COMP 3010 Notes

Saksham Bedi

Jan 8, 2025 – April 9, 2025

Contents

1 Week 1 : Distributed Systems **5**
1.1 Distributed Systems 5
1.2 2 Broad Models 5
1.2.1 Thin vs Thick Client in Server-Client 5
1.3 What is cloud? 5
1.4 Resource Location 6
1.5 URI vs URL's 6
1.6 Ports 6
1.7 DNS 7

2 Week 2 : Client Server **8**
2.1 Server Client 8
2.2 Design Issues 8
2.2.1 Division of Labour 8
2.2.2 Reliability of Transmission 8
2.3 Server Design 8
2.4 Managing State 8
2.5 Concurrency 9
2.6 Work Distribution 9
2.7 State 9
2.7.1 Case Study: Network File System (NFS) 9
2.7.2 Case Study: UM Learn 9
2.7.3 Case Study: Tesla 9
2.8 Work Distribution 9
2.9 Issues with Work Distribution 9
2.10 Client-Server Architecture 10
2.11 Thick and Thin Clients 10
2.11.1 Thick Client 10
2.11.2 Thin Client 10
2.12 Web Browser 11
2.13 When to Use Which? 11
2.14 Message Passing 11
2.14.1 What is a Message? 11
2.14.2 Uses 12
2.14.3 How to Send Messages? 12
2.14.4 Blocking and Non-Blocking 12
2.14.5 Rendezvous 12
2.14.6 Call Back 12
2.14.7 Relay Buffer 13
2.15 Dijkstra's Algorithm 13

3 Week 3 : Webserver and Sockets **14**
3.1 Practical Communication 14
3.1.1 Unicast 14
3.1.2 Multicast 14
3.2 Theoretical Background: State Machines 14
3.2.1 TCP State Machine 14
3.2.2 UDP (Connectionless) 14
3.3 Blocking vs Non-blocking 14
3.3.1 Blocking 14
3.3.2 Non-blocking 14
3.3.3 Industry Application 15
3.4 Sockets 15
3.4.1 Theoretical Background: Sockets and the OSI Model 15
3.4.2 Layer 4 Connection 15
3.4.3 Layer 5 Communication 16
3.4.4 Layer 6 Presentation 16
3.4.5 Theoretical Background: Data Serialization 16

3.5 Socket How-To's . . . 16
3.5.1 Socket connection types: . . . 16
3.5.2 Steps . . . 16
3.5.3 Client Steps (TCP): . . . 17
3.5.4 TCP vs. UDP (Order of Operations) . . . 17
3.5.5 Questions: . . . 17
3.6 Practical Communication (In Class) . . . 18
3.7 Flattening . . . 18
3.7.1 Challenges with Encoding . . . 18
3.8 Multicast . . . 18
3.8.1 Uses of UDP Multicast: . . . 18
3.8.2 Limitations of UDP Multicast: . . . 19
3.8.3 Theoretical Background: IP Multicast . . . 19
3.9 Questions on TCP and UDP . . . 19

4 Week 4 : Webserver 20
4.1 Continution of Week 3 . . . 20

5 Week 5 : Introduction to Web Computing 21
5.1 Web vs. Internet . . . 21
5.2 Web Computing Characteristics . . . 21
5.3 HTTP (HyperText Transfer Protocol) . . . 21
5.3.1 How HTTP Works . . . 21
5.3.2 HTTP is Stateless . . . 21
5.3.3 HTTP Request Structure . . . 21
5.3.4 Useful HTTP Headers . . . 22
5.3.5 HTTP Response Structure . . . 22
5.3.6 HTTP Status Codes . . . 22
5.4 Using Telnet to Request HTTP Resources . . . 23
5.5 Building an HTTP Server . . . 23
5.6 Understanding HTTP GET Requests . . . 23
5.7 Understanding HTTP POST Requests . . . 24
5.8 Parsing HTTP Requests . . . 24
5.9 State Management with Cookies . . . 24
5.10 Recommended Readings . . . 25

6 Week 6 : Introduction to Client-Side Web Computing 26
6.1 Key Technologies for Client-Side Web Interaction . . . 26
6.1.1 Forms (HTML Forms) . . . 26
6.2 Single Page Application . . . 26
6.3 XHR (XMLHttpRequest) . . . 26
6.4 DOM (Document Object Model) . . . 27
6.5 Cookies . . . 27
6.5.1 Cookies and State . . . 27
6.6 Server-Side Web Computing . . . 28
6.7 REST (Representational State Transfer) . . . 28
6.7.1 API Design for AURORA . . . 28
6.7.2 Twitter API Example . . . 28
6.7.3 Key Takeaways about REST . . . 29
6.8 CGI Scripts . . . 29
6.9 Recommended Readings . . . 29

7 Week 7 : Performance, Attacks, and Consistency 30
7.1 Scaling for High Traffic . . . 30
7.2 Strategies for Scaling . . . 30
7.2.1 Caching . . . 30
7.2.2 Client-Side vs. Server-Side Responsibilities . . . 30
7.3 Content Delivery Networks (CDNs) . . . 30
7.4 Attacks . . . 30
7.5 Error Handling . . . 31
7.6 Load Balancers . . . 31

7.7 Consistency . . . 31
7.7.1 Replicas . . . 31
7.7.2 Problems with replicas . . . 32
7.7.3 One solution : 2PC - Two Phase Commit . . . 32
7.7.4 3PC-Three Phase Commit . . . 32
7.8 Recommended Readings . . . 32

8 Week 8 : Elections **33**
8.1 Coordinators . . . 33
8.2 Bully Algorithm . . . 33
8.3 Ring Algorithm . . . 33

9 Week 9 : Consensus & Blockchain **35**
9.1 Byzantine Failures . . . 35
9.2 Consensus . . . 35
9.3 The Byzantine Generals Problem . . . 35
9.4 Byzantine Fault Tolerance . . . 35
9.4.1 Oral Message Algorithm (OM) . . . 35
9.5 Introduction to Blockchain . . . 36
9.5.1 Key Features of Blockchain . . . 36
9.5.2 Consensus and Blockchain Features . . . 36
9.5.3 Consistency vs Consensus . . . 36
9.5.4 Buzzwords . . . 37
9.6 Blockchain Components . . . 37
9.6.1 Blocks . . . 37
9.6.2 Mining and Proof-of-Work . . . 38
9.6.3 Consensus: Longest Chain Rule . . . 38
9.6.4 Race Conditions and 51% Attack . . . 38
9.7 Other Consensus Mechanisms . . . 39
9.8 Confirmations . . . 39
9.9 Implications for Distributed Systems . . . 39
9.10 Recommended Reading . . . 39

10 Week 10 : CAP Theorem **40**
10.1 CAP Theorem - Intro . . . 40
10.1.1 CAP Theorem Combinations . . . 40
10.2 Two-Phase Commit (2PC) as a CA System . . . 40
10.2.1 Partition Tolerance and Majority Rules . . . 41
10.3 Availability and Partition Tolerance (AP System) . . . 41
10.3.1 A Note About Even-Numbered Partition Tolerance . . . 41
10.4 Key Takeaways . . . 41

11 Week 11 : DHT(s) **42**
11.1 Introduction to Distributed Hash Tables (DHTs) . . . 42
11.2 Table ADT . . . 42
11.3 Basic Distributed Hash Table Implementation: A Linear Ring . . . 42
11.3.1 Concept . . . 42
11.3.2 Joining the Ring . . . 42
11.3.3 Lookup Operation . . . 43
11.3.4 Leaving the Ring (Graceful Shutdown) . . . 43
11.4 Handling Node Failure . . . 43
11.4.1 The Problem . . . 43
11.4.2 Detection . . . 43
11.4.3 Recovery (Basic) . . . 43
11.5 Improving Lookup Performance: Finger Tables . . . 44
11.5.1 The Problem with Linear Lookup . . . 44
11.5.2 The Solution: Finger Tables . . . 44
11.6 Race Conditions and Connection Issues . . . 44
11.6.1 Race Conditions . . . 44
11.6.2 Connection Issues . . . 44
11.7 Further Considerations . . . 44

1 Week 1 : Distributed Systems

This is the content for week 1.

1.1 Distributed Systems

A distributed system is a collection of independent computers that appear to users as a single coherent system. These systems are designed to work together, sharing resources and coordinating their actions to achieve common goals. Systems often called 'hosts'.

1.2 2 Broad Models

- **Server-Client**: A server with many clients. It's one machine (that is stateful) with all these dumb clients that are doing very little for us. Example: SSH is an example.
- **Peer to Peer**: A 2-way relation. (Example: Blockchain, torrent, Messaging app (Signal)).

1.2.1 Thin vs Thick Client in Server-Client

Thin client: A thin client is a lightweight computer device that relies heavily on a central server for most of its computational tasks.

Thick Clients: Also known as fat clients or rich clients, thick clients are more similar to traditional desktop computers. For Example: A modern web browser, such as when watching YouTube, receives a video stream from the server while offloading the decoding of the stream to the client.

Aspect	Thin Client	Thick Client
Processing Location	Relies on central servers for most processing tasks.	Performs significant processing locally with powerful hardware.
Network Dependency	Requires constant network connectivity to function.	Can function offline and perform tasks without server access.
Cost	Generally less expensive to deploy and maintain due to simpler hardware.	Higher cost due to more powerful hardware and maintenance needs.
Security	Enhanced security with centralized data storage.	Increased security risks due to local data storage.
Customization	Limited customization; centrally managed by IT administrators.	Allows more user customization and control over installed software.
Performance	May experience latency; dependent on network and server.	Better performance for resource-intensive tasks.
Local Resources	Minimal local storage and processing power (e.g., 8 GB storage).	Ample local resources, including storage and dedicated graphics.
User Interface	Simplified interface; primarily an access point to server-hosted applications.	Rich and advanced user interface with responsive user experience.
Energy Efficiency	Consumes less power, contributing to a smaller carbon footprint.	Consumes more power due to high-performance hardware.
Management	Centralized management makes updates and security policy enforcement easier.	Requires individual maintenance and management.

Table 1: Thin vs Thick Client Comparison

1.3 What is cloud?

Cloud computing refers to the delivery of computing services (such as servers, storage, databases, networking, software, analytics, and intelligence) over the internet ("the cloud") to offer faster innovation, flexible resources, and economies of scale. Elastic cloud is just adding more hosts to a distributed system.

1.4 Resource Location

How to discover resources, which can be the client itself. So how do we find them, how do we connect to them. There are different approaches:

- **Hard Code it:** Hard coding the address of the server ex: `130.179.28.115`. This is super hard to remember and it's not flexible. Flexible: if the server goes down there is no way to let people to know there is a different machine.
- **Configuration File:** Refer to the configuration file on the machine to find the IP address. For instances: IP ADDRESS. IPv4 address is a collection of 4 numbers in a format `1.1.1.1`. The numbers can go from 1 - 255.

```
ip addr
# or
ifconfig
# or
hostname
```

Codeblock 1: Getting the hostname from the configration file

- **Announce yourself:** When a laptop/printer joins a network usually a printer, or a fileshare and sometimes a laptop where the printer (for this instance) is connected to the router. The printer says *Hello, what's available for me* to the router and the router will repeat this message for the others. And you basically announce yourself that I am a host. This is part of the DHCP (Dynamic Host Configuration Protocol). This is a MAC protocol. This will also give an IP address back. Works well in the local network where we have 256 hosts.
- **Use a lookup service:** Announcing yourself doesn't scale well at a global (a billion, trillion or maybe even more scale) level. For instance: use a lookup service to translate a Domain Name into an IP Address from `Google.com` → `'123.345.456.678'`. Key/Value pair or a hash value. Lookups are $\mathcal{O}(1)$ operation.
- **Naming requirements:** Names are in a namespace. `Google.com` is unique. Conditions that need to be required for resource location requests:
  - Location Transparency: It doesn't matter what is the IP address of the computer that is serving the content. This is Local Transparency where the name of the website is decoupled from the location. Location transparency makes it fault-tolerant if any machine goes down, the requests can be resolved by the other servers but the requester won't see a difference.
  - Global Uniqueness: Also the resource location requests, the naming has been globally unique. The most popular DNS service is called bind.
  - Ability to access all names: It also needs the ability to access all names from all locations and it has to be fast.
  - Protocol Identification: If this is the ftp protocol we need to speak ftp to this thing and access/serve file and directory listings. Services run on different ports, for instance, http runs on port 80, ftp on port 22. In the case of `https://2.2.2.2./page.html`. The request will go to the machine's port 80 associated with the http port and get an http response.

1.5 URI vs URL's

URL is a special case (subset) of URI. For Example:

```
protocol://user:password@host:port/dir?query#fragment
https://e xample.com/path/resource.txt?GET=query&key=value#fragment
```

1.6 Ports

Ports 0 - 1023 are reserved well-known ports. Must be admin/root to listen to these ports. Ports 1024 - 65535 are general use.

We need 2 ports, 1 port accepts the connection and then transfers other to another port that establishes a socket.

