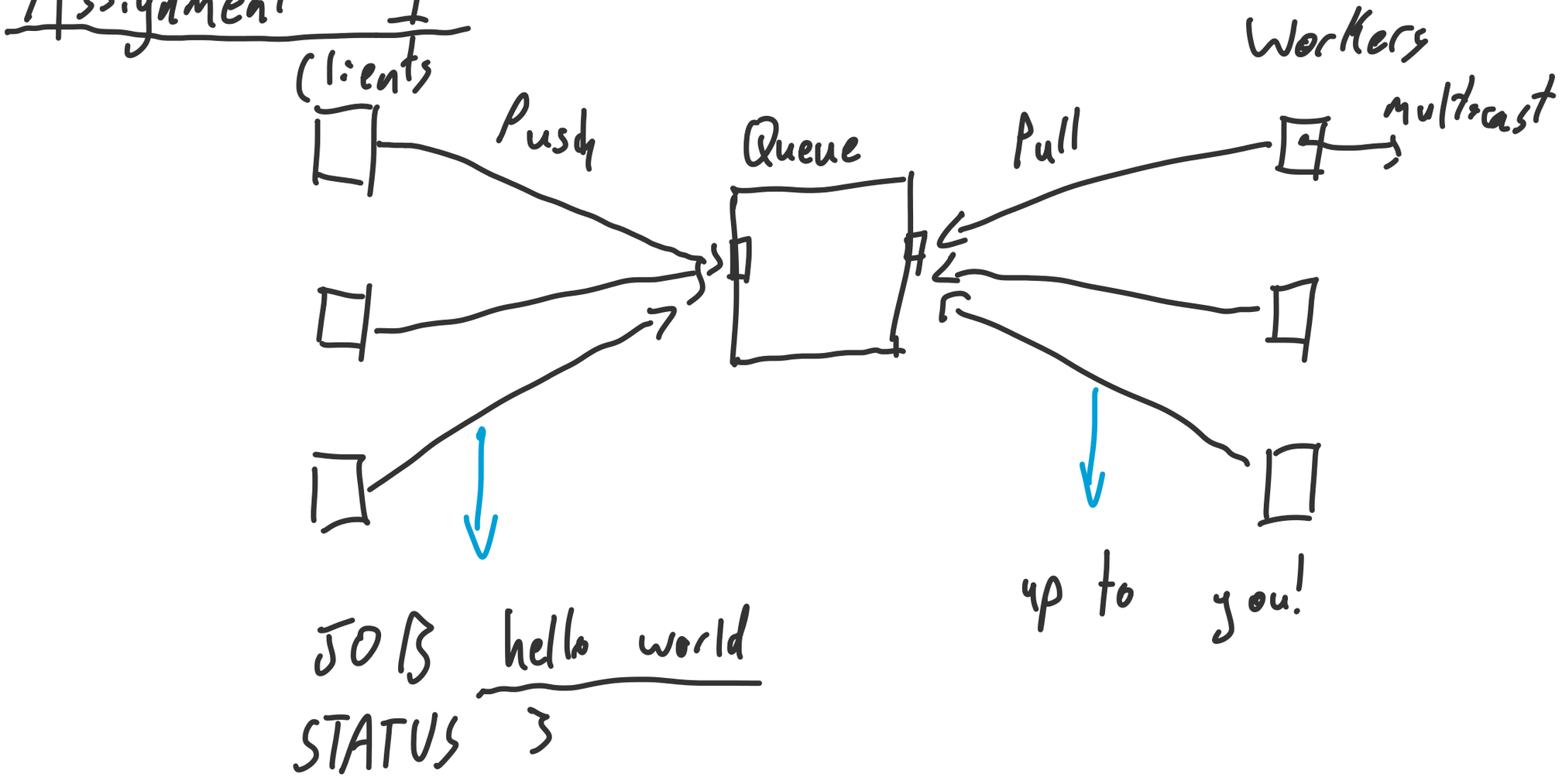
```
readable, writable, exceptional = select.select(inputs, outputs, inputs, timeout)
```

October 2

Thursday, October 2, 2025 11:22 AM

Practical Communication (continued)

Assignment 1



---

unix tools:

- telnet - wraps a socket, input through terminal
- netcat/nc - wraps a socket, input through stdin
- curl - downloads a resource at a URL
- wget - downloads a resource at a URL (but worse)

- We can use these to get web pages

Design Case study:

- Suppose you are designing a system that needs to transfer a lot of data from one machine to another, and it needs to happen quickly. How do you design it? What protocols/socket types might you use?

