Fundamentals of Networking for Effective Backend Applications

Understanding the first principles of networking to build low latency and high throughput backends

Introduction

Introduction

- Welcome
- Who this course is for?
- Course Outline

Fundamentals of Networking

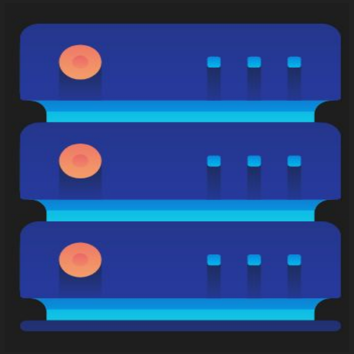
The first principles of computer networking

Client-Server Architecture

A revolution in networking

Client-Server Architecture

- Machines are expensive, applications are complex
- Seperate the application into two components
- Expensive workload can be done on the server
- Clients call servers to perform expensive tasks
- Remote procedure call (RPC) was born



Client-Server Architecture Benefits

- Servers have beefy hardware
- Clients have commodity hardware
- Clients can still perform lightweight tasks
- Clients no longer require dependencies
- However, we need a communication model

OSI Model

Open Systems Interconnection model

Why do we need a communication model?

- Agnostic applications
  - Without a standard model, your application must have knowledge of the underlying network medium
  - Imagine if you have to author different version of your apps so that it works on wifi vs ethernet vs LTE vs fiber
- Network Equipment Management
  - Without a standard model, upgrading network equipments becomes difficult
- Decoupled Innovation
  - Innovations can be done in each layer separately without affecting the rest of the models

What is the OSI Model?

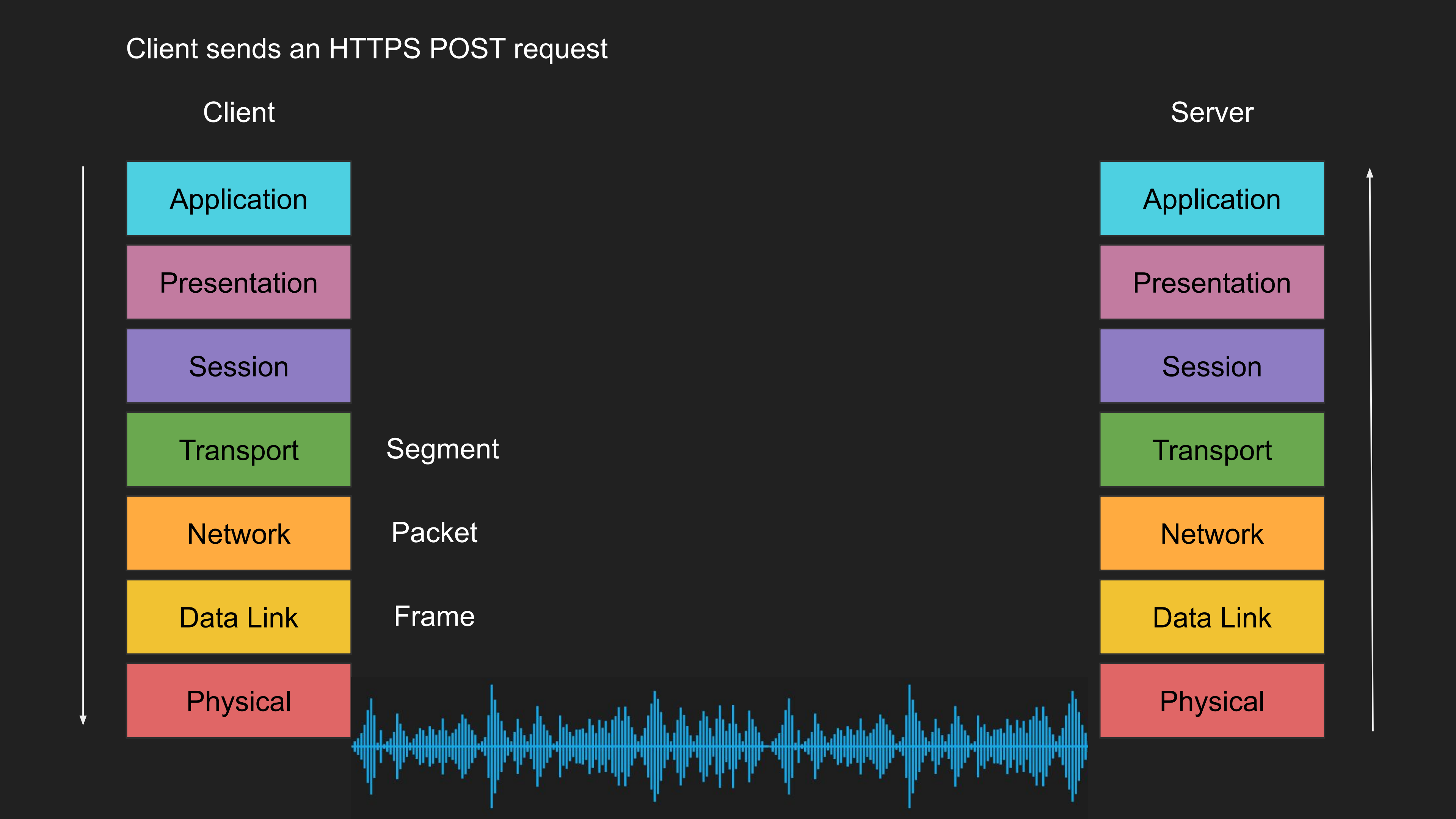
- 7 Layers each describe a specific networking component
- Layer 7 - Application - HTTP/FTP/gRPC
- Layer 6 - Presentation - Encoding, Serialization
- Layer 5 - Session - Connection establishment, TLS
- Layer 4 - Transport - UDP/TCP
- Layer 3 - Network - IP
- Layer 2 - Data link - Frames, Mac address Ethernet
- Layer 1 - Physical - Electric signals, fiber or radio waves

The OSI Layers - an Example (Sender)

- Example sending a POST request to an HTTPS webpage
- Layer 7 - Application
  - POST request with JSON data to HTTPS server
- Layer 6 - Presentation
  - Serialize JSON to flat byte strings
- Layer 5 - Session
  - Request to establish TCP connection/TLS
- Layer 4 - Transport
  - Sends SYN request target port 443
- Layer 3 - Network
  - SYN is placed an IP packet(s) and adds the source/dest IPs
- Layer 2 - Data link
  - Each packet goes into a single frame and adds the source/dest MAC addresses
- Layer 1 - Physical
  - Each frame becomes string of bits which converted into either a radio signal (wifi), electric signal (ethernet), or light (fiber)
- Take it with a grain of salt, it's not always cut and dry

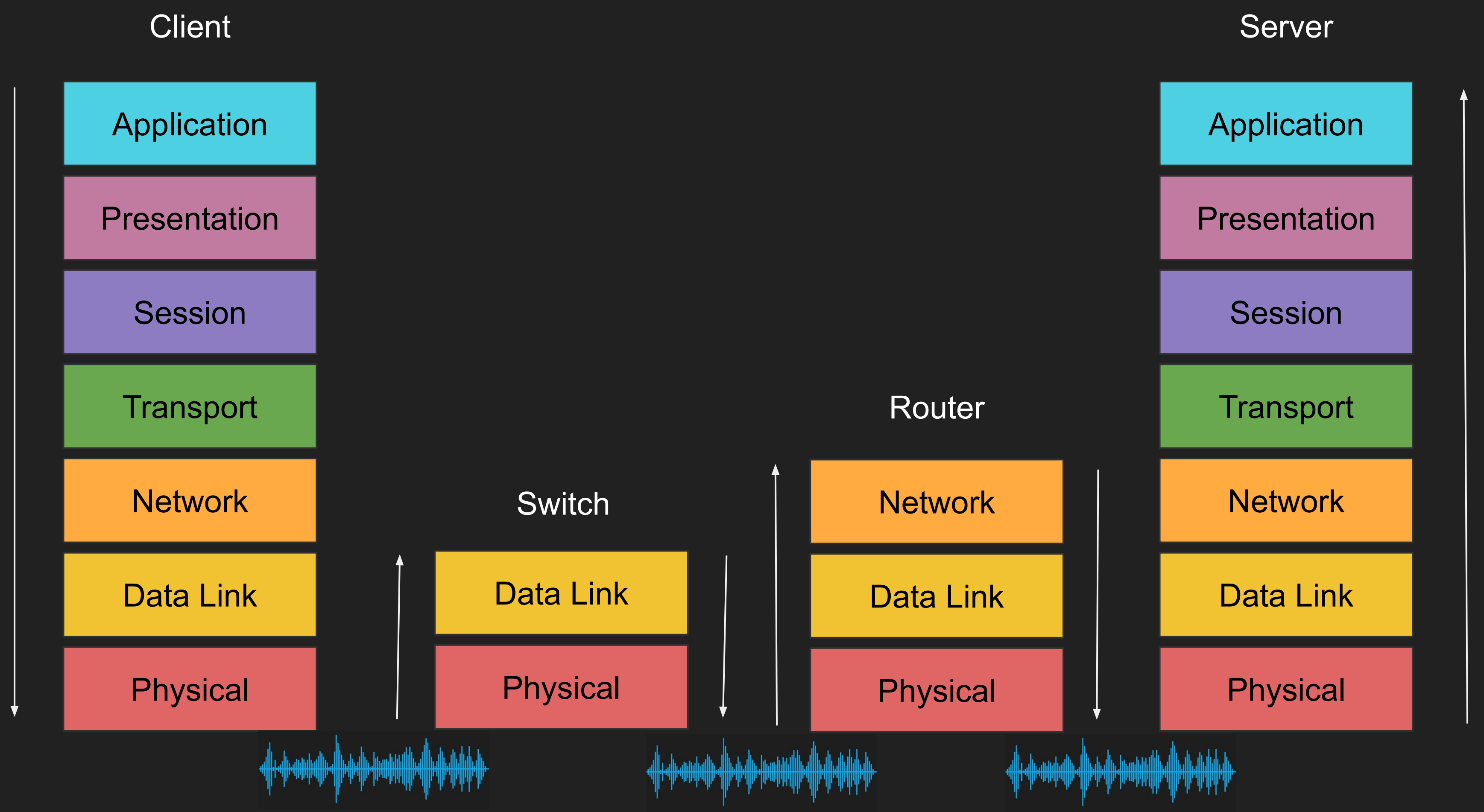
The OSI Layers - an Example (Receiver)

- Receiver computer receives the POST request the other way around
- Layer 1 - Physical
  - Radio, electric or light is received and converted into digital bits
- Layer 2 - Data link
  - The bits from Layer 1 is assembled into frames
- Layer 3 - Network
  - The frames from layer 2 are assembled into IP packet.
- Layer 4 - Transport
  - The IP packets from layer 3 are assembled into TCP segments
  - Deals with Congestion control/flow control/retransmission in case of TCP
  - If Segment is SYN we don't need to go further into more layers as we are still processing the connection request
- Layer 5 - Session
  - The connection session is established or identified
  - We only arrive at this layer when necessary (three way handshake is done)
- Layer 6 - Presentation
  - Deserialize flat byte strings back to JSON for the app to consume
- Layer 7 - Application
  - Application understands the JSON POST request and your express json or apache request receive event is triggered
- Take it with a grain of salt, it's not always cut and dry

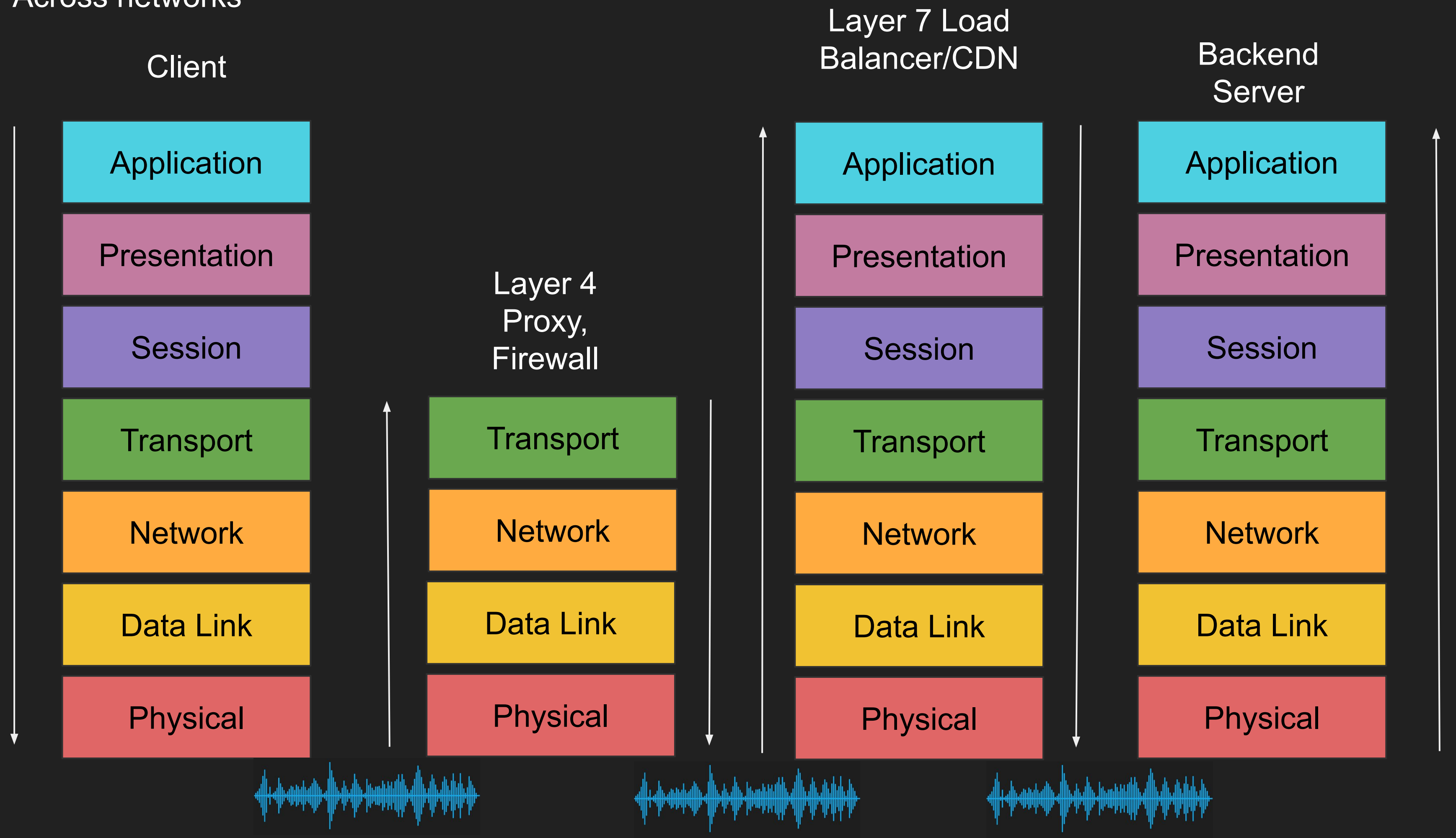

Client sends an HTTPS POST request
Client
Server
Application
Presentation
Session
Transport
Segment
Network
Packet
Data Link
Frame
Physical
Application
Presentation
Session
Transport
Network
Data Link
Physical

Client
Server
SPORT
DPORT
SIP
DIP
SMAC
DMAC
SPORT
DPORT
SIP
DIP
SMAC
DMAC

Across networks



Across networks



The shortcomings of the OSI Model

- OSI Model has too many layers which can be hard to comprehend
- Hard to argue about which layer does what
- Simpler to deal with Layers 5-6-7 as just one layer, application
- TCP/IP Model does just that

TCP/IP Model

- Much simpler than OSI just 4 layers
- Application (Layer 5, 6 and 7)
- Transport (Layer 4)
- Internet (Layer 3)
- Data link (Layer 2)
- Physical layer is not officially covered in the model

OSI Model Summary

- Why do we need a communication model?
- What is the OSI Model?
- Example
- Each device in the network doesn’t have to map the entire 7 layers
- TCP/IP is simpler model

Host to Host communication

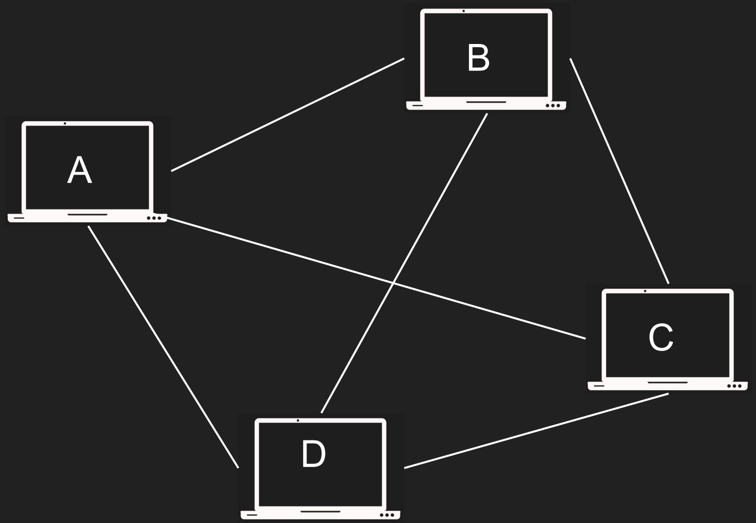
How messages are sent between hosts

Host to Host communication

- I need to send a message from host A to host B
- Usually a request to do something on host B (RPC)
- Each host network card has a unique Media Access Control address (MAC)
- E.g. 00:00:5e:00:53:af

Host to Host communication

- A sends a message to B specifying the MAC address
- Everyone in the network will “get” the message but only B will accept it



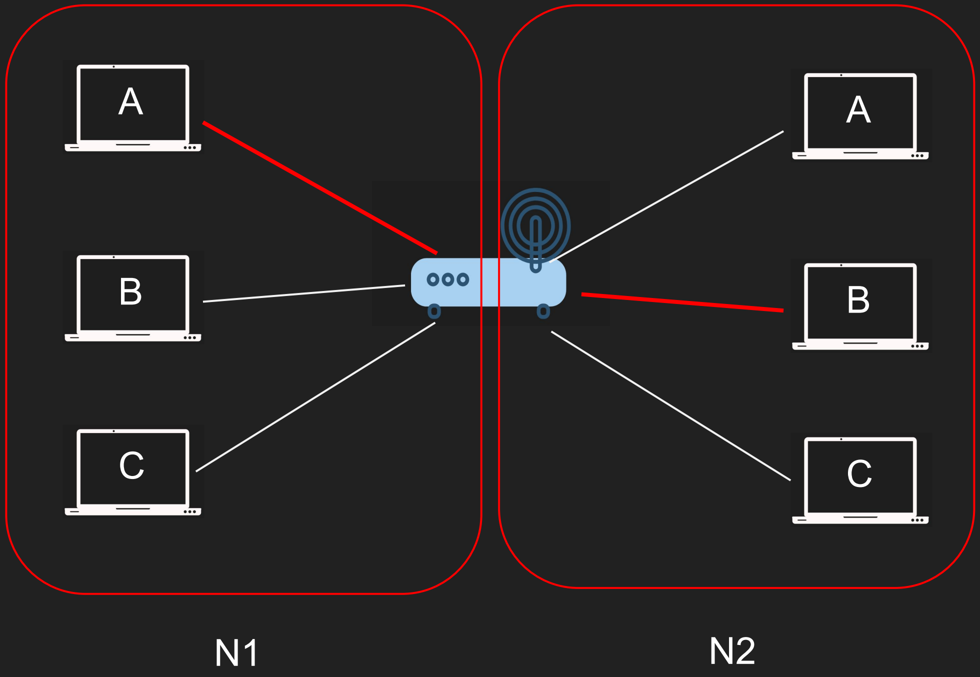
Host to Host communication

- Imagine millions of machines?
- We need a way to eliminate the need to send it to everyone
- The address needs to get better
- We need routability, meet the IP Address

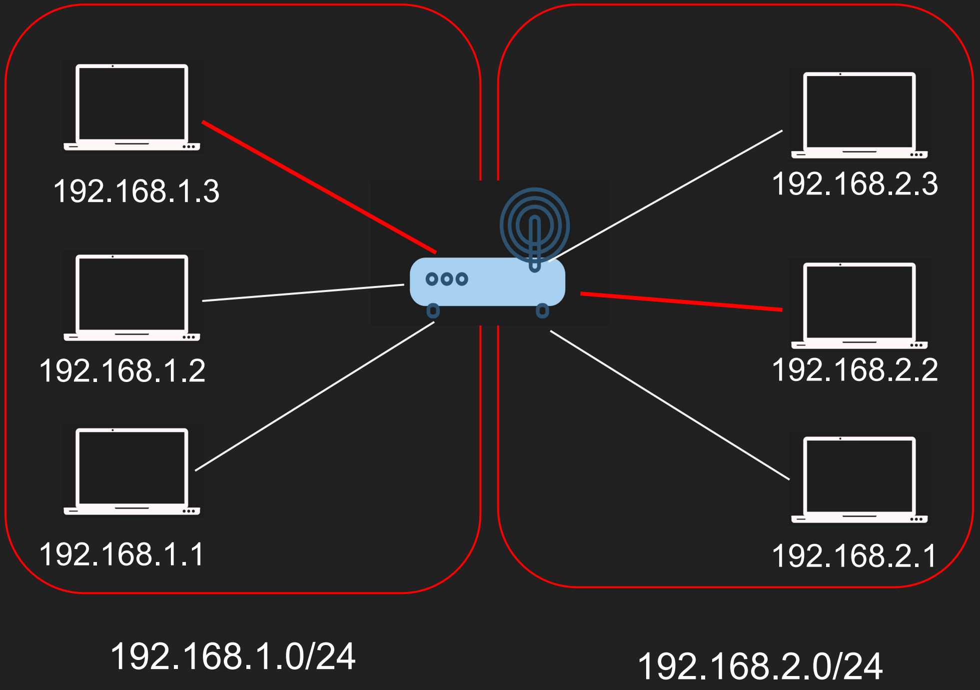
Host to Host communication

- The IP Address is built in two parts
- One part to identify the network, the other is the host
- We use the network portion to eliminate many networks
- The host part is used to find the host
- Still needs MAC addresses!