1.7 DNS

Domain Name System. This is a hierarchical lookup. For instance `robin.cs.umanitoba.ca`, CA is the highest hierarchy meaning located in CA (Canada).

But where are the .ca root servers? There is a list of root servers that can be searched using Google.

Also, our router also caches recently used DNS entries, so this reduces the load on the root server and returns a faster response. In case of failure cache hit, the request will go to the ISP who also has a cache. And if that fails again, then the request hits the root server.

The DNS records are cached. But that has a downside, one

End of Week 1

2 Week 2 : Client Server

2.1 Server Client

Distributed computing involves multiple independent computational entities (nodes) that communicate over a network to achieve a common goal. A fundamental model is the client/server architecture, where:

1. Client: Initiates requests for services.
2. Server: Processes requests and provides responses.

This is a fundamental concept used every day. For example, ordering food means that you are a client talking to a server. Other examples are web or mail servers.

2.2 Design Issues

These issues are primarily addressed later in the course, but an introduction is provided here.

2.2.1 Division of Labour

Concept: How to divide the overall task into smaller, manageable subtasks that can be executed concurrently by different nodes.

Theoretical Foundation: Amdahl's Law provides a theoretical limit on the speedup achievable through parallelization. It highlights the importance of minimizing the sequential portion of a task.

Practical Implication: Careful task decomposition is crucial for scalability. Overly fine-grained tasks can lead to excessive communication overhead, while overly coarse-grained tasks may not fully utilize available resources.

Industry Application: MapReduce (used by Google) is a prime example. A large dataset is split into smaller chunks (`"map"` phase), processed independently, and then the results are combined (`"reduce"` phase).

2.2.2 Reliability of Transmission

Concept: Ensuring data transmitted between nodes arrives correctly and in the intended order, despite potential network failures.

Theoretical Foundation: The Two Generals' Problem demonstrates the fundamental impossibility of achieving perfect agreement in the presence of unreliable communication channels.

Practical Implication: Reliable communication protocols (e.g., TCP) use techniques like acknowledgments, retransmissions, and checksums to detect and correct errors. However, these add overhead. In the example presented in the class, if a network is working all the time, if the server is online and functioning properly, and if the messages are not getting lost, the transmission is reliable.

Industry Application: *TCP/IP*, the foundation of the internet, provides reliable, ordered byte streams. Higher-level protocols (e.g., HTTPS) build on this foundation.

2.3 Server Design

Server design often involves transparency. This means that multiple machines can respond to a user's request, but they share a common endpoint. The user interacts with this single point, unaware of the underlying complexity. This is achieved using a load balancer, which acts as a traffic director. The load balancer is a device or software that distributes network or application traffic across multiple servers. This distribution prevents any single server from becoming overwhelmed, improving the responsiveness and availability of applications, and ensuring reliability and high performance.

2.4 Managing State

State Management is crucial between the client and the server. For example, in an email client, emails are stored on the server and should be accessible from anywhere at any time. If a user goes offline, the server and the client need to determine which emails the user has and hasn't accessed. Furthermore, in server-server communication, data across all servers must be synchronized and up-to-date.

1. **Stateful Server:** The server maintains the state. Clients are typically thin. This simplifies client logic but can create a single point of failure and a bottleneck.
2. **Stateless Server:** The server does not store any client-specific state between requests. Each request must include all necessary information. This improves scalability and fault tolerance but increases message size and complexity.

3. **Shared State:** A hybrid approach where some state is managed by the server, and some may be managed by the client or a separate distributed data store.

2.5 Concurrency

Concurrency refers to events happening simultaneously, often in an unpredictable order. When order is important, operating system concepts like locks are used to manage concurrent access to shared resources. Locks ensure that only one process can access a resource at a time, preventing data corruption and inconsistencies.

2.6 Work Distribution

Work distribution involves balancing the load across servers. This includes considering the number of clients each server handles, their memory load, processor load, and overall client interaction.

2.7 State

A state represents a piece of information. For example, if two web servers handle client requests and a user is logged in, this status (e.g., `UserLoggedIn=true`) must be maintained on both machines. A state machine transitions through different states. For instance, a user login might involve inputting a password, then proceeding to 2-factor authentication (2FA), and finally achieving a logged-in state. A stateful machine stores state information in the server-client connection or even in server-server interactions. For instance, a learning management system (LMS) server might store user IDs, interaction history, duration of interactions, IP addresses, and browser information. On the client side, a cookie can store state information, such as language preferences. A stateless machine, on the other hand, does not store any state. Examples include simple HTTP servers and DNS servers. TCP maintains a connection state, storing information about connected hosts. UDP, used for live streaming and video calls, does not maintain connection state; the order of packets is not guaranteed.

2.7.1 Case Study: Network File System (NFS)

If two different machines can write to files on the same server, file locking is necessary. The server maintains state information about which machine has a lock on a file. In the example below, the server holds the state indicating which machine first requested to edit the file. Only changes from that machine are executed; other requests fail. The NFS server is a stateful machine, holding the locks on files. However, if the NFS server fails before completing a write operation, the lock might persist, leading to inconsistencies. This illustrates a trade-off between stateful and stateless designs. In practice, NFS servers employ complex solutions to manage state and handle potential failures.

2.7.2 Case Study: UM Learn

UM Learn (a learning management system) exhibits state transitions. A user might transition from a "not logged in" state to a "2FA" state and finally to a "logged in" state. A failed 2FA attempt might return the user to the "not logged in" state.

2.7.3 Case Study: Tesla

A Tesla car is an example of a stateful system. It maintains state information about power/battery levels and various other car-related parameters. The Full Self-Driving (FSD) system on the car can be considered a thick client.

2.8 Work Distribution

A single computer often cannot handle the workload. Work can be "farmed out" to multiple machines. If one server goes down, other servers must be able to handle requests. This necessitates transparency, where the client is unaware of the multiple servers.

2.9 Issues with Work Distribution

In a stateful system, if a machine goes down, what happens to its state? In a cluster, if a file is modified on one server and that server fails, the change must be reflected on other servers. The other servers should have the latest committed data. Therefore, state sharing is essential.

2.10 Client-Server Architecture

The most fundamental division of labour is the client-server model. This model, introduced by Xerox PARC in the 1970s, distinguishes between:

- **Clients**: Initiate requests for services. They are typically the user-facing part of the application.
- **Servers**: Respond to requests from clients, providing resources or performing computations.

This architecture promotes modularity, allowing for independent scaling and maintenance of clients and servers. The client-server model is a foundational concept underlying many distributed system designs, including web applications, databases, and file systems. It is a specialization of the more general concept of interacting processes in a distributed system.

2.11 Thick and Thin Clients

Two main approaches exist:

2.11.1 Thick Client

Processing occurs primarily on the client. The server primarily handles storage and communication.

1. **Pros:**
   (a) *Reduced Server Load*: The server handles less processing, potentially reducing infrastructure costs.
   (b) *Improved Responsiveness*: Local processing can lead to faster response times for user interactions.
   (c) *Offline Capabilities*: Thick clients can often function, at least partially, without a network connection.
2. **Cons:**
   (a) *More Expensive Clients*: Requires clients with sufficient processing power and storage.
   (b) *Client Management Complexity*: Updates and maintenance must be performed on each client, potentially increasing deployment overhead.
   (c) *Distributed State Management Challenges*: Managing state consistency across multiple clients can be complex, particularly in offline scenarios.
   (d) *Security concerns*: The larger attack vector.

Industry Applications: Desktop applications (e.g., Microsoft Office, Adobe Photoshop), mobile applications, video games (especially multiplayer games, where the client handles rendering and game logic). PowerBuilder applications interacting with an SQL database.

2.11.2 Thin Client

The client acts primarily as a view or display terminal, connecting to a powerful mainframe (e.g., via SSH).

1. **Pros:**
   (a) *Cheap Clients*: Minimal hardware requirements translate to lower costs.
   (b) *Simplified Client Management*: Updates and maintenance are centralized on the server, simplifying deployment.
   (c) *Centralized Security*: Security enforcement is primarily on the server, reducing client-side vulnerabilities.
2. **Cons**:
   (a) *Server Dependency*: The client is unusable without a network connection to the server.
   (b) *Server Scalability Bottleneck*: The server must handle the computational load for all connected clients, potentially requiring significant resources (what the notes referred to as a "BEEFY" server, a non-technical, but descriptive term).
   (c) *Network Latency Impact*: Every interaction requires a round-trip to the server, making the application sensitive to network latency.

Industry Applications: Point-of-sale (POS) systems in some retail environments (e.g., Home Depot, Canadian Tire, grocery stores), where terminals primarily display data from a central inventory system. SSH sessions to remote servers.
Note: A modern SSH connection holds state. The state is the TCP connection.
The optimal solution often lies on a spectrum between thick and thin clients, depending on the specific application.

2.12 Web Browser

A web browser can be considered a "chubby" client, neither strictly thick nor thin. When browsing a website with pure HTML, it acts like a thin client. However, when rendering video, playing JavaScript games locally, or using a local database, it resembles a thick client. Its behavior varies depending on the task.

- *Pure HTML*: Viewing a static HTML page is closer to the "thin client" end of the spectrum. The browser renders the HTML, but the server provides the content.
- *Video Rendering*: Streaming a video from YouTube involves the browser decoding the video stream and handling playback, requiring significant local processing.
- *JavaScript Games*: Playing a complex game written in JavaScript within the browser shifts the architecture further towards the "thick client" end. The JavaScript code executes locally, handling game logic and interactions.
- *Local Storage/Database*: Modern web browsers support local storage (e.g., `‘localStorage’`, `‘IndexedDB’`), allowing for offline data storage and manipulation. This further blurs the line between client and server, as the client now maintains its own state.

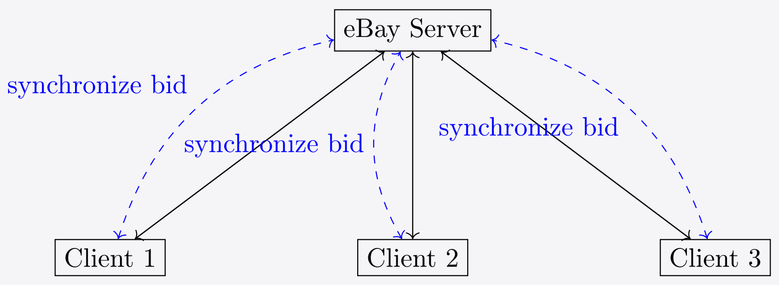
2.13 When to Use Which?

There is no universally right answer for whether to use a thick or thin client. The optimal choice depends on the specific requirements of the application. Some key factors to consider:

1. **Resource Constraints**: If specialized hardware or software is required (e.g., a specific database, a high-performance GPU), and that resource is only available on the server, then a thinner client architecture might be necessary. The work ***must*** be done where the resource resides.
2. **Data Transfer Volume**: If the application requires transferring large amounts of data between the client and server, minimizing communication becomes critical. It may be more efficient to perform processing on the server and transfer only the results to the client, rather than transferring the raw data for the client to process. For example, performing a database join on the server and returning only the matching rows is much more efficient than transferring entire tables to the client.
3. **Synchronization Requirements**: If the application involves concurrent operations that must be synchronized (e.g., bidding in an online auction), a centralized server is typically required to manage the synchronization and ensure consistency. The clients cannot directly synchronize with each other in a client-server architecture, as they only communicate with the server.
4. **Computational Workload Balance**: In cloud computing environments, where computation time is directly billed, balancing the workload between clients and servers can be a significant cost optimization strategy. Offloading work to clients reduces server-side computation, potentially leading to lower costs. However, this must be balanced against the increased complexity of managing thicker clients.

Example 2.1: Ebay Bidding Example

In the case of Ebay, the clients do not communicate the bids with one other. The server is the main source of truth for the active bid price on the item. This diagram represents a client-server interaction where client machines Client 1, Client 2, and Client 3 all talk to the server eBay servers, but do not talk amongst themselves.



2.14 Message Passing

2.14.1 What is a Message?

Definition: A message is any form of information or data that is transmitted between hosts in a distributed system. At a high level, this can be considered as any sequence of bits (ones and zeros).

2.14.2 Uses

Message passing is crucial for designing and implementing distributed systems. Common uses include:

1. **Synchronization between Hosts:** Ensuring that multiple hosts have a consistent view of data or state, at least momentarily.
2. **Data Dissemination:** Broadcasting data to multiple hosts, making sure all hosts receive the relevant information. This is very similar to synchronization of data.
3. **Notifications of Events:** Informing hosts about specific events, such as a printer coming online or a mobile device receiving a notification.
4. **Information Retrieval:** Requesting data from a server, which is the foundation of most HTTP traffic.
5. **Sneakernet**: Even physically delivering data with, for example, a USB key, is considered message passing.