-> mix of TCP and UDP
-> buffer between them
-> send data in chunks via UDP, use TCP to verify
-> give each a packet/chunk a number, communicate number of chunks & missed chunks with TCP

In-class Exercise

I have a CTF running on goose.cs.umanitoba.ca on port 9001. Try to connect to it to get the word of the day!

October 7

Tuesday, October 7, 2025 11:20 AM

Web Computing

HTTP: What does "idempotent" mean?
-> if you perform the same operation many times, you get the same result.

Is a google search idempotent?
-> it isn't, performing a search changes the state in the server.

Request types! Case study - adding a discussion post. Consider the following sequence of actions:

(1) Go to Piazza GET
(2) Log into Piazza GET? POST? CONNECT?
(3) Move to 3010 (if you aren't there already) GET
(4) Add a node POST
(5) Update the node PUT/PATCH POST??

Which HTTP methods will be used? What data will be sent?

October 9

Thursday, October 9, 2025 11:11 AM

Web Computing (continued)

Coding Exercise: Take the web server code from last class and add a hit counter (ie, "you are user number x").

In-Class Example: Tina Docs - like google docs, but not.
-> create text files
-> edit text files
-> save text files and open them later

Two pages:

Menu
-> select from all files
-> make new file

GET /
-> print the menu outline
-> generate a list of all existing files

POST /
-> create new file
-> refresh menu page

Compose
-> edit an existing file
-> save the file
-> move back to menu

GET /file_name
-> show compose page
-> fill compose page with contents of file

POST /file_name
-> update file contents
-> refresh compose page (?)

October 14

Tuesday, October 14, 2025 11:20 AM

Web Computing (continued)

Continuing example: Tiny Docs

→ Recall: Knockoff Google docs.

→ Show all files, send post to create new file, then nothing. So, let's finish it!

October 16

Thursday, October 16, 2025 11:41 AM

Web Computing (continued)

Tina Docs V2: what is going on?

/

GET - gives menu page with all files as buttons
POST - create a new (empty) file

/compose

GET - skeleton of a compose page

/compose/filename
GET - compose page, triggers GET /filename
POST - saves file contents (triggers POST /filename)

/filename
GET - file text
POST - save file text

Active server pages (Asp.net php) are dynamic web sites that update on the server.
-> JS runs on the client and requests things from the server
-> PHP (eg) runs on the server and dynamically sends updates to the client

What are some problems with PHP?
(1) Server is doing "display work", taking up resources
(2) Hard to cache
(3) Too much in one place - server code in client file (HTML)
(4) Can be insecure - code is running on your server

More modern server frameworks:
- Node.js Express
- Python Flask
- Java Spring / Spring boot

Case study: Rewrite aurora with a RESTful backend
- Make paths/routes:
  -> Login
  -> List courses
  -> View a specific course
  -> Register for a course

root page - login
↳ Post to log in

/courses
↳ GET
↳ probably no post unless we need to make a new course

/courses/coursenum
↳ GET
↳ probably no post (maybe to register?)

/users/usernum
↳ POST to register

both okay

October 21

Tuesday, October 21, 2025 11:13 AM

Server Performance

How can we divide up work on a server?

What does a web server do?
-> Listen on a port (port 80?)
-> Get new TCP client
-> Get the request
-> Fulfill that request

What if we get a lot of clients at once?
-> we fulfill requests one at a time, leaving clients waiting

Instead, we can handle this with multithreading.

Two basic techniques:
(1) Accept the client, read the request, and send it to a thread to be fulfilled.
(2) Accept the client and send it to the thread to do everything.

