CS 355 Internet and Web Technologies - Topic 1

Nicholas Farkash

Saturday, August 31, 2024

1 Compiled vs Interpreted Language

A compiled language must have its source code ran through a compiler to turn it all into a machine code executable before it can be executed. The compilation process is slow, as the compiler looks for and performs optimizations. However, once compiled, the executable runs fast.

An interpreted language does not need a compiler, but rather an interpreter. The code is passed through the interpreter and code is executed line by line. The speed of execution depends on what is in the source file. It is not always faster than a compiled language.

JavaScript is both compiled and interpreted. When JS code is ran, it is duplicated, with one source code file being fed into a compiler while the other is sent through an interpreter. When one of them completes an action (such as printing to the console or performing a calculation), the engine keeps track of how far each is in its execution process, so the one that is lagging behind does not produce duplicate results. When the lagger surpasses the leader, it begins producing the results instead. This is called "Just In Time" (JIT) compilation, when whichever runs faster produces results while the slower is "muted".

2 Basic JavaScript Syntax

2.1 Basic Functions

Printing to the console can be accomplished via the "console.log()" function.
"console.table()" can also be used for printing tabular data.
typeof x gives the data type of the variable x.
let/const to define variables.

2.2 Data Types

The JavaScript language features 7 built in data types, 6 of which are primitives, each with its own wrapper class with methods, constructors, and properties:

1. Number - all-encompassing class for anything that resembles a number. Following the IEEE754 double precision format, "Number" can be thought of as a "double".
2. String - functions the same as a Java "String" object. Can be created using single or double quotation marks or backtick/grave (‘) to create a template string, which allows for expressions to be evaluated within it. For example, console.log(‘Hello 1 + 1‘); would print "Hello 2" to the console.
3. Boolean - standard Boolean True/False. Note that 0 evaluates to false, while any other number will evaluate to true.
4. Undefined - type for a variable that has been declared, but no value has been assigned to it.
5. Null - a value assigned to a variable. Not the same as "Undefined".
6. Symbol - a primitive that is unique; no two symbol variables are the same.
7. Object - NOT A PRIMITIVE - stores a collection of other data types.

3 Objects and Other Complex Types

Objects {a:3, b:'John', c:false} are similar to classes/structs - they store a combination of different data types in a single variable. Dot notation is used to access variables within an object. Access feels similar to Python in that it is very flexible.

4 Functions

4.1 Function Declaration (Traditional Function)

Always starts with "function" keyword:
```
function sum(a,b){
    return a+b;
}
```

4.2 Function Expression

Assigning a function declaration to a variable. The function does not need a name. If you want to apply the function to all elements of an array, use arr.map(varNameStoringFunction):

```
let mySum = function sum(a,b){
        return a+b;
    }
console.log(mySum(3,4)) prints 7
console.log(mySum) → function
```

4.3 Arrow Expressions

AKA lambda expression - () => - convenient for simple calculations, but anything requiring multiple lines is error prone - avoid at all costs.

let sq = (x) => x*x

console.log(sq(3)) prints 9

5 Spread and Rest Operators

The "spread" and "rest" operations use the same operator (...). It is used to convert an array into a parameter list (spread) and vice versa (rest). Rest is ONLY ever used in function definition parameter lists; it essentially says "I'll take any number of parameters". Anywhere else the operator appears, it is a Spread operation

CS 355 Internet and Web Technologies - Topic 2

Nicholas Farkash

Saturday, September 7, 2024

1 Stack vs. Heap

The engine that runs a program is composed of two memory parts:

1. Stack - stores values from live execution, including those in function calls. Mostly used for primitives that are fixed in size.
2. Heap - raw chunk of memory used to store dynamic variables, such as objects, that can change in size.

When a new primitive is created, a new entry is placed in the stack. That entry contains the primitive's identifier, address in memory, and value. For example, "let x = 10;" will place an entry in the stack that looks like:

Stack		
ID	Addr	Value
x	0x01	10

When a new object is created, a new entry is placed in the stack. That entry contains the object's identifier, address in memory, and value. Since an object's value can dynamically change in size, the value of an object cannot be stored in the stack - it must be stored in the heap. Therefore, the value stored in the stack entry is the memory address of (pointer to) that object's value in the heap. For example, "let f1 = {a:3, b:4};" will place an entry in both the stack and heap that look like:

Stack			Heap	
ID	Addr	Value	Addr	Value
f1	0x01	0xA1	0xA1	{a:3, b:4}

Memory is essentially a lookup table to the stack and heap. Everything in JavaScript is passed by value. Nothing is pass by reference. However, sometimes the value that is passed is a reference, such as in the case of objects, whose value is a reference to an address in the heap.

1.1 Notes on Objects and Memory

1. You can add attributes to an already-existing object by using the dot operator. "obj.attr = 10;" will add the attribute "attr" to the object "obj" and assign it the value of 10.

2. Nested objects use pointers in the heap to keep track of relationships between objects.

3. The Shallow Copy Operator (...{}) is used to create a copy of an object. It copies the attributes of the object which are one level deep. It does not create an exact copy of the object with all of the reference relationships in tact. To get around this, you can use:

   (a) Method 1: Modern Approach - use a structuredClone(), which copies itself recursively to reach all levels. Only possible on updated machines.

   (b) Method 2: Legacy Approach - use JSON to turn the object into a string and then back into an object, allocating new memory for all nested objects. This works on all machines, but requires the object to be represented by a string, which is not always possible.

2 Scope for Variables

JavaScript has 4 different scopes:

1. Module Scope - Will not be discussed.

2. Global Scope - variables "declared" without using "let" or "var" (like Python where you just write "x=10;" instead of "var x = 10;"). Global variables are accessible from anywhere, including other files that the variable is not declared inside of.

3. Function Scope - variables declared using "var". Function scoped variables are accessible from anywhere inside the function, even if the variable is declared after it is used or within a block that is unreachable. This is because JavaScript moves declarations to the top of the function block. If it is an initialization statement, the declaration moves to the top and the assignment stays where it is.

4. Block Scope - variables declared using "let". Block scoped variables work as one would expect; they can only be accessed after being declared and only exist for the block they are declared inside of.

3 JavaScript Closures

A closure is the preservation of ACCESS to a variable that would otherwise be out of scope. It does NOT preserve the variable itself - only ACCESS to the variable. Take the function:

```
function outer(a){
    let b=20;
    let unused = 50;

    return function inner(c){
        let d=40;
        return `${a}, ${b}, ${c}, ${d}`;
    }
}

let i = outer(10);
let j = i(30);
console.log(j);
```

In this code, the variable 'i' stores a function that requires access to variables 'a' and 'b', but once 'i' is called, there is no reference to 'a' and 'b' anymore, as they are out of scope. JavaScript avoids this potential issue with closure. When the stack frame that stores 'a' and 'b' is popped from the activation record, the associated pieces of memory are "closured", meaning that access to those variables is still possible. This allows for the expected output of "10, 20, 30, 40".

CS 355 Internet and Web Technologies - Topic 3

Nicholas Farkash

Saturday, September 14, 2024

1 Asynchronous Programming (C05 - C06)

"Kinda like threads, but without the problems of threads"

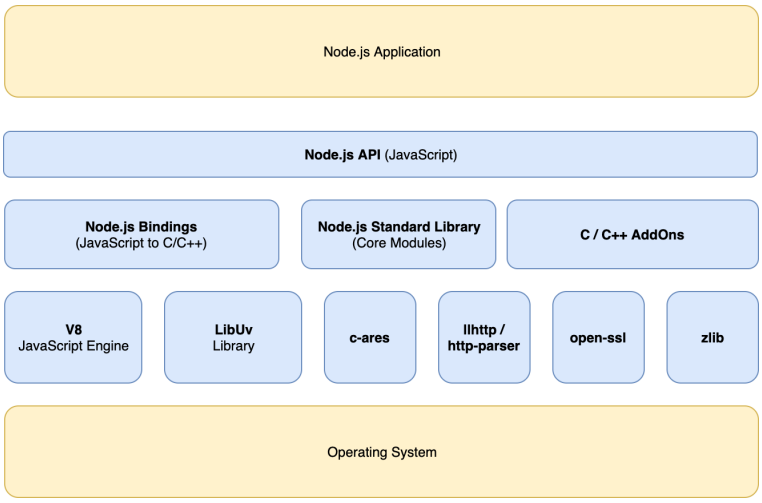
Note: (#) - On the website, go to today's lecture, **Code**, **Download Files**, and see file # for an example.

1.1 Blocking Code

Blocking code takes up 100% of the CPU, not allowing anything else to run until it finishes. If you have blocking code on a website, clicking a button has no immediate effect; instead, it adds the action to a queue. Once the blocking code finishes, it dequeues the actions, resulting in all those button clicks activating at once and opening a bunch of tabs. We do not want this.

1.2 Node.js Architecture

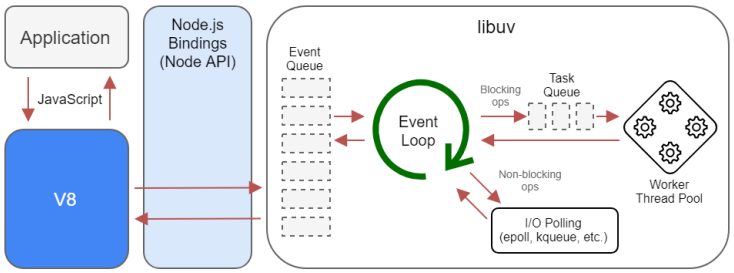
Node.js's architecture interacts with the operating system via 6 features, which are written in C++:



- V8 - Stack/Heap/JIT Compiler, the engine that processes code
- libuv - support for asynchronous programming
- c-ares - allows for translation between IP address and domain names (google.com <=> 192.158. 1.38)
- http-parser - parses http text into a JS object
- open-ssl - standard for internet security encryption (lock icon by URL)
- zlib - compression and decompression of files

1.3 libuv

Runs asynchronous code separate from the V8 engine. Works as follows:



1. The V8 engine runs the JS code. If the V8 engine encounters a piece of asynchronous code (task), it delegates that task to libuv and continues running the next line of code. It does not wait for libuv to process and finish its task. (6)
2. When libuv receives an asynchronous task, it goes into the Task Queue, where it waits for a Worker Thread to take it and execute it. When the Worker Thread finishes executing the task, the task goes into the Event Queue.
3. The Event Loop monitors the interactions between all parts. It is notified when a Worker Thread places a finished task in the Event Queue and checks if V8 still has code in it. If so, it waits and checks again. If not, it pushes the first item in the Event Queue into V8. This action (the callback specifically) will place more code into V8, and the process repeats until the Event Queue is empty.

1.4 Asynchronous Functions

1.4.1 Callbacks (2)

All asynchronous functions take a parameter "callback". This is a function that is executed when the asynchronous task is completed and returns to the V8 engine. Some callback functions have output parameters - check the documentation (`https://nodejs.org/api/`) to see the specifics for each function.

1.4.2 fs.readFile(path[, options], callback) => (err, data)

Reads data from a file located at **path** and stores it in output parameter **data**. Also takes in optional parameters **options**. Also returns an **err** code. Executes **callback** when completed.

1.4.3 dns.resolve(hostname[, rrtype], callback) => (err, records)

Returns the IP address (stored in **records**) of **hostname**. Also takes in optional parameters **rrtype**. Also returns an **err** code. Executes **callback** when completed. (3), (5)

Note: DNS - Domain Name System - pairs a url/hostname to an IP Server address. Since servers are connected to the internet, an internet connection is required for DNS.

1.4.4 fs.writeFile(file, data[, options], callback) => (err)

Writes **data** to a specified **file**. Also takes in optional parameters **options**. Returns an **err** code. Executes **callback** when completed.

Note: For **options**, our "encoding" will always be 'utf8' for string data and 'null' for binary data.