Host A on network N1 wants to talk to Host B on network N2



Host 192.168.1.3 wants to talk to 192.168.2.2



But my host have many apps!

- It's not enough just to address the host
- The host is runnings many apps each with different requirements
- Meet ports
- You can send an HTTP request on port 80, a DNS request on port 53 and an SSH request on port 22 all running on the same server!

Host to Host communication - Summary

- Host needs addresses
- MAC Addresses are great but not scalable in the Internet
- Internet Protocol Address solves this by routing
- Layer 4 ports help create finer addressability to the process level

1.2.3.4

The IP building blocks

Understanding the IP Protocol

IP Address

- Layer 3 property
- Can be set automatically or statically
- Network and Host portion
- 4 bytes in IPv4 - 32 bits

Network vs Host

- a.b.c.d/x (a.b.c.d are integers) x is the network bits and remains are host
- Example 192.168.254.0/24
- The first 24 bits (3 bytes) are network the rest 8 are for host
- This means we can have 2^24 (16777216) networks and each network has 2^8 (255) hosts
- Also called a subnet

Subnet Mask

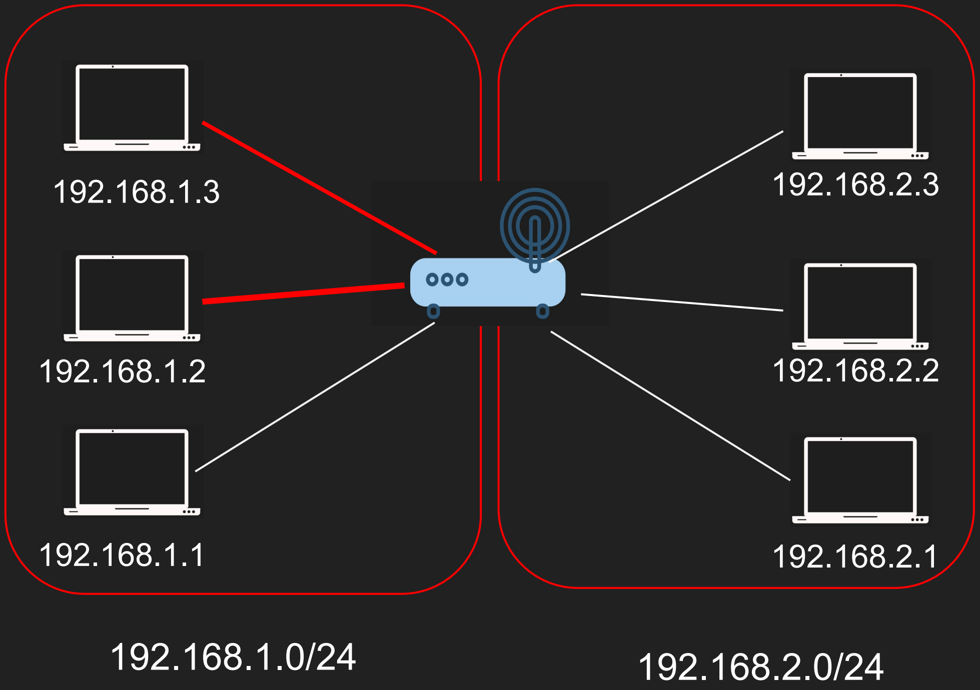
- 192.168.254.0/24 is also called a subnet
- The subnet has a mask 255.255.255.0
- Subnet mask is used to determine whether an IP is in the same subnet

Default Gateway

- Most networks consists of hosts and a Default Gateway
- Host A can talk to B directly if both are in the same subnet
- Otherwise A sends it to someone who might know, the gateway
- The Gateway has an IP Address and each host should know its gateway

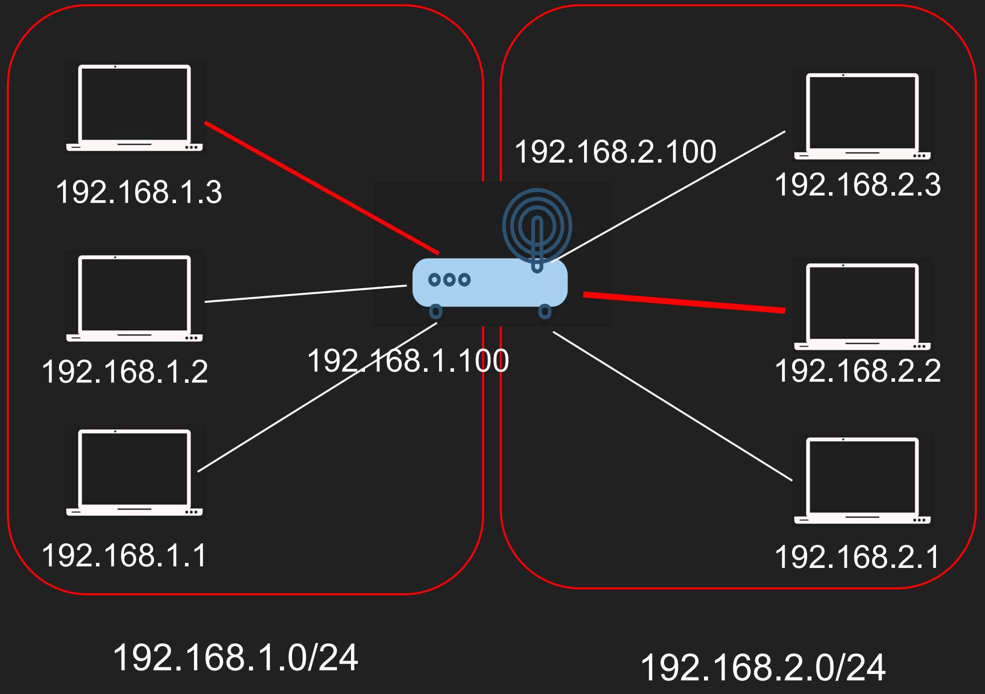
E.g. Host 192.168.1.3 wants to talk to 192.168.1.2

- 192.168.1.3 applies subnet mask to itself and the destination IP 192.168.1.2
- 255.255.255.0 & 192.168.1.3 = 192.168.1.0
- 255.255.255.0 & 192.168.1.2 = 192.168.1.0
- Same subnet ! no need to route



E.g. Host 192.168.1.3 wants to talk to 192.168.2.2

- 192.168.1.3 applies subnet mask to itself and the destination IP 192.168.2.2
- 255.255.255.0 & 192.168.1.3 = 192.168.1.0
- 255.255.255.0 & 192.168.2.2 = 192.168.2.0
- Not the subnet ! The packet is sent to the Default Gateway 192.168.1.100



Summary

- IP Address
- Network vs Host
- Subnet and subnet mask
- Default Gateway

The IP Packet

Anatomy of the IP Packet

IP Packet

- The IP Packet has headers and data sections
- IP Packet header is 20 bytes (can go up to 60 bytes if options are enabled)
- Data section can go up to 65536

IP Packet to the Backend Engineer

Actual IP Packet

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><th>0</th><th>0</th><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><th>4</th><th>32</th><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><th>8</th><th>64</th><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><th>12</th><th>96</th><td colspan="32">Source IP Address</td></tr>
<tr><th>16</th><th>128</th><td colspan="32">Destination IP Address</td></tr>
<tr><th>20</th><th>160</th><td colspan="32" rowspan="3">Options (if IHL > 5)</td></tr>
<tr><th>⋮</th><th>⋮</th></tr>
<tr><th>56</th><th>448</th></tr>
<tr><td></td><td></td><td colspan="32">Data</td></tr>
</table>

https://datatracker.ietf.org/doc/html/rfc791
https://en.wikipedia.org/wiki/IPv4

Version - The Protocol version

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>4</td><td>32</td><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>8</td><td>64</td><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>12</td><td>96</td><td colspan="32">Source IP Address</td></tr>
<tr><td>16</td><td>128</td><td colspan="32">Destination IP Address</td></tr>
<tr><td>20</td><td>160</td><td colspan="32" rowspan="3">Options (if IHL > 5)</td></tr>
<tr><td>⋮</td><td>⋮</td></tr>
<tr><td>56</td><td>448</td></tr>
<tr><td></td><td></td><td colspan="32">Data</td></tr>
</table>

Internet Header Length - Defines the Options length

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>4</td><td>32</td><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>8</td><td>64</td><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>12</td><td>96</td><td colspan="32">Source IP Address</td></tr>
<tr><td>16</td><td>128</td><td colspan="32">Destination IP Address</td></tr>
<tr><td>20</td><td>160</td><td colspan="32" rowspan="3">Options (if IHL > 5)</td></tr>
<tr><td>⋮</td><td>⋮</td></tr>
<tr><td>56</td><td>448</td></tr>
<tr><td></td><td></td><td colspan="32">Data</td></tr>
</table>

Total Length - 16 bit Data + header

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>4</td><td>32</td><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>8</td><td>64</td><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>12</td><td>96</td><td colspan="32">Source IP Address</td></tr>
<tr><td>16</td><td>128</td><td colspan="32">Destination IP Address</td></tr>
<tr><td>20</td><td>160</td><td colspan="32" rowspan="3">Options (if IHL > 5)</td></tr>
<tr><td>⋮</td><td>⋮</td></tr>
<tr><td>56</td><td>448</td></tr>
<tr><td></td><td></td><td colspan="32">Data</td></tr>
</table>

Fragmentation - Jumbo packets

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>4</td><td>32</td><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>8</td><td>64</td><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>12</td><td>96</td><td colspan="32">Source IP Address</td></tr>
<tr><td>16</td><td>128</td><td colspan="32">Destination IP Address</td></tr>
<tr><td>20</td><td>160</td><td colspan="32" rowspan="3">Options (if IHL > 5)</td></tr>
<tr><td>⋮</td><td>⋮</td></tr>
<tr><td>56</td><td>448</td></tr>
<tr><td></td><td></td><td colspan="32">Data</td></tr>
</table>

Time To Live - How many hops can this packet survive?

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>4</td><td>32</td><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>8</td><td>64</td><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>12</td><td>96</td><td colspan="32">Source IP Address</td></tr>
<tr><td>16</td><td>128</td><td colspan="32">Destination IP Address</td></tr>
<tr><td>20</td><td>160</td><td colspan="32" rowspan="3">Options (if IHL > 5)</td></tr>
<tr><td>⋮</td><td>⋮</td></tr>
<tr><td>56</td><td>448</td></tr>
<tr><td></td><td></td><td colspan="32">Data</td></tr>
</table>

Protocol - What protocol is inside the data section?

<table>
<tr><td>Offsets</td><td>Octet</td><td colspan="8">0</td><td colspan="8">1</td><td colspan="8">2</td><td colspan="8">3</td></tr>
<tr><td>Octet</td><td>Bit</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td>0</td><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>4</td><td>32</td><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>8</td><td>64</td><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>12</td><td>96</td><td colspan="32">Source IP Address</td></tr>
<tr><td>16</td><td>128</td><td colspan="32">Destination IP Address</td></tr>
<tr><td>20</td><td>160</td><td colspan="32" rowspan="3">Options (if IHL > 5)</td></tr>
<tr><td>⋮</td><td>⋮</td></tr>
<tr><td>56</td><td>448</td></tr>
<tr><td></td><td></td><td colspan="32">Data</td></tr>
</table>

Source and Destination IP

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><th>0</th><th>0</th><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><th>4</th><th>32</th><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><th>8</th><th>64</th><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><th>12</th><th>96</th><td colspan="32">Source IP Address</td></tr>
<tr><th>16</th><th>128</th><td colspan="32">Destination IP Address</td></tr>
<tr><th>20</th><th>160</th><td colspan="32" rowspan="3">Options (if IHL > 5)</td></tr>
<tr><th>⋮</th><th>⋮</th></tr>
<tr><th>56</th><th>448</th></tr>
</table>

Data

Explicit Congestion Notification

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td>0</td><td>0</td><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="6">DSCP</td><td colspan="2">ECN</td><td colspan="16">Total Length</td></tr>
<tr><td>4</td><td>32</td><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td>8</td><td>64</td><td colspan="8">Time To Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td>12</td><td>96</td><td colspan="32">Source IP Address</td></tr>
<tr><td>16</td><td>128</td><td colspan="32">Destination IP Address</td></tr>
<tr><td>20</td><td>160</td><td colspan="32" rowspan="3">Options (if IHL > 5)</td></tr>
<tr><td>⋮</td><td>⋮</td></tr>
<tr><td>56</td><td>448</td></tr>
<tr><td></td><td></td><td colspan="32">Data</td></tr>
</table>

Summary

- The IP Packet has headers and data sections
- IP Packet header is 20 bytes (can go up to 60 bytes if options are enabled)
- Data section can go up to 65536
- Packets need to get fragmented if it doesn’t fit in a frame

ICMP

Internet Control Message Protocol

ICMP

- Stands for Internet Control Message Protocol
- Designed for informational messages
  - Host unreachable, port unreachable, fragmentation needed
  - Packet expired (infinite loop in routers)
- Uses IP directly
- PING and traceroute use it
- Doesn’t require listeners or ports to be opened

ICMP header

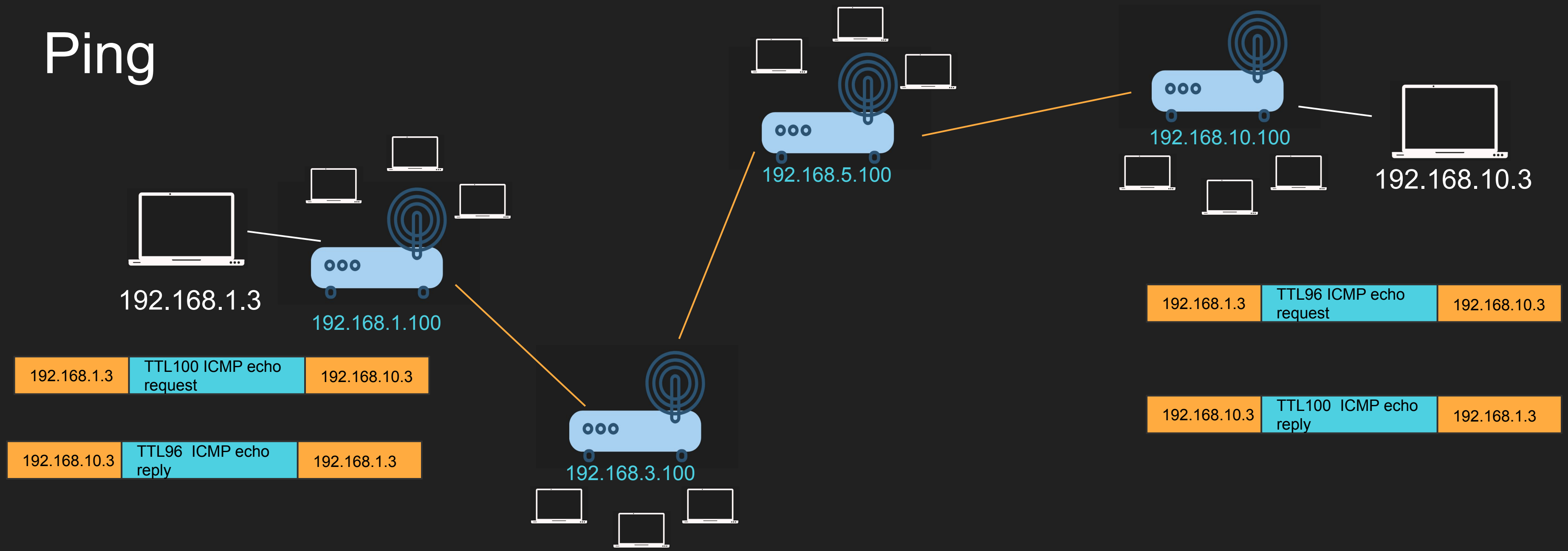
Offsets	Octet	0								1								2								3							
Octet	**Bit**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**	**31**
0	**0**	Type								Code								Checksum															
4	**32**	Rest of header																															

https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol
https://datatracker.ietf.org/doc/html/rfc792

ICMP

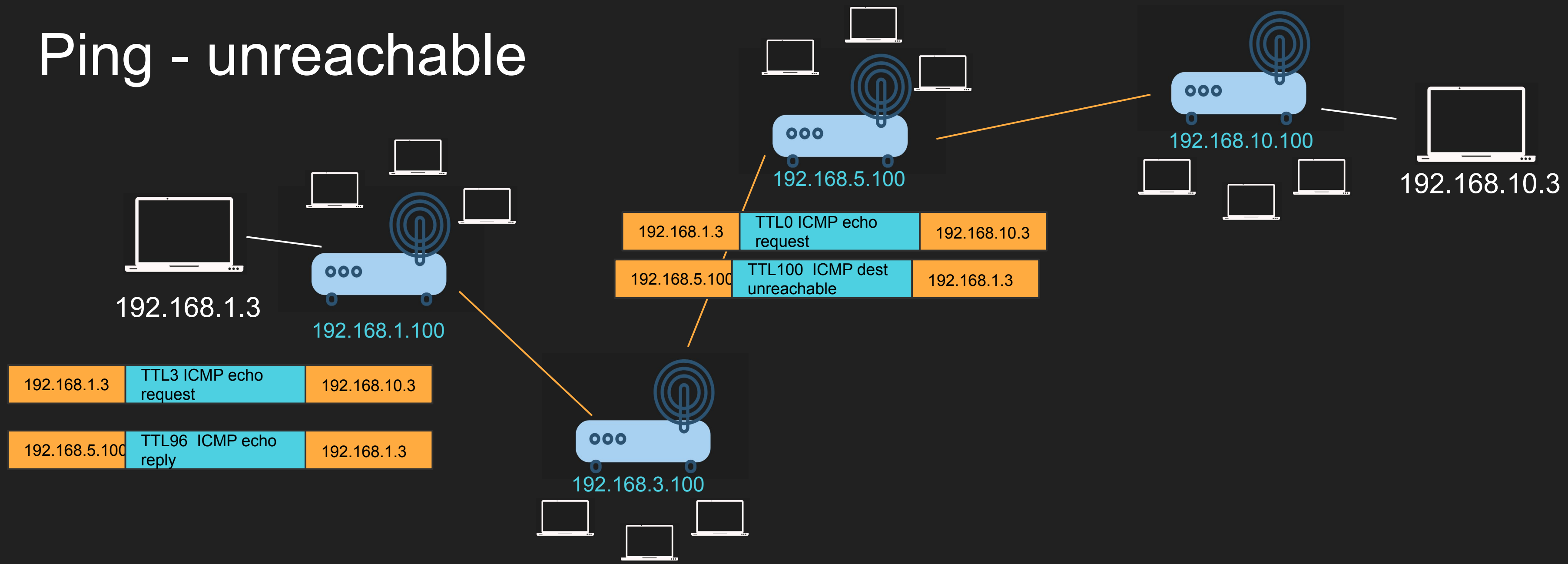
- Some firewalls block ICMP for security reasons
- That is why PING might not work in those cases
- Disabling ICMP also can cause real damage with connection establishment
  - Fragmentation needed
- PING demo

Ping


192.168.1.3
192.168.1.100
192.168.3.100
192.168.5.100
192.168.10.100
192.168.10.3
192.168.1.3 TTL100 ICMP echo request 192.168.10.3
192.168.10.3 TTL96 ICMP echo reply 192.168.1.3
192.168.1.3 TTL96 ICMP echo request 192.168.10.3
192.168.10.3 TTL100 ICMP echo reply 192.168.1.3

husseinnasser

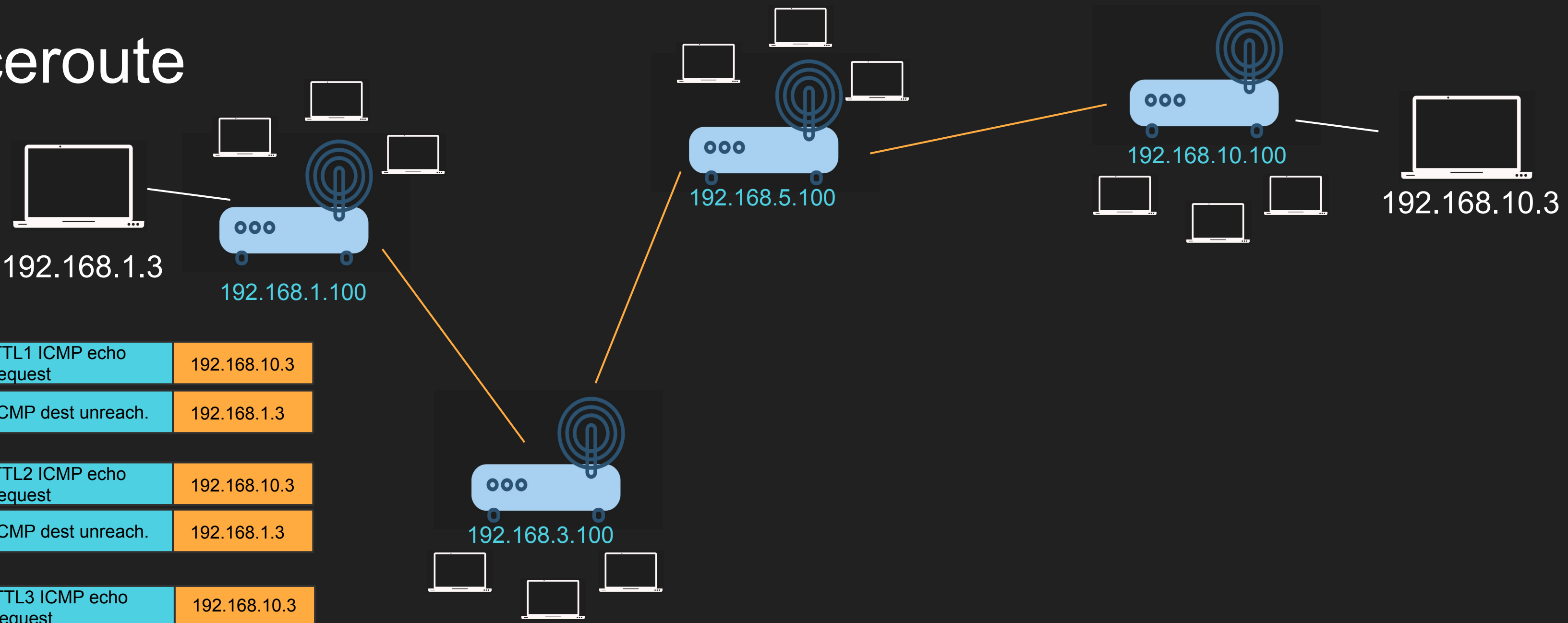
Ping - unreachable



TraceRoute

- Can you identify the entire path your IP Packet takes?
- Clever use of TTL
- Increment TTL slowly and you will get the router IP address for each hop
- Doesn’t always work as path changes and ICMP might be blocked

Traceroute



192.168.1.3	TTL1 ICMP echo request	192.168.10.3
192.168.1.100	ICMP dest unreach.	192.168.1.3
192.168.1.3	TTL2 ICMP echo request	192.168.10.3
192.168.3.100	ICMP dest unreach.	192.168.1.3
192.168.1.3	TTL3 ICMP echo request	192.168.10.3
192.168.5.100	ICMP dest unreach.	192.168.1.3
192.168.1.3	TTL4 ICMP echo request	192.168.10.3
192.168.10.100	ICMP dest unreach	192.168.1.3
192.168.1.3	TTL5 ICMP echo request	192.168.10.3
192.168.10.3	ICMP Echo reply	192.168.1.3

Summary

- ICMP is an IP level protocol used for information messages
- Critical to know if the host is available or port is opened
- Used for PING and TraceRoute
- Can be blocked which can cause problems

ARP

Address Resolution Protocol

Why ARP?