We usually use the second option — we want to divide work, the "main" thread just needs to listen for new connections

What is a thread?
-> a thread is a stack and instruction pointer within a program. So, multiple threads can be running different code concurrently, but they share the same memory.

Python threads are weird and a bit of a lie.
-> they run concurrently but not in parallel
-> Python has a Global Interpreter Lock (GIL), which means that at most one Python thread can be running at a time.

It is, however, good for parallel I/O — other threads can continue working while one thread is waiting, which is what we need it for.

Code is pretty easy, just a function call.

How else can we handle scale?
-> Caching - the art of not doing the same work twice

Very common in web computing - browsers do this automatically

We can cache static responses, but what if it isn't static?
-> depending on the situation, we may be able to cache, but we can't if the operation isn't idempotent

Caching can happen at the server or the client - we usually do it at the client, but it matters more at the server when we have multiple layers (eg, what if the server needs to query a database to fulfill a client's request?)

HTML5 provides a client-side database, localStorage, so our client code can implement a cache, querying it with JS and making an XHR request if necessary (not there or old)

Performance attacks - DoS (denial of service), DDoS (distributed denial of service)
-> How do we handle this? - first, we probably need an error message to say we're temporarily down.

How do we recover and rebuild our deployment? No one answer, but broadly we want to reconnect. This should not involve human interaction. We need an automatic way of restarting and reporting the error.

What are the best requests for DDoS-ing? Why?
-> lots of I/O (because waiting)
-> small request size, big response size

What parts of a system can be DoS'd?
-> bandwidth
-> CPU time
-> memory
-> connections

How do we address these?
- CPU:
  -> write better code
  -> add more machines or better machines
  -> break up the problem, add a new layer

- Bandwidth:
-> load balancing, multiple locations
-> can also pass less data
-> rate limit

- Connections (port saturation):
-> shorter timeouts (if possible)
-> add more machines

- Memory: difficult, it depends

October 23

Thursday, October 23, 2025 11:40 AM

Server Performance (continued)

Q: Is rate-limiting a solution to DoS?
-> Kind of? It depends how you do it

What are the HTTP headers related to rate-limiting? What response codes might we use?
-> 429 Too Many Requests
-> what if it's malicious? This won't actually stop them
-> it does help, however, if it's an accident - good users will respect a 429 (eg, Reddit hug of death)

Consistency

Why do we want consistency?
-> how do you determine "real" data? Human intervention?
-> a unique key should not be associated with more than 1 value.
-> everyone should have the same view of the world (within reason...)

Is consistency inherently a distributed systems problem?
-> no? Kind of?

Are all distributed systems subject to consistency/inconsistency?
-> no, if all the state is in one place, it isn't an issue. However, state usually isn't all in one place, we usually replicate something.

What could impact consistency? Why might our system be inconsistent?
-> latency
-> crashing / someone goes offline
-> concurrency issues: poor programming, lack of coordination
-> resource limitations in general (eg processor speed)

Who is responsible for consistency?
-> whoever is managing state (us)

Do we actually need consistency? It's hard, and we are lazy.
-> it depends...

How could we work around it? What strategies could we use?
-> avoid the need for consistency at all (eg by splitting up the state)
-> relax the consistency constraints (eg, maybe it's good enough if we are eventually consistent)

How do we detect inconsistency?
-> some sort of Monte Carlo technique? Statistical evaluation?
-> random spot checks?
-> operation logging
-> checksums / comparing hashes
-> timestamps (not perfect - what if there is a malicious host or it updates at the same time?)
-> version data
=> also hard

Once we detect inconsistency, how can we resolve it?
-> lots of ways! Depends on the application

-> automically take the most recent
-> git pull - more generally, leave it up to the user to choose which state to use
-> go with the majority
-> have a "leader" that dictates the state
-> Two-phase commit (not perfect, but good) - doesn't fix it once it has happened

October 30

Thursday, October 30, 2025 11:13 AM

Elections

Up to this point: one server (potentially multiple copies), many clients.
→ can clients talk to each other? No... but they can in a peer-to-peer (p2p) network.