1.4.5 zlib.deflate(content, callback) => (data)

Takes **content**, compresses it, and returns the compressed **data**. Executes **callback** when completed.

1.4.6 zlib.inflate(content, callback) => (data)

Takes **content**, decompresses it, and returns the decompressed **data**. Executes **callback** when completed.

1.5 Generalizing an Asynchronous Program

When calling one async function, we need to specify the "callback" function for each function. In the case of multiple DNS resolves, it is very costly to have a separate function for each resolve. (5)

We can generalize by using closure - wrap the callback function in another function. This will create a new frame, and the value will be preserved in the Environment Variable in the heap. (7) (Recall: A closure is created when a function has a dependency on something **outside** of the function - the garbage collector will collect everything that is not being pointed to from the current scope, so if we needed access to something outside of the function, and enclosure would be created to preserve **access** to the variable.)

In general, for the repetition of function calls, we can use either recursion or a loop. However, these two methods work differently with asynchronous functions. Moving forward, recursion will be used on synchronous code and loops will be used with asynchronous code.

1.5.1 Loops and Asynchronous Functions

We can use a counter to bring n asynchronous threads into one synchronous line of execution. We can also guarantee the order of return by returning items to an array via the index they were called in. (12)

CS 355 Internet and Web Technologies - Topic 4

Nicholas Farkash

Saturday, September 21, 2024

1 OSI & TCP/IP Model (C05)

OSI - Open Systems Interconnected Model

The OSI Model has 7 layers. The TCP/IP Model has same layers as the OSI Model, but three of the layers are grouped into one. The goal of layering is to provide abstraction; higher layers assume the lower layers work.

1.1 OSI Model

1. Application
2. Presentation
3. Session
4. Transport - ensures delivery to correct program; get to right program
5. Network - provides end to end delivery; get to end machine
6. Data Link - facilitates delivery across single hop; get to next machine
7. Physical - facilitates delivery to the first machine

Note: TCP/IP Model groups Application, Presentation, and Session into one Application Layer

1.2 Communication Protocols

A set of strict rules that allow 2+ entities to transmit/receive information:

Syntax: how messages are structured
Semantics: underlying meaning of the message
Synchronization: timing and rate of transmission
Error Recovery*(Optional)*: Detecting and correcting error states

1.3 Computer Protocols

Rigidly defined with no ambiguity allowed.

RFC - Request For Comments - provides documentation for a protocol.

CS 355 Internet and Web Technologies - Topic 4

Nicholas Farkash

Saturday, September 21, 2024

1 Physical Layer (C06)

Involves the bit by bit delivery of data and how that data is encoded on a transition medium.

1.1 Cables

Provide support for end to end connections. Can only connect 2 devices

1.1.1 Copper Cables

Carry voltage to encode data:

0: No Electricity
1: Electricity

Network Interface Card (NIC) - used to translate signals into bits, which can then be organized into DLL frames

Cross-talk - electricity in wires produces a magnetic field. Other wires within the magnetic field will have their current affected, causing interference. This can be reduced by keeping each cable's return cable partner close by (twisted pair).

1.1.2 Fiber (Optic) Cable

Bits are transmitted using light in a glass tube.

0: No Light
1: Pulse of Light

Attenuation Loss / Smearing - light is a wave and scatters in all directions. When the pulse is emitted, some light travels straight, but some light will go at an angle and bounce off the glass tube many times before reaching the receptor at the end. The bouncing light travels a longer distance and takes

longer to reach the end, resulting in a smearing effect between the straight light and the bouncing light.

A drawback of optical cables is that they do not send power like copper wires do. As such, devices like keyboards, mice, and monitors use copper wires. We use Fiber Optic cables to get into buildings, while copper wires are used to connect devices within the building.

1.2 Channel Types

- Simplex (Unidirectional)
- Duplex (Bidirectional)
  - Half Duplex - one party sends at a time (Walkie-Talkie)
  - Full Duplex - both parties can send simultaneously (Cables)

Hubs *(historical device)* - devices in a system are connected to a hub. Information from one device is sent to the hub, and the hub then sends that information to all other connected devices. The hub is "dumb", meaning it has no CPU and cannot do anything of much importance, such as view the information. It can only copy and send it.

The problem with hubs is that they create a **collision domain**, when a network connection is downgraded to a half-duplex because of corruption.

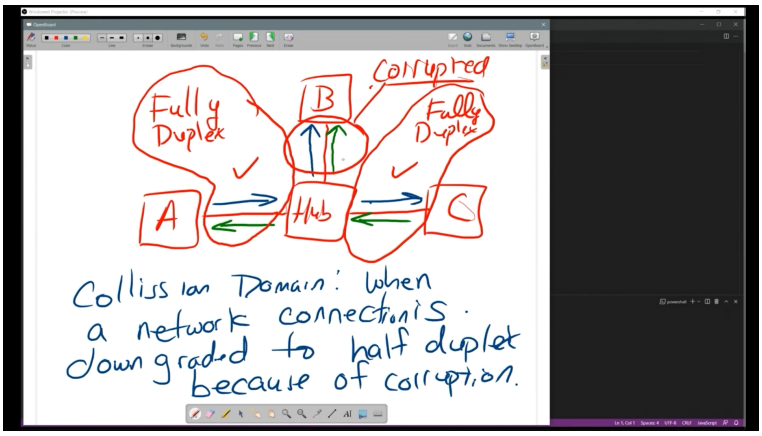


Figure 1: If A and C send data at the same time, B must receive 2 messages at once, which Full Duplex cannot support.

Carrier Sense Multiple Access (CSMA) - procedure to avoid the collision domain issue.

1. Verify that there is no signal being received
2. Send your signal to the hub
3. At any point, if you detect data, stop transmitting

3. is a step added by the **CSMA/CD (collision detection)** protocol, which was created to work around the problem of some devices not having the CSMA protocol.

There is also the **CSMA/CA (collision avoidance)** protocol, which prevents live lock (both repeatedly send data together and lock out the other, like two people walking towards one another shuffling left and right trying to move out of the other's way).

CS 355 Internet and Web Technologies - Topic 6

Nicholas Farkash

Saturday, September 28, 2024

1 Data Link Layer

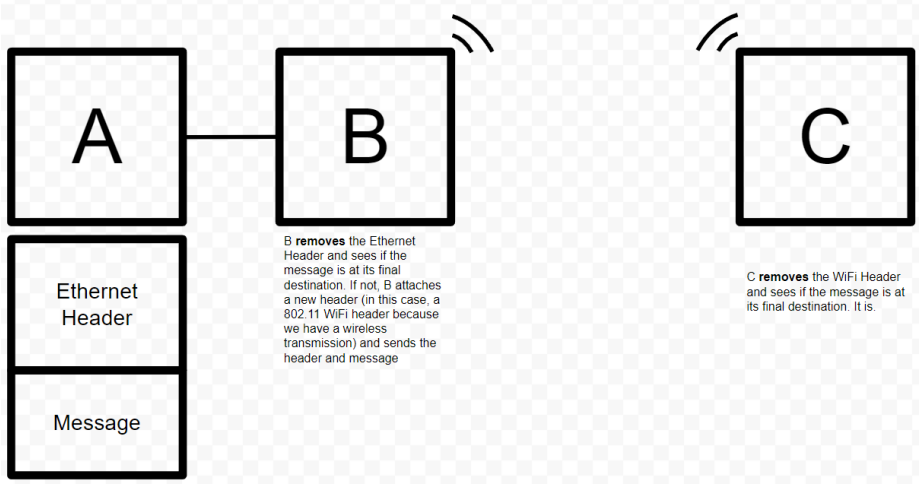
The **Data Link Layer** provides support for hop-to-hop delivery of data between two ***directly connected***, adjacent nodes. The nodes can be connected via a wired or wireless connection. It provides the translation from physical signal to digital bits.

The Physical layer property dictates the frame structure. Wireless and wired connections operate using their own frameworks.

1.1 Ethernet (802.3)

Ethernet is used for all wired connections.

Data & Header Protocol - attach a header to the data and send. On hop complete, the header is removed, signaling the end of DLL. The header divides hops. Think of the header as the mailing address and the data as the envelope contents.



The **Ethernet Header** is comprised of 7 sections:

1. Preamble - 7 alternating bits followed by a '1' - signal to other side that signal transmit is about to begin - used for synchronization of clocks.
2. Source MAC - Media Access Control point from which the signal is being sent from - device ID (*see MAC Address below*).
3. Destination MAC - MAC point from which the signal is being sent to.
4. Ethertype - 2 byte field used to identify which protocol is being used.
5. Payload - the message being sent.
6. Frame Check Sequence - provides data integrity.
7. Interframe Gap - gap of bits for timing/spacing.

1.2 MAC Address

MAC Address - globally unique hardware address used to identify devices in transmission. IEEE manages MAC Address distribution to ensure all unique addresses. Each hardware manufacturer can purchase a **prefix** that identifies the manufacturer. The values after the prefix count up, and the manufacturer is responsible for ensuring that those are all unique.

A MAC Address is comprised of 6 pairs of hexadecimal numbers.

Ex: 3E-AA-01-00-22-A9
Prefix: 3E-AA-01
NIC: 00-22-A9

1.3 Ethertypes

Ethertypes are used to identify the type of protocol used:

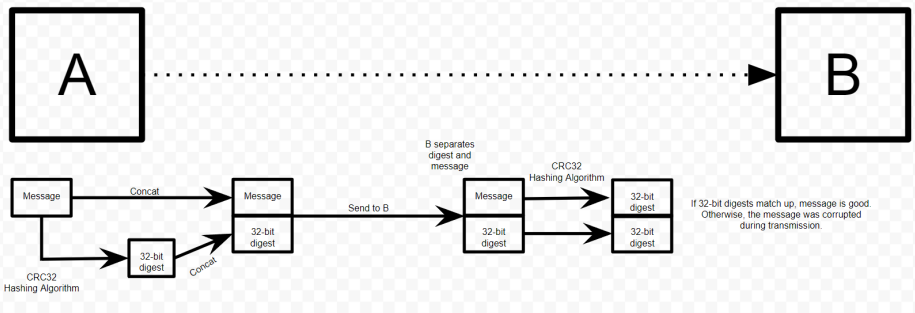
IPV4	0800
IPV6	86DD
ARP	0806
IPV4 w/ VLAN	8100

1.4 Frame Check Sequence

A 4-byte sequence used to provide data integrity. It checks if there was an error in transmission that could lead to corruption. It does NOT have a recovery system built-in. FCS uses a hashing algorithm:

Input: Any variable-sized message

Output: fixed length **digest** (summary of message)



1.5 Network Switch

A **network switch** is an upgraded version of a hub. It is a DLL device with a CPU and memory (not dumb like a hub).

A switch acts as an n-dimensional cable. It does **NOT** have a MAC address, so it is **NOT** a hop destination. A switch is a passive device, meaning that it never sends messages out on its own; rather, it only forwards messages it receives.

When switches are connected to devices, it cannot tell what device is connected to what port. It uses a switch device discovery algorithm to learn.