- We need the MAC address to send frames (layer 2)
- Most of the time we know the IP address but not the MAC
- ARP Table is cached IP->Mac mapping

Network Frame

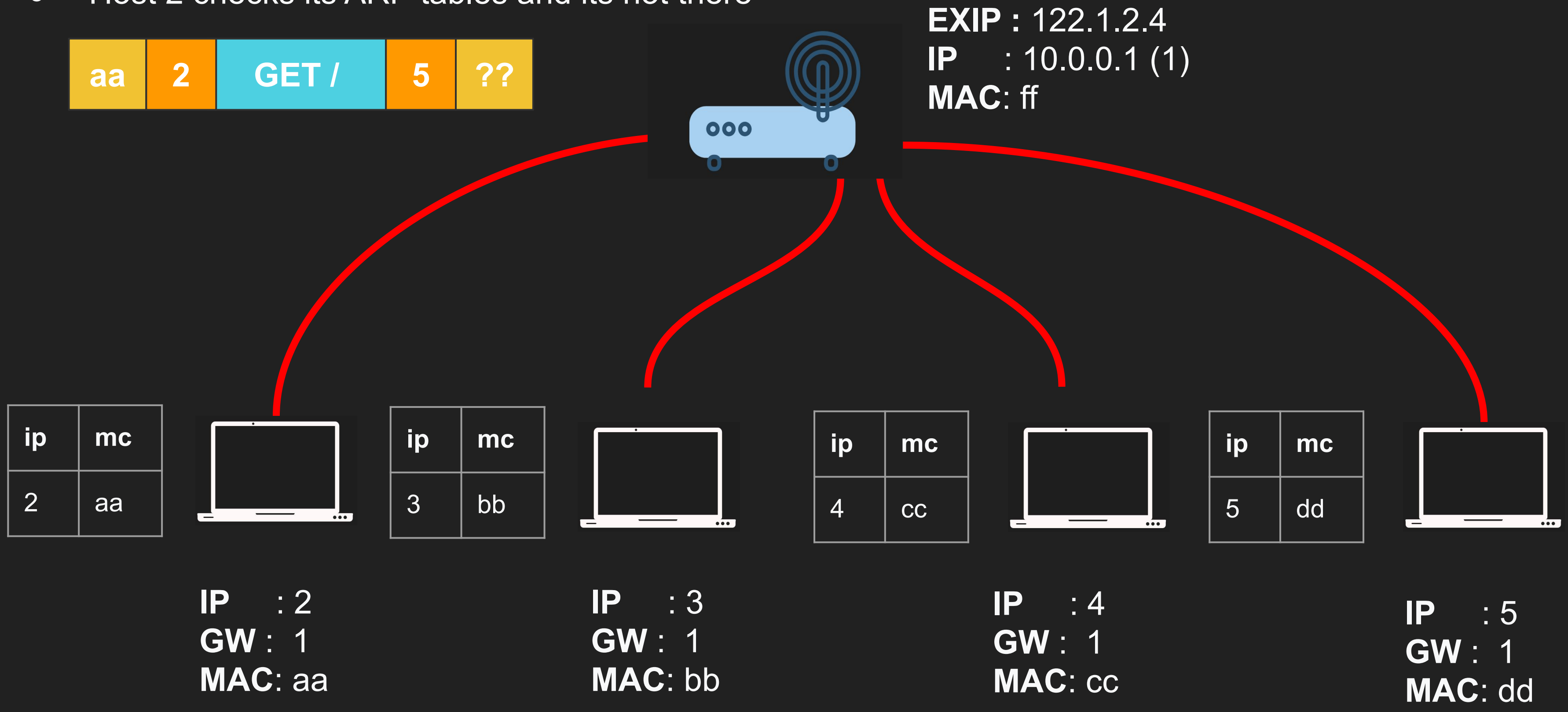
aa:bc:32:7f:c0:07	10.0.0.2	2312	GET /	8080	10.0.0.3	bb:ab:dd:11:22:33

aa	2	GET /	3	bb

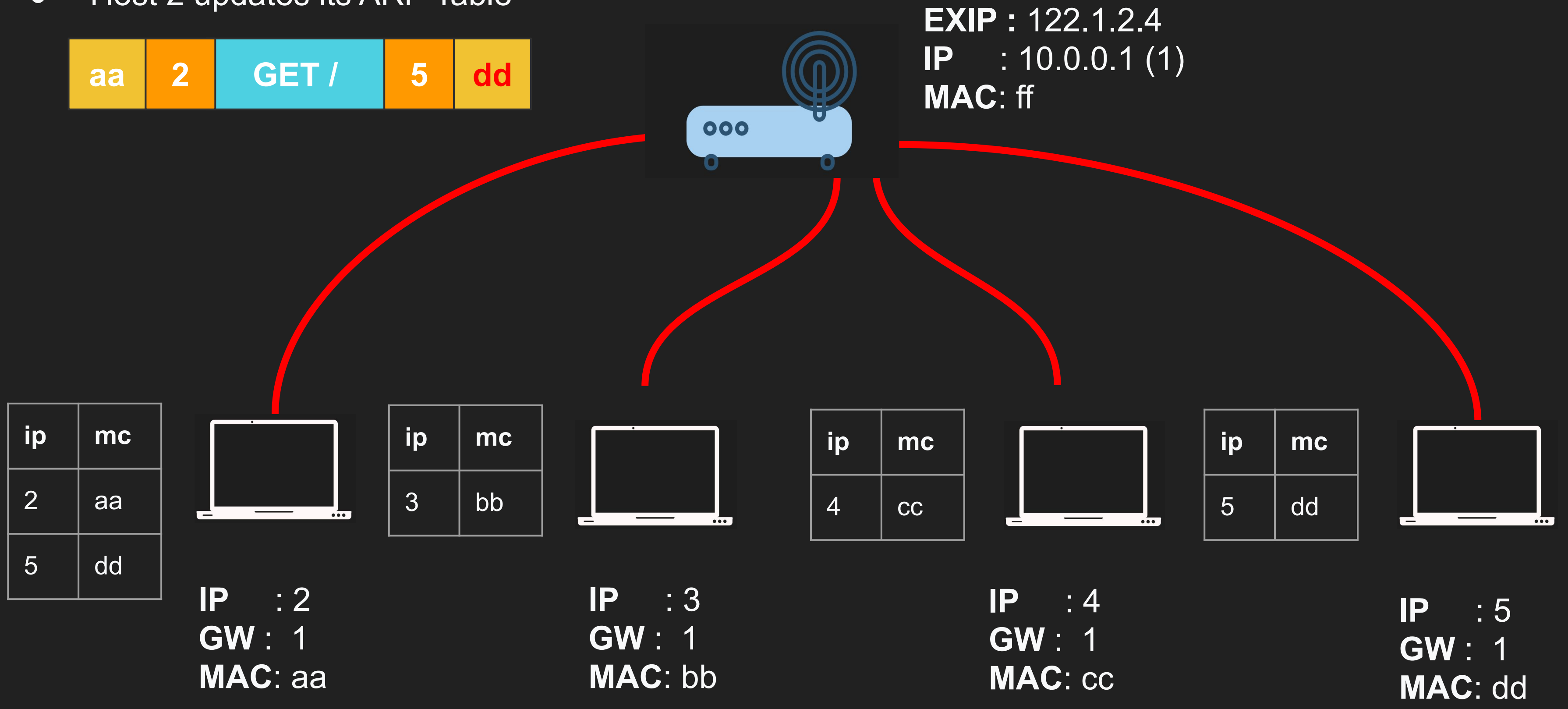
IP : 10.0.0.2
MAC: aa:bc:32:7f:c0:07

IP : 10.0.0.3
MAC: bb:ab:dd:11:22:33
Port: 8080

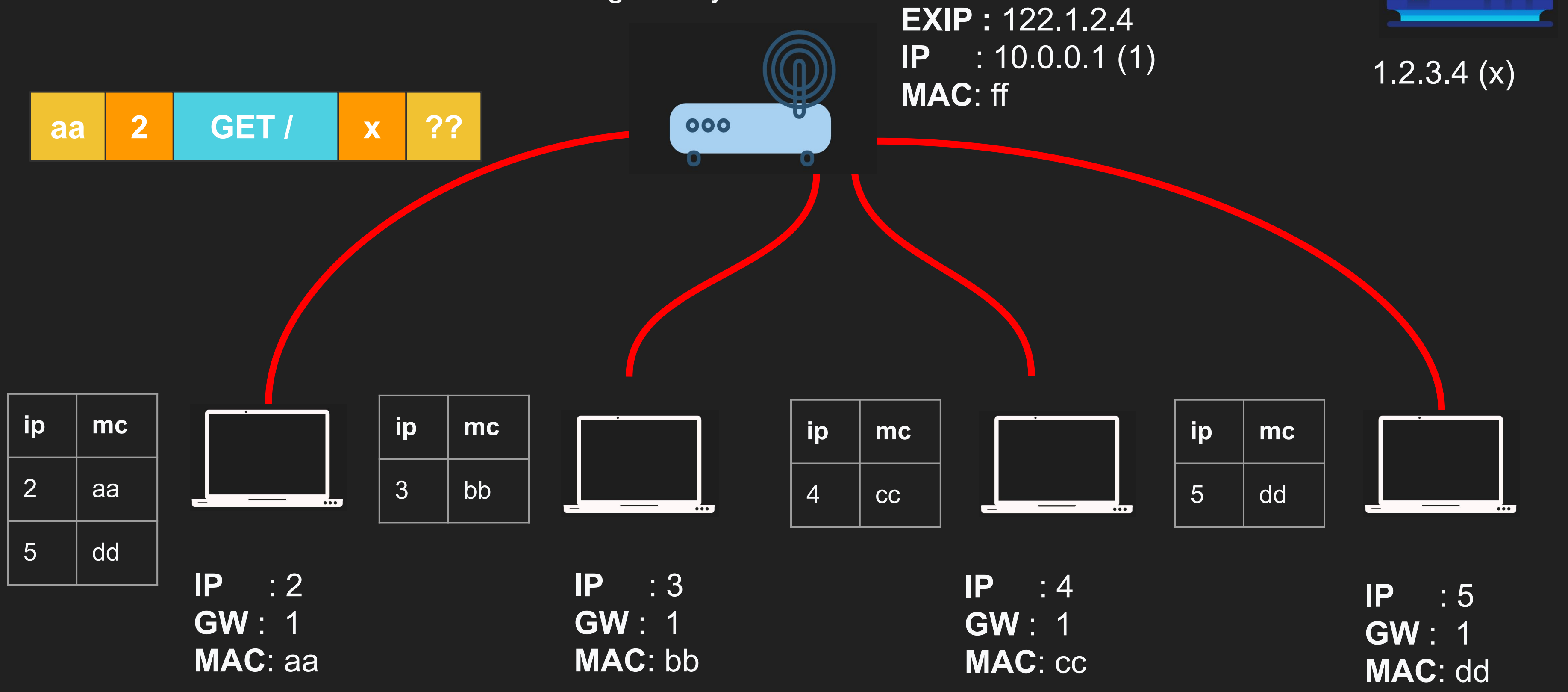
- IP 10.0.0.2 (**2**) wants to connect to IP 10.0.0.5 (**5**)
- Host 2 checks if host 5 is within its subnet, it is.
- Host **2** needs the MAC address of host **5**
- Host 2 checks its ARP tables and its not there



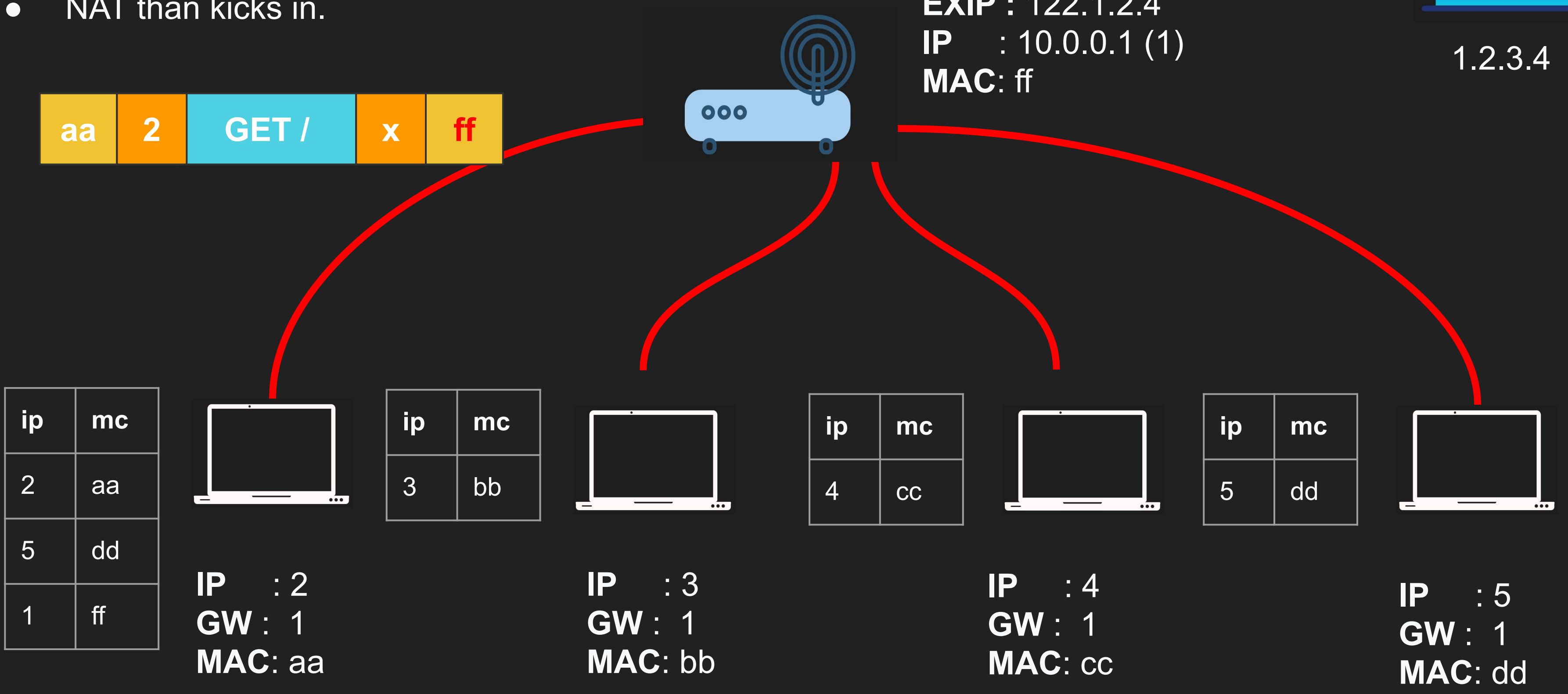
- Host 2 sends an ARP request broadcast to all machines in its network
- Who has IP address 10.0.0.5?
- Host 5 replies with dd
- Host 2 updates its ARP Table



- IP 10.0.0.2 (**2**) wants to connect to IP 1.2.3.4 (x)
- Host 2 checks if 1.2.3.4 is within its subnet, it is NOT!
- Host 2 needs to talk to its gatway
- Host **2** needs the MAC address of the gateway



- Host 2 checks its local ARP table, 10.0.0.1 is not it in
- Host 2 sends an ARP request to everybody in the network
- Who has 10.0.0.1? (A DANGEROUS QUESTION)
- Gateway reply with ff
- NAT than kicks in.

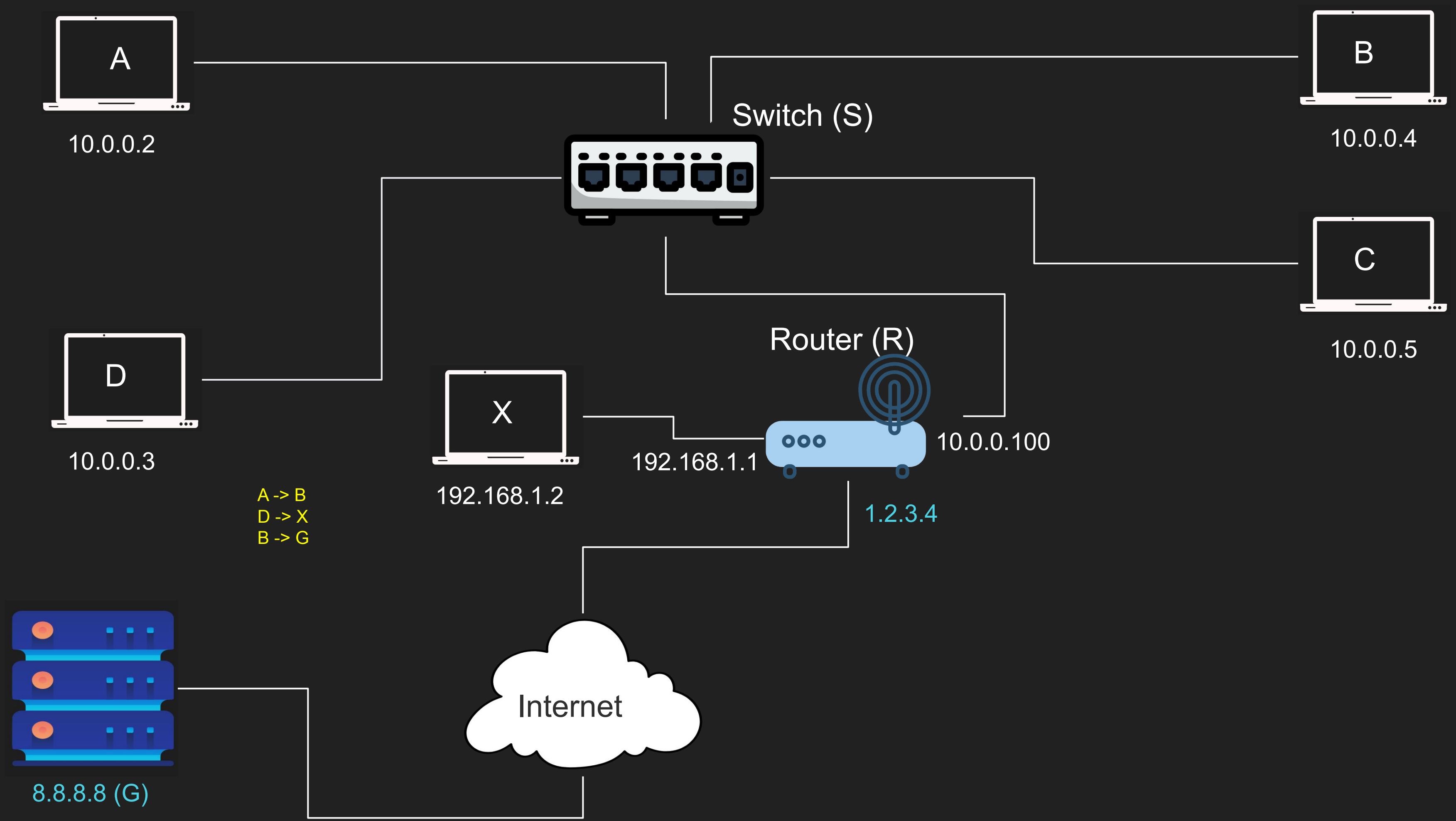


Summary

- ARP stands for Address resolution protocol
- We need MAC address to send frames between machines
- Almost always we have the IP address but not the MAC
- Need a lookup protocol that give us the MAC from IP address
- Attacks can be performed on ARP (ARP poisoning)