2.14.3 How to Send Messages?

1. **Push:**
   (a) *Description:* Announcements where data is sent to peers without expecting a direct response. It's a one-way message, like a notification that might say, `"I am here."`
   (b) *Example:* A laptop sending a one-way message to a server, which might then do things but does not need to reply.
   (c) *Use Case:* Typically used in peer-to-peer problems where a response might not be necessary. An example is using `DHCP` and `multicast DNS`.
   (d) *Technical Considerations:* Often implemented with protocols like *UDP*, where delivery is not guaranteed.
2. **Pull:**
   (a) *Description:* A request/response protocol where a client requests a resource and receives that resource.
   (b) *Example:* HTTP traffic, where a client requests a website and receives the text of the website.
   (c) *Use Case:* Common in client-server models, such as fetching a webpage. The key is that communication happens on request.
3. **Push/Pull:**
   (a) *Description:* A combination where hosts can both announce (push) and respond to requests (pull).
   (b) *Example:* Peer-to-peer networks where hosts act as both clients and servers.
   (c) *Challenges:* Implementing this in a distributed system can be complex, requiring careful consideration of synchronization and consistency.

2.14.4 Blocking and Non-Blocking

1. **Blocking:**
   - The sender waits for a response before proceeding. The operation blocks other processes until it completes. For Example, Logging into a server; the client sends a password and must wait for validation before proceeding.
   - *Use Case:* Necessary when the response is critical for subsequent actions.
2. **Non-blocking:**
   - The sender does not wait for a response and continues other operations. For Example, Requesting images from a webpage after the initial HTML is loaded.
   - *Use Case:* Useful when immediate response is not critical, allowing for concurrent operations and improved efficiency.

2.14.5 Rendezvous

Rendezvous involves synchronizing state between a client and a server. The client waits for a response (e.g., during login).

2.14.6 Call Back

A callback is an asynchronous request. The client sends a request and continues execution while the server processes the request and eventually sends a response. This is a non-blocking approach.

2.14.7 Relay Buffer

In a relay system, a request might travel from host 1 to host 2 to host 3, and the response follows the same path. This is common in location transparency with web proxies. Host 1 sends a message and blocks, waiting for a response from host 2. Host 2, in turn, sends a request to host 3 and blocks.

Flooding is a very simple algorithm. If you receive a message you forward the message to every node connected to you, except for the one that you just received the message from. It works well if the number of nodes is small, but the number of messages quickly becomes large.

Gossiping is just like flooding, but you only send the message to a few neighbours.

2.15 Dijkstra's Algorithm

Dijkstra's algorithm is a well-known algorithm for finding the shortest paths between nodes in a graph.

- **Algorithm Steps:**
  1. Find the current minimum cost node.
  2. "Push" the cost to get to this point plus the cost to all its peers.
- **Distributed Implementation:** In a distributed system, implementing Dijkstra's algorithm is complex because there is no shared memory.

We have a min-heap or a min-queue, but it is not trivial to understand where it is or how to synchronize the data.

End of Week 2

3 Week 3 : Webserver and Sockets

3.1 Practical Communication

3.1.1 Unicast

A communication method where a message is sent from a single sender to a single receiver. This is the most common form of communication on the internet for one-to-one interactions. Think of a direct phone call.

3.1.2 Multicast

A communication method where a message is sent from a single sender to multiple, specific receivers simultaneously. This differs from broadcast, where the message goes to ***all*** nodes on the network. Multicast is more efficient for group communication. An example is a video conference call where only subscribed participants receive the stream. The Internet Group Management Protocol (IGMP) is commonly used to manage multicast group memberships. We communicate over connections. Connections, particularly in the context of TCP, require state machines to manage the establishment, maintenance, and termination of the communication link.

3.2 Theoretical Background: State Machines

A finite-state machine (FSM) is a mathematical model of computation used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of *states*. The machine is in only one state at a time; the state it is in at any given time is called the *current state*. It can change from one state to another when initiated by a triggering event or condition; this is called a *transition*. A particular FSM is defined by a list of its states, its initial state, and the triggering condition for each transition.

3.2.1 TCP State Machine

TCP uses an 11-state machine to manage connections. Key states include: `‘LISTEN’`, `‘SYN-SENT’`, `’SYN-RECEIVED’`, `‘ESTABLISHED’`,`‘FIN-WAIT-1’`, `‘FIN-WAIT-2’`, `‘CLOSE-WAIT’`, `‘CLOSING’`, `‘LAST-ACK’`, `‘TIME-WAIT’`, and `‘CLOSED’`. The three-way handshake (`‘SYN’`, `‘SYN-ACK’`, `‘ACK’`) is used to establish a connection (transitioning from `‘CLOSED’` to `‘ESTABLISHED’`).

3.2.2 UDP (Connectionless)

UDP does not maintain a connection in the same way *TCP* does. It’s a `"fire-and-forget"` protocol. However, applications can implement connection-like behavior on top of *UDP*, adding reliability and ordering if needed (this is often called `"faking"` a connection, as the underlying transport layer remains connectionless).

3.3 Blocking vs Non-blocking

Regardless of the underlying connection (or lack thereof), operations on sockets can be either blocking or non-blocking. This refers to the behavior of the system call (e.g., `‘send’`, `‘recv’`) made by the application.

3.3.1 Blocking

A blocking operation will pause the execution of the thread (or process) until the operation completes.

- **Receive Blocking**: The process waits until data is available to be read from the socket. For a web server, this is typical behavior: the server blocks on a `‘recv’` call, waiting for a client request.
- **Send Blocking**: The process waits until the data has been successfully sent (and, in the case of TCP, acknowledged by the receiver).

Theoretical Background: Blocking operations rely on the operating system’s scheduler to suspend the process and wake it up when the I/O operation is complete. This can be efficient if the process has nothing else to do while waiting, but can lead to wasted resources if the process could be doing other useful work.

3.3.2 Non-blocking

A non-blocking operation will return immediately, even if the operation is not yet complete.

- **Receive Non-blocking**: The process checks if data is available. If data is available, it’s read; otherwise, the call returns immediately (often with an error code indicating no data was available). This often requires *polling* – repeatedly checking the socket for data.

- **Send Non-blocking**: The process attempts to send the data. If the send buffer is full, the call may return immediately (again, often with an error code), indicating that the data could not be sent at this time.

Theoretical Background: Non-blocking operations allow a single thread to handle multiple I/O operations concurrently, improving responsiveness and throughput. However, they require careful handling of error conditions and often involve more complex programming logic (e.g., using `select`, `poll`, or `epoll` to monitor multiple sockets).

3.3.3 Industry Application

Web servers often use a combination of blocking and non-blocking techniques. For example, a multi-threaded server might use a blocking 'accept' call to wait for new connections, but then handle each connection in a separate thread (which could use blocking or non-blocking I/O). High-performance servers often use non-blocking I/O and event loops to handle a large number of concurrent connections efficiently.

3.4 Sockets

Sockets provide an abstraction for network communication, allowing applications to send and receive data as if they were reading and writing to a file.

3.4.1 Theoretical Background: Sockets and the OSI Model

The statement mentions layers of the OSI (Open Systems Interconnection) model. It is important to understand that, while conceptual, the OSI model doesn't perfectly map to real-world implementations like TCP/IP. However, it provides a good framework:

- **Layer 4 (Transport):** This layer provides end-to-end communication services. TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are the two main protocols at this layer. TCP provides reliable, ordered, connection-oriented communication, while UDP provides unreliable, unordered, connectionless communication. When you create a socket, you specify the transport protocol (TCP or UDP).
- **Layer 5 (Session):** This layer manages the dialogue (session) between communicating applications. In practice, this layer is often handled within the application itself or combined with the transport layer. Socket APIs often blur the lines between layers 4 and 5. The "session" can be thought of as the ongoing communication between two processes.
- **Layer 6 (Presentation):** This layer is concerned with the format of the data being exchanged. It handles things like data representation, encryption, and compression. Examples include converting data to/from JSON, XML, or using encryption like TLS/SSL.
- **Layer 7 (Application):** This layer is where the application-specific protocols reside (e.g., HTTP, FTP, SMTP). Your application code, running at this layer, uses the socket API to interact with the lower layers.

The phrase ***"Anything"*** can be sent over the sockets refers to the fact that sockets operate on a byte stream, meaning that any kind of information can be converted to a sequence of bytes for transferral.

3.4.2 Layer 4 Connection

This section discusses the two primary transport layer protocols:

1. **Connection-oriented (TCP)**:
   - **Data is ordered**: TCP guarantees that data will be delivered in the order it was sent. This is achieved through sequence numbers and acknowledgments.
   - **Reliable**: TCP provides mechanisms for error detection and correction, including checksums, acknowledgments, and retransmissions. If a packet is lost or corrupted, TCP will automatically retransmit it.
   - **Theoretical Background**: TCP uses a sliding window protocol to manage data flow and ensure reliable delivery. The window size determines how much data can be sent without waiting for an acknowledgment.
   - **Industry Application**: TCP is used for most internet applications that require reliable data transfer, such as web browsing (HTTP/HTTPS), file transfer (FTP), email (SMTP), and secure shell (SSH).
2. **Connection-less (UDP)**:
   - **Best Local Route**: UDP packets are routed independently, and there's no guarantee of order or delivery. Each packet takes the best available route at the time it's sent, which might differ for subsequent packets.

- **Problematic (depending on application)**: The lack of reliability and ordering can be problematic for applications that require these features. However, it can be advantageous for applications where speed is more important than reliability, or where occasional packet loss is acceptable.
- **Theoretical Background**: UDP is a simple protocol with minimal overhead. It's often used for real-time applications or applications that can tolerate some packet loss.
- **Industry Application**: UDP is used for applications like DNS (Domain Name System), streaming media (video conferencing, online gaming), and some network management protocols (SNMP).

3.4.3 Layer 5 Communication

The statement `"Everything is files"` is a common Unix/Linux philosophy. It means that many system resources, including devices, pipes, and sockets, are treated as files and can be accessed using standard file I/O operations. This simplifies programming and allows for a consistent interface. The `"OS PTSD"` refers to the complexities that can arise when dealing with low-level operating system details.

3.4.4 Layer 6 Presentation

This section discusses the different ways data can be represented for transmission over a socket:

1. **Text**: Plain text data, often encoded using ASCII or UTF-8.
2. **HTML**: HyperText Markup Language, used for structuring web pages.
3. **Structured Data (JSON, XML)**
   - **JSON (JavaScript Object Notation)**: A lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. A JSON object is essentially a collection of key-value pairs, where keys are strings and values can be primitive types (string, number, boolean, null) or nested JSON objects or arrays. ***Note***: While often described as a dictionary, JSON *objects* do not inherently guarantee key order, although many implementations preserve insertion order.
   - **XML (Extensible Markup Language)**: A markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. XML uses tags to define elements and attributes to provide additional information about those elements. XML is more verbose than JSON. The note's recommendation against XML is a common sentiment, as JSON has largely replaced XML for many applications due to its simplicity and smaller size.
4. **Binary**: Raw binary data, which can represent any type of information. Binary formats are often more compact and efficient to process than text-based formats, but they are less human-readable and can be more difficult to debug.

3.4.5 Theoretical Background: Data Serialization

Sending data over a network often requires *serialization*, which is the process of converting a data structure or object into a format that can be stored or transmitted and reconstructed later. JSON, XML, and binary formats are all used for serialization.

3.5 Socket How-To's

3.5.1 Socket connection types:

1. `SOCK_DGRAM` (UDP): Specifies a datagram socket, used for UDP communication.
2. `SOCK_STREAM` (TCP): Specifies a stream socket, used for TCP communication.

3.5.2 Steps

Server Steps (TCP):

1. **Create a socket**: `socket()` system call. This creates a new socket and returns a socket descriptor (an integer). The call takes arguments specifying the address family (e.g., `AF_INET` for IPv4, `AF_INET6` for IPv6), the socket type (`SOCK_STREAM` or `SOCK_DGRAM`), and the protocol (usually 0, which lets the system choose the default protocol for the given socket type).
2. **Bind**: `bind()` system call. This associates the socket with a specific address and port number on the local machine. This is like assigning a phone number to a phone.

3. **Listen**: ‘listen()‘ system call (TCP only). This puts the socket into a listening state, waiting for incoming connections. The argument to ‘listen‘ specifies the maximum number of pending connections that can be queued.
4. **Accept**: ‘accept()‘ system call (TCP only). This blocks until a client connects to the server. When a client connects, ‘accept‘ returns a *new* socket descriptor that is used for communication with that specific client. The original listening socket remains open and can accept further connections. This is crucial for handling multiple clients concurrently.
5. **Begin receive and send**: ‘recv()‘ and ‘send()‘ system calls (or variants like ‘recvfrom‘ and ‘sendto‘ for UDP). These are used to receive data from and send data to the connected client.

3.5.3 Client Steps (TCP):

- **Create a socket**: ‘socket()‘ system call, similar to the server.
- **Connect**: ‘connect()‘ system call. This establishes a connection to the server’s listening socket. The arguments specify the server’s address and port number. This initiates the TCP three-way handshake.
- **Begin send and receive**: ‘recv()‘ and ‘send()‘ system calls, similar to the server.