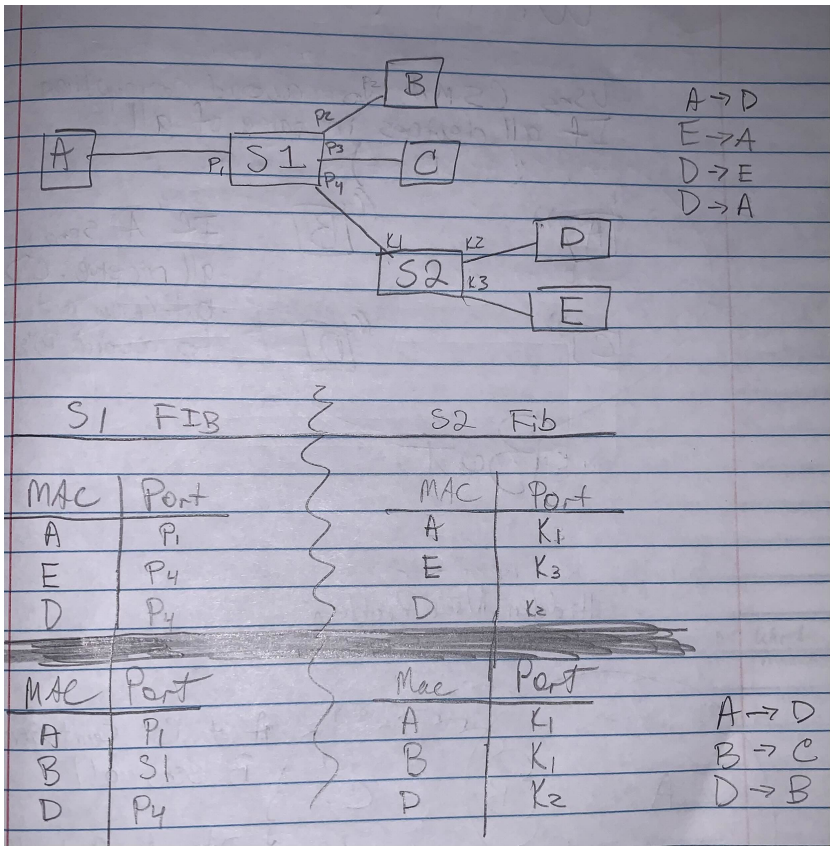
1.5.1 Switch Device Discovery Algorithm

MAC	Physical Port #	Expiration
A	1	****
C	3	****
B	2	****

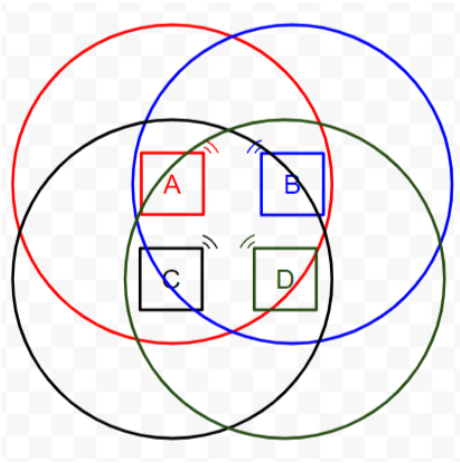
Table 1: A Forwarding Information Base (FIB)

Step 1) When a message arrives on a switch, the switch records the source MAC and the physical port the message arrived on. It stores the information in the FIB.

Step 2) If the destination MAC is already in the FIB, forward the message to that MAC's Physical Port. Otherwise, convert to 'Hub Mode' and broadcast the message to all other ports (not including where the message came from).

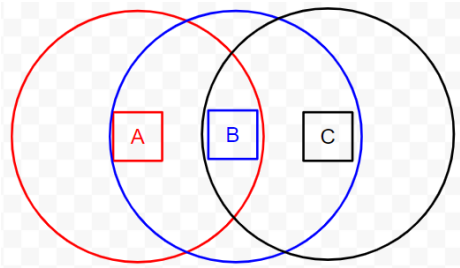


1.6 WiFi (802.11)



Each device has a range associated with it. If A is sending a message to B, then devices B, C, and D will all receive the message. B will accept the message, while C and D will ignore it. However, all devices now know that another device is currently sending data, and that they should not send their own data, so as to avoid corrupting the message from A to B (CSMA is applied locally).

1.6.1 Hidden Node Problem



B can see everything, but A and C can't see each other. A and C might transmit a message to B at the same time, unaware of the other's attempt, and CSMA will not work.

To avoid this problem, RTS/CTS messages are used. In a wireless system, if someone wants to send a message, say A to B:

1. A sends an RTS (Request to Send) message to everyone in its range.
2. If any device in the range is currently sending its own message, they will stop their transmission (CSMA is locally applied).
3. If B is currently receiving data from a device, B will reject the RTS from A.
4. If B is free, B sends out a CTS (Clear to Send) message to all devices in range. All devices in range of B will see this and know not to send an RTS for a set duration.

1.7 WiFi Frame Types

There are 3 major categories of WiFi Frames:

1. Management Frames - connect/disconnect to wireless network
    - Beacon - signal with information about the network - says "Hey, you can connect to me! Here's what you need to know about me..."
    - Probe - WiFi signal ping that attempts to connect to a network.
    - Authentication - network password security.
    - Association - device connected to a network has access to other devices/resources connected to that network.
2. Control Frame - WiFi loses a lot (majority) of messages; manage the connection for when messages do and don't make it through
    - ACK - data was sent out, acknowledge it was received.
    - Block ACK - multiple pieces of data were sent, ACK them all at once.
    - RTS/CTS - Request to Send/Clear to Send message
3. Data Frame - the message being sent and the header sent alongside the message.

CS 355 Internet and Web Technologies - Topic 7

Nicholas Farkash

Saturday, October 26, 2024

1 Network Layer

The **Network Layer** is responsible for end-to-end delivery. It connects any two devices that are connected to the internet.

Why can't we use the Data Link Layer? DLL is responsible for single hops; we can chain single hop DLLs together, but if the route to a machine were to change, the chain of single hops would break and no longer be valid.

Therefore, we build **networks**, collections of computers which are generally organized by geography - computers close to one another are likely to be grouped into a network, like in an office building or school.

1.1 IP Address

IP Address - the Network Layer level's device-specific identification - An IP Address is to the Network Layer as a MAC Address is to the Data Link Layer.

IPV4 format: 4 octets (8 bits) [0-255] - ex: 62.41.8.124

There are 2^{32} different IPV4 addresses $\approx$ 4.2 billion

The US uses IPV4 while the rest of the world primarily uses IPV6. This is because the US has exhausted IPV4 addresses.

1.1.1 Classful Addressing (Historic)

Historically, IP addresses were distributed in classes.

Network Size	Class	First Code	User Pattern	Max Num Devices
Big	A	0-126	MIT: 8.X.X.X	2^{24} = 16 Million
Medium	B	128-191	CUNY: 170.30.X.X	2^{16} = 65,536
Small	C	192-223	Cafe: 200.1.1.X	2^{8} = 256

Similar to a MAC Address on the DLL, the IP has a prefix that identifies the network and the rest of the address specifies the host device. When a router sees an IP address, it looks at the prefix and says, "Hey, this IP starts with an 8. That means it's on the MIT network, so I'll send this to MIT. They'll know exactly what machine it needs to get to over there."

1.1.2 Issues with Classful Addressing

1. Wasteful - MIT doesn't need 16 million devices.
2. Restrictive - There can only be 127 Class-A corporations.
3. Scaling - Sizes change too drastically; if you need an upgrade, you'll waste a lot of space.

We want to keep the concept of using a prefix to identify organizations while fixing the sizing issues.

1.1.3 Classless Addressing (Modern)

Classless InterDomain Routing (CIDR) - modern IPV4 - Features 2 Addresses:

1. Network Address with **12-Subnetwork ID**:
   8.16.0.0/12
2. Device (Host) Address:
   8.31.255.6

The decimal point does not determine the prefix; you can't look and see if a device belongs to a network. Instead, it must be converted into binary:

1. Network Address with **12-Subnetwork ID**:
   00001000.00010000.00000000.00000000
2. Device Address:
   00001000.00011111.11111111.00000110

With a **12-subnet ID**, we look at the first 12 bits and see that they match. Therefore, the Device belongs to the Network:

1. Network Address with **12-Subnet ID**:
   00001000.00010000.00000000.00000000
2. Device Address:
   00001000.00011111.11111111.00000110

1.1.4 Network Size

For an n-subnet ID network, the first n bits determine the prefix, while the remaining $32-n$ bits determine the device ID. The size of a network is determined by this number; the number of devices that can belong to a network.

size of network $= 2^{(32-n)} - 2$
For $n = 12$, size of network $= 2^{20} - 2 \approx 1.05$ million
If we run out of space on a network, no problem! Just ask the ISP to exchange for a new, larger Network Address:

1. Network Address: 8.16.0.0/11 **(11-subnet ID)**

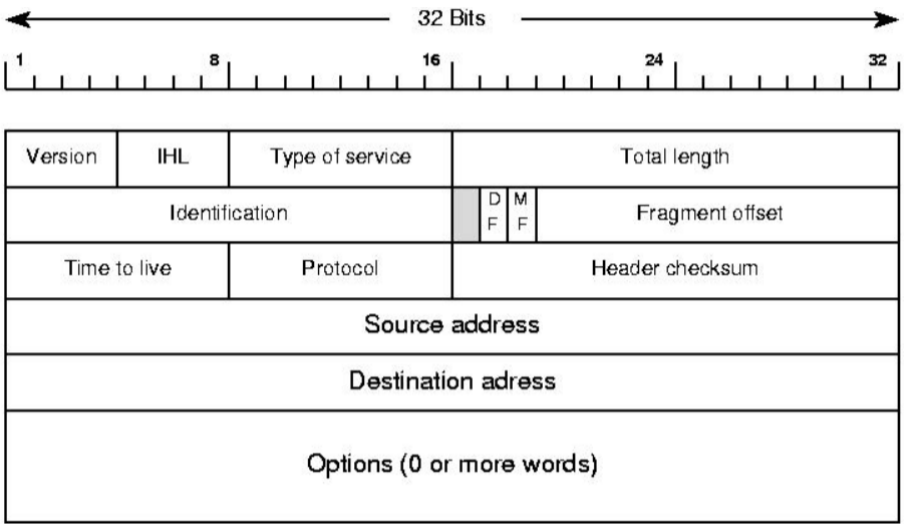
For $n = 11$, size of network $= 2^{21} - 2 \approx 2.10$ million, which is only 2x as big, not 256x as big, solving the issues of Classful Addressing.

1.1.5 Reserved IPs Network

Notice the -2 at the end; that's because two device IPs are reserved on each network:

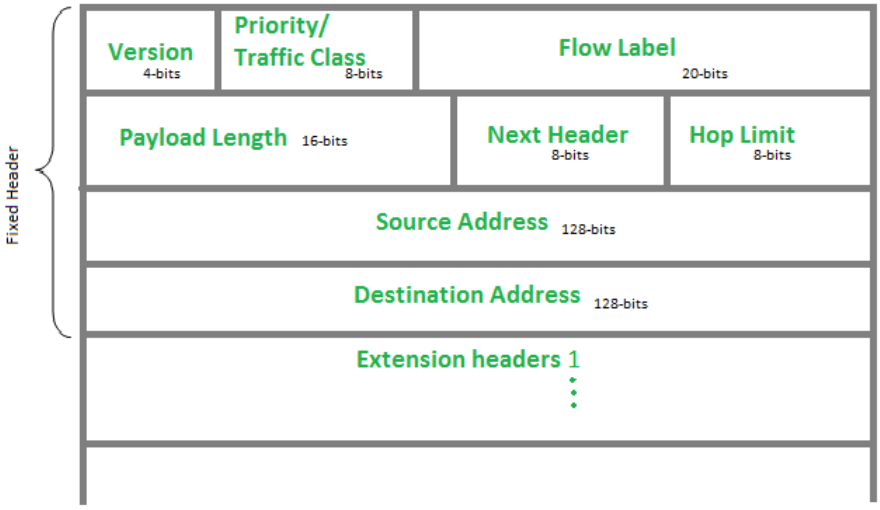
1. prefix + all 0's - network address $\rightarrow$ refers to the network as a whole
2. prefix + all 1's - broadcast address $\rightarrow$ used to send a message to all devices on the network
3. 127.0.0.0/8 - Loopback Range - send message back to self; useful for testing in web development. 127.0.0.1 is LOCALHOST
4. 192.168.0.0/16 or 192.0.0.0/24 - LOCAL NETWORK (LAN) - used for connecting multiple devices without requiring an IP address for each. Acts as temp IP addresses that are only accessible on the network. Communication on these devices can never leave the network, since each network has its own LAN range.
5. 224.0.0.0/4 (technically 224.0.0.0 - 239.255.255.255) - MULTICAST - used when there is one sender and multiple receivers, as in sending a mass update to devices on a network, streaming systems with many people watching one feed, etc.