In many P2P algorithms, we need to choose a leader/coordinator. How do we do this?

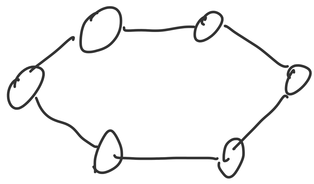
Simplest is the Bully Algorithm
- Each node has an ID/priority value
- To start an election, a host broadcasts its priority
- If a host receiving this message has a higher priority, it broadcasts its priority
- After 2 "rounds", everyone knows who has the highest priority, and they are the leader
- In practice, the leader then broadcasts to tell everyone they won

How could this fail?
→ what if a message was lost or delivered late?
→ what if two nodes have the same priority? (probably a bad network)

How many messages does this take? Let n be the number of hosts.
- Each host that broadcasts sends n-1 messages
- How many hosts have to broadcast? If the initiating host has the k-th largest priority, then k. In the best case, k=1. In the worst case, k=n.

⇒ number of messages is between n-1 and n(n-1) (potentially with an additional n-1 for the confirmation).

Another approach is a Ring Election.



Each host has a "previous" and "next" host in the ring. Every host still has a priority.
- The initiating node starts a message that initially contains only its "name" and priority. It sends this message to the next node.
- If a node receives a message that does not include its name & priority, it adds itself to the message and sends it to the next node. It then echoes the message it received to the previous node (think: read receipt).
- If a node receives a message that does contain its name & priority, it knows all the nodes and selects the largest priority as the leader. Then, it removes itself from the message and forwards it to the next node, echoing what it received to the previous node.

Note that each node needs to keep track of all names & priorities it receives for this to work.

Activity

Your name: your name/pseudonym
Your priority: the last 2 digits of your student number

November 4

Tuesday, November 4, 2025 11:09 AM

Byzantine Generals

What is a lie? Our distributed systems don't have generals or wars.
- Transmission failure
- Malfunction / bit flip
- Bugs
- Actual bad actors - happen in distributed systems where the host could be anyone.

What is consensus?
- The non-traitors agree
- If the general is not a traitor, the value agreed on must be whatever they sent

Activity
- In each group, the general is the person with the closest birthday.
- Tear group paper into 4/5 pieces. On one, write "Traitor"

The algorithm:
(1) The general sends everyone a message (either attack or retreat)
(2) everyone else sends the non-generals a message with what the general said
(3) output / choose the majority

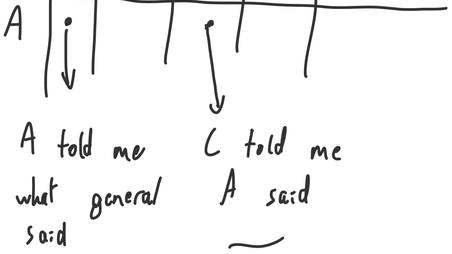
Important: the traitor can send whatever they want!

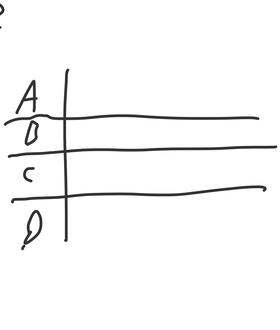
Now: do it again, but with 2 traitors!
-> still possible to achieve consensus, but it is no longer guaranteed

1 more round! - merge into bigger groups
- Go down to 2 traitors

New algorithm:
(1) General sends a message to everyone
(2) Everyone sends a message to non-generals with what the general said AND their name
(3) For each message you receive in step 2, send a message to everyone who isn't a general or the person who sent the message saying your name, the person who sent the message, and what they said

(1) A / R
(2) [name] A/R
(3) [name] [other name] said A/R





Conclusion: so, so many messages - grows exponentially with the number of traitors

November 6

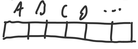
Thursday, November 6, 2025 11:19 AM

Byzantine Generals (continued)

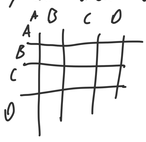
Activity Debrief

- 4/5 peers, 1 traitor => always get consensus
- 4/5 peers, 2 traitors => may or may not get consensus
  - ↳ why is it still possible to get consensus?
    - the traitors could just play along
    - the traitors can't coordinate. If they could, it would be possible to always break consensus.
- 8/9 peers, 2 traitors => no one finished (or came close). Why?