3.5.4 TCP vs. UDP (Order of Operations)

1. TCP: The order of ‘send‘ and ‘receive‘ operations can be interleaved; the connection is full-duplex.
2. UDP: Since UDP is connectionless, there is no ‘connect‘ or ‘accept‘. The server typically binds to a port and then uses ‘recvfrom‘ to receive data, which also provides the sender’s address. The server can then use ‘sendto‘ to send a response to that address.

```python
import socket

# Create a TCP socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect to a server

s.connect(("someserver.com", 200))  # Replace with actual server and port
s.sendall(b'Blast a message') # Send data (encode the string to bytes)
data = s.recv(1024)# Receive data (up to 1024 bytes)
print(data.decode("utf-8")) # Decode the received bytes to a string

s.close() # Close the socket (important!)
```

Codeblock 2: Creating a socket in Python

The socket API is a set of functions provided by the operating system that allows applications to create and use sockets. The API is typically similar across different operating systems, making it relatively portable.

3.5.5 Questions:

1. **Is the session layer just for opening and closing sockets?** No, the session layer is more than just opening and closing. It encompasses managing the dialogue between applications. While socket creation and destruction are part of establishing and terminating a session, the session layer also concerns itself with things like checkpointing, recovery, and potentially authentication and authorization. In many practical implementations, these functionalities are often intertwined with or handled directly by the application layer or are implicitly managed within the transport layer (especially with TCP).
2. **What is the difference between flooding (closely and loosely connected)?** "Flooding" isn’t a standard term directly related to sockets in the context of the OSI model layers we’ve discussed. However, it likely refers to network flooding techniques. In a *closely connected* network (e.g., a local area network), flooding might involve sending packets to all nodes on the network, potentially overwhelming the network. In a *loosely connected* network (e.g., the internet), flooding is typically associated with denial-of-service attacks, where a large number of packets are sent to a target, consuming its resources and preventing legitimate users from accessing it. The concept doesn’t map directly to a specific socket operation, but rather describes a *use* (or misuse) of network communication.
3. **When is binary format better than JSON and XML?** Binary formats are generally preferred over JSON and XML when:

- **Performance is critical:** Binary formats are typically smaller and faster to parse than text-based formats like JSON and XML. This is because they don't require character encoding/decoding and often have a more compact representation.
- **Bandwidth is limited:** Smaller message sizes translate to less data transmitted over the network, which is crucial in environments with limited bandwidth.
- **Storage space is a concern:** Binary formats generally require less storage space.
- **Data types are complex:** Binary formats can directly represent complex data types without the need for custom serialization/deserialization logic, which might be required with JSON or XML.

However, binary formats are less human-readable and can be more difficult to debug. The choice between binary and text-based formats depends on the specific requirements of the application.

3.6 Practical Communication (In Class)

Sockets fundamentally transmit data as a stream of bytes (binary data). Encoding schemes like JSON, XML or custom binary formats are used on top of the socket layer.

Serialization: Converting an object (in memory) into a byte stream (or a string representation like JSON) so it can be sent over a network or stored. The reverse process is called deserialization.

Python JSON Documentation

3.7 Flattening

"Flattening" refers to the process of converting a complex data structure (like a tree, graph, or object with nested objects) into a linear representation suitable for transmission or storage.

3.7.1 Challenges with Encoding

i) **Circular Linked Lists:** A circular linked list (where the last node points back to the first node) cannot be directly serialized using simple techniques because the serialization process would enter an infinite loop.

ii) **File-like Objects:** File-like objects (objects that represent open files, network connections, or other I/O resources) typically cannot be serialized directly because they represent a *handle* to a resource, not the resource's data itself. You can't serialize the *state* of an open file or network connection in a meaningful way.

Solutions

- **Metadata:** Provide additional information (metadata) to describe the structure of the data. For example, you could represent a circular linked list by identifying the starting node and indicating the circular relationship explicitly.
- **Specialized Encoding:** Use encoding schemes specifically designed to handle complex data structures. For example, graph serialization libraries can handle circular references and other complex relationships. For file-like objects, you might serialize the *data* contained in the file, rather than the file handle itself.

3.8 Multicast

Multicast is a network addressing method where data is sent to a *group* of destination computers simultaneously in a single transmission from the source.

3.8.1 Uses of UDP Multicast:

- **DDoS (Distributed Denial of Service) Attacks**: Multicast can be exploited to amplify DDoS attacks, where a small number of attackers can send requests to a multicast group, causing all members of the group to respond to the victim, overwhelming it.
- **Live Feeds**: Video and audio streaming to multiple recipients simultaneously (e.g., live sports events, online lectures).
- **Notifications**: Sending notifications to a group of subscribers (e.g., stock price updates, news alerts).
- **Game State Updates**: Multiplayer online games often use multicast to distribute game state updates to all players efficiently.

3.8.2 Limitations of UDP Multicast:

- **Unknown Receiver Status**: The sender doesn't know which receivers have successfully received the data (no acknowledgments in UDP).
- **Unordered Packets**: UDP doesn't guarantee packet order.
- **Packet Loss**: UDP is unreliable; packets can be lost in transit.

3.8.3 Theoretical Background: IP Multicast

IP multicast uses a special range of IP addresses (Class D addresses: `224.0.0.0` to `239.255.255.255`). Hosts join multicast groups using the Internet Group Management Protocol (IGMP). Routers use multicast routing protocols (e.g., PIM - Protocol Independent Multicast) to forward multicast traffic efficiently.

3.9 Questions on TCP and UDP

Why do most ports and protocols use TCP?
TCP's reliability (guaranteed delivery, ordering, error detection and correction) is crucial for many applications. Most applications prioritize data integrity over speed.

Why does NTP not use TCP?
NTP (Network Time Protocol) is designed to synchronize clocks across a network. Accuracy and low latency are paramount.

- **Time-sensitive**: TCP's overhead (connection establishment, retransmissions) can introduce unpredictable delays, which would negatively impact the accuracy of time synchronization.
- **Operates like live streaming**: NTP doesn't need to retransmit lost packets. A slightly older time update is better than waiting for a retransmitted, but potentially much older, packet. It's similar to live streaming, where occasional packet loss is acceptable, but low latency is essential.
- **Theoretical Background:** NTP uses a hierarchical system of time servers, with highly accurate atomic clocks at the top (Stratum 0). Clients synchronize with servers at lower strata. The protocol uses statistical algorithms to filter out network jitter and estimate the round-trip delay.

End of Week 3

4 Week 4 : Webserver

4.1 Continution of Week 3

Week 4 focuses on webserver with focus on practical portion from section 3

End of Week 4

5 Week 5 : Introduction to Web Computing

Web computing is a subset of distributed computing and it is based on HTTP, which is a simple, text-based protocol. It was originally developed at CERN in 1989 by Tim Berners-Lee.

5.1 Web vs. Internet

The Internet is a broad, interconnected framework that supports many different protocols. The Web, specifically, is the subset of the Internet that uses the HTTP protocol, normally operating on ports 80 (HTTP) and 443 (HTTPS). Other protocols, such as email (SMTP) or file transfers (FTP), operate on the Internet but are not technically part of the Web.

5.2 Web Computing Characteristics

- **Server-Oriented**: Web computing is primarily a client-server model. Clients (usually web browsers) make requests to servers, which then respond with data.
- **Platform Independent**: The HTTP protocol is platform-independent. This means that clients and servers can run on any operating system (Windows, macOS, Linux, etc.) and still communicate effectively.
- **Based on HTTP Protocol**: Any device or application that can understand and respond to HTTP requests can participate in web computing.

5.3 HTTP (HyperText Transfer Protocol)

HTTP is a text-based protocol that allows the client to retrieve specific resources from the server.

5.3.1 How HTTP Works

1. **Client Request**: The client (e.g., a web browser) initiates a TCP connection to a web server, typically on port 80 (for HTTP) or 443 (for HTTPS). It then sends an HTTP request message.
2. **Server Response**: The server processes the request and sends back an HTTP response message.
3. **Connection Closure**: After the response is sent, the server closes the TCP connection (though keep-alive connections are possible). Each time the user comes up as a new client.

Example 5.1: Example

Consider you're using a web browser to visit `home.cs.umanitoba.ca`.

1. Your browser initiates a TCP connection to `home.cs.umanitoba.ca` on port 80.
2. Your browser sends an HTTP request, looking something like:

```
GET /~robg/index.html HTTP/1.1
Host: home.cs.umanitoba.ca
```

Codeblock 3: HTTP Request

3. The server processes the request and sends back an HTTP response.
4. The server closes the connection.

5.3.2 HTTP is Stateless

Each HTTP request is independent of previous or subsequent requests. The server does not maintain any session state between requests. That is made by creating a new TCP connection per request.

5.3.3 HTTP Request Structure

An HTTP request has three main parts:

1. **Message Type (Start Line)**:
   (a) **Request Type**: Specifies the action to be performed, such as GET, POST, PUT, DELETE, etc.
   (b) **Absolute Path**: The path of the resource being requested (e.g., '/index.html').

   (c) **HTTP Version**: The version of the HTTP protocol being used (e.g., 'HTTP/1.1').

2. **Header Lines**:
   (a) These are key-value pairs providing additional information about the request, such as the user agent (browser), accepted content types, and more.
   (b) The header section is terminated by a blank line (two newlines in a row).

3. **Body**:
   (a) This is the optional data being sent with the request, typically used with POST or PUT requests.
   (b) Only the message type is required.

Example 5.2: Example

```
GET /~/robg/index.html HTTP/1.1
Host: home.cs.umanitoba.ca
```

The code above represents a simple HTTP GET request.

- `GET` is the request type.
- `/~/robg/index.html` is the absolute path.
- `HTTP/1.1` is the HTTP version.

And `Host: home.cs.umanitoba.ca` is a header.

5.3.4 Useful HTTP Headers

- **Content-Length**: Required if you are sending a body. It indicates the size of the response body in bytes.
- **Content-Type**: Specifies the media type of the resource (e.g., 'text/html', 'application/json', 'image/jpeg'). This uses MIME types.
- **User-Agent**: Identifies the client (browser) making the request. For analytics.
- **Referer [sic]**: Indicates the URL of the page that linked to the requested resource. (The misspelling is historical.)

MIME Types: MIME (Multipurpose Internet Mail Extensions) types specify the nature and format of a document or file. They are typically two-part identifiers: a type and a subtype, separated by a slash (e.g., `text/html`, `image/gif`, `application/json`).

5.3.5 HTTP Response Structure

HTTP responses have a structure similar to requests:

1. **Message Type (Status Line)**: Indicates the HTTP version and a status code (e.g., `HTTP/1.1 200 OK`).
2. **Headers**: Key-value pairs providing information about the response (e.g., content type, content length).
3. **Body**: The actual content being returned (e.g., HTML, JSON, an image).

5.3.6 HTTP Status Codes

Status codes are three-digit numbers that indicate the outcome of the request. They are grouped into five classes:

1. **1xx (Informational)**: Indicates that the request was received and the process is continuing. (Rarely seen in practice.)
2. **2xx (Successful)**: The request was successfully received, understood, and accepted. The most common is `200 OK`.
3. **3xx (Redirection)**: Further action needs to be taken to complete the request. `301 Moved Permanently` is common.
4. **4xx (Client Error)**: The request contains bad syntax or cannot be fulfilled. `403 Forbidden`. `404 Not Found` and `401 Unauthorized` `400 Bad Request` are common. `405` (Mostly seen from the API servers.) if you hosting something that can only 1 kind of response.
5. **5xx (Server Error)**: The server failed to fulfill an apparently valid request. `500 Internal Server Error` is the most common, when the databse is locked, or not running, basically anything on the server side fault.

5.4 Using Telnet to Request HTTP Resources

Telnet is a simple, text-based network protocol that can be used to make raw TCP connections. Since HTTP runs on top of TCP and is text-based, we can use Telnet to demonstrate HTTP requests.

Example 5.3: Telnet Example for UManitoba website

```
telnet home.cs.umanitoba.ca 80
GET /~robg/index.html HTTP/1.1
Host: home.cs.umanitoba.ca
```

Codeblock 4: Telnet Message

The command above is a simple HTTP request by using telnet:

i) `telnet home.cs.umanitoba.ca 80`: This opens a TCP connection to `home.cs.umanitoba.ca` on port 80.

ii) `GET /~/robg/index.html HTTP/1.1`: This is the HTTP request line.

iii) `Host: home.cs.umanitoba.ca`: This is a required header in HTTP/1.1. The two enters, which represent a blank line, this signals the end of the headers.

```
HTTP/1.1 200 OK
Date: Fri, 22 Sep 2023 20:29:51 GMT
Server: Apache/2.4.6
Last-Modified: Wed, 14 Sep 2022 20:37:42 GMT
ETag: "295-5e8a91b959180"
Accept-Ranges: bytes
Content-Length: 661
Permissions-Policy: interest-cohort=()
X-Frame-Options: SAMEORIGIN
X-Content-Type-Options: nosniff
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8

<!doctype html>
<html ...>  (The rest of the HTML content)
```

Codeblock 5: Response from the server

The server responds with the HTTP headers and the content of ‘index.html‘, after which it will typically close the connection.