Routing Example

How IP Packets are routed in Switches and Routers


A
10.0.0.2
B
10.0.0.4
Switch (S)
C
10.0.0.5
D
10.0.0.3
X
192.168.1.2
Router (R)
192.168.1.1
10.0.0.100
1.2.3.4
A -> B
D -> X
B -> G
Internet
8.8.8.8 (G)

UDP

User Datagram Protocol

UDP

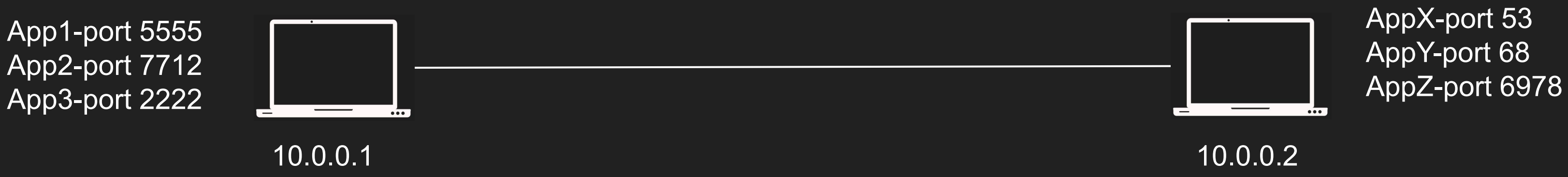
- Stands for User Datagram Protocol
- Layer 4 protocol
- Ability to address processes in a host using ports
- Simple protocol to send and receive data
- Prior communication not required (double edge sword)
- Stateless no knowledge is stored on the host
- 8 byte header Datagram

UDP Use cases

- Video streaming
- VPN
- DNS
- WebRTC

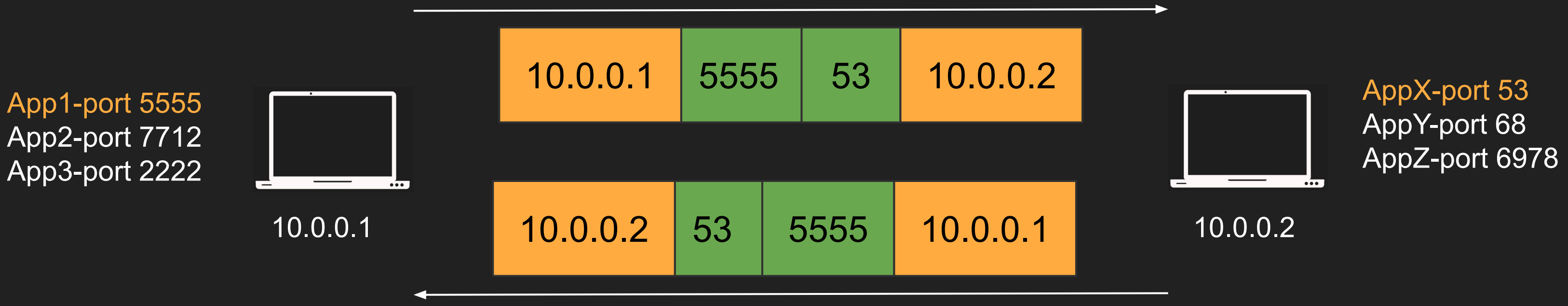
Multiplexing and demultiplexing

- IP target hosts only
- Hosts run many apps each with different requirements
- Ports now identify the “app” or “process”
- Sender multiplexes all its apps into UDP
- Receiver demultiplex UDP datagrams to each app



Source and Destination Port

- App1 on 10.0.0.1 sends data to AppX on 10.0.0.2
- Destination Port = 53
- AppX responds back to App1
- We need Source Port so we know how to send back data
- Source Port = 5555



Summary

- UDP is a simple layer 4 protocol
- Uses ports to address processes
- Stateless

UDP Datagram

The anatomy of the UDP datagram

UDP Datagram

- UDP Header is 8 bytes only (IPv4)
- Datagram slides into an IP packet as “data”
- Port are 16 bit (0 to 65535)

UDP Datagram header

Offsets	**Octet**	**0**								**1**								**2**								**3**							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Length																Checksum															
		Data																															

https://www.ietf.org/rfc/rfc768.txt
https://en.wikipedia.org/wiki/User_Datagram_Protocol

Source Port and Destination Port

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><th>0</th><td>0</td><td colspan="16">Source port</td><td colspan="16">Destination port</td></tr>
<tr><th>4</th><td>32</td><td colspan="16">Length</td><td colspan="16">Checksum</td></tr>
<tr><td></td><td></td><td colspan="32">Data</td></tr>
</table>

Length & Checksum

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Length																Checksum															
		Data																															

UDP Pros and Cons

The power and drawbacks of UDP

UDP Pros

- Simple protocol
- Header size is small so datagrams are small
- Uses less bandwidth
- Stateless
- Consumes less memory (no state stored in the server/client)
- Low latency - no handshake , order, retransmission or guaranteed delivery

UDP Cons

- No acknowledgement
- No guarantee delivery
- Connection-less - anyone can send data without prior knowledge
- No flow control
- No congestion control
- No ordered packets
- Security - can be easily spoofed

TCP

Transmission Control Protocol

TCP

- Stands for Transmission Control Protocol
- Layer 4 protocol
- Ability to address processes in a host using ports
- “Controls” the transmission unlike UDP which is a firehose
- Connection
- Requires handshake
- 20 bytes headers Segment (can go to 60)
- Stateful

TCP Use cases

- Reliable communication
- Remote shell
- Database connections
- Web communications
- Any bidirectional communication

TCP Connection

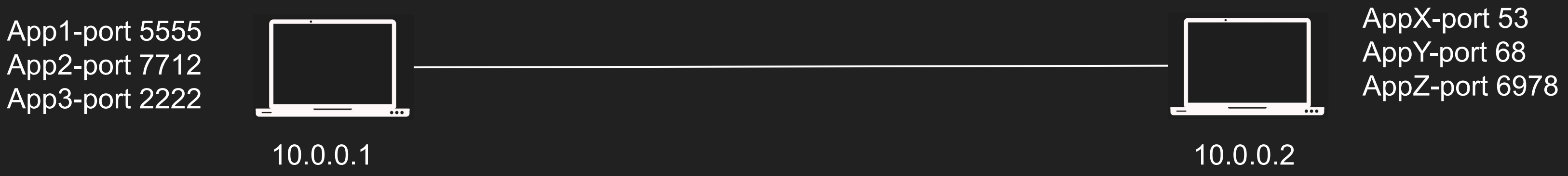
- Connection is a Layer 5 (session)
- Connection is an agreement between client and server
- Must create a connection to send data
- Connection is identified by 4 properties
    - SourceIP-SourcePort
    - DestinationIP-DestinationPort

TCP Connection

- Can’t send data outside of a connection
- Sometimes called socket or file descriptor
- Requires a 3-way TCP handshake
- Segments are sequenced and ordered
- Segments are acknowledged
- Lost segments are retransmitted

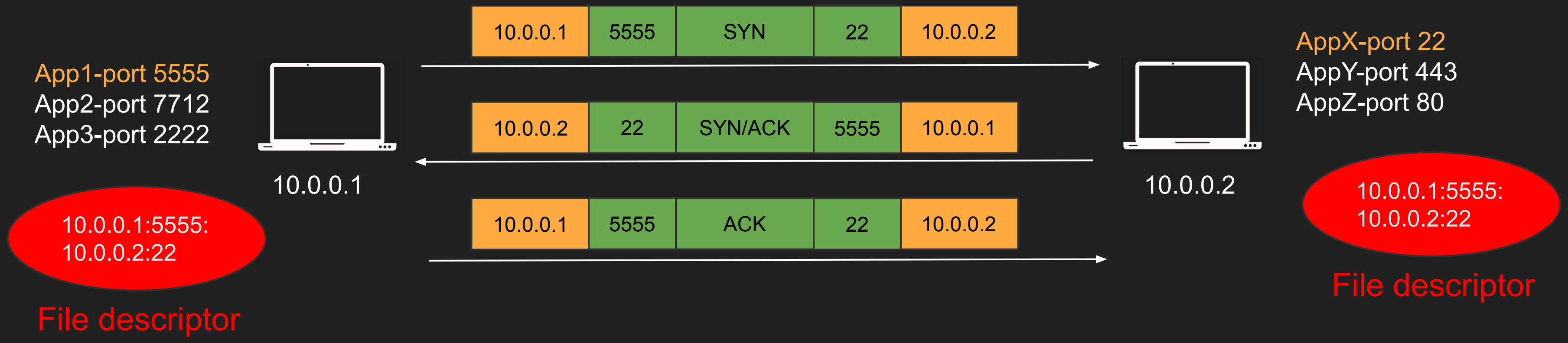
Multiplexing and demultiplexing

- IP target hosts only
- Hosts run many apps each with different requirements
- Ports now identify the “app” or “process”
- Sender multiplexes all its apps into TCP connections
- Receiver demultiplex TCP segments to each app based on connection pairs



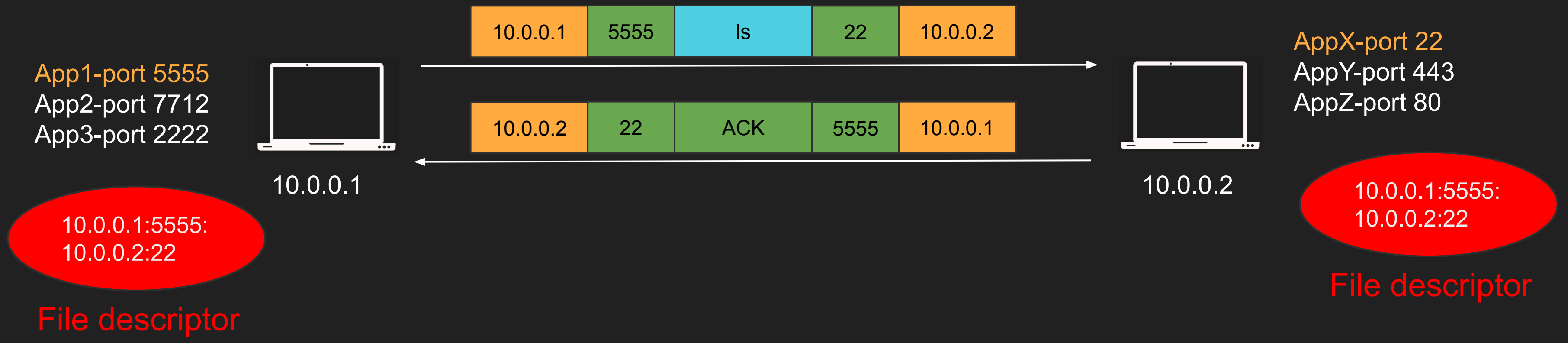
Connection Establishment

- App1 on 10.0.0.1 want to send data to AppX on 10.0.0.2
- App1 sends SYN to AppX to synchronous sequence numbers
- AppX sends SYN/ACK to synchronous its sequence number
- App1 ACKs AppX SYN.
- Three way handshake



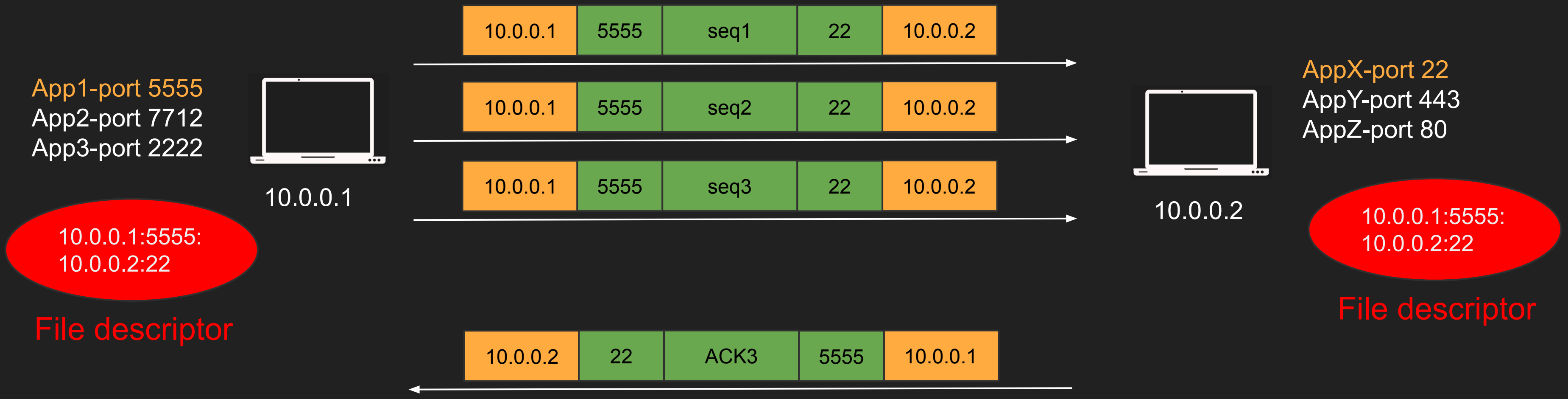
Sending data

- App1 sends data to AppX
- App1 encapsulate the data in a segment and send it
- AppX acknowledges the segment
- Hint: Can App1 send new segment before ack of old segment arrives?



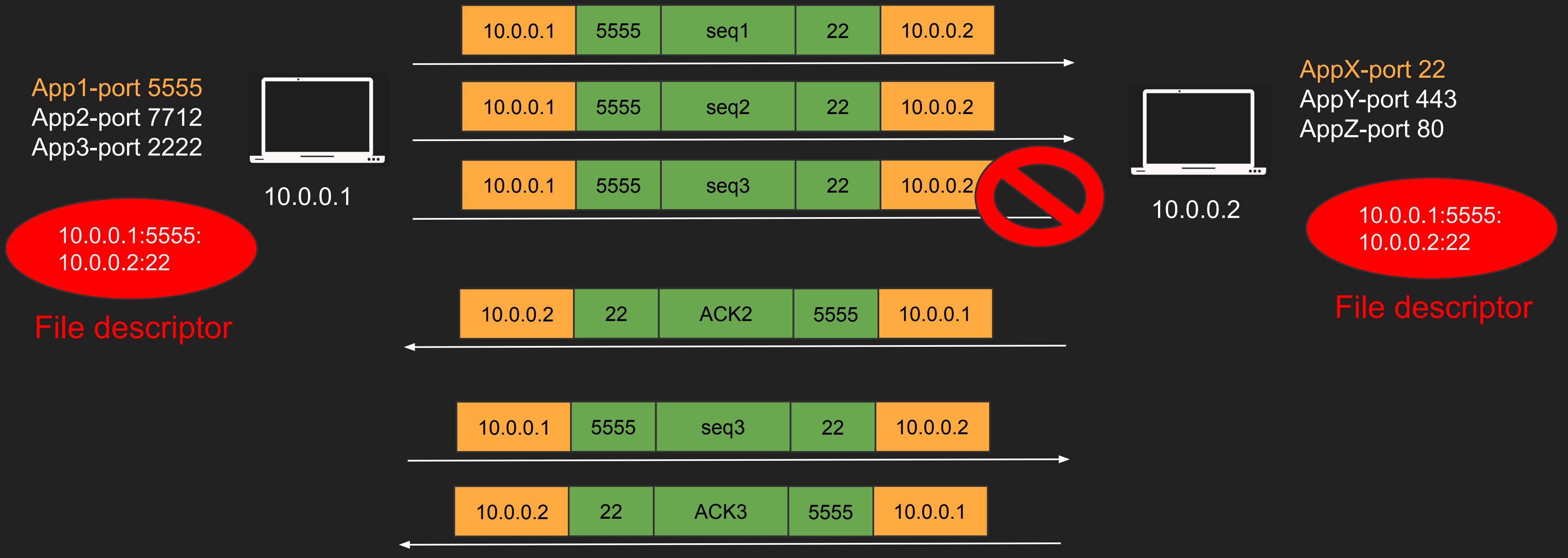
Acknowledgment

- App1 sends segment 1,2 and 3 to AppX
- AppX acknowledge all of them with a single ACK 3



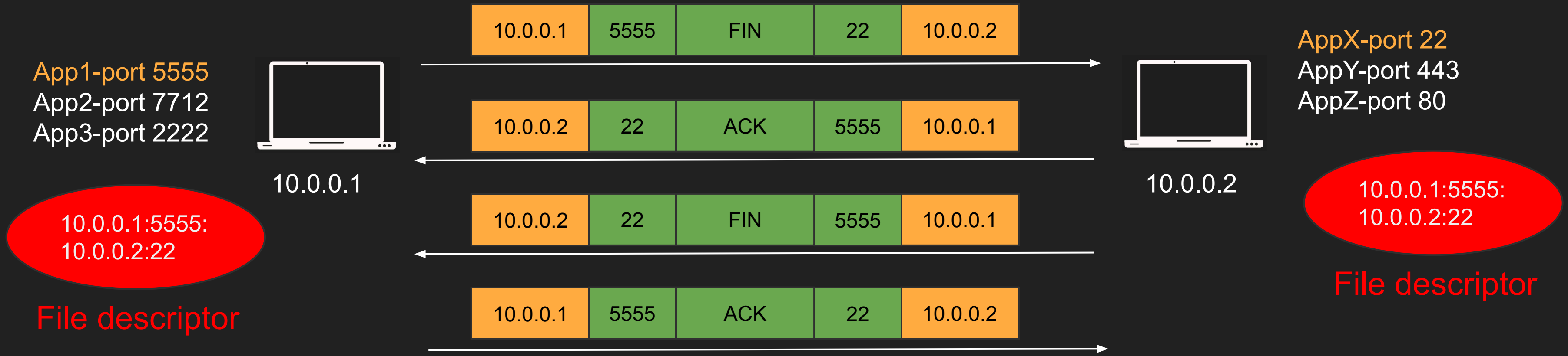
Lost data

- App1 sends segment 1,2 and 3 to AppX
- Seg 3 is lost, AppX acknowledge 3
- App1 resend Seq 3



Closing Connection

- App1 wants to close the connection
- App1 sends FIN, AppX ACK
- AppX sends FIN, App1 ACK
- Four way handshake



Summary

- Stands for Transmission Control Protocol
- Layer 4 protocol
- “Controls” the transmission unlike UDP which is a firehose
- Introduces Connection concept
- Retransmission, acknowledgement, guaranteed delivery
- Stateful, connection has a state

TCP Segment

The anatomy of the TCP Segment

TCP Segment

- TCP segment Header is 20 bytes and can go up to 60 bytes
- TCP segments slides into an IP packet as “data”
- Port are 16 bit (0 to 65535)
- Sequences, Acknowledgment, flow control and more

TCP Segment

Offsets	Octet	0								1								2								3							
Octet	**Bit**	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	**0**	Source port																Destination port															
4	**32**	Sequence number																															
8	**64**	Acknowledgment number (if ACK set)																															
12	**96**	Data offset				Reserved **0 0 0**			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	**128**	Checksum																Urgent pointer (if URG set)															
20	**160**	Options (if *data offset* > 5. Padded at the end with "0" bits if necessary.)																															
⋮	⋮																																
60	**480**																																

https://en.wikipedia.org/wiki/Transmission_Control_Protocol
https://datatracker.ietf.org/doc/html/rfc793

Ports

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><th>0</th><th>0</th><td colspan="16">Source port</td><td colspan="16">Destination port</td></tr>
<tr><th>4</th><th>32</th><td colspan="32">Sequence number</td></tr>
<tr><th>8</th><th>64</th><td colspan="32">Acknowledgment number (if ACK set)</td></tr>
<tr><th>12</th><th>96</th><td colspan="4">Data offset</td><td colspan="3">Reserved
0 0 0</td><td>NS</td><td>CWR</td><td>ECE</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="16">Window Size</td></tr>
<tr><th>16</th><th>128</th><td colspan="16">Checksum</td><td colspan="16">Urgent pointer (if URG set)</td></tr>
<tr><th>20</th><th>160</th><td colspan="32" rowspan="3">Options (if data offset > 5. Padded at the end with "0" bits if necessary.)</td></tr>
<tr><th>⋮</th><th>⋮</th></tr>
<tr><th>60</th><th>480</th></tr>
</table>

Sequences and ACKs

Offsets	Octet	0								1								2								3							
Octet	**Bit**	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	**0**	Source port																Destination port															
4	**32**	Sequence number																															
8	**64**	Acknowledgment number (if ACK set)																															
12	**96**	Data offset				Reserved 0 0 0			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	**128**	Checksum																Urgent pointer (if URG set)															
20	**160**	Options (if *data offset* > 5. Padded at the end with "0" bits if necessary.)																															
⋮	⋮																																
60	**480**																																

Flow Control Window Size

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td colspan="16">Source port</td><td colspan="16">Destination port</td></tr>
<tr><td>4</td><td>32</td><td colspan="32">Sequence number</td></tr>
<tr><td>8</td><td>64</td><td colspan="32">Acknowledgment number (if ACK set)</td></tr>
<tr><td>12</td><td>96</td><td colspan="4">Data offset</td><td colspan="3">Reserved
0 0 0</td><td>N
S</td><td>C
W
R</td><td>E
C
E</td><td>U
R
G</td><td>A
C
K</td><td>P
S
H</td><td>R
S
T</td><td>S
Y
N</td><td>FI
N</td><td colspan="16">Window Size</td></tr>
<tr><td>16</td><td>128</td><td colspan="16">Checksum</td><td colspan="16">Urgent pointer (if URG set)</td></tr>
<tr><td>20</td><td>160</td><td colspan="32" rowspan="3">Options (if data offset > 5. Padded at the end with "0" bits if necessary.)</td></tr>
<tr><td>⋮</td><td>⋮</td></tr>
<tr><td>60</td><td>480</td></tr>
</table>

9 bit flags

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td colspan="16">Source port</td><td colspan="16">Destination port</td></tr>
<tr><td>4</td><td>32</td><td colspan="32">Sequence number</td></tr>
<tr><td>8</td><td>64</td><td colspan="32">Acknowledgment number (if ACK set)</td></tr>
<tr><td>12</td><td>96</td><td colspan="4">Data offset</td><td colspan="3">Reserved
0 0 0</td><td>NS</td><td>CWR</td><td>ECE</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="16">Window Size</td></tr>
<tr><td>16</td><td>128</td><td colspan="16">Checksum</td><td colspan="16">Urgent pointer (if URG set)</td></tr>
<tr><td>20</td><td>160</td><td colspan="32" rowspan="3">Options (if data offset > 5. Padded at the end with "0" bits if necessary.)</td></tr>
<tr><td>⋮</td><td>⋮</td></tr>
<tr><td>60</td><td>480</td></tr>
</table>

Maximum Segment Size

- Segment Size depends the MTU of the network
- Usually 512 bytes can go up to 1460
- Default MTU in the Internet is 1500 (results in MSS 1460)
- Jumbo frames MTU goes to 9000 or more
- MSS can be larger in jumbo frames cases

Flow Control

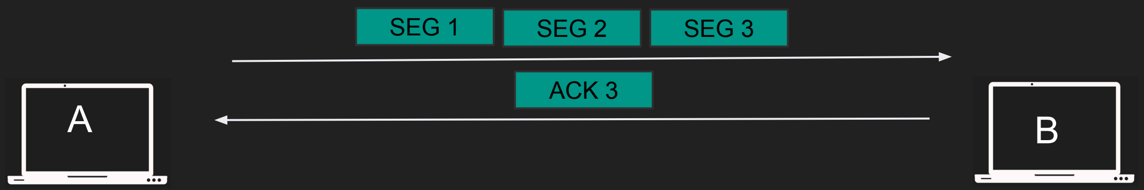
How much the receiver can handle?