Suppose you write down every message you get. For OM(1), how many do you get? n-1, you receive a message from everyone else.



For OM(2), we drew a table:



There are $\Theta(n^2)$ messages / we receive $\Theta(n^2)$ messages.

In general, for OM(m), we receive $\Theta(n^m)$ messages. This is a lot! It grows exponentially with m.

Blockchain

Quick note: in the videos, Rob talks about CAP theorem — don't worry about this, we will talk about it soon.

Blockchain is an... interesting consensus model.

Recap: The state is just a linked list, and all peers have a full copy of it.

First: consistency vs consensus
-> consistency means "we have the same data"
-> consensus means "we think this data is right"

Blockchains almost never have strong consistency, they usually only guarantee weak at best. They do achieve consensus, eventually (ish).

Blockchains achieve consensus... unusually. Normally, we look at all the peers and take the majority opinion. Here, we focus solely on the chain.
-> whichever chain is longest is "correct" and we should choose it for our consensus, even if only 1 peer has that chain.
-> For blockchains, what is a "traitor"?
  ↳ Someone who has any other chain

At what point would a blockchain be considered consistent?
-> possibly when there are blocks that all participating peers have received and committed. This is definitely weak consistency at best.

To add a block, we need to correctly hash the chain. Why does this work?
- Growing the chain is hard, which helps solve some problems:
  (1) Race conditions
  (2) Attacks
  (3) Trust

Race conditions: what if two peers add a block at the same time? How does growing the chain being hard help with race conditions?
- It makes it less likely by slowing the growth of the chain.

Attacks: what would happen if it was easy to create blocks?
-> It would be very easy to make the "bad" chain the longest one.

Trust: we assume peers willing to do the work of adding a block are likely not bad actors.

Most PoW blockchains dynamically adjust the difficulty of adding blocks. Why?
-> race conditions! If it's too fast, make it slower
-> if it's too slow, the network can get stalled, so we can make it faster.

November 20

Thursday, November 20, 2025 11:19 AM

Clocks

Quick Recap:
- Event: could be anything, context dependent
- "Happened-Before": what it sounds like!
- Partial Ordering: can determine "happened-before" for some event pairs
- Total Order: for all event pairs

- Clock: measures time
- Clock drift/skew: the idea that no two physical clocks are perfectly in sync
-> also can mean the amount they differ by

Does NTP solve our problem? No!
-> Doesn't have enough precision
-> Drift can still happen after a sync
-> How long does it take for messages to arrive? Imprecise!

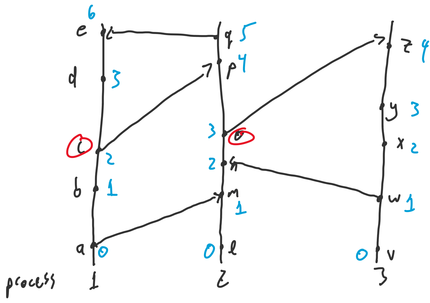
Why is the Facebook Time Card interesting?
-> it's cheap & open source
-> very precise
-> not perfect, two things could conceivably happen at the same microsecond

Instead, we can share some state to help order the events.

Quick recap:
- Lamport Timestamps: Monotonically increasing event numbers/indices
- Logical Clock: System for relative ordering of events

To implement this, each process has a clock. The clock is incremented every time an event occurs.
-> If we are sending a message, include your current clock count
-> If you receive a message, set your clock to max(curr Clock, rec Clock) + 1

Example:



- What partial ordering do we get?

$$(a,l,v) \to (b,m,w) \to (c,n,x) \to (d,o,y) \to (p,z) \to q \to e$$
$$0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6$$

- This does not give a total ordering, because we do not know if, eg, a or l happened first

- To turn this into a total ordering, we can just assign each event a priority!