Insomnia is a tool to communicate and test web servers and APIs. It can send different request, as GET, POST, PUT, PATCH, DELETE and others.

5.5 Building an HTTP Server

The most basic function of a web server is to serve files in response to requests. In Python, you can spin up a simple HTTP server with a single command in the terminal:

```
python -m http.server 8000
```

Codeblock 6: Running the http.server

This command will start a server on port 8000, serving files from the current directory.

5.6 Understanding HTTP GET Requests

The server that was started will respond to HTTP GET requests, which are used for retrieving data and can be used to build a web server. It can respond to GET requests, which are used to retrieve files from the server. The structure of a basic GET request is as follows:

a) **Verb**: This indicates the type of request, typically `GET`.

b) **Path**: The location of the requested resource (file or folder).

c) **HTTP Protocol Version**: Usually `HTTP/1.1`.

Example 5.4: Example for the http request

```
GET /week01/class_01-intro/ HTTP/1.1
```

Codeblock 7: Example

If a requested resource is not found, the server will return a `404 error`.

5.7 Understanding HTTP POST Requests

HTTP POST requests are used to send data to the server. It is used for

i) Login
ii) Reply to a forum post
iii) Should make a new item.

A typical POST request includes: The verb POST, The path of the file or folder, HTTP protocol (HTTP/1.1). After that comes the header which contains Host, Content-Length and Content-Type. You can also post json object.

5.8 Parsing HTTP Requests

Parsing an HTTP request involves breaking it down into its constituent parts. For a GET request, this is relatively simple:

1. Split the first line into three parts: the verb, the path, and the HTTP version.
2. Subsequent lines represent headers, which are key-value pairs.
3. Two consecutive newlines indicate the end of the headers and the beginning of the body (if any).

Parsing POST requests is slightly more complex because they can contain data in the body, which is often URL-encoded. URL encoding replaces spaces with plus signs (+) and special characters with their hexadecimal ASCII codes (e.g., `&` becomes `%26`).

5.9 State Management with Cookies

HTTP is a stateless protocol, meaning each request is independent of previous ones. To maintain state (e.g., remember a user is logged in), cookies are used.

i) **Server Sets Cookies**: The server sends a 'Set-Cookie' header in its HTTP response, instructing the browser to store a cookie.
ii) **Browser Sends Cookies**: The browser then includes the cookie in subsequent requests to the same domain.
iii) **Key-Value Pairs**: Cookies are essentially key-value pairs (e.g., `username=RobertGuderian`).
iv) **Domain-Specific**: Cookies are associated with a specific domain. A browser will send cookies for `www-test.cs.umanitoba` and all its parent domains (`cs.umanitoba.ca`, `umanitoba.ca`).

Example 5.5: Example of a server setting multiple cookies in its response

```
HTTP/1.1 200 OK
Set-Cookie: key=value
Set-Cookie: username=what_they_logged_on_with
```

Example of a browser sending a cookie to the server:

```
GET /something.html HTTP/1.1
Cookie: key=value; username=what_they_logged_on_with
```

The professor showed an example of how the cookies change with different requests to different sites.

Security Implications: Because cookies are stored in the client's browser, they can be modified. It is very important to rely on cookies for client identification, with server-side verification using that session ID. The lecturer showed how easy is to change the cookies using the browser developer tools, so It's very important to keep the state in the server. By understanding HTTP requests (GET and POST), headers, and cookies, you can build rich web applications. Cookies allow for state management by identifying clients across multiple requests. However, for secure applications, the actual state (e.g., logged-in status, user data) should be stored on the server, with the cookie serving as an identifier (session ID).

5.10 Recommended Readings

i) HTTP Methods
ii) Morilla documentation on HTTP Methods
iii) HTTP semantics
iv) Documentation on HTTP messages

End of Week 5

6 Week 6 : Introduction to Client-Side Web Computing

Much of modern computing is done in-browser, a popular way to distribute applications. This approach has pros and cons, creating "chubby clients" (more processing on the client-side). It approaches, but doesn't fully reach, the capability of a "thick client" (stand-alone application). Still relies on a web server to fetch files.

In this implementation web server has endpoints for API. And each endpoint is running on an individual thread. When the client is reaching out to '/' root endpoint and that will return the website with CSS and JS. If the client reaches out any other implemented endpoint, then a thread gets spun up and that task is performed.

6.1 Key Technologies for Client-Side Web Interaction

6.1.1 Forms (HTML Forms)

A traditional (though slightly outdated) way to interact with servers. Have fields (for input) and actions (what happens when submitted). Uses submit button.

- **Action attribute** specifies the web page to be fetched when form is submitted.
- **Method** can be `put`, `post`, `delete`, or `get`.

Example 6.1: Example of a form submit onAction

Example: A simple form with "Name of poster" and "The message" fields, and a "go" (submit) button.

The instructor demonstrates inspecting the form's HTML using the browser's developer tools. The form's `action` attribute is `newnote.cgi`, which uses a Common Gateway Interface (CGI) script. This is code on the server. The server in this case is Apache. The form's `method` is set to `"POST."` This means data is sent in the body of the HTTP request. The "Network" tab of the developer tools shows the request and response, including the "200 OK" status and the "POST" method. The "Payload" tab shows the data sent: `name=Robert+Guderian&message=test`. This is URL-encoded.

6.2 Single Page Application

This is built using AJAX (Asynchronous JavaScript and XML). That mean we do no request the whole HTML, instead we request the small missing information to fill that data field. AS a result of it, it doesn't require reloading the page and let the work be done in the JavaScript.

6.3 XHR (XMLHttpRequest)

Allows making requests "behind the scenes" without reloading the entire page. XHR is used to update the DOM (Document Object Model), which represents the structure of the HTML page. The "XML" part of the name is becoming less relevant; JSON is more common now. Has callbacks (asynchronous functions) that run when certain events occur (like the request completing). Look at the `example.html` code.

```
<html>
<head>
<script>
  function doneBeenClicked() {
    function loadedEventCallback () {
          alert(oReq.responseText);
        }

        function loadedEventCallback2 () {
          var theData = JSON.parse(oReq.responseText);

          var theDiv = document.getElementById("fillHere");
          //theDiv.innerText = theData.someKey;
          theDiv.innerHTML= "<h1>" + theData.someKey +"</h1>";

        }

        // basis from
        //
           https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest
        // probably will not work from localhost!
        var oReq = new XMLHttpRequest();
        oReq.addEventListener("load", loadedEventCallback2);
        // a static page we have access to
        oReq.open("GET", "./simple.json");
        oReq.send();
  }
</script>
</head>
<body>
    <button type="button" onClick='doneBeenClicked()'>Do it!</button>
    <div id='fillHere'>Stuff will go here.</div>
</body>
</html>
```

Codeblock 8: HTML

- `oReq = new XMLHttpRequest();` creates the request object.
- `oReq.addEventListener("load", loadedEventCallback2);` sets up a callback function to run when the request loads.
- `oReq.open("GET", "./simple.json");` specifies the request method (GET) and the URL.
- `oReq.send();` sends the request.

`LoadedEventCallback2` function:

- Parses the JSON response: `var theData = JSON.parse(oReq.responseText);`
- Gets a DOM element by its ID: `var theDiv = document.getElementById("fillHere");`
- Changes the inner HTML of the element: `theDiv.innerHTML = theData.someKey;`

He also shows how the "inner text" command can be changed to "inner HTML" to create an h1 tag. Clicking the "Do it!" button triggers the XHR request, fetches the JSON, and updates the page content without a full reload.

6.4 DOM (Document Object Model)

The Document Object Model (DOM) serves as a crucial Application Programming Interface (API) that allows programs to interact with the structure of HTML and XML documents. Fundamentally, the DOM represents a document as a tree-like hierarchy, often referred to as the DOM tree, where every element, attribute, and piece of text within the document is treated as a distinct node in this tree. This tree representation enables dynamic manipulation of web page content, most notably through JavaScript. By providing programmatic access to the DOM tree, JavaScript can dynamically create new nodes, modify existing ones, move elements within the structure, and generally alter the content and presentation of a web page in response to user actions or data updates, which is essential for building interactive and dynamic web applications within distributed systems.

6.5 Cookies

Cookies provide a mechanism for maintaining client-side state in web applications. It is crucial capability where interactions are inherently stateless. Primarily, cookies are employed to remember user-specific information across multiple HTTP requests, enabling functionalities like persistent logins, the storage of user preferences, and, controversially, user tracking for targeted advertising. These small pieces of data are set by the server in the `Set-Cookie` header of HTTP responses. Once set, the browser automatically and persistently includes these cookies in the headers of all subsequent requests directed to the same domain or its subdomains. Cookies sent by the server can be observed in response headers, such as within a "messages" field which, in the example provided, contained a numerical identifier. While the presenter will later elaborate on the role of cookies in website performance optimization, it's important to understand that cookies are a cornerstone of client-side state management, enabling personalized and persistent user experiences in distributed web environments despite the underlying stateless nature of the HTTP protocol.

The web browser send a request cookie that contains the unique ID for the client. The web server accepts this cookie and validated if the userIsLogged in the system. This is done by having a state stored on a database, the information is extracted in $\mathcal{O}(1)$ time. If the stored value for is `TRUE`, then show the contents of the page. If the stored value is `FALSE`, then prompt a login page.

6.5.1 Cookies and State

We can set the cookies to expire, in case where we want to forget the user. There are 2 two ways to do this.

- Set the expiry date to the past date
- Set the expiry at the current time, the cookie will expire in a second.

```
Set-Cookie:<cookie-name>=<cookie-value>; Expires=<date>
Set-Cookie:username=Rob; Expires=Wed, 21 Oct 2025 07:28:00 GMT
```

Codeblock 9: Set cookie to expire

6.6 Server-Side Web Computing

The lecture shifts focus to server-side web computing. It emphasizes the importance of organizing the server in a meaningful and usable way for clients.

6.7 REST (Representational State Transfer)

REST (Representational State Transfer) is introduced as a key concept for organizing web servers in modern web development. It focuses on the semantic meaning of HTTP verbs such as GET, POST, PUT, DELETE, and others. REST provides a more intuitive path structure compared to traditional methods.

- **GET** : This will get an response from the server, depending upon the parameters passed, Example: GET /user/11
- **POST** : Post request will make a new post. For instance, making a post using the Twitter API.
- **PUT** : This would update the information. For Example: update the name for the username
- **DELETE** : This will delete the item requested by the user.

Example 6.2: CGI → REST API

Example: Instead of using a parameterized URL like `myScript.cgi?user=11&getItem=firstname`, a RESTful path would be structured as `/user/11/firstname`. This RESTful path clearly indicates the resource being accessed (user), the identifier of the resource (11), and the specific attribute being requested (firstname).

6.7.1 API Design for AURORA

Imagine we have an oppurtunity to rewrite Aurora with a RESTful API. Design the route for AURORA.

- Login : `POST /api/v1/auth/` and the body of the HTTP request
  a. JSON with username and password
  b. cookie
- List Courses `GET api/v1/courses` and pass the parameter for the seach in the JSON Body.
- View a course `GET api/v1/courses/crnid`
- Register for a course `POST api/v1/courses/register/crnid` and send the cookie

6.7.2 Twitter API Example

The Twitter developer platform is provided as a practical example of RESTful API design. Here are some Twitter API endpoints that illustrate REST principles:

Tweets:

- `DELETE /2/users/:id/bookmarks/:tweet_id`
- `GET /2/users/:id/bookmarks`
- `POST /2/users/:id/bookmarks`

Filtered stream:

- `GET /2/tweets/search/stream`
- `GET /2/tweets/search/stream/rules`
- `POST /2/tweets/search/stream/rules`

Hide replies:

- `PUT /2/tweets/:id/hidden`

Likes:

- `DELETE /2/users/:id/likes/:tweet_id`
- `GET /2/users/:id/likes/:tweet_id`
- `GET /2/users/:id/liking_users`
- `GET /2/users/:id/liked_tweets`
- `POST /2/users/:id/likes`

6.7.3 Key Takeaways about REST

There are several key takeaways regarding RESTful architecture. Firstly, REST utilizes intuitive paths that clearly represent resources and their identifiers, making APIs easier to understand and use. Secondly, a single path, such as `/user/11`, can be used with different HTTP verbs to perform various actions on that resource. For instance, `GET` would be used to fetch user information, `PUT` to update user details, and `DELETE` to remove the user. Lastly, Twitter's API serves as a real-world example of a system that effectively employs RESTful principles in its design.

6.8 CGI Scripts

During the lecture, it was noted that the presenter is using a UNIX system and is logged in as user `rogb`. Accessing their environment allows for demonstration of system settings. CGI is an acronym for Common Gateway Interface. The server-side code examples used in the lecture are implemented in Python. For demonstrating and testing API calls, the presenter is using the software Insomnia.

```python
#!/usr/bin/python3
import sys
import os

print('''Content-type: text/text

{:}'''.format(dict(os.environ)))
```

Codeblock 10: CGI script

The CGI script notifies the server that something needs to run. The SGI script is going to run python. The CGi scripts need to be add the content type since it can return anything, the user/programmer needs to specify it. The browser handles the rest of the it including the date, content-length.