1.2 IPV4 Format



- **Version** - determines what version of IP the message is using.
- **IHL** - Internet Header Length
- **Type of Service** - unused priority field
- **Total Length** - size of message in bytes
- **Fragmentation Row** - unused method to break up and piece together messages for smaller packet transportation
- **Time To Live** - Number of remaining hops until the message is terminated (used to prevent routing loops)
- **Protocol** - Transport Layer's protocol
- **Header Checksum** - unused data integrity
- **Source Address** - IP address of sender
- **Destination Address** - IP address of recipient
- **Options** - additional information

1.3 IPV6 Format



- **Version** - determines what version of IP the message is using.
- **Priority** - ISP controlled priority value
- **Payload Length** - size of message in bytes
- **Next Header** - Transport Layer's protocol
- **Hop Limit** - prevents router loop
- **Source Address** - IP address of sender
- **Destination Address** - IP address of recipient

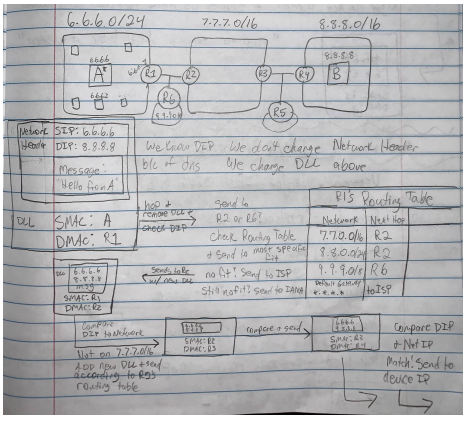
1.4 Sending Messages

Messages contain a DLL Header around a Network Header. The DLL header is responsible for single hops, getting to the next device. When the message reaches a new device, the DLL header is removed and the Network Header is checked to see if the device the message is currently on matches the destination IP address. If it does, the message has arrived. Otherwise, a new DLL header is attached and the message is sent out again. THE NETWORK HEADER DOES NOT CHANGE. ONLY THE DLL HEADER CHANGES.

Routers work similar to switches in how they know where to forward a message to. Each router has a routing table that stores networks that have been communicated with previously as well as the next hop that should be taken to reach that network. It also contains an entry called *Default Gateway* that leads to the ISP. If an IP address isn't known to a router, the router performs ARP (address resolution protocol).

1. ARP converts the IP address to a MAC address
2. Router broadcasts ARP message to all devices on the network
3. All wrong devices (not that MAC address) ignore the ARP message
4. Correct devices accepts and shares its MAC address to the router

it gets sent up the chain until it reaches someone who does know.



1.5 DHCP

Dynamic Host Configuration Protocol - the process by which IP addresses are automatically assigned to devices for leasing purposes. IP addresses are owned by the Network; the network leases an IP address to each device the user connects to the network. The steps for DHCP are as follows:

1. A device is added to the Network - brand new so no IP has been assigned yet
2. The device sends a discovery request on the network to the DHCP device(s) (routers), requesting an IP address.
3. Each DHCP device offers an IP address in its available IP pool to the requesting device.
4. The device accepts whichever IP address it receives first
5. DHCP device logs the IP and MAC addresses in its DHCP table alongside its expiration date, acknowledging that the device has been assigned an IP address

1.5.1 Shortcomings of DHCP

DHCP doesn't work well for server-based behavior. For example, you buy a new printer and automatically connect it to your network. DHCP assigns the printer an IP address, which is stored in the network and is hard-coded on your computer. When the printer's IP expiration data passes, a new IP address is assigned by DHCP. If this IP address is different from the original one, the computer's hard-coded IP address for the printer no longer works.

The solution to this is to statically assign the printer's IP address so it doesn't change. Otherwise, you have to reconnect the printer constantly.

CS 355 Internet and Web Technologies - Topic 8

Nicholas Farkash

Saturday, October 19, 2024

1 Observer Pattern

The "Asynchronous Programming with Callback Function" pattern has some limitations:

1. An asynchronous function call only applies to a single input. *fs.readFile* only reads a single file. *dns.resolve* only resolves a single host name. You need extra code (loop or recursion) to perform the action multiple times.
2. There is an expectation that the function will finish - if the function does not complete, it is considered an error.

The **observer pattern** is used to model behavior that is indeterminate and optional. It uses a tight coupling between two agents: a subject and zero or more observers.

The format for a subject sending a signal is **subject.emit(eventName, [...args])**, where "eventName" is a string and any number of optional arguments are allowed. Note that signals can be emitted without being received, and all observers act independently of one another

The format for an observer receiving a signal and taking action is **subject.on(eventName, listenerFunction())**, where "eventName" is a string that matches the "eventName" signal the subject sends and listenerFunction() is the function that gets called when the signal is received.

The same subject and "eventName" must be used in order to transmit the signal. This is because there is a **tight bind** between an object and its event name.

1.1 Example

There is a subject lion and two observer photographers. If the lion performs:

lion.emit("Roar")

then the lion will send out a signal that it has performed the action "Roar". If both photographer observers are:

lion.on("Roar", listenerFunction())

then listenerFunction() will be called twice. However, if one of the photographers is instead:

cat.on("Roar", listenerFunction())

then listenerFunction() will only be called once. This is because there is a **tight bind** between an object and its event name. Despite the signal names being the same ("Roar"), one photographer is listening for a lion's roar, whereas the other is listening for a cat's roar.

1.2 Libraries/APIs

The "readline" library contains the InterfaceConstructor class, which is used with I/O streams (will be discussed in detail later in the semester). This is used to interact with the terminal (input and output) or other inputs and outputs.

The "events" library contains the EventEmitter class, which contains the "emit" and "on" functions.

1.2.1 Misc. Notes on Programming

The "line" event in "readline" emits a signal when a new line character is detected ($\backslash r, \backslash n, \backslash r\backslash n$).

Using require() with a .json file as its parameter turns the .json object into a JS object.

clearTimeout(id) cancels whichever setTimeout()'s id number is passed in. This can be used to simulate "pausing".

The **map()** function applies a function to all elements of a collection.
array = [a, b, c]
array.map(fn) $\rightarrow$ [fn(a), fn(b), fn(c)]

1.3 Publisher-Subscriber Pattern

The "PubSub" pattern addresses and builds upon the shortcomings of the Observer pattern; namely, it removes the requirement for a tight coupling between the subject and the observer. However, this comes at a cost, as more overhead code is required to account for this feature.

CS 355 Internet and Web Technologies - Topic 9

Nicholas Farkash

Saturday, November 2, 2024

1 Routing Protocols

When a message arrives on a network, the message is checked to see if the header's Destination IP matches the network's IP. If not, the message must be forwarded, but how does the network know who to forward the message to, and how does it do so in the most cost-effective way?

A **Network Operator** is a person/organization/entity in charge of all traffic that flows through a network.

An **Autonomous System** is a collection of networks owned and operated by a single Network Operator with a unified routing policy.

1.1 EGP

Routing Algorithms can be broken into 2 categories: EGP and IGP

Exterior Gateway Protocols - routing negotiations done between DIFFERENT Autonomous Systems

- **Peering Relationship** - traffic between the two (or more) Autonomous Systems are roughly equal (Verizon and AT&T), so they would both benefit from using the other's network infrastructure and are both okay with allowing the other to use their network infrastructure for the sake of their customers - a mutual agreement.
- **Transit Relationship** - one autonomous system (Netflix) sends a lot more traffic to the other autonomous system (Verizon) than vice versa. Customers who don't use Netflix are paying to allow Netflix to use Verizon's network, and they experience slower connections because the infrastructure is saturated with Netflix (which they aren't even using). Verizon charges Netflix to use their network infrastructure so Verizon can use the money to upgrade and maintain the infrastructure so the customers who don't use Netflix don't suffer for no reason.

2 IGPs

Interior Gateway Protocols - routing negotiations done inside of a SINGLE Autonomous System. Examples include RIP, IS-IS, and OSPF

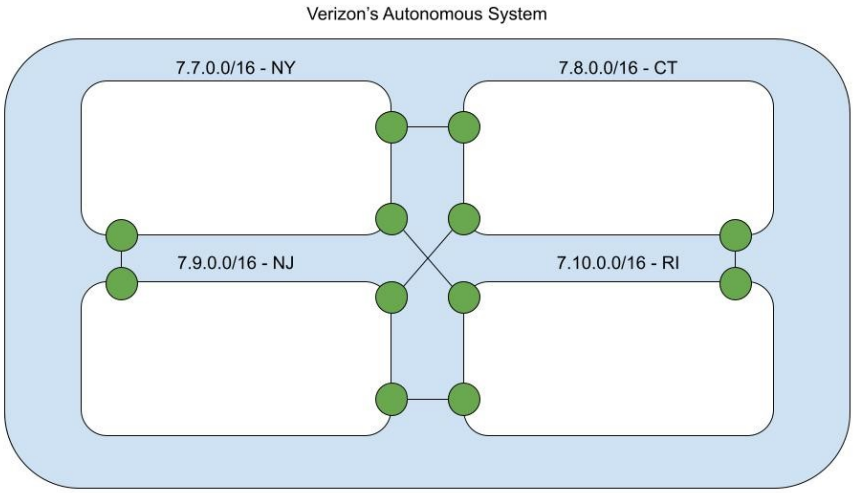


Figure 1: Because Verizon owns all these networks, no money concerns. But which path is fastest?

3 Shortest Path Algorithms

Given a graph G, find the shortest path between any 2 nodes in G

Dijkstra - solves single source* shortest path Dijkstra(G,S,E) - takes the graph and the start and end nodes - O($n$lg n)

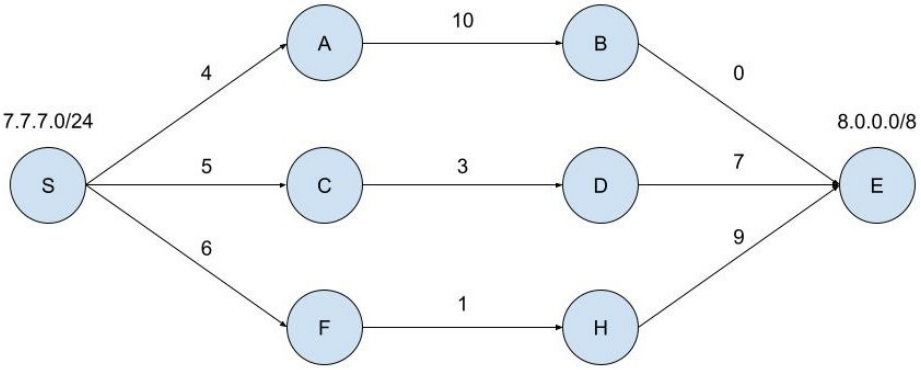


Figure 2: Graph G

Priority Queue	Pop Queue, Add to Results,Relax/Explore	Results
(S,0)	→	(S,0)
(A,4) (C,5) (F,6)	→	(A,4)
(C,5) (F,6) (B,14)	→	(C,5)
(F,6) (D,8) (B,14)	→	(F,6)
(H,7) (D,8) (B,14)	→	(H,7)
(D,8) (B,14) (E,16)	→	(D,8)
(B,14) (E,15) (E,16)	→	(B,14)
(E,14) (E,15) (E,16)	→	(E,14)