Flow Control

- A want to send 10 segments to B
- A sends segment 1 to B
- B acknowledges segment 1
- A sends segment 2 to B
- B acknowledges segment 2
- VERY SLOW!

Flow Control

- A can send multiple segments and B can acknowledge all in 1 ACK
- The question is … how much A can send?
- This is called flow control



Flow Control

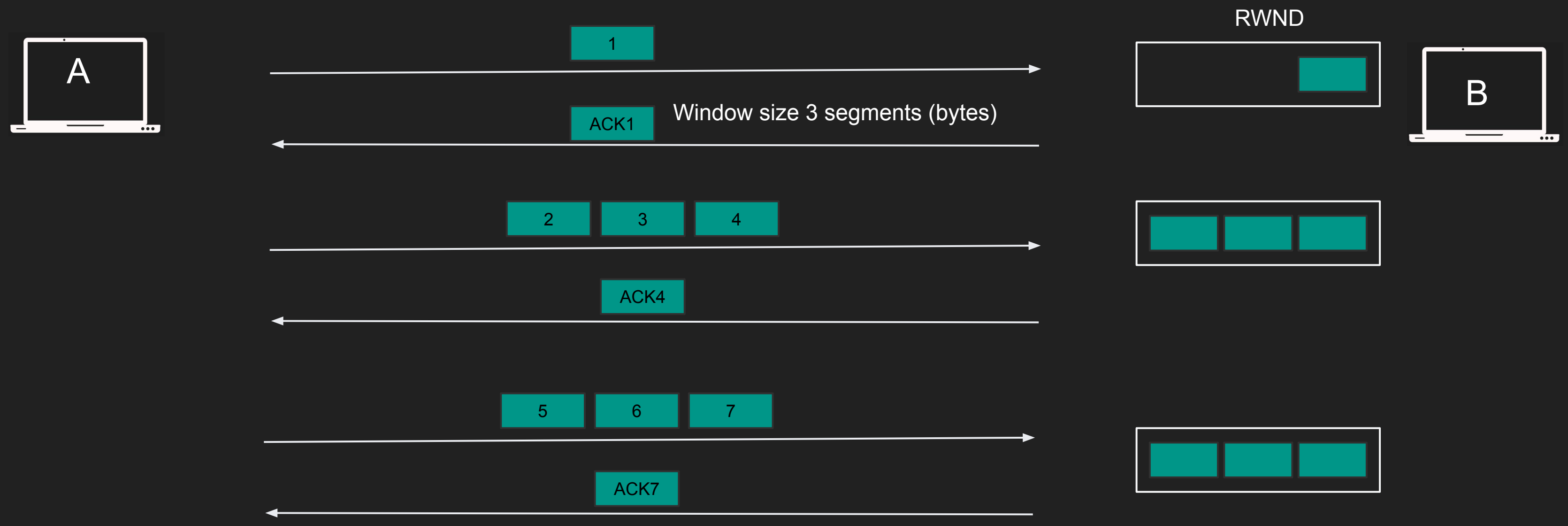
- When TCP segments arrive they are put in receiver's buffer
- If we kept sending data the receiver will be overwhelmed
- Segments will be dropped
- Solution? Let the sender know how much you can handle

Flow Control Window Size (Receiver Window)

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><th>0</th><th>0</th><td colspan="16">Source port</td><td colspan="16">Destination port</td></tr>
<tr><th>4</th><th>32</th><td colspan="32">Sequence number</td></tr>
<tr><th>8</th><th>64</th><td colspan="32">Acknowledgment number (if ACK set)</td></tr>
<tr><th>12</th><th>96</th><td colspan="4">Data offset</td><td colspan="3">Reserved
0 0 0</td><td>NS</td><td>CWR</td><td>ECE</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="16">Window Size</td></tr>
<tr><th>16</th><th>128</th><td colspan="16">Checksum</td><td colspan="16">Urgent pointer (if URG set)</td></tr>
<tr><th>20</th><th>160</th><td colspan="32" rowspan="3">Options (if data offset > 5. Padded at the end with "0" bits if necessary.)</td></tr>
<tr><th>⋮</th><th>⋮</th></tr>
<tr><th>60</th><th>480</th></tr>
</table>

Window Size (Receiver Window) RWND

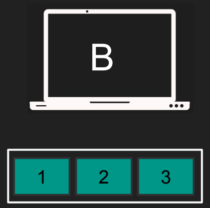
- 16 bit - Up to 64KB
- Updated with each acknowledgment
- Tells the sender how much to send before waiting for ACK
- Receiver can decide to decrease the Window Size (out of memory) more important stuff



Sliding Window

- Can’t keep waiting for receiver to acknowledge all segments
- Whatever gets acknowledge moves
- We “slide” the window
- Sender maintains the sliding window for the receiver

Window Scaling



- 64 KB is too small
- We can’t increase the bits on the segment
- Meet Window Scaling factor (0-14)
- Window Size can go up to 1GB ((2^16-1) x 2^14)
- Only exchanged during the handshake

Summary

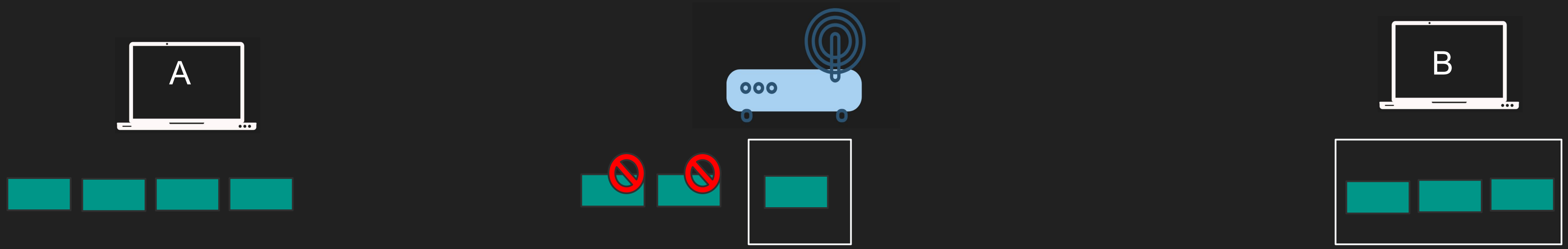
- Receiver host has a limit
- We need to let the sender know how much it can send
- Receiver Window is in the segment
- Sender maintains the Sliding Window to know how much it can send
- Window Scaling can increase that

Congestion Control

How much the network can handle?

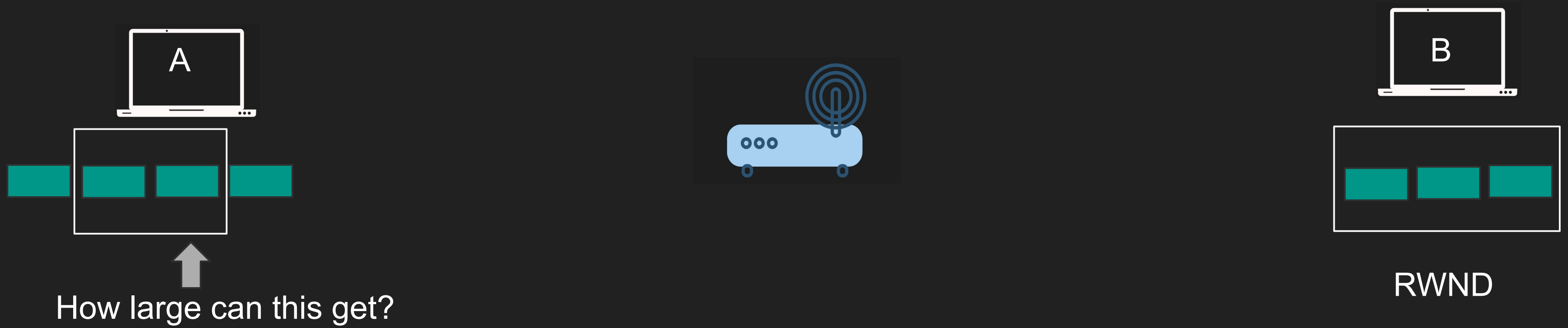
Congestion Control

- The receiver might handle the load but the middle boxes might not
- The routers in the middle have limit
- We don't want to congest the network with data
- We need to avoid congestion
- A new window: Congestion Window (CWND)



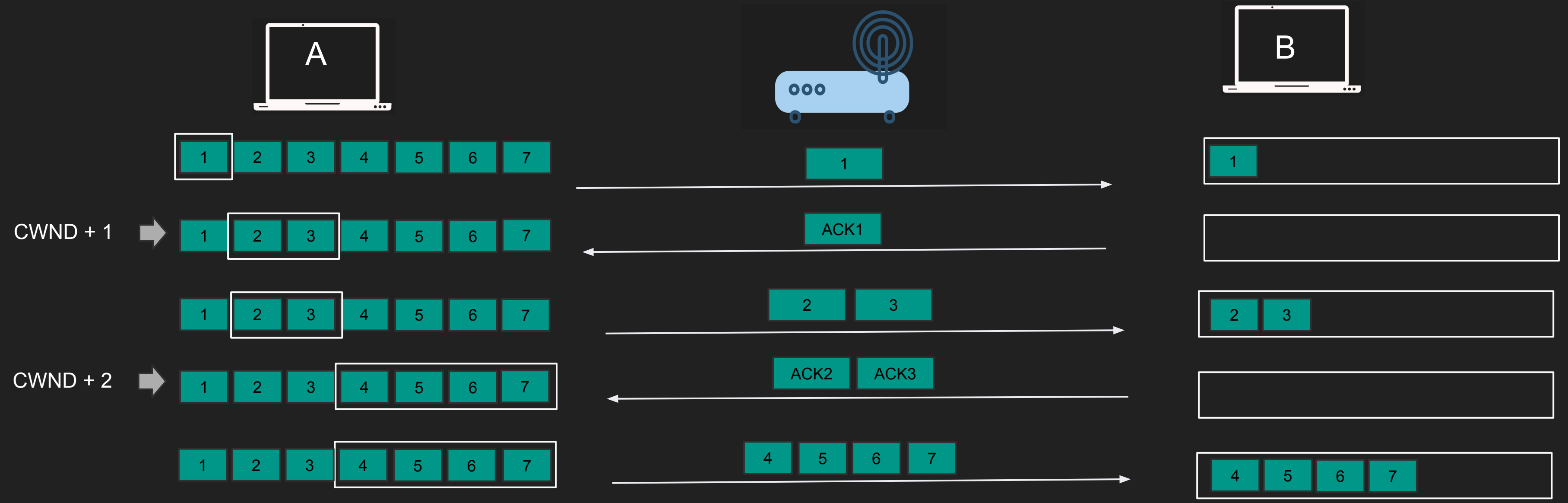
Two Congestion algorithms

- TCP Slow Start
  - Start slow goes fast!
  - CWND + 1 MSS after each ACK
- Congestion Avoidance
  - Once Slow start reaches its threshold this kicks in
  - CWND + 1 MSS after complete RTT
- CWND must not exceeds RWND



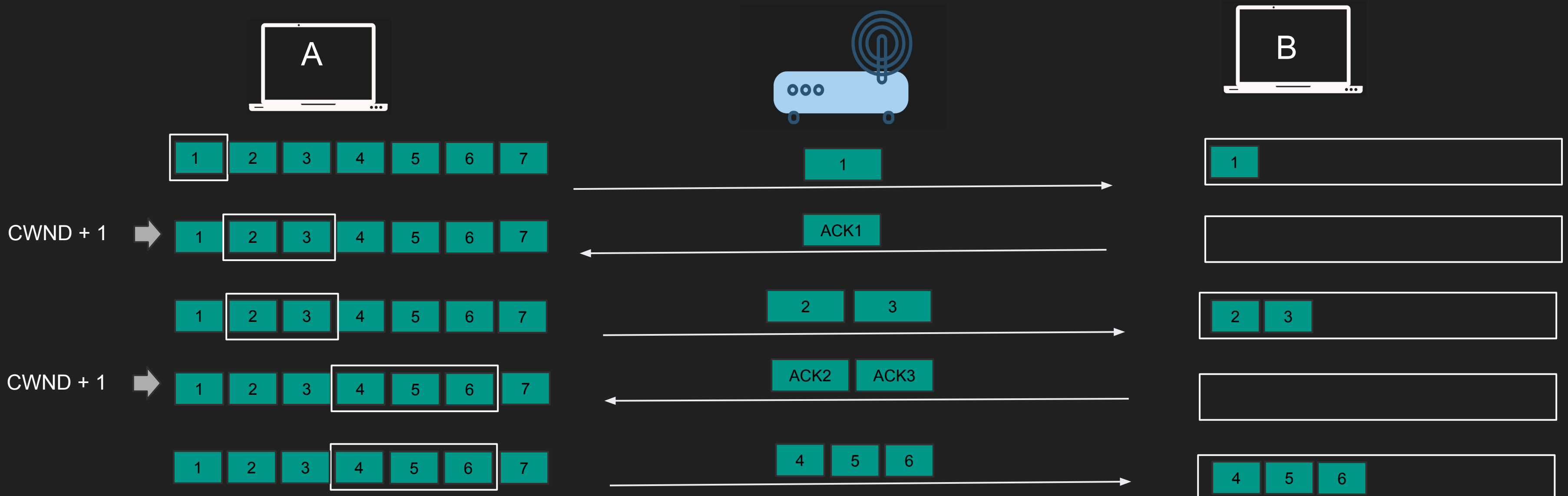
Slow Start

- CWND starts with 1 MSS (or more)
- Send 1 Segment and waits for ACK
- With EACH ACK received CWND is incremented by 1 MSS
- Until we reach slow start threshold we switch to congestion avoidance algorithm



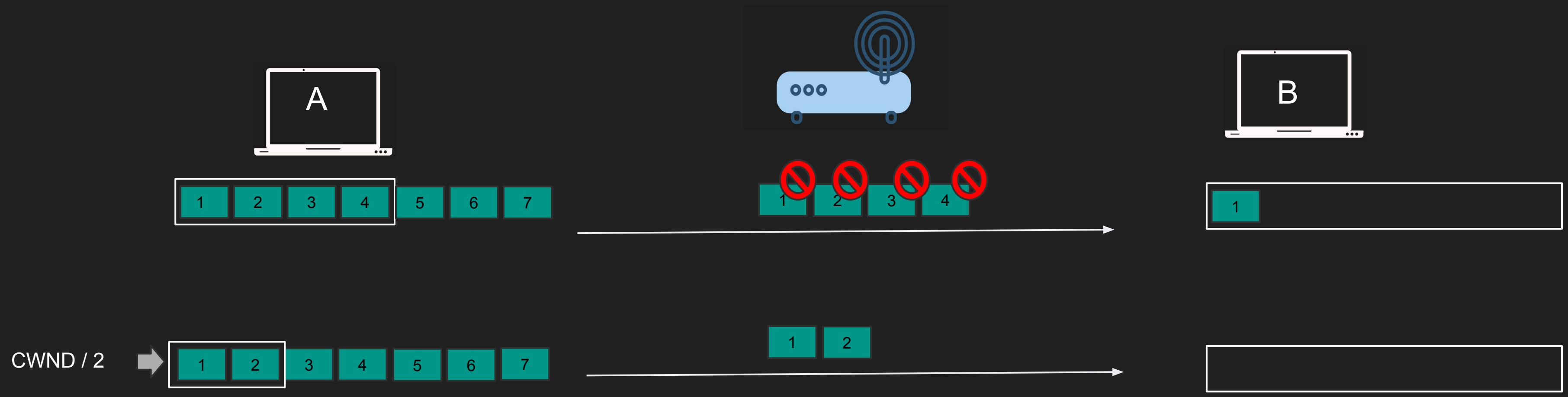
Congestion Avoidance

- Send CWND worth of Segments and waits for ACK
- Only when ALL segments are ACKed add UP to one MSS to CWND
- Precisely CWND = CWND + MSS*MSS/CWND



Congestion Avoidance

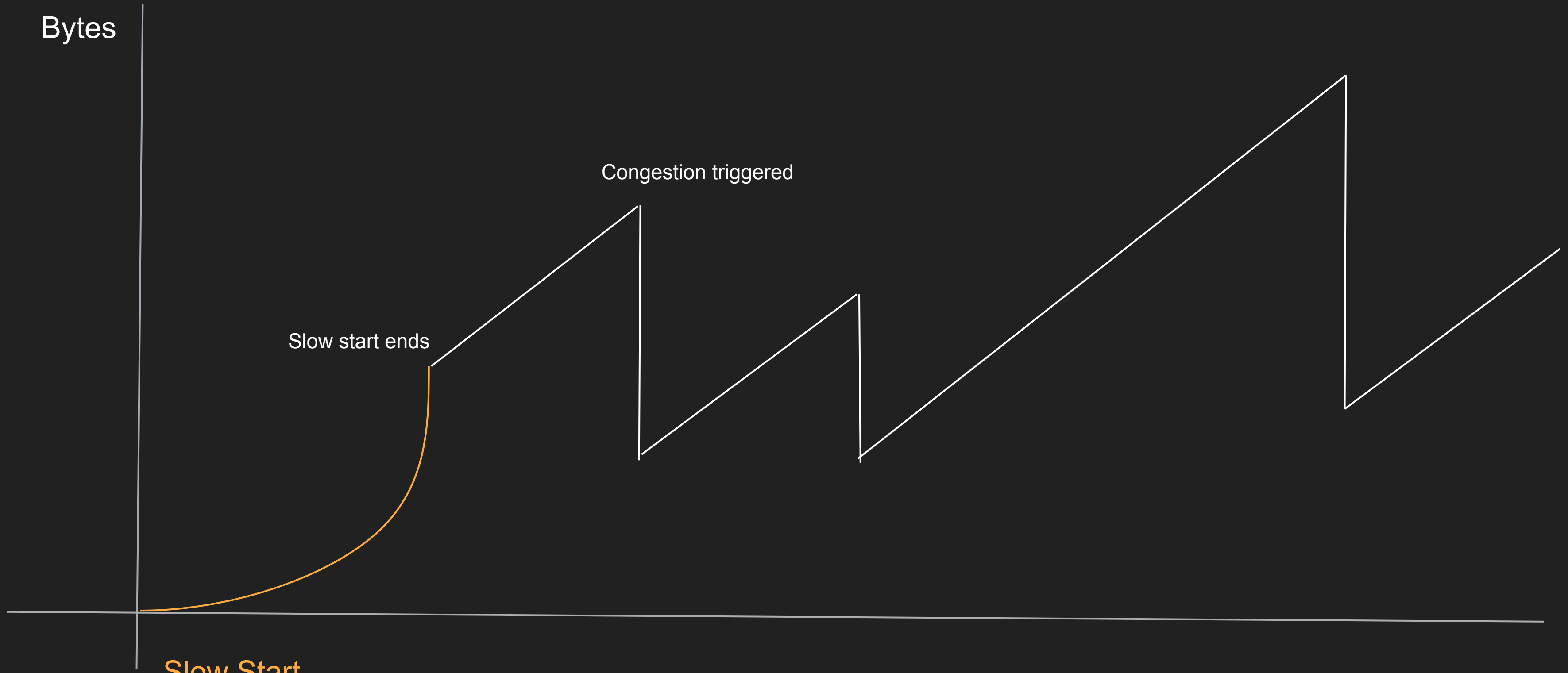
- The moment we get timeouts, dup ACKs or packet drops
- The CWND is halved



Congestion Notification

- We don’t want routers dropping packets
- Can Routers let us know when congestion hit?
- Meet ECN (Explicit Congestion Notification)
- Routers and middle boxes can tag IP packets with ECN
- The receiver will copy this bit back to the sender
- ECN is IP Header bit
- So Routers don’t drop packets just let me know you are reaching your limit

Congestion Control



Slow Start
Congestion Avoidance

Summary

- While the receiver may handle large data middle boxes might not
- Middle routers buffers may fill up
- Need to control the congestion in the network
- Sender can send segments up to CWND or RWND without ACK
- Isn’t normally a problem in hosts connected directly (LAN)