We can do anything with CGI ? basically anything as long as it complies with HTTP protocol (not necessarly strictly). But it doesn't work well with REST API.

6.9 Recommended Readings

- Documentation on Post
- Documentation on Cookie
- XMLHttp
- Documentation on using XMLHttpRequest

End of Week 6

7 Week 7 : Performance, Attacks, and Consistency

This week covers scaling for high-performance web applications, focusing on issues that arise when scaling distributed systems.

7.1 Scaling for High Traffic

A major challenge in web applications is handling large volumes of traffic. When a website experiences a sudden, massive increase in traffic – perhaps because content has gone viral (terms like `"Reddit hug-of-death"`, `"Slashdot effect"`, and `"farked"` all describe this phenomenon) – it can resemble a Denial of Service (DoS) attack. The objective is to make the system faster. Scaling is the solution to maintain responsiveness under such loads.

7.2 Strategies for Scaling

There are a few general ways to make a system scale to a high number of requests. There are several main strategies for scaling:

- **Adding more servers:** This increases capacity, but it comes with a cost. Remember, the cloud is essentially renting someone else's servers, so this isn't free.
- **Writing better code:** Code optimization can certainly improve efficiency. However, this is often a time-consuming and expensive process.
- **Doing less work:** This is often the most effective approach, and a key technique is caching.

7.2.1 Caching

Caching is the art of avoiding redundant work. By storing the results of previous computations or data retrievals, we can reuse those results later, eliminating the need to repeat the same operations. The HTTP protocol itself incorporates caching mechanisms, such as the `"304 Not Modified"` response and HEAD requests. We have browser cache at our local storage and some browser built-in automatic caching. webservers also have a cache (webcache) that is very close to the server with atleast a 1 GiG of bandwidth. Memcache is a 3rd party caching solution that that the developer installs and setups.

7.2.2 Client-Side vs. Server-Side Responsibilities

A common strategy is to shift more of the processing burden to the client, creating what's often called a `"chubby client"`. This contrasts with a `"thin client"` model, where most of the processing is performed on the server. We can use techniques like AJAX (Asynchronous JavaScript and XML) and XHR (XMLHttpRequest) to make many small requests, building up the web page incrementally using Microservices. Local storage on the client-side (in the browser) allows us to cache data, further reducing the load on the server. Caching is implemented using key-value pairs. When we are able to store results of a task to retrieve at a later time, we call the results idempotent.

7.3 Content Delivery Networks (CDNs)

CDNs act as sophisticated caches, distributed geographically. They are particularly effective for handling static content like images, videos, and large JavaScript files. By offloading this content to a CDN, we free up our own servers and improve overall speed. URIs for resources hosted on a CDN often include a random element or a date/timestamp. This helps ensure that updated versions of the content are fetched when necessary (a technique called `"cache busting"`).

7.4 Attacks

Distributed systems are vulnerable to various attacks.

- **Denial of Service (DoS) Attack:** The goal of a DoS attack is to prevent a server from servicing legitimate requests, effectively taking it offline.
- **Distributed Denial of Service (DDoS) Attack:** This is a more potent form of DoS attack, where many compromised computers (often called `"zombies"` or a `"botnet"`) simultaneously flood the target server with requests. This distributed nature makes it much harder to block, as simply blocking a single IP address won't suffice.

- **Reflection Attack:** This attack type sends messages in the User Diagram Protocol (UDP) which does not guarantee the delivery or ordering of messages. A reflection attack exploits services like NTP, sending a small request that generates a much larger response. This response is directed at the victim, amplifying the attack.

These attack commonly target the following vectors:

i) Maxing out network bandwidth.
ii) Overloading CPU time.
iii) Consuming excessive memory.

7.5 Error Handling

Distributed systems introduce new failure modes. We need to consider extended failure modes, where some parts of the system are functioning while others are not. The key is to fail gracefully. This means:

i) Having functional pieces, but not a functional whole.
ii) The system should strive to do what it can, even if some components are unavailable.
iii) Provide meaningful error messages to the user (e.g., `Temporarily down for maintenance`).

Recovery often involves automatic reconnection attempts, potentially with timeouts. A `‘while true’` loop is sometimes used to constantly attempt reconnection. Use code with caution.

7.6 Load Balancers

Load balancers are crucial for scaling. They distribute incoming traffic across multiple servers (often organized in `tiers`). This prevents any single server from being overwhelmed. A simple load balancing strategy is `round-robin`(often used for download requests) where requests are sent to servers in a sequential order.

For example, consider a system with a web browser client, a gateway, and a database. If we create replicas (clones) of the database, a load balancer can distribute requests between them. However, if one database goes down, and we're using something like a two-phase commit protocol, problems arise if the databases are no longer synchronized.

Two-Phase Commit (2PC): Two-phase commit is a protocol designed to ensure data consistency across multiple databases. The process happens over two phases, and each uses messages between the database and the client.

1. **Voting Phase (Query):** The coordinator sends a `"query"` message to all workers. The workers respond with a `"vote"` (e.g., "true" for "yes, I can commit").
2. **Commit Phase:** If all workers vote `"true"`, the coordinator sends a `"commit"` message. If any worker votes `"false"` (or doesn't respond), the coordinator sends a `"rollback"` message. The message is a `"ack"` (acknowledgement).

Even with 2PC, problems can occur. For example, if a worker crashes before receiving the `"commit"` message, the system will be in an inconsistent state. This illustrates that while distributed systems offer advantages, they introduce significant complexity in error handling and ensuring data consistency. There exist other distributed consensus protocols, such as Raft.

7.7 Consistency

The need to scale by adding more hosts brings the issue of consistency. This is a the scenario: We have data that is getting fetched all the time, because it has gone viral. We use a single database, that is maxed out and it can't handle any more requests.

There are a web browser and a client that sends requests, and the access is through a gateway(API server, website) that gets the data from the database host, and that is currently maxed out.

7.7.1 Replicas

We are setting a replica(clone, exact copy) for the current database. In case the first database burns, the other one can do all the work. We also gain bandwidth, and CPU time. Using more database hosts by using replicas, in order to create redundancy, which will have some perks:

i) Full redundancy
ii) More bandwidth (assuming they don't share)
iii) More processing power

7.7.2 Problems with replicas

Are networks reliable? Are the systems on?

Example 7.1: What if message is dropped (garbled or intercepted)

1. The two database values for Z, for instance, are not equal
2. When the value of A in the database will be set to one and to two, the network is now restored, the question is what value will hold as a result

If this is financial data, if the message is lost, and the databases are not in sync, this could mean that we lost information about how much money is there.

A system goes offline: Crash, reboot(planned or otherwise), network makes it unavailable. One database is down before it gets commit. The values are not synchronized in the databases.

7.7.3 One solution : 2PC - Two Phase Commit

In this method, there is a voting phase, and a commit phase. It can be done as a

i) Client-server
ii) Coordinator
iii) Workers

Also, it can be done as peer-to-peer client-server network.

There are the following phases for the protocol:

1. Query - lock
2. Vote
3. Commit
4. Acknowledgement

Rob presented in order message passing to show the 2PC protocol. Let assume there are n workers:

1. When worker 1 sends to the other $n-1$ workers the query, this will represent the lock
2. The fleet of workers will send to $n-1$ messages back to the worker 1, with *vote=True*
3. If all of the workers agree, that value will lock all of the databases.
4. Then work 1 sends a commit messages to rest of the $n-1$ workers and the rest of the workers will set the updated value for that variable.
5. The $n-1$ workers will send back `ACK` message, stating the value is updated.

This is all rendevouz messages.

Example 7.2: Explanation for 2 PC Protocol

There is a request to write $Z = 3$. The Worker 1 will send a request, a query for the key Z, a lock request for Z. If Z is not locked, the workers will reply `"vote=true"` and the data for Z will be locked, but not written. After worker 1 gets the `"vote=true"` message from all of the other workers, it will send a commit message to every worker, the database will set $Z = 3$. If it does not get `"vote=true"` from all the databases, it can say `"unlock"`.

With this method, there is still a problem with consistency. If the commit message is not sent, then a message saying `"roll back"` is sent.

7.7.4 3PC-Three Phase Commit

It adds another layer of acknowledgments when the client the first acknowledgement happens, the worker 1 sends another acknowledgement message for `"OK Everyone's commited, right?"`. This creates a lot of communation/messages sending requests back and forth.

7.8 Recommended Readings

- What Is a DNS Amplification Attack?
- Window: localStorage property

End of Week 7

8 Week 8 : Elections

8.1 Coordinators

Many peer-to-peer algorithms require a coordinator. This coordinator manages the system and simplifies algorithm design. If a coordinator goes offline we can choose a new one.

8.2 Bully Algorithm

The Bully algorithm is a method for electing a leader in a distributed system. The basic principle is that the node with the highest unique ID (sometimes referred to as process ID) becomes the leader.
Assume a network of five homogeneous hosts.

i) Each host is assigned a unique ID.
ii) Hosts are aware of other hosts in the network.
iii) If a host does not hear from a coordinator in an expected timeout it may try sending a message to the coordinator.
iv) If the coordinator is offline (fails to respond within a timeout period), a host initiates an election.
v) The host will broadcast an `"I'm the boss"` message, including its ID, to all known hosts.
    i) Each node in the network has a unique ID which serves as its priority (higher ID equals higher priority).
    ii) If a host receives an `"I'm the boss"` message and it has a higher priority, it broadcasts its own `"I'm the boss"` message with its ID.
vi) The highest priority host that gets no objections during a timeout is considered the leader.

Suppose node 2 is the initiator of the election, the messages may appear as shown in Table 2:
The efficiency of the Bully algorithm can vary. In the worst-case scenario, there could be n^2 messages where n is the number of hosts. The minimal amount of messages would be $n-1$, in the case where no other hosts sends a reply back.

Node	Unique ID
1	10
2	5
3	4
4	3
5	2

Table 2: Bully Algorithm Table

8.3 Ring Algorithm

The Ring algorithm is another approach to leader election, particularly useful when nodes don't need to know all other peers in the network, only their successor and predecessor. Assume nodes arranged in a logical ring. Each node (peer) has a priority and maintains a vector of k elements, tracking the priorities of other nodes. When a node initiates an election, it sends an election message, $\{i, P_i\}$, to its successor, where i is the node's index and P_i is its priority. Upon receiving an election message, a node:

i) Records the priorities in its vector.
ii) Prepends its priority to the message and forwards it.

Example 8.1: Ring Algorithm Explanation

Suppose node 2 is the initiator of the election. The table on the right is the index i of the node and the corresponding P_i for the i^{th} node. The messages may appear as follows:

i) Node 2 sends {2:5}
ii) Node 3 adds its information and sends {3:4, 2:5}
iii) Node 4 adds its information and sends {4:3, 3:4, 2:5}
iv) Node 5 adds its information and sends {5:2, 4:3, 3:4, 2:5}
v) Node 1 adds its information and sends {1:10, 5:2, 4:3, 3:4, 2:5}

Node	Unique ID
1	10
2	5
3	4
4	3
5	2

Table 3: Ring Algorithm Table

- When a node receives its own election message back, it can determine the coordinator by identifying the highest priority in the accumulated vector.
- The message circulation ensures that all nodes have consistent information about node priorities.

Implementation Details

The ring algorithm is a more structured approach to leader election, reducing some of the message overhead compared to broadcast-heavy methods like the Bully algorithm. However, its message complexity is still $O(n^2)$ in the worst case, where n is the number of nodes

End of Week 8

9 Week 9 : Consensus & Blockchain

9.1 Byzantine Failures

Byzantine failures relate to problems with distributed systems where nodes (or people) are dispersed and communication is unreliable. It stems from the Byzantine Empire era, specifically illustrating armies coordinating an attack. The attack is successful only if all armies agree on when to attack. The agreement is a binary choice: *YES* or *NO*.

9.2 Consensus

Consensus is the process of getting everyone on the same idea. The main challenge is that the nodes are everywhere, and have trouble communicating because of the communication medium. The idea of the nodes, and consensus, can be applied, and related, to a distributed system, and getting all of the people to agree on one idea. The problem is that these nodes can be anywhere and can have trouble communicating through the communication medium.

9.3 The Byzantine Generals Problem

The Byzantine Generals Problem conceptualizes this, referring to armies communicating an attack in tandem. It's not a new problem, dating back to approximately 600 AD.

The main problem is that the armies must coordinate and attack will only be successful if all of the armies agree to attack at the time that they agree on, and will lose, if only a fraction of the army attacks. There is also the possibility that not everyone is being honest, so there is the potential for somebody to act unreliably.

9.4 Byzantine Fault Tolerance

It is about achieving consensus even with unreliable actors, with examples in:

1. Cryptocurrencies like Bitcoin
2. Aircraft systems with sensors

Example 9.1: Bitcoin Example

Bitcoin uses an open ledger system, and transactions are sent to be permanent, and it has to reach consensus as to whether the transactions occurred or not.
`Person A` wants to send Bitcoin to `Person B`. Workers in the network need to agree on the validity of the transaction. The challenge is dealing with potential bad actors or unreliable messages.