Problem: this says that $c \to o$. Did it? Not necessarily!
- This isn't exactly wrong, but it can ignore possibilities

Fidge's Algorithm addresses this:
- Each process keeps a list of the latest clock count from each process
- Each message is sent with the list of what the sender currently knows
- Recipient updates with new info (basically a gossip!)

This algorithm is more complicated and requires more communication, but it doesn't exclude possibilities.

Now, is ordering necessary?
- Not always!
- This can be very expensive, so we try to avoid it if possible.

Summary:
- Timing & ordering events are hard
- Even with perfect time, we have concurrency issues
- Logical clocks can help.

November 18

Tuesday, November 18, 2025 11:05 AM

CAP Theorem

C - Consistency
A - Availability
P - Partition tolerance

The theorem says we can have at most 2 of them.
-> Availability: all of the non-faulty hosts function no matter what.
-> Partition tolerance: the hosts (some of them) should continue functioning if the network is partitioned into 2 or more sub-networks.

Examples: where do each of the following sit?
- Blockchain:
  -> C? No, different peers may have different chains
  -> A? Yes!
  -> P? Yes! The nodes will continue to function, though their state will almost certainly not be consistent.

=> AP

- IRC:
-> C? No...
-> A? Yes
-> P? Yes
=> AP - messages are not sent between the partitions and will be lost. Alternatively, we could have CA, and a partition would break the system (we would need to stop)

- 2PC Database:
-> C? Yes, all hosts will always have the same state
-> A? Yes
-> P? No - in a partition, the hosts all stay available, but can no longer write new state

- Youtube: Let's think about the data, not the streaming part.
-> Maybe AP? - the updates propagate across the system over time, so we don't have consistency.

Now, let's design a file share, once for each possible CAP design

(1) PC - we want consistency and partition-tolerance but not availability. What could this look like?
-> a partition would leave part of the system unaccessable (one of the sub-networks would just crash).

(2) CA - consistency, availability, not partition tolerance.
-> If there is a partition, we need to stop writes - the system still responds to reads at any host and it will return the same value.

(3) AP - availability, partition tolerance, not consistency
-> Ignore partitions, there are just sub-networks now

November 27

Thursday, November 27, 2025 11:09 AM

Distributed Hash Tables

- A hash table, but distributed !!
- A system by which a Key space is distributed across multiple nodes for load distribution, availability, and/or resiliency
  -> Key space: the range of all possible values for a key

- Chord: a specific DHT protocol that we're focusing on
- Successor: the node after a given node in a ring
- Predecessor: the node before a given node in a ring
- Finger table: a collection of state for accelerating lookups

Why would we create a network as a ring?
-> leader elections and DHT's
-> simple & predictable
-> minimizes local state

Why not trees?
-> hard to repair
-> hard to route
-> root is extremely important - single point of failure
-> still sometimes used!

Rings and fallacies - do we fall prey to them?

(1) Network is reliable
  -> if the network is a physical ring, we are in trouble
  -> if not, we need to be able to recover

(2) Latency is zero
  -> Usually okay, but messages can still arrive out of order due to latency

(3) Topology doesn't change
-> yes, if the network is a physical ring, less of a problem if not

Now, how do we join the ring?
- Need to query a peer that is already in the ring - doesn't necessarily need to be the bootstrap, the bootstrap is there to ensure that the ring always exists

CAP theorem - probably AP. In the case of a network partition, each sub-network will be a correctly functioning ring, but the state between them is no longer consistent
-> possible to do a different design depending on requirements

Next, how do we query/look up a value?
- Query can come in at any node
- Go around the ring until we get to the node responsible for that key
- How does the message get back to the querier?
-> send return address with the query
-> go back around to the original node

What happens if a node goes offline?
- Repair the ring
- Reshuffle/reorganize the shares of the Key space
- This is slow! - re-insert all nodes
- All data is lost

Could we be nice on exit? What would we have to do?
- Connect predecessor and successor
- Transfer data, by default it all goes to the successor

How do we detect faults?
- Timeouts
- Ask predecessor who their successor is. If it isn't you, rejoin the ring.