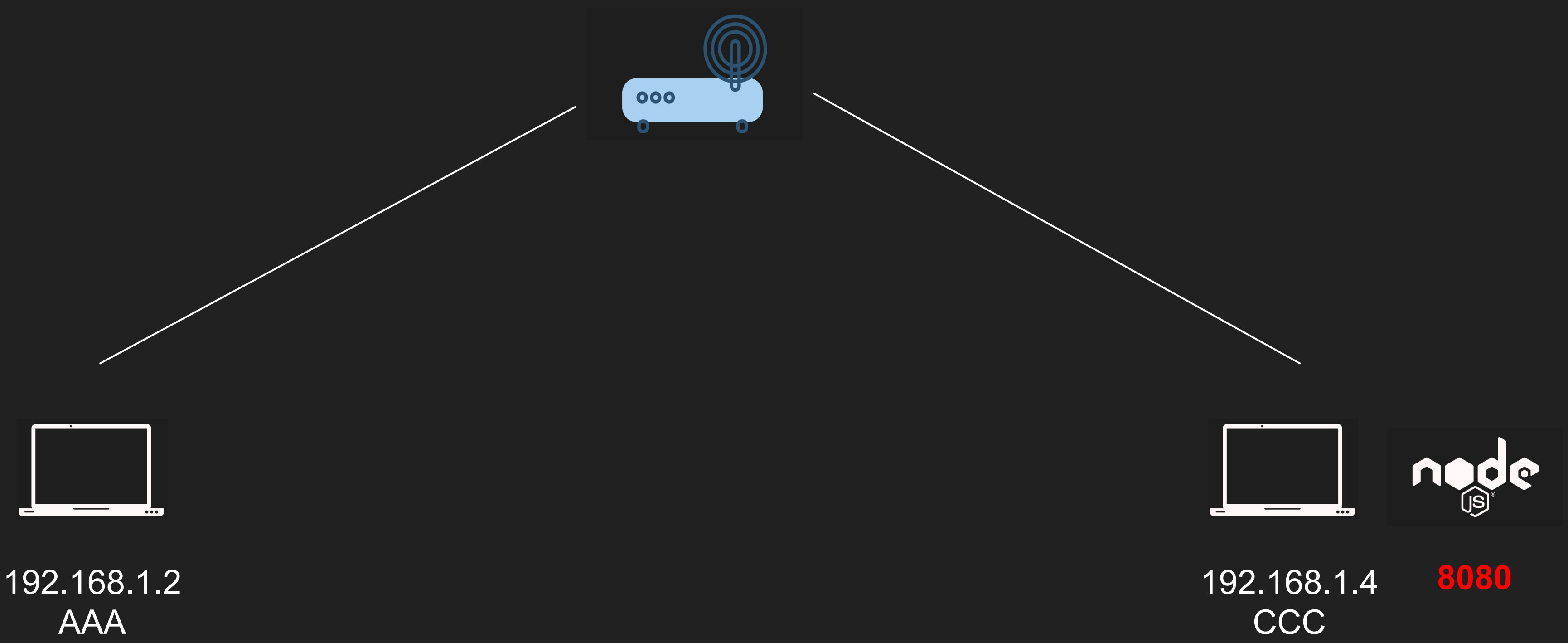
Network Address Translation

How the WAN sees your internal devices

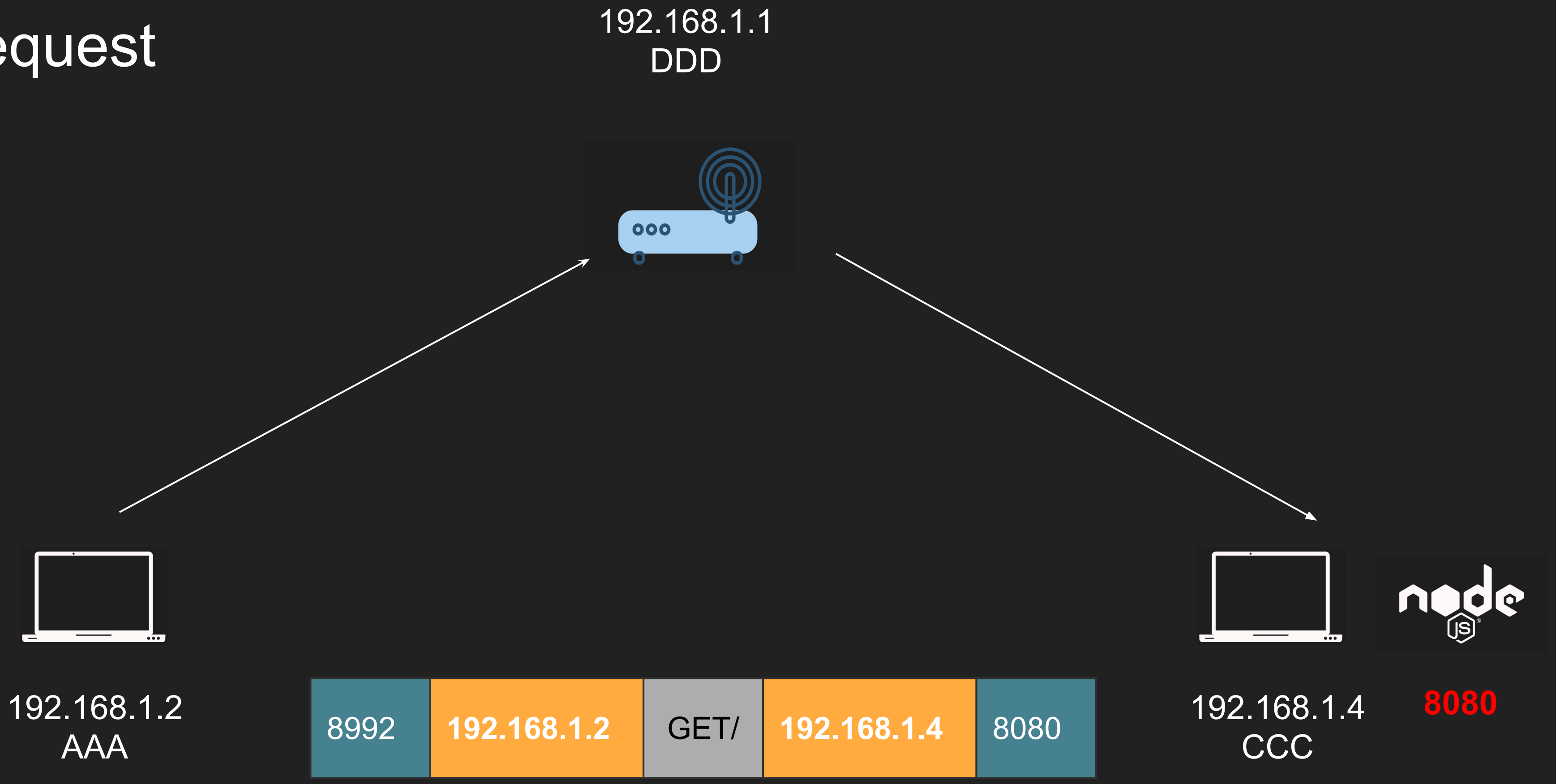
NAT

- IPv4 is limited only 4 billion
- Private vs Public IP Address
- E.g. 192.168.x.x , 10.0.0.x is private not routable in the Internet
- Internal hosts can be assigned private addresses
- Only your router need public IP address
- Router need to translate requests

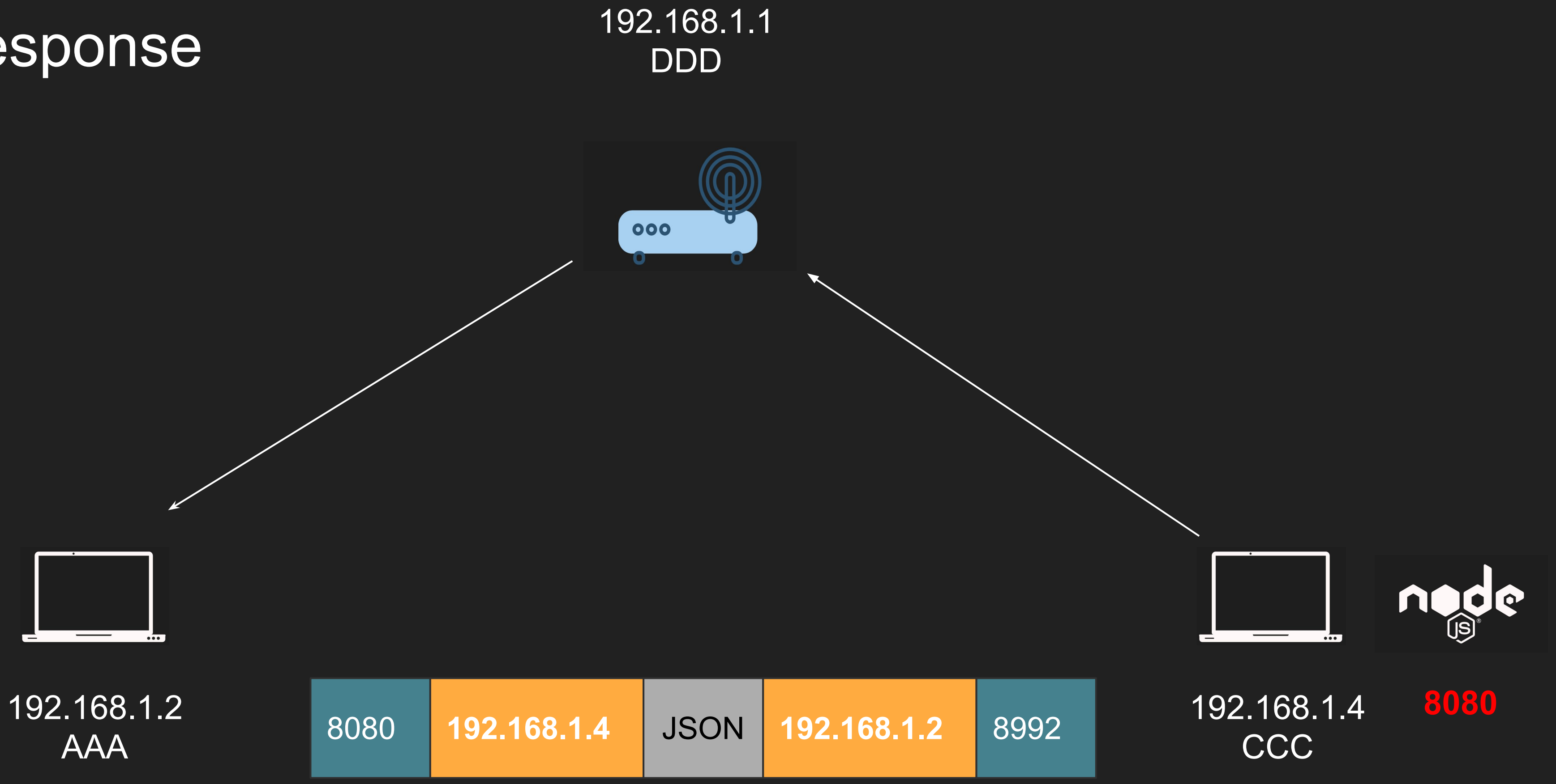
Local Network



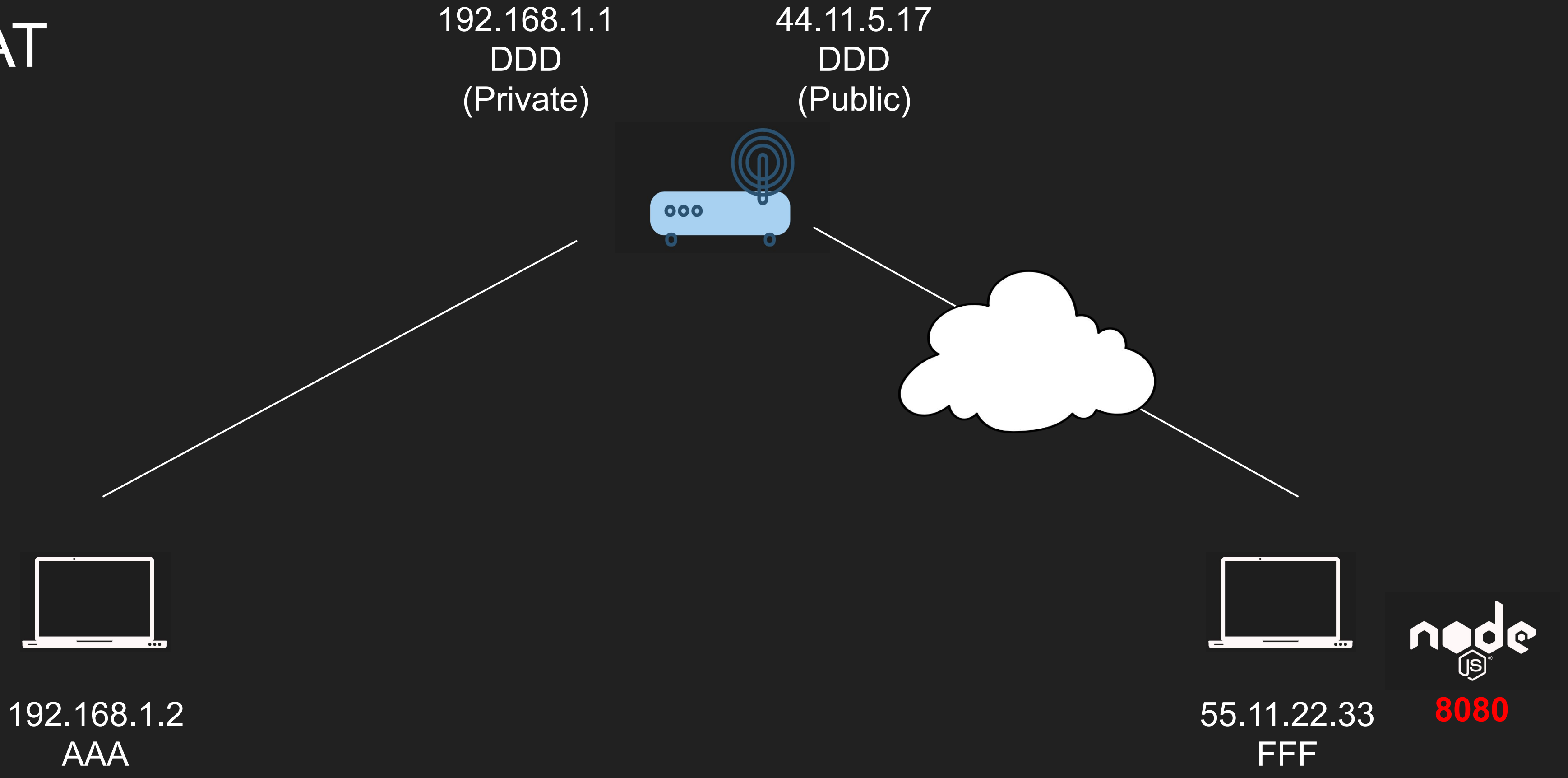
Request


192.168.1.1
DDD
192.168.1.2
AAA
8992
192.168.1.2
GET/
192.168.1.4
8080
192.168.1.4
CCC
8080

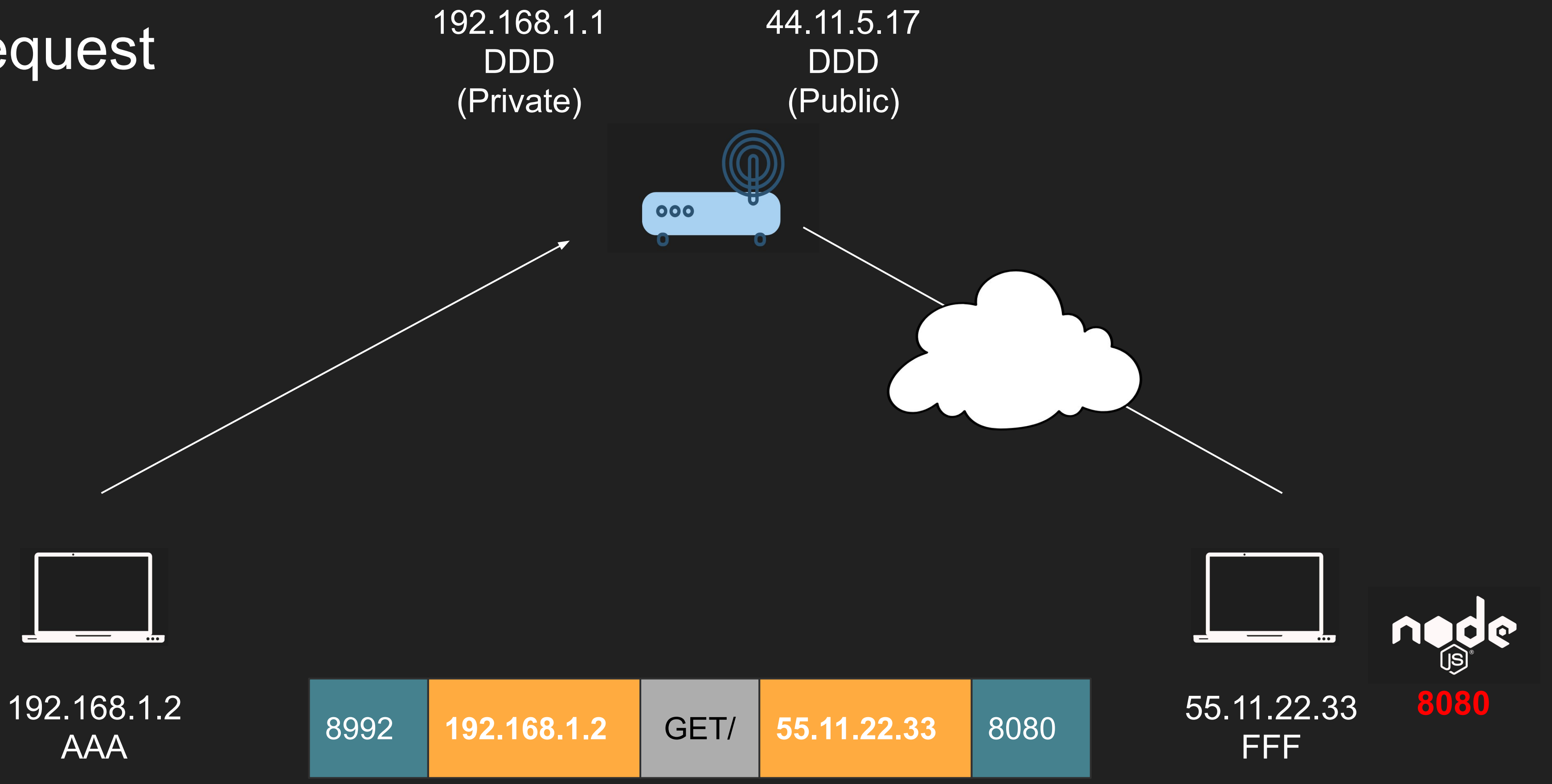
Response


192.168.1.1
DDD
192.168.1.2
AAA
8080
192.168.1.4
JSON
192.168.1.2
8992
192.168.1.4
CCC
8080

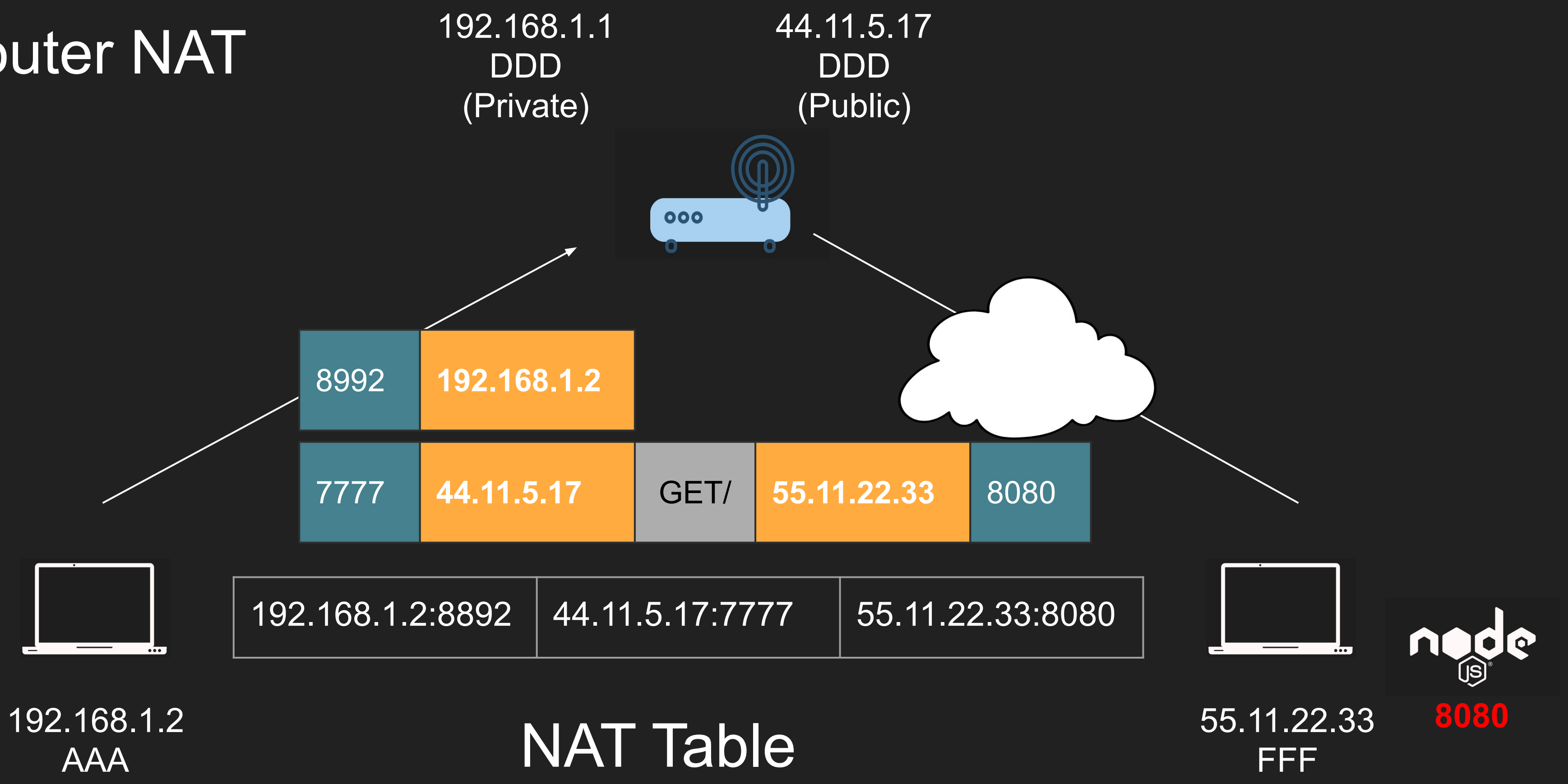
NAT


192.168.1.1
DDD
(Private)
44.11.5.17
DDD
(Public)
192.168.1.2
AAA
55.11.22.33
FFF
node
JS
8080

Request


192.168.1.1
DDD
(Private)
44.11.5.17
DDD
(Public)
192.168.1.2
AAA
8992
192.168.1.2
GET/
55.11.22.33
8080
55.11.22.33
FFF
8080