Example 9.2: Aircraft Example

Aircraft have multiple sensors that need to agree on actions:

1. Sensors detect a situation and suggest a response.
2. Sensors can be faulty, leading to incorrect inputs.
3. A minority report situation arises when sensors disagree.

The problem can be summarized as needing:

1. Agreement among all loyal nodes.
2. A small number of traitors cannot force a bad plan.

9.4.1 Oral Message Algorithm (OM)

The Oral Message (OM) algorithm is a recursive method for achieving consensus in the presence of traitorous generals. It is defined as follows:

- **OM(m)**: An oral message algorithm where m represents the maximum number of traitors.
- **Base Case:**
  - **OM(0)**: If there are no traitors ($m = 0$), all generals simply follow the commander's order, ensuring consensus.
- **Recursive Case:**
  - Each general takes on the role of the commander and sends an order to the remaining $n - 1$ generals.
  - Each recipient then recursively applies $OM(m - 1)$.

– After receiving orders from all possible commanders, each general determines the final decision based on the majority value.

Example 9.3: Byzantine Generals with 3 Generals

The commander sends an `attack` message to two generals: General A and General B. General B is a traitor and sends conflicting information.

With $N = 3$, and $3 \geq 3M + 1$ or the equivalent $2/3 \geq M$. Since the number of traitors M must be an integer there is NO solution.

G	1	2
	a	r

Example 9.4: Byzantine Generals with 4 Generals

The commander sends an `"attack"` message to three generals, 1, 2, and 3. General 2 is a traitor. The messages received are present on the table on the right.

With $N = 4$, and $4 \geq 3M+1$ or the equivalent $3/3 \geq M$ that means that a solution can work with one traitor.

G	1	2	3
	a	a	r

With the use of recursion, and the fact that the generals send messages to each other and compare results, the algorithm works.

9.5 Introduction to Blockchain

Blockchain systems are a fascinating application of distributed systems principles. They provide a distributed, trustless, and immutable store of information, operating on a peer-to-peer (P2P) network. This means there's no central authority, participants don't necessarily trust each other, and once data is added, it's exceptionally difficult to alter.

9.5.1 Key Features of Blockchain

1. **Distributed:** The data and operational logic are spread across multiple nodes (computers) in the network, rather than being centralized. This enhances resilience against single points of failure.
2. **Trustless:** Participants don't need to trust each other. Trust is established through cryptographic mechanisms and consensus protocols, ensuring data integrity and validity.
3. **Immutable:** Once a `block` of data is added to the `chain`, it becomes extremely difficult, if not practically impossible, to alter it. This immutability is a cornerstone of blockchain's security and reliability.
4. **Peer-to-Peer (P2P) Network:** A network architecture where each node can act as both a client and a server, communicating directly with other nodes.
5. **Ledger:** The data is stored in an append-only ledger. The most common representation of such a ledger is Bitcoin.
6. **Contracts:** Smart contracts, such as those used by Ethereum. These are essentially programs that execute on the blockchain and can be seen as a piece of information that the blockchain stores.

9.5.2 Consensus and Blockchain Features

There is some hype around blockchain systems. A lot of the hype relates to various blockchains and cryptocurrencies and their perceived value or `get rich quick` schemes associated with them. However, the underlying technology has interesting computer science implications for security, networks, and trust.

9.5.3 Consistency vs Consensus

- **Consistency** refers to ensuring that all nodes have a uniform and up-to-date view of the system's state. If multiple copies of a document exist, everyone should see the same version at all times.
- **Consensus** refers to the process by which nodes in the system agree on a single value or decision, despite potential failures. If a group is voting on a decision, they must all agree on one final choice.

Aspect	Consistency	Consensus
Definition	Ensuring that all nodes see the same data at the same time.	Agreement among nodes on a single value or decision.
Scope	Data synchronization across replicas.	Agreement on an operation or leader.
Goal	Maintain a consistent state across nodes.	Achieve agreement in the presence of failures.
Example	Linearizability, Sequential Consistency.	Paxos, Raft, Byzantine Agreement.
Failure Handling	A node may lag in consistency but eventually catch up (eventual consistency).	Consensus protocols tolerate node failures and network partitions.
Relation	Requires a mechanism (like consensus) to maintain consistency in distributed databases.	Consensus is a fundamental step in ensuring consistency in stateful distributed systems.

Table 4: Differences between Consistency and Consensus

9.5.4 Buzzwords

Blockchain is a combination of many features, but they can be boiled down to the following description: a distributed trustless store of immutable information on a P2P network. Let's unpack those terms:

- **Distributed:** The data is replicated across many nodes in the network, making the system robust against failures. This aligns with the core principle of distributed systems: avoiding single points of failure and improving availability.
- **Trustless:** No single entity controls the system, and participants do not need to trust one another. Cryptographic techniques (like hashing and digital signatures) and consensus mechanisms ensure data integrity and prevent tampering.
- **Immutable:** Once data is added to the blockchain, it cannot be easily altered. This is achieved through the use of cryptographic hashing (as we'll see below) and the chaining of blocks. The data up to the `top few blocks`(-6 depth in case of Bitcoin) is unchangeable, but the more recent blocks can be changed.
- **Peer-to-Peer (P2P):** Nodes in the network communicate directly with each other, without a central server.

9.6 Blockchain Components

Blockchain data is put into `blocks` which is a grouping of data that has a maximum size. The contents of this block must have some cryptographic properties.

9.6.1 Blocks

A block is a fundamental unit of a blockchain. It typically contains:

1. **Previous Block Hash:** A cryptographic hash of the previous block in the chain. This creates the `chain` and is crucial for immutability. A hash function takes an arbitrary input and produces a fixed-size output (the `hash`). Crucially, even a tiny change in the input produces a drastically different hash output (the `avalanche effect`). This means that if you change any data in a previous block, its hash will change, and so will the `previous hash` field of the subsequent block, and so on, making tampering immediately evident.

Example 9.5: Hashing Example

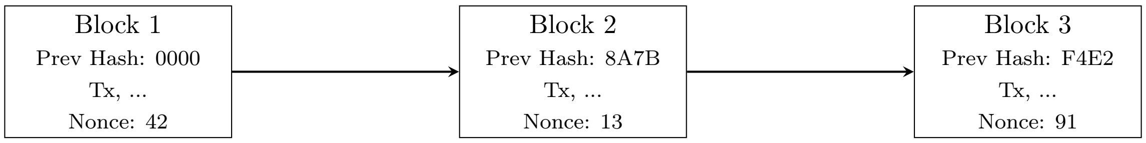
You can test this with command-line tools like `md5` (though MD5 is cryptographically broken and should not be used in a real blockchain) or `shasum` (which uses SHA algorithms). For example in a Unix-like terminal:

```
$ echo test | md5
d8e8fca2dc0f896fd7cb4cb0031ba249
$ echo test1 | md5
3e7705498e8be60520841409ebc69bc1
```

Codeblock 11: Block chain hash Example

Notice the very large changes after changing the input even slightly.

2. **Data:** This is the actual information stored in the block. In cryptocurrencies like Bitcoin, this is primarily transaction data (e.g., `Account A sent X amount to Account B`). In other blockchain systems, it could be any kind of data, including code such as in the case of smart contracts.
3. **Nonce:** A number that is adjusted by `miners` to achieve a specific cryptographic property for the block's hash (more on this later). The nonce is the key element in the ***proof-of-work*** consensus mechanism.



9.6.2 Mining and Proof-of-Work

The process of adding new blocks to the blockchain is called `mining`. The goal of mining is to find a nonce such that when the block's data (including the previous hash and the nonce) is hashed, the resulting hash meets a specific criterion, usually that it starts with a certain number of zeros. This is the `difficulty` of the blockchain.

Why leading zeros? Because it's computationally hard to find a hash that starts with many zeros. Hash functions are designed to be unpredictable; the only way to find a suitable hash is to try many different nonces (brute-force search).

Proof-of-Work (PoW): The process of finding a suitable *nonce* (and thus, a valid block hash) is called `proof-of-work`. It demonstrates that computational effort has been expended. This makes it costly to tamper with the blockchain, as an attacker would need to redo the proof-of-work for any block they change, and *all* subsequent blocks (because of the chaining via previous hashes). The difficulty is adjusted dynamically to maintain a consistent block creation rate (e.g., about every 10 minutes for Bitcoin).

Example 9.6: Imagine

Let's imagine a simplified difficulty requirement: the hash must start with one leading zero. Miners would try different nonces until they find one that, combined with the other block data, produces a hash meeting that criterion. The hash is going to be a hexadecimal value, which means that each digit could be 0-9 or a-f. So for each additional leading zero that is required, this multiplies the amount of work by 16x, because each hexadecimal digit represents four bits, and there is a 1 in 16 chance of getting a zero with each attempt.

9.6.3 Consensus: Longest Chain Rule

In a distributed system, multiple miners might find valid blocks around the same time. This can lead to temporary forks in the blockchain. The consensus mechanism resolves these forks. The most common rule is the *longest chain rule.*

Longest Chain Wins: The chain with the most blocks (representing the most accumulated proof-of-work) is considered the valid chain. Nodes will switch to the longest chain they see.

Why does this work? Because, statistically, the chain with the most work (longest chain) is the most likely to represent the ***true*** state of the network, assuming the majority of the network's hashing power is honest.

9.6.4 Race Conditions and 51% Attack

Race Conditions: Since multiple miners are working simultaneously, it's possible for two or more miners to find valid blocks around the same time. This creates temporary forks in the chain. The network resolves these forks naturally over time because the chain with the most computational work put into it (i.e., the longest valid chain) is the chain that is accepted as the true chain by design.

51% Attack: A theoretical vulnerability of blockchain systems. If a single entity (or a colluding group) controls more than 51% of the network's hashing power, they could, in theory:

1. Mine a `secret` chain longer than the public chain.
2. Release their longer chain, causing the network to switch to it (because of the longest chain rule).
3. This could allow them to `double-spend` coins (spend the same coins twice) or censor transactions.

This attack is extremely expensive to pull off on a large, well-established blockchain like Bitcoin, making it highly unlikely. It requires more hashing power (and consequently, more energy and dedicated hardware) than the rest of the network combined.

9.7 Other Consensus Mechanisms

Proof-of-Stake (PoS): Instead of relying on computational work, PoS relies on `staking`. Users `stake` a certain amount of cryptocurrency, and their chance of being selected to validate a block (and earn rewards) is proportional to the amount they stake. This is much more energy-efficient than PoW. Ethereum is transitioning to a Proof-of-Stake consensus mechanism.

9.8 Confirmations

Confirmations refer to the number of blocks that have been added to the chain on top of the block containing a given transaction. Each new block makes it exponentially harder to revert the previous transaction. A given vendor/business may require 5 or more confirmations before a transaction is `cleared`, which is the point at which the transaction is considered irreversable.

9.9 Implications for Distributed Systems

Blockchain demonstrates several important principles of distributed systems:

1. **Consensus:** Achieving agreement among distributed nodes in the presence of failures and malicious actors. Consensus algorithms, such as Proof-of-Work (PoW) and Proof-of-Stake (PoS), are used to ensure that all nodes agree on the state of the blockchain. These protocols must handle situations where nodes might crash, be unreachable, or even be intentionally trying to disrupt the network (Byzantine Fault Tolerance). See Lamport, Shostak, and Pease's work on the Byzantine Generals Problem for a foundational understanding of this challenge The Byzantine Generals Problem By Lamport Paper.
2. **Fault Tolerance:** The system can continue to operate correctly even if some nodes fail (e.g., crash, become unresponsive) or become malicious (attempt to corrupt the data or disrupt the network). Blockchain achieves fault tolerance through data replication (each node has a copy of the blockchain) and the consensus mechanism, which ensures that a majority of honest nodes can maintain the integrity of the chain.
3. **Data Consistency:** While *strict* consistency (where all nodes see the *exact* same data at the *exact* same time) is often impossible in a distributed system due to network latency and potential partitions, blockchain achieves *eventual* consistency. The longest chain rule ensures that, over time, the network converges on a single, agreed-upon history. Any temporary forks (due to race conditions in block creation) are resolved as the chain with the most accumulated proof-of-work becomes dominant. This relates to the CAP theorem (Consistency, Availability, Partition Tolerance), where blockchain generally prioritizes Availability and Partition Tolerance over strict Consistency.
4. **Immutability:** Data, once added to a block that has sufficient confirmations on the blockchain, is extremely difficult to alter. This provides a tamper-proof record. This is primarily enforced via the use of cryptographic hashing that is used in each block, where that block's contents as well as the hash of the preceeding block are hashed. This means to make a change would require changing all of the blocks, which is computationally very expensive, and requires controlling a majority of the system's computational resources, which is referred to as a 51% attack.