Table 1: Results of Performing Dijkstra's Algorithm on G

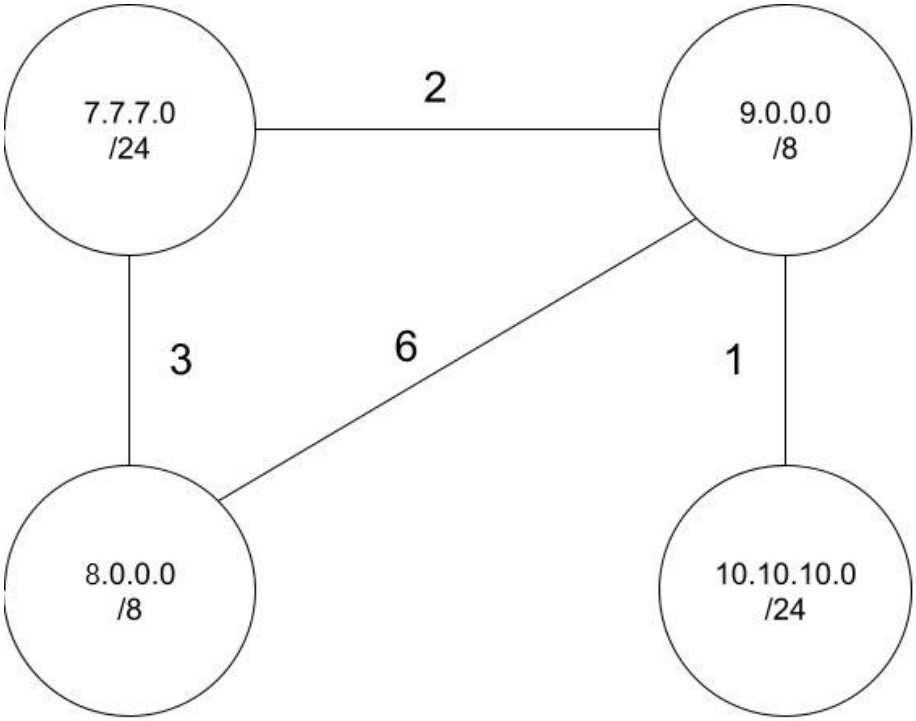
This is great for sending a message from S to E, but if we want to now send a message from S to H, we have no information to use. We would have to calculate Dijkstra(G,S,H).

The problem is that Dijkstra requires the end node to be specified, as it calculates the shortest path TO THAT NODE. We could do Dijkstra for each node (network), but that would be O(n^2lg n), very expensive.

3.1 Distance Vector Algorithm

Distance Vector - a distributed variant of Bellman-Ford - O(n^2), but BF(G) - no need to specify Start or End. Give Bellman-Ford a graph and it outputs a table with the shortest path from each node to each node

Distributed - all networks act independently of one another - each network has a brain (network router that shares info with other network routers)



Network 7 RT

Dest	Cost	NextHop
7	0	7
8	3	8
9	2	9
10	3	9

Network 9 RT

Dest	Cost	NextHop
9	0	9
7	2	7
8	65	87
10	1	10

Network 8 RT

Dest	Cost	NextHop
8	0	8
7	3	7
9	65	97
10	6	7

Network 10 RT

Dest	Cost	NextHop
10	0	10
9	1	9
7	3	9
8	~~76~~	9

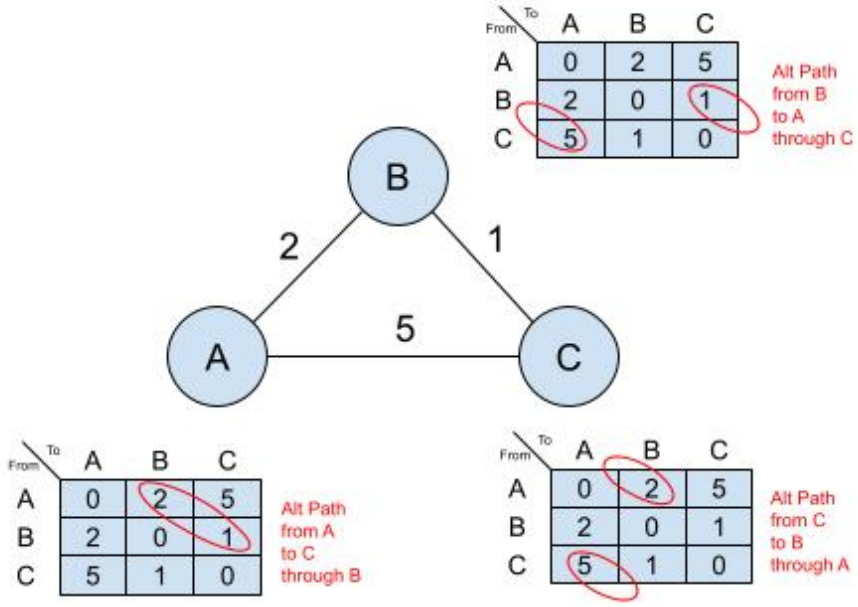
Steps for filling in a Routing Table for a Network: *The order of execution must be given. In this example, the order is increasing numerical order*

1. **Initialize each Network's RT with an entry to itself, where the destination is itself, the cost is 0, and the NextHop is itself.**
2. **Network 7 sends its distance vectors [7|0] to all of its neighbors; 8 and 9. Each neighbor takes the distance vector and adds its edge cost to the Cost field and adds the entry to its own table.**
3. **Network 8 sends its distance vectors [8|0], [7|3] to all of its neighbors; 7 and 9. The neighbors add their edge cost and log the entry in their RT** ONLY IF **the new entry has lower cost. Otherwise, it is ignored.**
4. **Network 9 sends its distance vectors [9|0], [7|2], [8|6] to all of its neighbors; 7, 8 and 10.**
5. **Network 10 sends its distance vectors [10|0], [9|1], [7|3], [8|7] to all of its neighbors; 9.**
6. **Repeat this process for a total of n times, updating if a shorter path is found, to ensure that all shortest paths are properly updated.**

3.2 Bellman-Ford Algorithm

The distance vector algorithm has a problem - it doesn't model breakage. If a link between networks breaks, we can't use that path anymore. If the cost associated with a link changes, that path may no longer be the shortest path. In this case, the networks who were connected would update their cost to the other network to be ∞ and share their info again, but this is very messy.

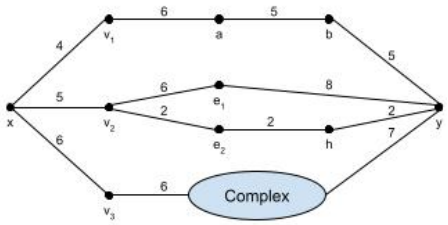
The solution is to have each network maintain a complete table of all connections among networks. Keep track of all paths and update them if breakage occurs and re-share information. Each network uses its own table to calculate (look up) the shortest path in real time.



3.3 Bellman-Ford Equation

$\delta_x(y) = \min_v \{c(x,v) + \delta_v(y)\}$

- $\delta_x(y)$: cost of shortest path from x to y
- v: collection of all nodes connected to x
- $c(x,v)$: direct edge cost from x to v
- $\delta_v(y)$: cost of shortest path from v to y



$\delta_x(y) = \min_v \{c(x,v) + \delta_v(y)\}$
$v_1 : 4 + \delta_{v_1}(y)$
$+6 + \delta_a(y)$
$+5 + \delta_b(y)$
$+5 + \delta_y(y)$
$+0$
$v_1 = 20$

$v_2 : 5 + \min_e \{c(v_2,e) + \delta_e(y)\}$
$e_1 : 6 + 8 = 14$
$\boxed{e_2 : 2 + 2 + 2 = 6}$
$v_2 = 11$

$v_3 : 6 + 6 + \min_v \{c(v_3,v) + \delta_v(y)\}$
v_3 : 12+ something. We can prune this branch because we have a complete solution better than 12 already.

$\delta_x(y) = \min\{20, 11, 12+\} = 11$

CS 355 Internet and Web Technologies - Topic 10

Nicholas Farkash

Saturday, November 2, 2024

1 Transport Layer

Transport Layer - responsible for ensuring that messages are delivered to the correct *application* on a machine. There are two major protocols:

2 TCP vs UDP

1. **Transmission Control Protocol (TCP)**
   (a) Guarantees Reliable Delivery - if a message reaches the other side, it is guaranteed to be error-free
   (b) Guarantees No Duplicates - messages will not be duplicated, ensuring proper action (think toggles - a duplicate toggle will show no change)
   (c) Guarantees No Out-of-Order Messages - messages will be displayed in the order in which they are sent
   (d) Congestion Control (sometimes) - limits the number of messages that can be sent in a batch
2. **User Datagram Protocol (UDP)**
   (a) Guarantees NOTHING
   (b) Provides the bare minimum to send a message - this allows UDP to be cheaper in cost and time, and it is used more commonly than TCP
   - Data and Header protocol - take your message, add a header, and send it out

UDP Header

Source Port (16 bits)	Dest Port (16 bits)
Length (16 bits)	Checksum (16 bits)

Table 1: UDP Header (8 byte size)

3 Ports

Logical Port Number - a number assigned by an OS to an application in order for the OS to tag all messages enter/exiting for ID purposes

When a message arrives on a device, the OS looks at the Destination Port and sends the message to the application associated with that port number.

3.1 Port Ranges

There are $2^{16} = 65,536$ possible port numbers

1. (0 - 1023): Well-Known Ports - reserved for OS protocols - not used by people
   - HTTP: 80
   - HTTPS: 443
   - SSH: 22
   - SFDP: 23
2. (1024 - 49,151): Registered Ports - Ports for programmers to use (server.listen(8000);) Make sure that, when you use a port number for your own application, it isn't the same port number as the default port of another application that may be used in parallel with your application.

   If MYSQL's port number is 3016, don't make your application's port number 3016 if you plan to use it with MYSQL running at the same time.
3. (49152 - 65535): Ephemeral Ports - Used by client applications in order to support receiving responses