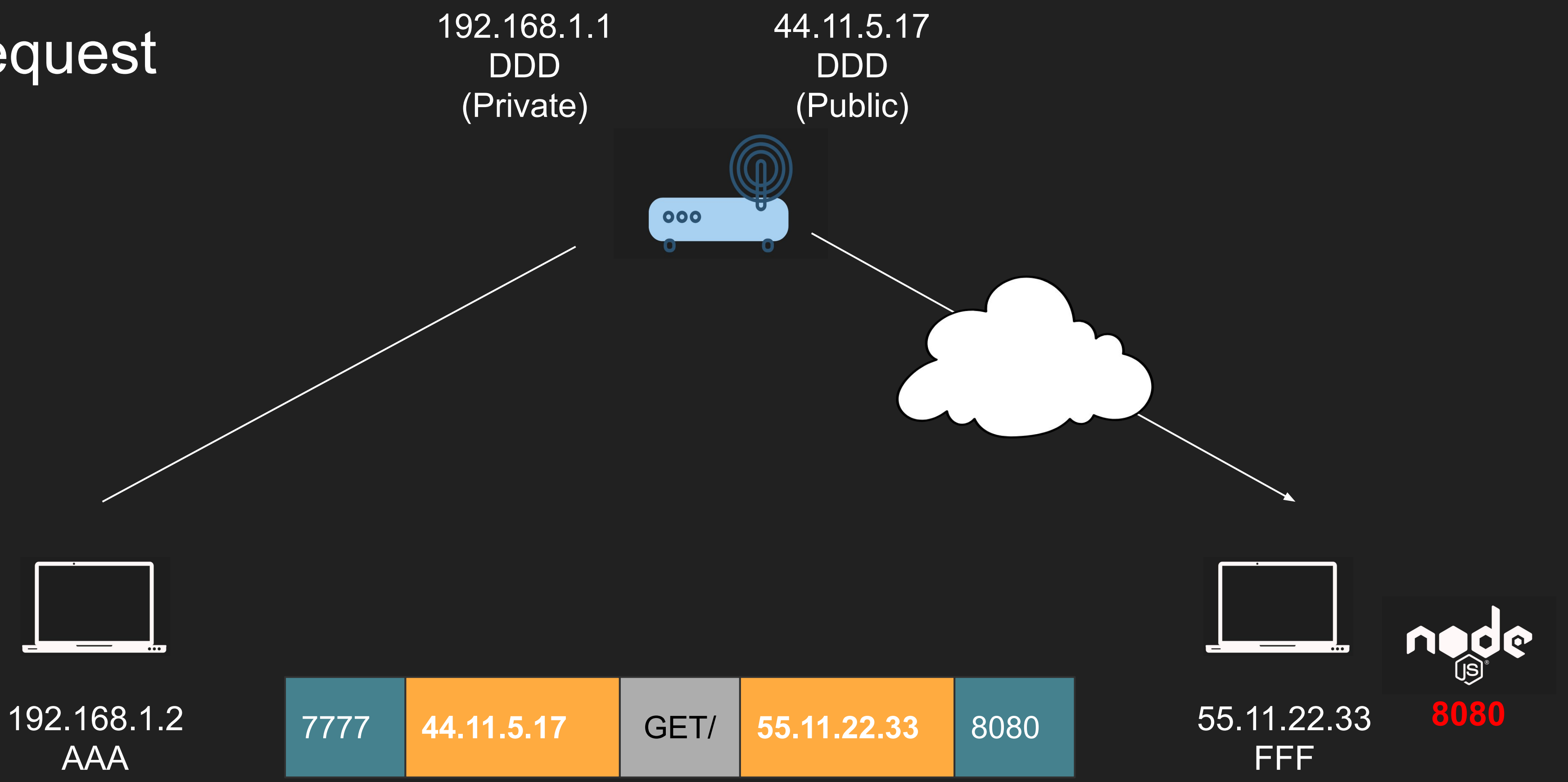
Router NAT



192.168.1.2:8892	44.11.5.17:7777	55.11.22.33:8080

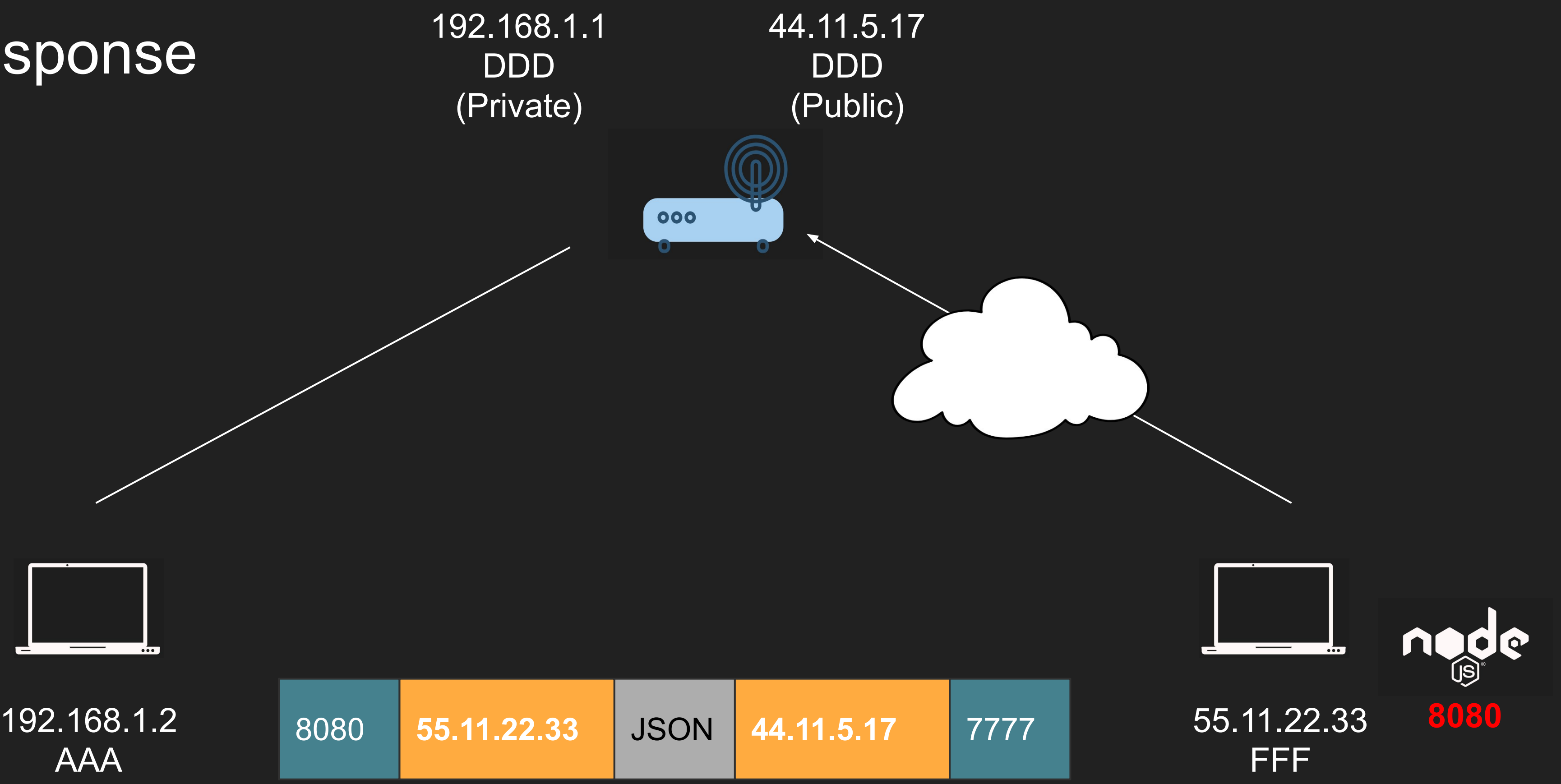
NAT Table

Request

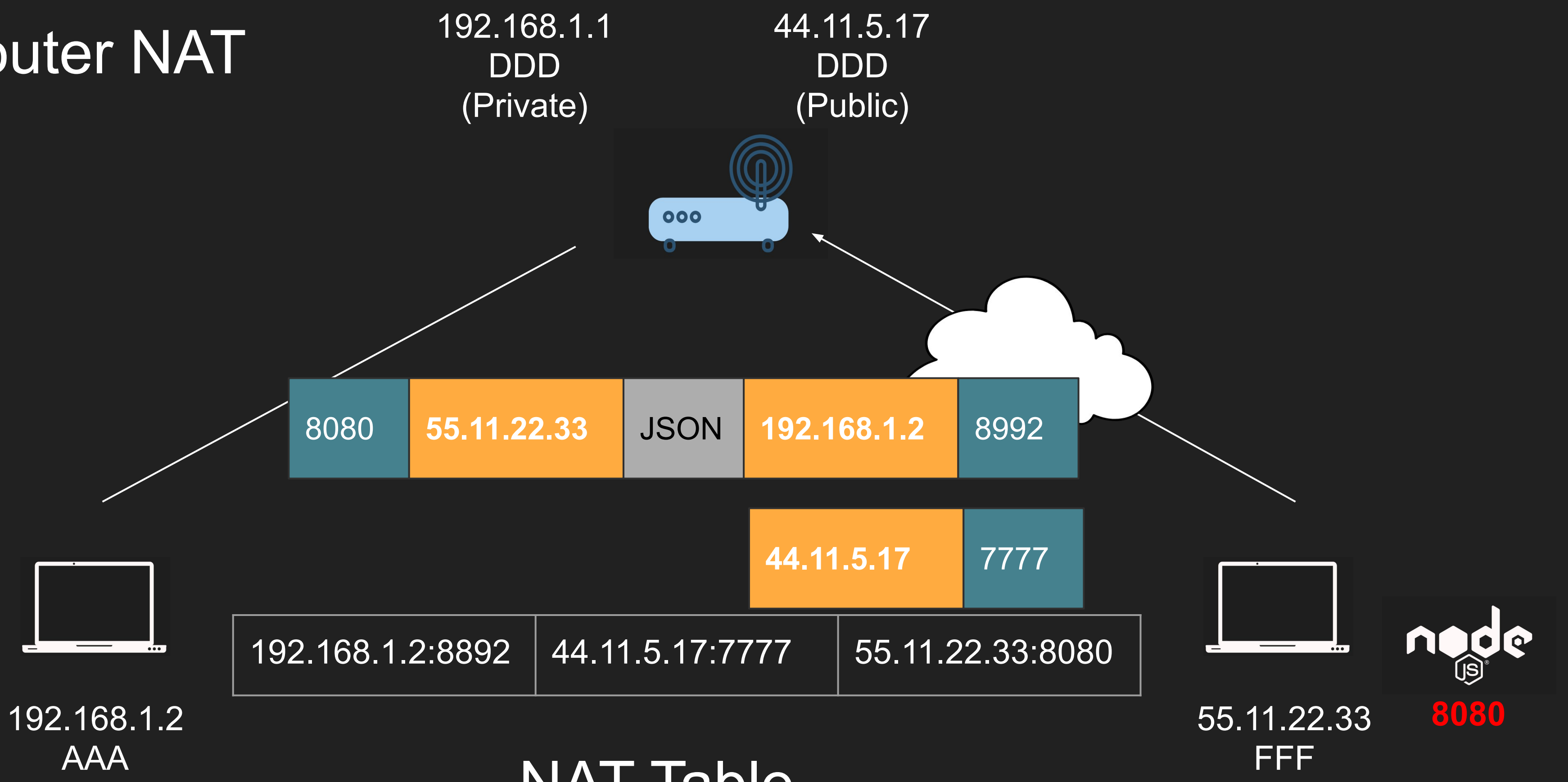


192.168.1.1
DDD
(Private)
44.11.5.17
DDD
(Public)
192.168.1.2
AAA
7777
44.11.5.17
GET/
55.11.22.33
8080
55.11.22.33
FFF
node JS
8080

Response

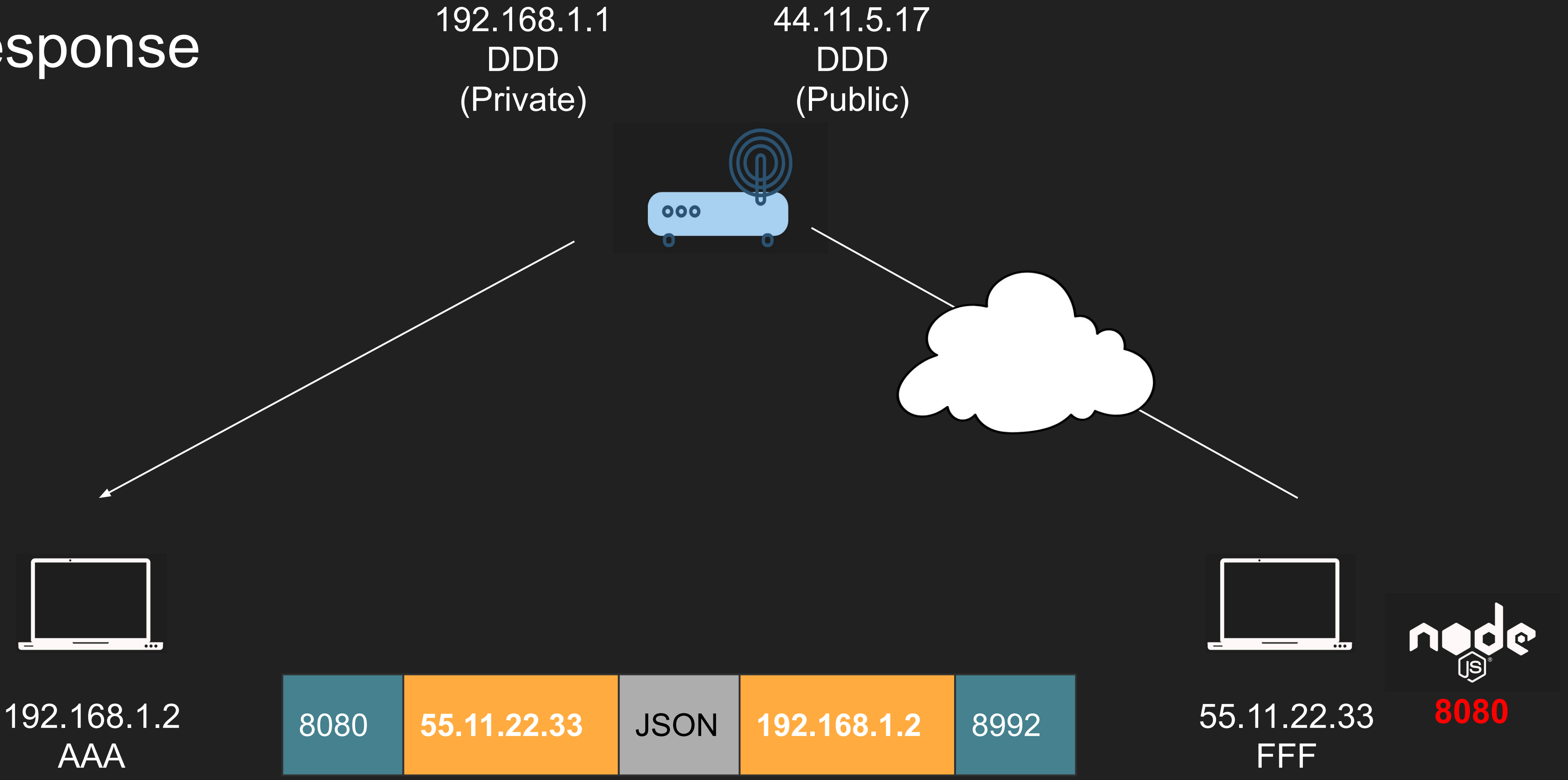

192.168.1.1
DDD
(Private)
44.11.5.17
DDD
(Public)
192.168.1.2
AAA
8080
55.11.22.33
JSON
44.11.5.17
7777
55.11.22.33
FFF
8080

Router NAT



NAT Table

Response


192.168.1.1
DDD
(Private)
44.11.5.17
DDD
(Public)
192.168.1.2
AAA
8080
55.11.22.33
JSON
192.168.1.2
8992
55.11.22.33
FFF
8080

NAT Applications

- Private to Public translations
  - So we don't run out IPv4
- Port forwarding
  - Add a NAT entry in the router to forward packets to 80 to a machine in your LAN
  - No need to have root access to listen on port 80 on your device
  - Expose your local web server publically
- Layer 4 Load Balancing
  - HAProxy NAT Mode - Your load balancer is your gateway
  - Clients send a request to a bogus service IP
  - Router intercepts that packet and replaces the service IP with a destination server
  - Layer 4 reverse proxying

Summary

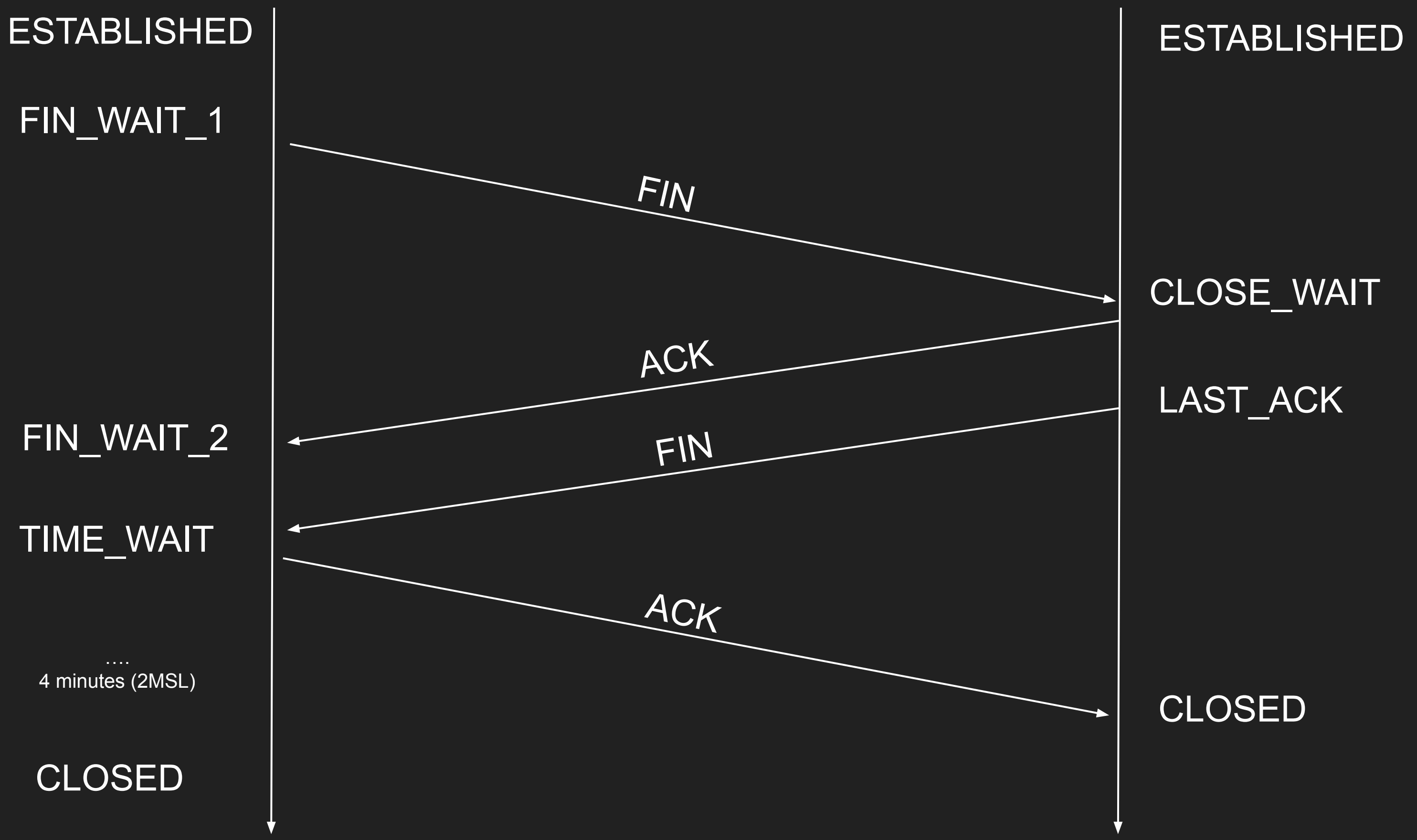
- IPv4 is limited only 4 billion
- Need to translate private to public
- Port forward/load balancing

TCP Connection States

Stateful protocol must have states

TCP Connection States

- TCP is a stateful protocol
- Both client and server need to maintain all sorts of state
- Window sizes, sequences and the state of the connection
- The connection goes through many states


ESTABLISHED
ESTABLISHED
FIN_WAIT_1
FIN
CLOSE_WAIT
ACK
LAST_ACK
FIN_WAIT_2
FIN
TIME_WAIT
ACK
....
4 minutes (2MSL)
CLOSED
CLOSED

TCP Pros and Cons

The power and drawbacks of TCP

TCP Pros

- Guarantee delivery
- No one can send data without prior knowledge
- Flow Control and Congestion Control
- Ordered Packets no corruption or app level work
- Secure and can’t be easily spoofed

TCP Cons

- Large header overhead compared to UDP
- More bandwidth
- Stateful - consumes memory on server and client
- Considered high latency for certain workloads (Slow start/ congestion/ acks)
- Does too much at a low level (hence QUIC)
  - Single connection to send multiple streams of data (HTTP requests)
  - Stream 1 has nothing to do with Stream 2
  - Both Stream 1 and Stream 2 packets must arrive
- TCP Meltdown
  - Not a good candidate for VPN

Networking Concepts for Effective Backend Applications

MSS/MTU and Path MTU

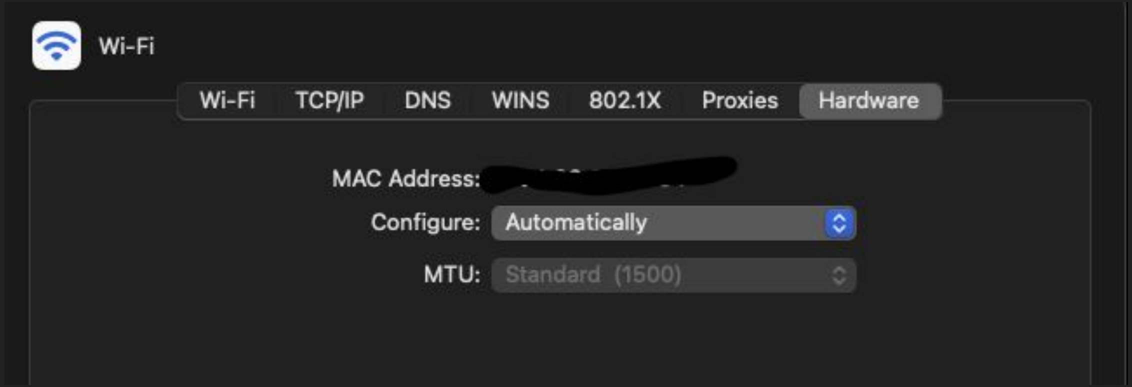
How large the packet can get

Overview

- TCP layer 4 unit is segment
- The segment slides into an IP Packet in layer 3
- The IP Packet now has the segment + headers
- The IP Packet slides into a layer 2 frame
- The frame has a fixed size based on the networking configuration.
- The size of the frame determines the size of the segment

Hardware MTU

- Maximum Transmission Unit (MTU) is the size of the frame
- It is a network interface property default 1500
- Some networks have jumbo frames up to 9000 bytes
- Are there are networks with larger MTUs?