9.10 Recommended Reading

- CAP Theorem

End of Week 9

10 Week 10 : CAP Theorem

10.1 CAP Theorem - Intro

The CAP Theorem states that in a distributed system, you can only guarantee two out of the following three properties:

1. **Consistency (C):** Every read receives the most recent write or an error. All nodes see the same data at the same time.
2. **Availability (A):** Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
3. **Partition Tolerance (P):** The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

The core idea is that when a network partition (a communication breakdown) occurs, you must choose between consistency and availability. You cannot have all three.

Rob G. references "Distributed systems for fun and profit on chapter 2", which has a helpful Venn diagram, as well as a link to the reading at book.mixu.net.

10.1.1 CAP Theorem Combinations

Because of the "choose two" nature, there are three primary system types:

1. **CA (Consistency and Availability):** These systems prioritize consistency and availability in the absence of network partitions. However, when a partition occurs, they may become unavailable to maintain consistency. Examples include traditional relational databases that do not account for partition tolerance, such as with a two-phase commit.
2. **CP (Consistency and Partition Tolerance):** These systems prioritize consistency even during network partitions. If a partition occurs, they may become unavailable on some nodes to ensure that data remains consistent across the accessible nodes.
3. **AP (Availability and Partition Tolerance):** These systems prioritize availability, even during network partitions. They will continue to serve requests, but some nodes might return stale or inconsistent data.

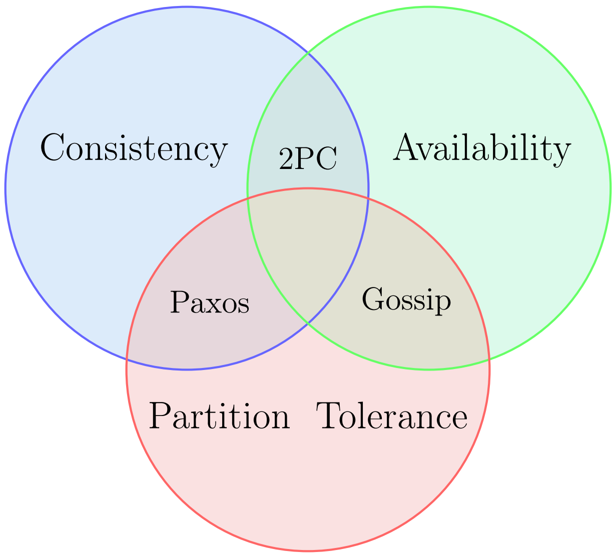


Figure 1: CAP theorem Venn Diagram

10.2 Two-Phase Commit (2PC) as a CA System

In 2PC, there are n nodes in a distributed database.

1. A coordinator node initiates a transaction by sending a `lock` request to all other $n-1$ nodes.
2. All $n-1$ nodes respond with `OK` to acknowledge the lock request.
3. The coordinator makes the change.
4. The coordinator sends a message, to all other nodes, to indicate that the change has been made.

5. All $n-1$ nodes respond with `OK` to aknowledge.

This protocol ensures **strong consistency** because all nodes must agree before a change is committed and the data is available. However, if any node fails or a network partition occurs, the entire system can become unavailable. The coordinator needs all of the responses.
This type of communication requires $4(n-1)$ messages in total.

10.2.1 Partition Tolerance and Majority Rules

Consider a system with five nodes. If a network partition splits the nodes into two groups (e.g., three in the USA and two in Asia), the group with the majority of nodes (three in this case, in the USA) can continue to operate and make changes.

The disconnected nodes (in Asia) become unavailable to maintain consistency. When the partition is resolved, the nodes can rejoin, and the data can be synchronized. This system chooses Consistency and Partition-tolerance, sacrificing availability on all nodes.

10.3 Availability and Partition Tolerance (AP System)

With an AP system, all partitions of nodes can continue to operate even in the face of a total partition. There is no lock, or strong consistency, so the changes that happen on a given side of the partition are not guarenteed to be the same changes that happen on another node.

When a merge of the partitions happen, there is no way to know which changes are `right`. The system continues to operate, but this may result in the nodes having conflicts, and stale or inconsistent data.

10.3.1 A Note About Even-Numbered Partition Tolerance

When making thought experiments of partitioning, it can be important to maintain an odd number of nodes for the sake of determining majority. With an even number of nodes, it can be impossible to determine, as two equal partitians are not a majority.

10.4 Key Takeaways

- The CAP Theorem states that you can only guarantee two of Consistency, Availability, and Partition Tolerance in a distributed system.
- When a network partition occurs, you must choose between consistency and availability.
- There is a strong tension between consistency and availability.
- Different system designs (CA, CP, AP) make different trade-offs based on the application's requirements.

End of Week 10

11 Week 11 : DHT(s)

11.1 Introduction to Distributed Hash Tables (DHTs)

A Distributed Hash Table (DHT) is a decentralized distributed system that provides a lookup service similar to a hash table; (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

DHTs are a fundamental building block for many distributed systems. They form the basis of many peer-to-peer (P2P) applications, distributed file systems, and databases.

11.2 Table ADT

A table, as an Abstract Data Type (ADT), is a fundamental concept in computer science. It's a structure that stores key-value pairs and supports, at a minimum, the following operations:

1. `add(key, value)`: Inserts a new key-value pair into the table.
2. `get(key)`: Retrieves the value associated with a given key.

This seemingly simple structure allows for great flexibility and, more importantly, hides implementation details from the user. The data could be stored on your local machine, in the cloud, or, in our current topic of discussion, distributed across nodes in a network.

11.3 Basic Distributed Hash Table Implementation: A Linear Ring

11.3.1 Concept

The simplest way to visualize a DHT is as called **Chord** which is essentially a ring. Imagine all possible keys (which we'll assume are integers for now) laid out in a circular fashion, from some minimum value (often 0) to some maximum value (e.g., $2^n - 1$, where n is the number of bits in the key). Each node in the system is assigned a random ID within this key space. We will use a hash function to map data keys to this key space.

Each node is responsible for a segment of the key space. Specifically, a node is responsible for all keys between its *predecessor's ID* (exclusive) and its own *ID* (inclusive). This forms a *keyspace*.

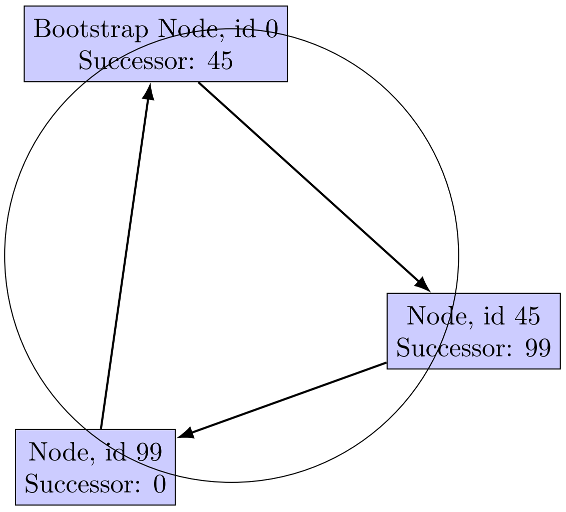


Figure 2: Chord Ring Structure with Three Nodes

11.3.2 Joining the Ring

When a new node wants to join the ring, it does the following:

1. **Bootstrap Node**: The new node contacts a known node in the ring. This initial contact point is often called the *bootstrap node*. In the simplest case, the bootstrap node could initially hold all keys and values.

2. **Key Range Calculation**: Upon connection, the new node announces a random ID chosen from within a specific key range. If the random key already exists in the network, leave the network and connect again with a new key.
3. **Insertion**: The new node is inserted into the ring, and the keyspace responsibilities are adjusted. The new node becomes the *successor* of its *predecessor*, and the *predecessor* of its *successor*.

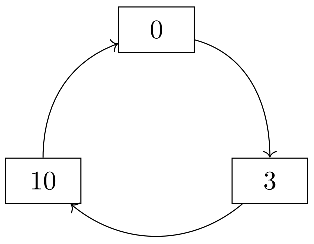
11.3.3 Lookup Operation

The lookup operation is the core of a DHT. Here's how it works in this basic linear ring:

1. **Start Query**: A node receives a query for a specific key.
2. **Check Keyspace**: The node checks if the key falls within its own keyspace (between its predecessor's ID and its own ID).
3. **Forward Query (if necessary)**:
    - If the key is within the node's keyspace, the node returns the corresponding value.
    - If the key is *not* within the node's keyspace, the node forwards the query to its *successor*.
4. **Iterative Process**: This process (check keyspace, forward if necessary) repeats until the node responsible for the key is found.

Initially, you may have node 0 as the bootstrap, which `owns` the entire key space. Then, let's imagine nodes with IDs 3 and 10 join. The ring might look like the graph on the right:
In this case, we can view the connections between the nodes as a doubly linked list. This list represents the ring.

- Node 3 is responsible for keys 1, 2, and 3.
- Node 10 is responsible for keys 4, 5, 6, 7, 8, 9, and 10.
- Node 0 is responsible for keys 0, 11, 12, 13, 14, and 15.


$$\begin{tikzcd} & 0 \arrow[dr, bend left] & \\ 10 \arrow[ur, bend left] & & 3 \arrow[ll, bend left] \end{tikzcd}$$

11.3.4 Leaving the Ring (Graceful Shutdown)

If a node wants to leave the ring gracefully (i.e., not a sudden failure), it should:

1. **Notify Successor**: The leaving node tells its `successor` that it has a new `predecessor` (the leaving node's predecessor).
2. **Pointer Update**: The successor's `prev` pointer should be updated to the same value that was in the exiting node's `prev` pointer.

11.4 Handling Node Failure

11.4.1 The Problem

In a distributed system, nodes can fail (e.g., crash, lose network connectivity). If a node fails, queries that need to traverse that node will also fail. In our simple linear ring, if the successor node is unreachable, there's no way to route around the failure.

11.4.2 Detection

How do we detect that a node has failed? One approach is to have nodes periodically send `ping` messages to their successors. If a node doesn't receive an `acknowledgment (ACK)` message back within a certain timeout period, it assumes its successor has failed.

11.4.3 Recovery (Basic)

If a node detects that its successor has failed, it can:

1. **Query the Bootstrap**: The node contacts the bootstrap node (or any other known node in the ring).
2. **Rejoin the Ring**: The node essentially rejoins the ring as if it were a new node. It contacts another node in the ring to insert itself, taking over the keyspace of the failed node.

11.5 Improving Lookup Performance: Finger Tables

11.5.1 The Problem with Linear Lookup

With a simple linear ring, lookup time is $O(n)$, where n is the number of nodes. This is because, in the worst case, a query might need to traverse the entire ring. This is unacceptably slow for a large distributed system.

11.5.2 The Solution: Finger Tables

To speed up lookups, each node maintains a *finger table*. The finger table is a set of pointers (or indexes) to other nodes in the ring, spaced at exponentially increasing distances.

The i^{th} entry in the finger table of node **n** points to the first node that succeeds $n + 2^i \pmod{2^n}$.

So, a node with ID 0 would have a finger table containing:

- id + 2^0
- id + 2^1
- id + 2^2
- id + 2^3
- ...
- id + 2^{n-1}

And each successive entry would store a node that is at least that far away from it in the ring.

Example 11.1: Example Finger Table for 8 nodes

In Table **??**:

- The ID is implicit (the node for which this is a table).
- id + 2^i is the target key.
- "successor" is the ID of the node responsible for that target key (or the next closest node in the ring).

The advantage of a finger table is that it enables logarithmic lookup. With each hop, the distance to the destination key is roughly halved.

i	id + 2^i	successor
0	1	3
1	2	3
2	4	6
3	8	10

Table 5: Finger Table

11.6 Race Conditions and Connection Issues

11.6.1 Race Conditions

Because nodes can join and leave concurrently, race conditions can occur. For example, two nodes might try to become the successor of the same node simultaneously. The Chord protocol uses the "last writer wins" rule (using timestamps or some other ordering mechanism) to resolve these conflicts.

11.6.2 Connection Issues

Nodes might be temporarily unreachable due to network problems. The system needs to be robust to this. Periodic checks (pings) and retries are essential. UDP is typically used for communication due to its connectionless nature, allowing messages to be sent and received without maintaining a persistent connection. However, this also means UDP is unreliable, and messages might be lost.

11.7 Further Considerations

- **Consistency Models**: Distributed systems often use *eventual consistency*, meaning that updates will eventually propagate through the system, but there might be temporary inconsistencies.
- **Fault Tolerance**: DHTs are designed to be fault-tolerant. The failure of a single node should not bring down the entire system. Data replication is commonly used to further enhance fault tolerance.
- **Security**: In real-world deployments, DHTs need to consider security aspects. How do you prevent malicious nodes from joining the ring and disrupting lookups or corrupting data?
- **Chord and Other DHT Algorithms**: Chord is a specific, well-known DHT algorithm. Other algorithms like Kademlia, Pastry, and Tapestry exist, each with different trade-offs in terms of performance, complexity, and robustness.

- **Real-World Applications**: DHTs are used in BitTorrent (for tracking peers), distributed databases (like Cassandra and Riak), and many other distributed systems.

End of Week 11