   Client Application - the application that sends the first message

   Server Application - the application that receives the first message

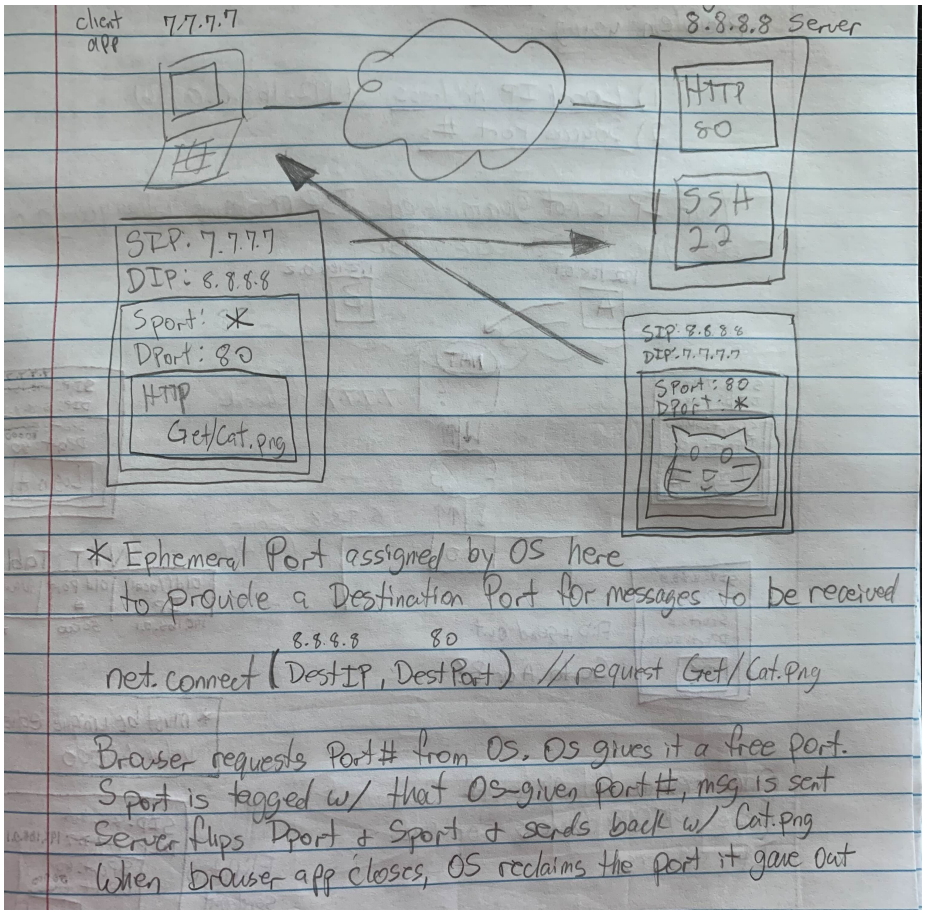
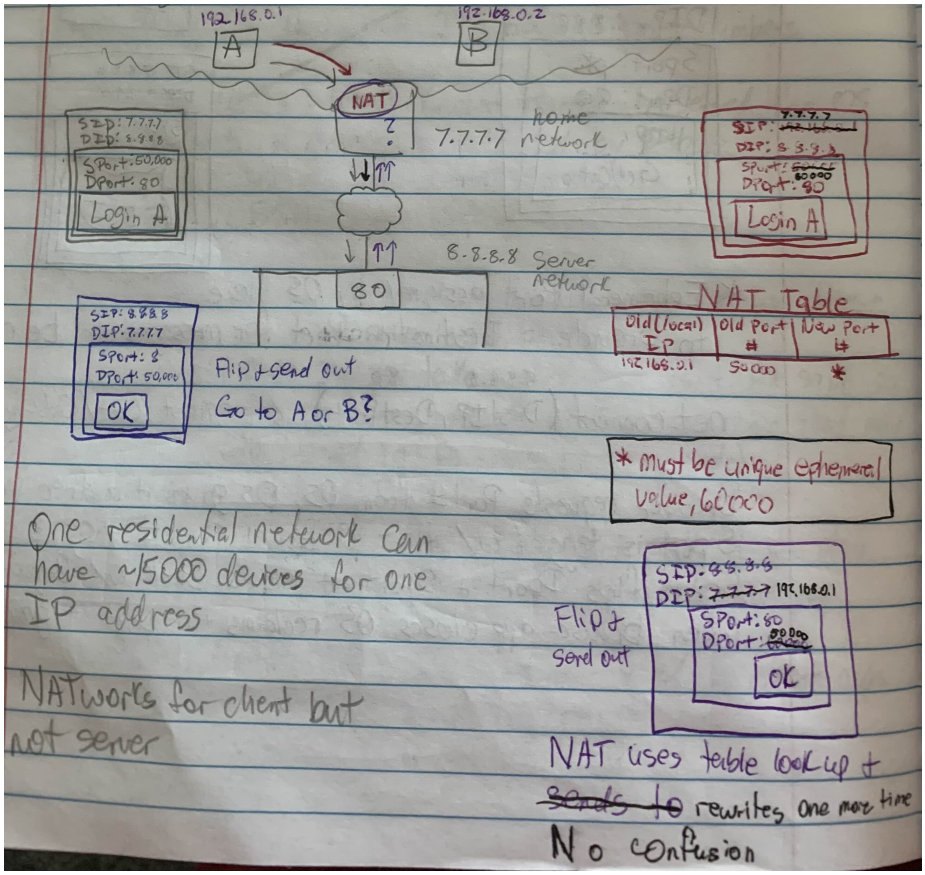


Figure 1: A diagram showing how Ephemeral Ports are assigned when a client sends a message to a server

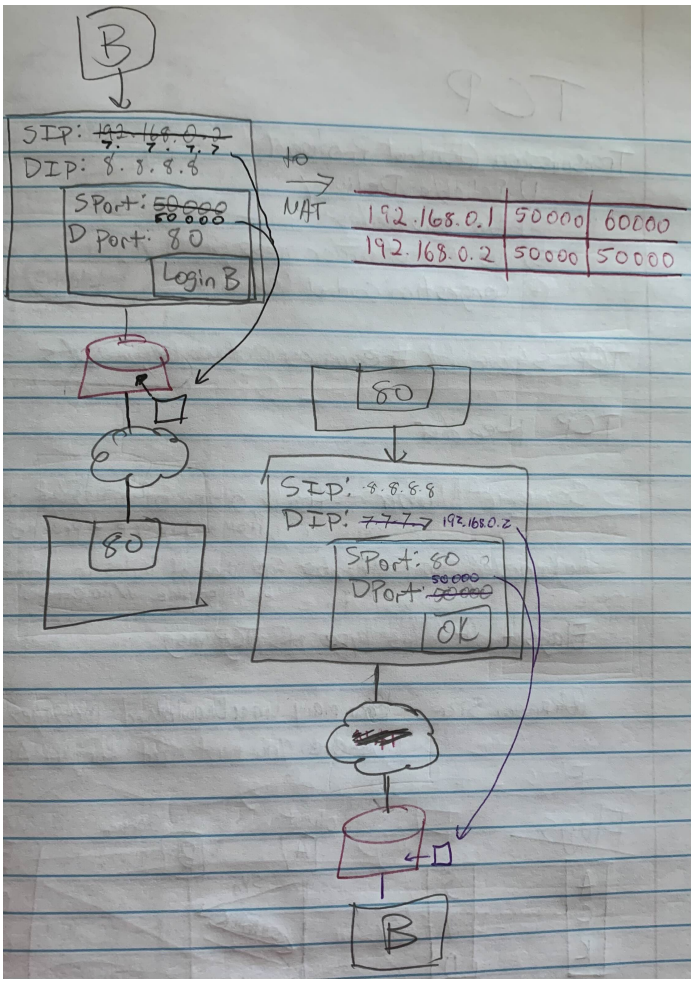
4 Network Address Translation

NAT - A *HACK* solution to the IPV4 exhaustion problem using:

1. Local IP Address (192.168.0.0/16)
2. Source Port Number



1. If A sends a message to the server, on the return trip, the network router doesn't know if it should send the message to A or B.
2. Instead, the router assigns each device a local IP address. In its NAT table, it record the local IP, the port it used, and assigns it a new port. This prevents the return trip forwarding issue.


B
SIP: 192.168.0.2
DIP: 8.8.8.8
SPort: 50000
D Port: 80
Login B
to
NAT
192.168.0.1 | 50000 | 60000
192.168.0.2 | 50000 | 50000
80
80
SIP: 8.8.8.8
DIP: 192.168.0.2
SPort: 80
DPort: 50000
OK
B

5 Transmission Control Protocol (TCP)

1. Guarantees Reliable Delivery - if a message reaches the other side, it is guaranteed to be error-free
2. Guarantees No Duplicates - messages will not be duplicated, ensuring proper action (think toggles - a duplicate toggle will show no change)
3. Guarantees No Out-of-Order Messages - messages will be displayed in the order in which they are sent
4. Congestion Control (sometimes) - limits the number of messages that can be sent in a batch

TCP is slower because it requires messages to be acknowledged, effectively doubling (at minimum) the time of UDP.

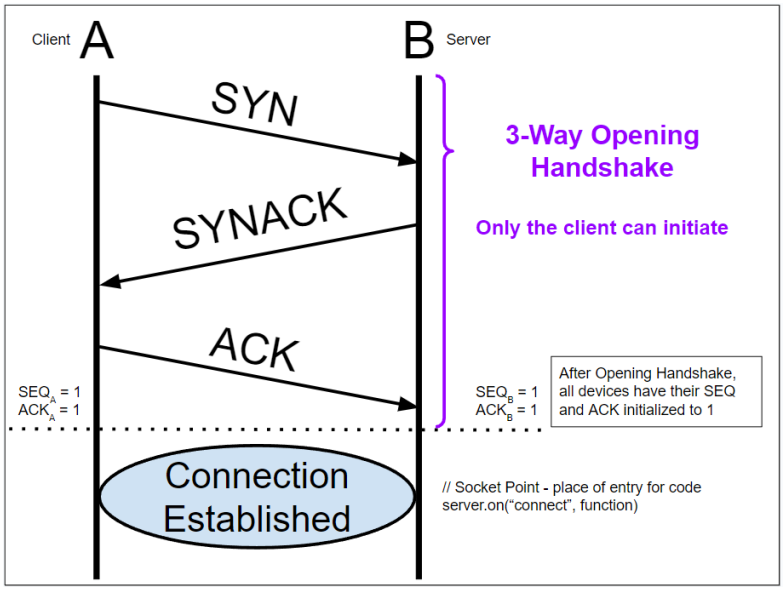
5.1 TCP Header

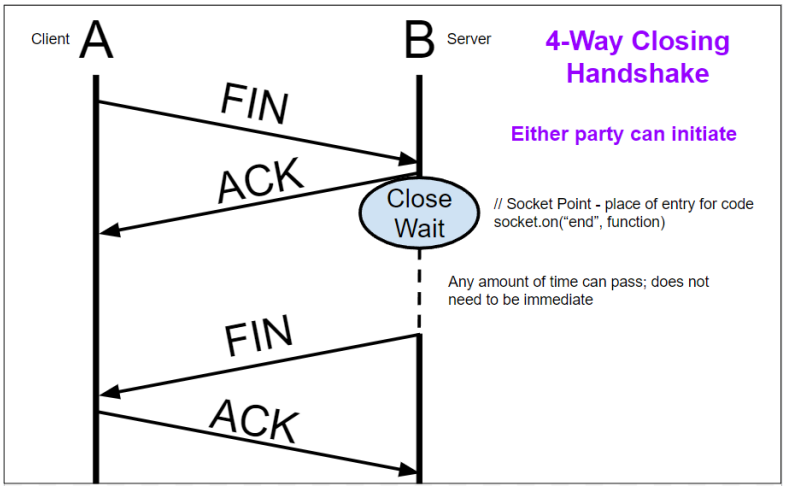
1. Sequence Number: tracks the order of messages to guarantee 2. and 3.
2. Acknowledgment Number: number of bytes received by the other side in the correct order with no errors
3. Flags - dictates the type of TCP message
4. Window Size - how many unacknowledged messages can be sent at a time before you must wait for an acknowledgment

5.2 Flags

- ACK - Acknowledge - confirm that a message was received
- SYN - Synchronize - request to communicate
- FIN - Finish - close the connection "gracefully"
- RST - Reset - something went wrong, reset connection
- PSH - Push - data is ready to be consumed

5.3 Handshakes





5.4 Example of Sequence Diagram

SEQ# - initialized to 1 after 3-Way Handshake

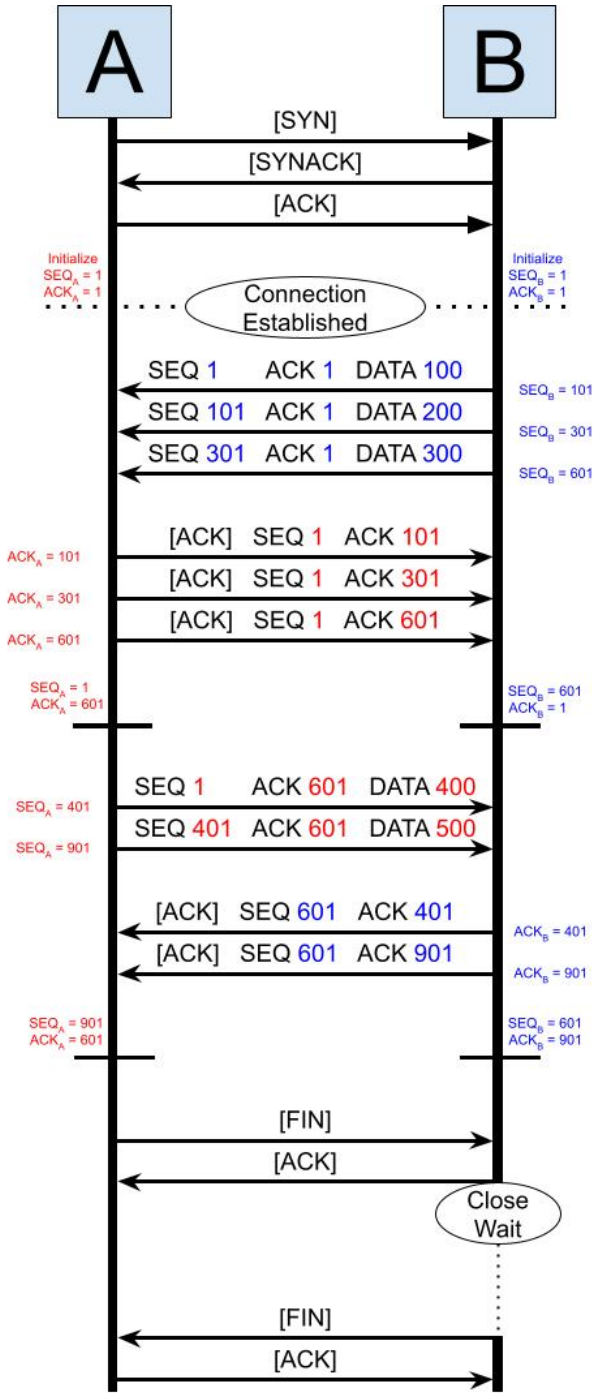
- Next SEQ# = Prev SEQ # + Prev Message Data Length