How do we rejoin? Do we need to go to the bootstrap? No!
- Even in a partition, it's okay, since we can just rejoin wherever our predecessor is

December 4

Thursday, December 4, 2025 11:20 AM

Review

CAP Theorem

Suppose there are two ATM's implementing a P2P banking system. Given the CAP Thm, what designs are possible?

(1) CP - we need consistency and partition tolerance
-> Suppose there is a network partition. How do we maintain consistency without sacrificing functionality? -> Kill one of the machines. We lose availability but the remaining machine is consistent with itself and, from a global view, the system has not lost functionality.

(2) CA - stop allowing balance updates. Both machines are still running and they will have the same state, but the system has lost functionality.

(3) AP -> just go !!

RESTful API's

Consider a web application for an online store. It must have the following features:
- Users can create an account
- Can log in to an account
- Can view a list of all items
- Can view an item's description
- Can buy items
- Can add items to a wishlist
- Can remove items from a wishlist
- Can view their wishlist

Design a RESTful API for this application. Specify all paths, HTTP verbs, and responses.

/account/create
- POST - create a new account. Server should send a 200 OK.

/account/login
- POST - logs in - verifies username/pwd pair. Server should send a 200 OK and set a login cookie

/store
- GET - returns a list of all items

/store/:item-id
- GET - returns the description of the item
- POST - triggers the buying. Server should send a 202

/wishlist
- POST - specifies :item-id in the body. Server should return OK
- GET - returns all the items in the user's wishlist
- DELETE - specifies :item-id in the body. That item is removed from the wishlist and the server returns a success.

If an item doesn't exist, the server will always return a 404.

Blockchain

What kind of race conditions can arise in a blockchain? Will the system recover from them? Why or why not?

A race condition can arise if two peers add a block at the same time and those chains are tied as the longest (both the longest).

The system will recover because, eventually, one of the chains will become longer (or another chain could overtake them) and become authoritative.

December 2

Tuesday, December 2, 2025 11:20 AM

DHT's (continued)

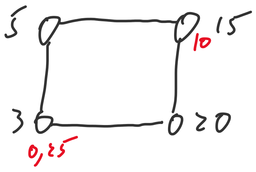
Activity

-> Form groups of at least 3
-> Select a bootstrap in some way
-> Determine your "node ID", student number mod 64
-> Form your ring!

Starting at bootstrap, add these keys:

9, 10, 17, 33, 38, 41, 44, 46, 55, 63

eg: if there are ID's:



insert ID

-> Key belongs to you if it is between your ID and your predecessor's ID
↳ [pred, my_id)

-> lookup for 11 -> start at the bootstrap, then go around the ring until it arrives at the node responsible for 11.

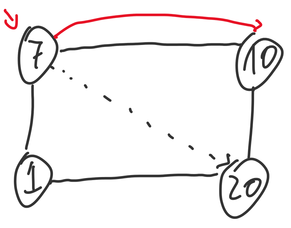
Next, we will fill 1 finger table entry at each node.
-> each node initiates a query for the key ID + 4. Then, this query will arrive at the node responsible for that key. This node will add their ID to the message and it will keep going around the ring until it reaches the initiator.

query: x
ID: —

-> when your query is returned to you, it will have exactly 1 ID on it, and this ID is the node responsible for your query. This gives a shortcut through the ring.

Example: If you are a node with ID 7, you could have the following finger table:

i	id + $2^{(i-1)}$	peer
1	8	10
2	9	10
3	11	20



So, if 7 is querying for 12, eg, it could skip 10 and forward it directly to 20 - if 20 controls 11, then 12 is held be someone at or after 20.

How do we update a finger table?
-> announce that you're leaving and your successor, allowing everyone to update finger tables. But UDP is unreliable, and a node might crash without updating. Can also just query again to get an updated value.