ACK# - initialized to 1 after 3-Way Handshake - number of bytes successfully received in the correct order + 1 OR next expected SEQ#

- Next ACK# = Received SEQ# + Message Data Length

Scenario: Alice and Bob communicate with TCP. Alice initiates the connection. Upon connecting, Bob immediately sends three messages of sizes 100 bytes, 200 bytes, 300 bytes. No data is lost. Upon receiving all messages, Alice sends two messages of sizes 400 bytes and 500 bytes. Upon receiving all messages, Bob closes the connection.

Note: If there was any data loss, retransmission occurs until all data is received. Tracing the SEQ# and ACK# remains the same


A
B
[SYN]
[SYNACK]
[ACK]
Initialize
SEQA = 1
ACKA = 1
Initialize
SEQB = 1
ACKB = 1
Connection Established
SEQ 1 ACK 1 DATA 100
SEQB = 101
SEQ 101 ACK 1 DATA 200
SEQB = 301
SEQ 301 ACK 1 DATA 300
SEQB = 601
[ACK] SEQ 1 ACK 101
ACKA = 101
[ACK] SEQ 1 ACK 301
ACKA = 301
[ACK] SEQ 1 ACK 601
ACKA = 601
SEQA = 1
ACKA = 601
SEQB = 601
ACKB = 1
SEQ 1 ACK 601 DATA 400
SEQA = 401
SEQ 401 ACK 601 DATA 500
SEQA = 901
[ACK] SEQ 601 ACK 401
ACKB = 401
[ACK] SEQ 601 ACK 901
ACKB = 901
SEQA = 901
ACKA = 601
SEQB = 601
ACKB = 901
[FIN]
[ACK]
Close Wait
[FIN]
[ACK]

CS 355 Internet and Web Technologies - Topic 11

Nicholas Farkash

Saturday, November 16 & 23, 2024

1 Streams and Sockets

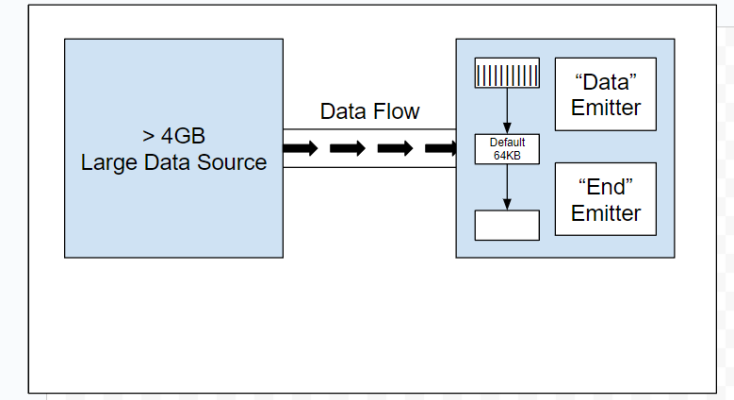
Streams (such as Java's BufferReader class) are an abstract data type. They allow for the movement of data from a source to a destination.

Stream Type	Source Data
File Stream	File
IO Stream	User Input
Socket Stream	Internet

Table 1: Types of File Streams and their Source Data

Buffer - a raw group of memory (64KB by default) **Chunk** - a full buffer
Streams have two components:

1. An "infinite linked list" of Buffers
2. Emitters (for each event, such as "Data" or "End")



readStream.on("Data") will connect the data source to the buffer and enable "flow mode", turning on the flow of data from source to buffer

When a buffer fills up and becomes a chunk, the "Data" emitter emits a "Data" event with a copy of the chunk - **readStream.DataEmitter.emit("data", chunk)**. The buffer linked list removes the head node, adds an empty node to the end, and increments the head node (it extends the linked list and gets rid of the old data)

When a stream detects the end of its source data, **readStream.EndEmitter.emit("end")** is called.

Buffers are sent out when the DataEmitter emits a "data" signal, which happens when the buffer fills and becomes a chunk. However, a buffer can also be sent out if it is not full. For example, an IO Stream will send out its buffer when a "newline" event is emitted. "PSH" has the same effect for Socket Streams.

readStream.pipe(writeStream) will move data from readStream to writeStream directly

Terminal IO:

- standard input = *process.stdin*
- standard output = *process.stdout*

Streams are easily interchangeable - you can replace one read/write stream with another with little (if any) change to code.

1.1 Summary of Streams

There are two types of streams: Read and Write

1. ReadStreams
    - "Data" Event - signals full buffer
    - "End" Event - data source depleted
    - pipe(ws) - method to redirect content to the WriteStream ws
2. WriteStreams
    - .write(chunk) - write the data in the chunk
    - .end() - signal that writing is finished
    - "Finish" Event - signals when WriteStream is closed

2 Sockets

A **socket** is a duplex stream, inheriting the properties of both the read and write streams

- The "net" package (const net = require("net") provides support for TCP servers
- server = net.createServer() - create a server
- server.on("connection", function()) - socket point after 3-Way Opening Handshake is established - when a connection is established with a new client
- server.listen(port) - if OS gets a message tagged with port number "port", send it to the server
- socket = net.createConnection(host, port) - establishes a client's connection with the host on the port
- socket.on("connect", function()) - socket point after 3-Way Opening Handshake is established - when connection is established with the server

It is good for the server to keep an Array of sockets - this allows clients to communicate with one another, as well as make it easy for the server to keep track of its clients.

2.1 Disconnection

A server and client can disconnect when either one:

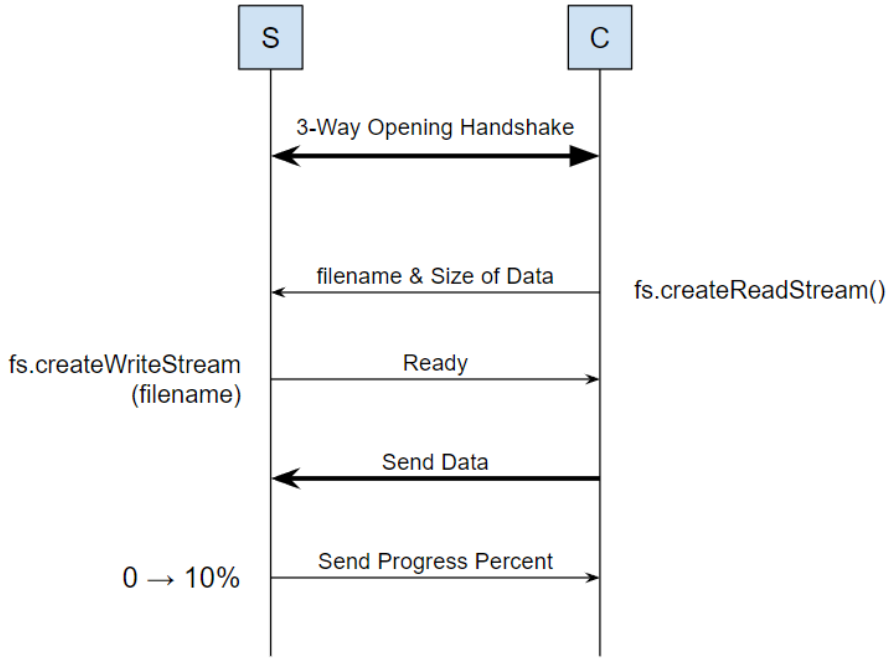
1. "X"'s out of the terminal (SIGHUP signal emitted)
2. CTRL + C's (SIGINT signal emitted)

A server can call socket.destroy() to kill its connection with the socket - "don't bother sending me an ACK FIN - I know I won't get one back because you're not there, you already left" (error occurred)

A client can call process.exit() or socket.destroy() to kill its connection with the server for the same reason.

2.2 File Uploader

See socket-demos exercise 9



CS 355 Internet and Web Technologies - Topic 11

Nicholas Farkash

Saturday, December 7, 2024

1 HTTP Protocol

HTTP is the layer on top of TCP, responsible for exchanging hypertext across the web. It operates on a client-server model:

- A HTTP client makes requests to a server, waits for the server's response, and processes the server's response.
- A HTTP server waits for a request to arrive from a client, processes the request, and returns a response.

1.1 Requests and Responses

An HTTP request is a piece of structured text that provides information. At the bare minimum, this is a URL to a desired resource, but most often also includes additional data that customizes the request. Requests can be seen by opening a browser, right-clicking and selecting "Inspect", and then navigating to the "Network" tab. There will be a bunch of requests made, and clicking on one and scrolling to "Request Headers" will show that additional request information.

Likewise, an HTTP response is very similar, but the contents of the message vary. It includes an HTTP status code (see below), header information, and sometimes a file depending on what the request was that is being responded to. After the last line of the header, the content being delivered is located.

1.2 Status Codes

Status codes are used to describe the response being returned. These status codes fall into 4 categories:

- 200s - Success (200 - OK)
- 300s - Redirection (304 - Not Modified)
- 400s - Client Errors (404 - Not Found)
- 500s - Server Errors (500 - Internal Server Error)

1.3 Browser

A web browser is client software that automates HTTP requests and responses. It abstracts away handling requests and responses, displaying to the user the contents of the message (payload) rather than the entire message (header and payload). Browers make it so the only thing a user needs to provide for an HTTP request is a URL.

Browsers start parsing HTML code as soon as it receives the HTML document, but that HTML document often has embedded resources, like images and videos. The browser will make a **secondary request** for these resources after it has loaded in the HTML code. This is shown in the diagram at `https://venus.cs.qc.cuny.edu/~rlaw/cs355/lectures/13-hypertext-transfer-protocol/hypertext-transfer-protocol.html`. As a web page is being rendered, secondary requests are sent out.

2 Code

2.1 01-http-servers

After requiring "const http = require("http");", access to the http library is granted.

server = http.createServer() creates an HTTP server.
server.listen(port) has the server listen on the specified port
When the server emits a "request" event, two objects are produced:

- **req** - A mock readStream - The incoming request. *req.socket.remoteAddress* will provide the address of the entity sending the request. *req.url* will provide the url being requested.
- **res** - A mock writeStream - The response back to the requesting socket (the website). The main communications for a response are:
  - **res.writeHead**(*status_code, status_message, headers_object*); - used to write the HTTP Headers part of the HTTP message.
  - **res.write**(*content*); - Writes the payload content of the HTTP message (the HTML code, for example).
  - **res.end**(); - sends out the response back to whoever made the request.

2.2 02-routing

Routing to different pages can be accomplished with if-else blocks. The requested URL (req.url) can be matched to strings, either exactly or comparing the beginning of the req.url. Different contents can be displayed depending on what the requested URL is. For example, if (req.url === "/") checks for the url (/index.HTML), so we can *res.write*() HTML text specifically for the root

page. if (req.url === "/page2") checks for a url named "index.html/page2", so we can *res.write*() different HTML text for that page. The "else" clase at the end is used for error catching. If the request is for a URL that we don't have (or maybe we don't want to give the requester access to it), we can *res.writeHead*() a 404 Not Found header and display HTML text as well.

2.3 03-piping

If an embedded resource, such as an image, is used in the HTML code, the browser will perform a secondary request for this resource. When it does this, it will make a HTTP request for that resource (req.url === "/cat.jpg"). To facilitate this, in our code, we will *res.writeHead()* to create a header with 200 status code and a header option for the content type (image/jpeg, don't worry about the specifics like that). As for the payload of the HTTP message, we want to send transfer the cat.jpg file, so we create an *fs.createReadStream("path_to_resource")* and pipe that content to res (remember, readStream.pipe(writeStream), which will serve as the content being sent over. Piping takes care of the *res.end()*, as it sends off the data. Note that you can also directly pipe an entire HTML file, rather than writing it out as a function parameter in *res.end()*.

2.4 04-forms-get

There are two main types of HTTP request methods. The first one is GET, which is used to retrieve data. The data being retrieved is stored in the URL, as it is okay for the user to see because the user is getting this data anyways. The URL can then be parsed using "const user_input = url.parse(req.url).query", which stores the information in an object to be extracted. For example, you can extract information from an HTML form by parsing the url. The information from each form entry is stored as a value in an object, with its key being the name of the input field in the HTML file (¡input name="dob" will have its contents stored in user_input.dob).

2.5 05-forms-post

The other HTTP request method is POST, which is used to provide data, like entering a password or uploading a file. Since we don't want this information visible to others, it is not stored in the URL. Rather, the content is retrieved via the "req.on('data')" method, loading in chunks (user_input_body += chunk). When all data has finished transferring, the information is parsed according to "user_input = querystring.parse(user_input_body); From there, the same method of parsing the object as above is used.

CS 355 Internet and Web Technologies - Topic 13

Nicholas Farkash

Saturday, December 7, 2024

1 Third Party APIs

API - Application Programming Interface - a software connection between computers. Programmers can write code (HTTP requests) to interact with services.

2 Code

2.1 01-dictionary

If the API does not require any authentication, such as the Dictionary API `https://dictionaryapi.dev/`, communicating with the API is very easy. First, it is important that you understand the format of the API. You need to know what the URL is that you need to request data from, as well as how to format the request. This can be found by consulting the API's documentation.

Once you know how to format a request, use the format:

const api = https.request("url");

where "url" is a string that the API accepts as a request URL (follows the API's expected format). For the dictionary API, this is:

const dictionary_api = https.request(`https://api.dictionaryapi.dev/api/v2/entries/en_US/$word`)

where 'word' is the word whose definition we want to get in the API's response.

Once the API responds (api.on("response")), we want to take the data returned in the response and process it. Generally, this will involve buffers/chunks processing to extract the data. Once this is done, you can parse the data into a JSON object to extract the information you want. The Dictionary API returns a lot of information, which is all stored in the JSON object. As such, we need to know how the JSON is formatted in order to extract the specific information we desire (see the APIs documentation or print the entire thing and see the path you need to extract).

2.2 02-usajobs.gov

Most APIs will require some form of authentication. The USAJobs API is an example of this. It uses "API-Key" authentication. This means that you must

make and verify an account in order to receive an API Key. Then, if you want to access the API's contents through code, you must provide your key, validating to the API's server WHO is accessing it.

This can be done by creating a JSON object to store any information needed for authentication. The API will provide information on how to send over your API key, but generally this information will be sent in the HTTP message's headers (). See line 37 for how the HTTPS request is made exactly, but to generically explain it, you execute the line:

https.get(endpoint, headers);

where "endpoint" is the API's endpoint to connect to and "headers" is an object that stores HTML header information (method:"GET", headers=credentials, etc). When the API responds (.once('response') in this case), it will return the data just like above. Perform buffers/chunks processing to extract the data, parse it into a JSON object, extract the desired information, and do whatever you need to do with the data.

2.3 03-dictionary x jobs

This example combines the Dictionary API and USAJobs API. You fill out the same form as before (job description and location), but before the results are shown, the definition of the job description is presented for a few seconds first. Tracing the code yields the following procedure:

1. The server is set up as usual. The request handler is also essentially the same. When the user hits "Search", the information is extracted from the HTML to JS (in the /search request handler if-else block). get_information(*description, location, res*) is called. This *res* is the response to the browser, which will be used to dynamically change pages and such down the line.

2. In get_information(*description, location, res*), the endpoint to each API is established and a request is made to each API. The USAJobs API gets a request from the browser saying "Here's my credentials. Will you please give me access to your API's entries for this job description??? :3 ¡3.", while the Dictionary API gets a request from the browser saying "Give me the definition of the Job Description!". When a request is fulfilled, the resulting stream (readStream of data chunks) is passed to the process_stream() function, which performs buffers/chunks processing and then the specified callback function when it finishes (parsing the data for each).

3. When the Dictionary API responds to the request, it provides a data stream that, when processed, returns the API's results for that word. After being processed, parse_dictionary() is called, which extracts the desired information (the definition and such), performs some CSS styling, and displays it to the browser using *res.write(results)*. After the results are displayed, the terminate() function is called, which increments the number

of tasks completed and checks to see if all tasks are done. If so, the browser's response terminates.

4. When the USAJobs API responds to the request, it provides a data stream that, when processed, returns the API's results for that job description. After being processed, parse_usajobs() is called, which extracts the desired information, stores it in an array, performs some CSS styling, and displays it to the browser using *res.write(results)*. After the results are displayed, the terminate() function is called, which increments the number of tasks completed and checks to see if all tasks are done. If so, the browser's response terminates.

The authentication is done at the same time the request is made for matching job descriptions. It is not a separate step when doing API-Key authentication. As a result, the user hits the "Submit" button, sees the Job Description's word and definition on the screen while the USAJob's API is authenticating and getting its information, and then sees the job listings.

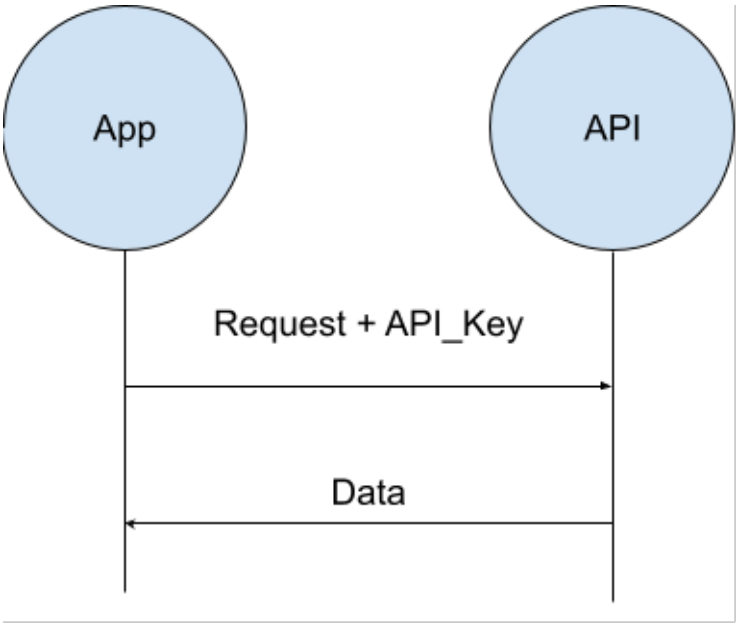
CS 355 Internet and Web Technologies - Topic 14

Nicholas Farkash

Saturday, December 14, 2024

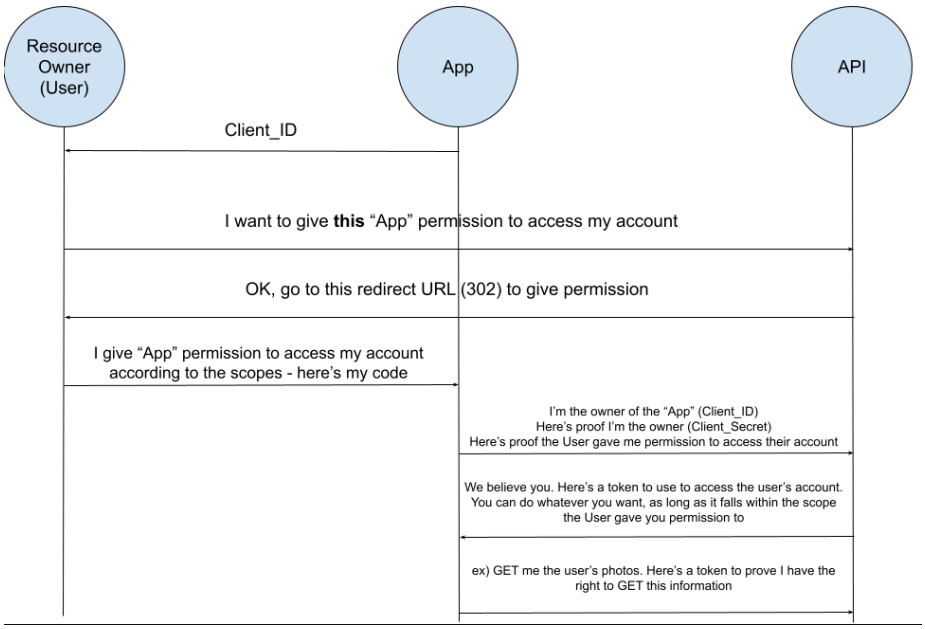
1 API Key

An API Key Authorization is a simple authorization that requires an API Key, essentially a username/password. It is used when the API owns all of the data on their platform. For example, USAJobs owns all of the information you can get from them, so all your app needs to do is provide an API key to say "Hey, I'm an authorized app you've given permission to before. I want to get something." and the API will say "Let me check real quick... yep we have you as a registered user in our records. Come on in and take whatever you want." It is as simple as the app sending a request to the API with its API key and receiving the requested data.



2 OAuth 2 - 3-Legged Authorization

3-Legged Authorization is used when the API doesn't own the data it contains. For example, Instagram does not own the pictures on its platform - the users who post them own the pictures. As such, if we want our app to utilize the Instagram API to download pictures on Instagram, we need permission from the users who uploaded those pictures. This means we need to get 3 parties in agreement - our app, the API, and the user, hence 3-Legged (3LO). Below is an example sequence diagram of what a 3LO Authorization would look like in this example.



This diagram essentially flows like this:

1. App to User: Hey! I'm an application. Here's my *Client_ID*. Can you take that, go to this website, and give them that ID? Then, tell them that you are giving me permission to do the following things: *Scope*
2. User to API: Hey API, I want to to give this App *Client_ID* permission to do the following things on my behalf: *Scope*
3. API to User: OK! I'm gonna redirect you to this website. You can give permission there. That will give your *Code* to the App, meaning that you have given the App permission to do the things you allowed them to do.
4. User to App: Hey, here's my *Code*. You're free to do whatever I gave you permission to do on my behalf.

5. App to API: Hey API! I'm the owner of app *Client_ID*. Here's proof that I'm the owner: *Client_Secret*. Here's proof that the User gave me permission to act on their behalf, provided that it falls within the *Scope* they agreed to: *Code*.

6. API to App: Yep, that all checks out. Okay, here's an *Access_Token* that's valid for a certain amount of time. You can do whatever you want, provided it falls within the agreed *Scope*, while that *Access_Token* is still valid. If it runs out of time, just request another one.

7. App to API: Great! I want to get the User's pictures, an action which falls within the *Scope* the User agreed to. Here's the valid *Access_Token* that you gave me, which shows I have permission to access the User's data.

8. *Not shown is the returned data from the API to the App.*