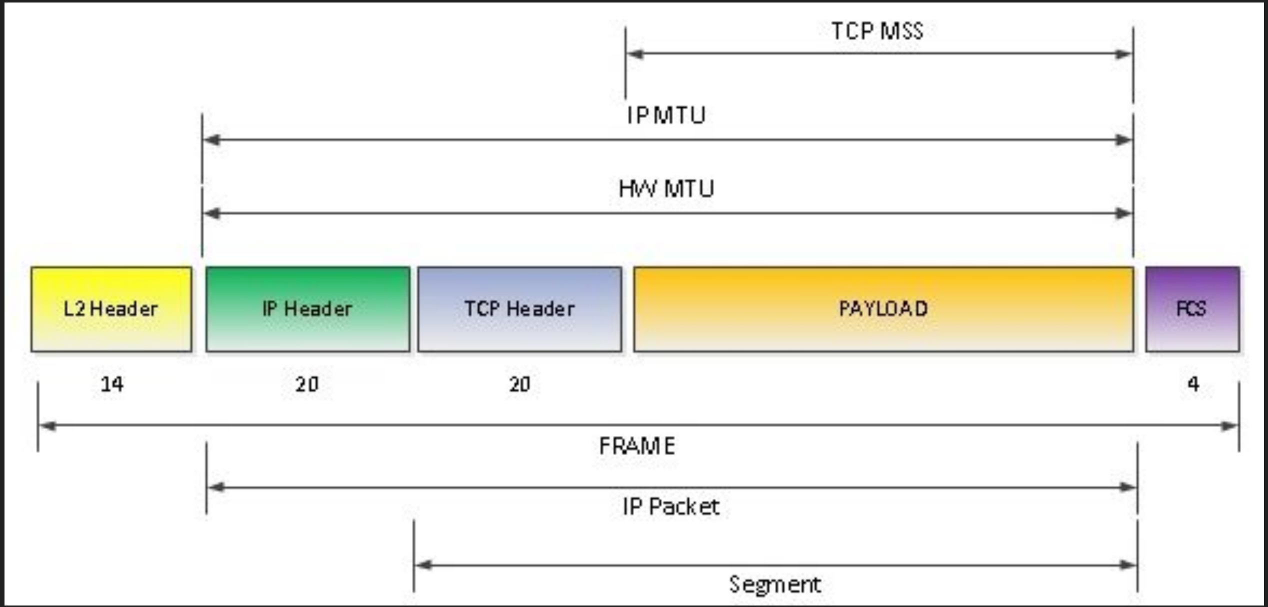


IP Packets and MTU

- The IP MTU usually equals the Hardware MTU
- One IP packet “should” fit a single frame
- Unless IP fragmentation is in place
- Larger IP Packets will be fragmented into multiple frames

MSS

- Maximum Segment size is determined based on MTU
- Segment must fit in an IP packet which “should” fit in a frame
- MSS = MTU - IP Headers - TCP Headers
- MSS = 1500 - 20 - 20 = 1460
- If you are sending 1460 bytes exactly that will fit nicely into a single MSS
- Which fits in a single frame

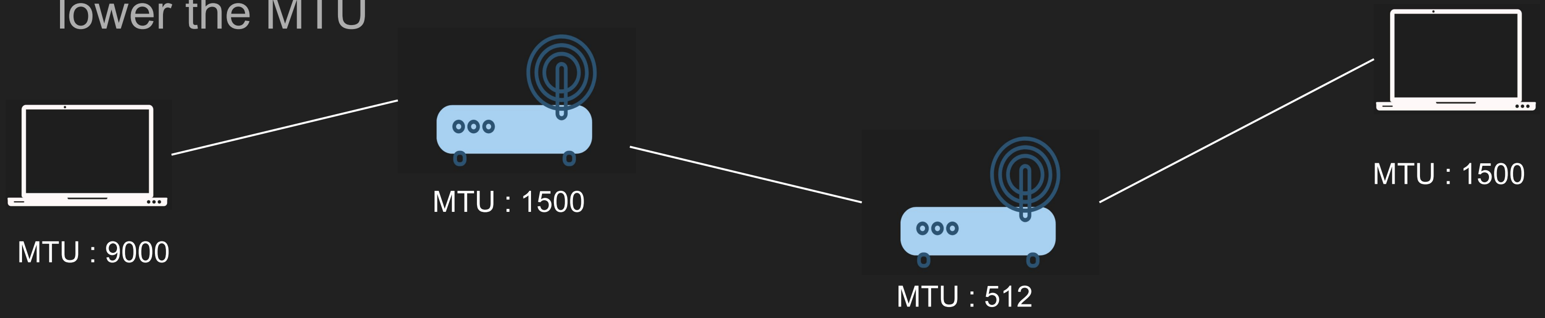


Credit Cisco

https://learningnetwork.cisco.com/s/question/0D53i00000Kt7CXCAZ/mtu-vs-pdu

Path MTU Discovery (PMTUD)

- MTU is network interface property each host can have different value
- You really need to use the smallest MTU in the network
- Path MTU help determine the MTU in the network path
- Client sends a IP packet with its MTU with a DF flag
- The host that their MTU is smaller will have to fragment but can't
- The host sends back an ICMP message fragmentation needed which will lower the MTU



Summary

- MTU is the maximum transmission unit on the device
- MSS is the maximum segment size at layer 4
- If you can fit more data into a single segment you lower latency
- It lowers overhead from headers and processing
- Path MTU can discover the network lowest MTU with ICMP
- Flow control/congestion control still allows sending multiple segments without ack

Nagle's algorithm

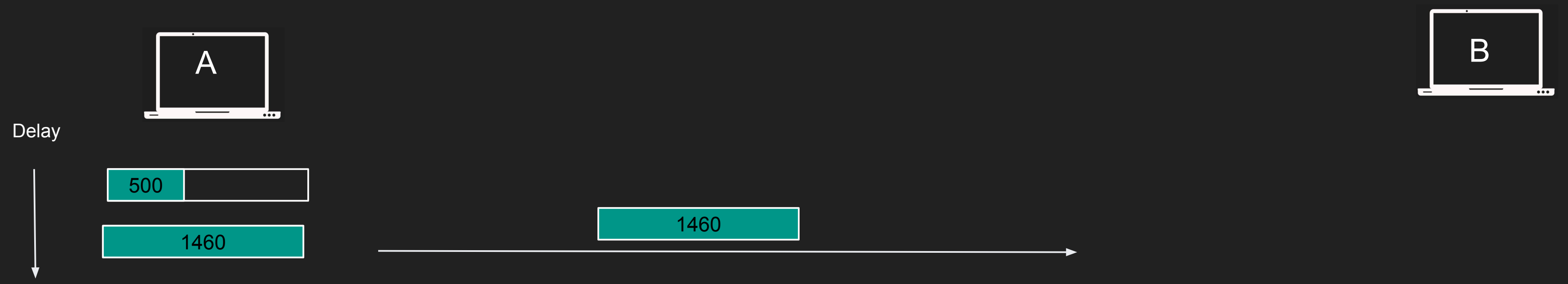
Delay in the client side

Nigel Algorithm

- In the telnet days sending a single byte in a segment is a waste
- Combine small segments and send them in a single one
- The client can wait for a full MSS before sending the segment
- No wasted 40 bytes header (IP + TCP) for few bytes of data

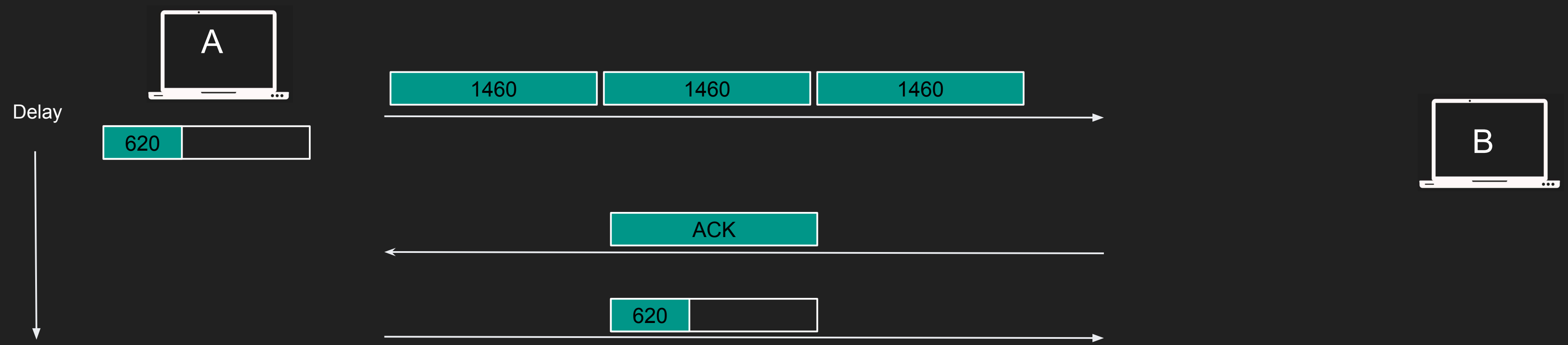
Nagle's algorithm

- Assume MSS = 1460, A sends 500 bytes
- 500 < 1460 client waits to fill the segment
- A sends 960 bytes, segment fills and send
- If there isn’t anything to ACK data will be immediately sent



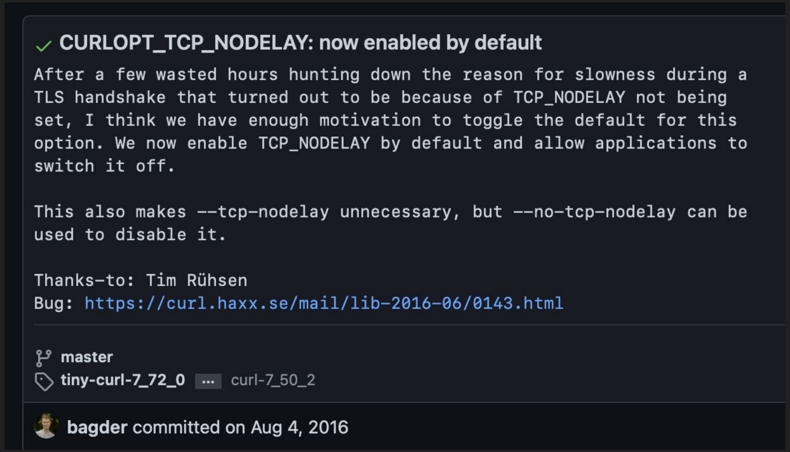
Problem with Nagle's algorithm

- Sending large data causes delay
- A want to send 5000 bytes on 1460 MSS
- 3 full segments of 1460 with 620 bytes
- 4th segment is not sent!
- 4th not full segment are only sent when an ACK is received



Disabling Nagle's algorithm

- Most clients today disable Nagle's algorithm
- I rather get performance than small bandwidth
- TCP_NODELAY
- Curl disabled this back in 2016 by default because TLS handshake was slowed down
- https://github.com/curl/curl/commit/4732ca5724072f132876f520c8f02c7c5b654d9


CURLOPT_TCP_NODELAY: now enabled by default

After a few wasted hours hunting down the reason for slowness during a TLS handshake that turned out to be because of TCP_NODELAY not being set, I think we have enough motivation to toggle the default for this option. We now enable TCP_NODELAY by default and allow applications to switch it off.

This also makes --tcp-nodelay unnecessary, but --no-tcp-nodelay can be used to disable it.

Thanks-to: Tim Rühsen
Bug: https://curl.haxx.se/mail/lib-2016-06/0143.html

master
tiny-curl-7_72_0 ... curl-7_50_2

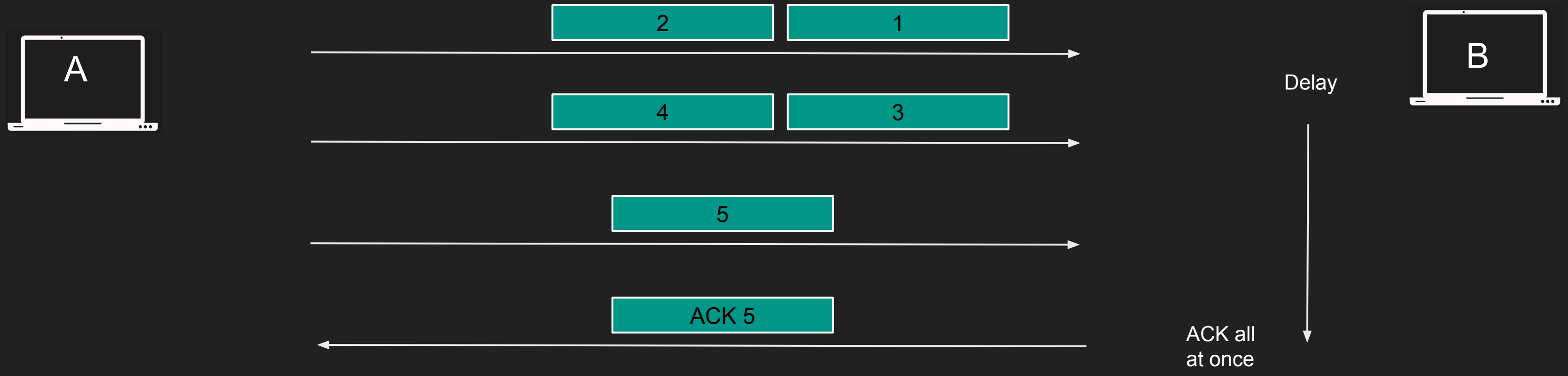
bagder committed on Aug 4, 2016

Delayed Acknowledgement

Less packets are good but performance is better

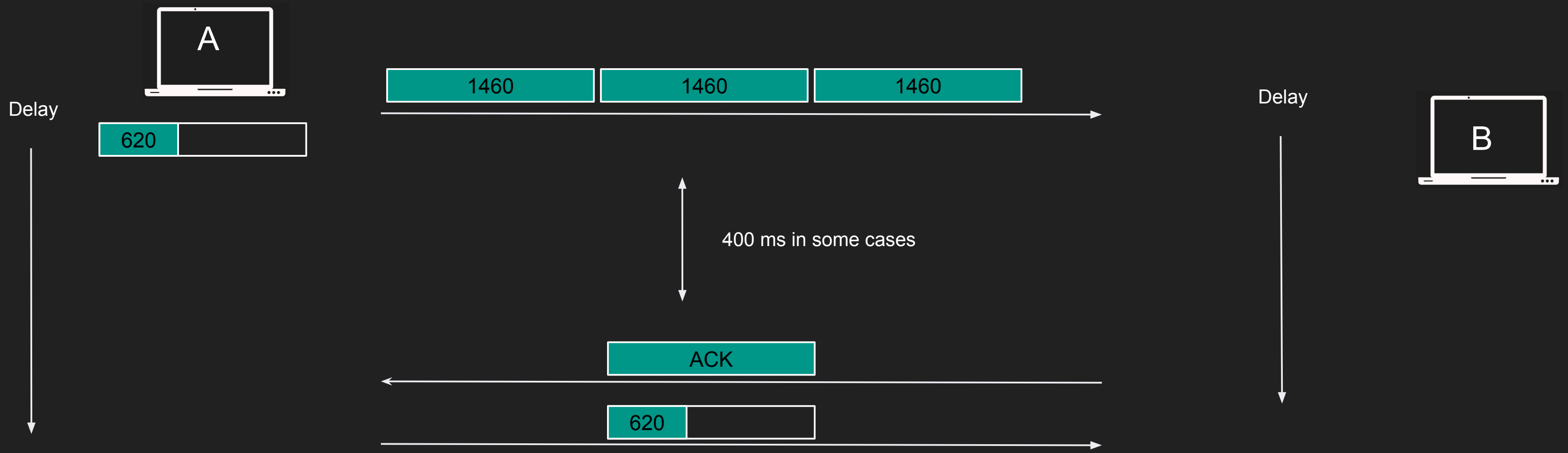
Delayed Acknowledgment algorithm

- Waste to acknowledge segments right away
- We can wait little more to receive more segment and ack once



Problem with delayed ACK

- Causes delays in some clients that may lead to timeout and retransmission
- Noticeable performance degradation
- Combined with Nagle's algorithm can lead to 400ms delays!
- Each party is waiting on each other



Disabling Delayed acknowledgement algorithm

- Disable delayed ack algorithm can be done with TCP_QUICKACK option
- Segments will be acknowledged “quicker”

The Cost of Connections

Understanding the cost of connections

Connection establishment is costly

- TCP three way handshake
- The further apart the peers, the slower it is to send segments
- Slow start keeps the connection from reaching its potential right away
- Congestion control and flow control limit that further
- Delayed and Nigel algorithm can further slow down
- Destroying the connection is also expensive

Connection Pooling

- Most implementation database backends and reverse proxies use pooling
- Establish a bunch of TCP connection to the backend and keep them running!
- Any request that comes to the backend use an already opened connection
- This way your connections will be “warm” and slow start would have already kicked in
- Don’t close the connection unless you absolutely don’t need it

Eager vs Lazy Loading

- Depending on what paradigm you take you can save on resources
- Eager loading -> Load everything and keep it ready
  - Start up is slow but requests will be served immediately
  - Some apps send warm up data to kick in the slow start but be careful of bandwidth and scalability
- Lazy Loading -> only load things on demand
  - Start up is fast but requests will suffer initially

TCP Fast Open

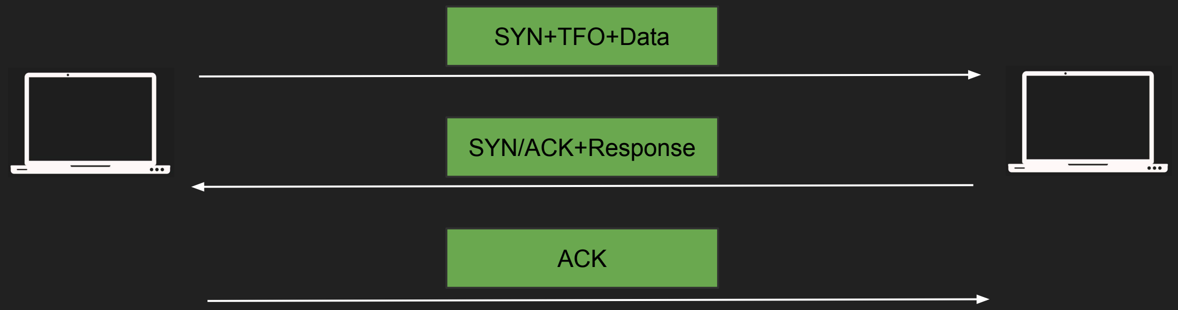
Wait I can send data during the handshake?

Handshake is Slow

- We know it, the handshake is slow
- I already know the server I have established a connection prior
- Can we use a predetermined token to send data immediately during the handshake?
- Meet TCP Fast open

TCP Fast Open (TFO)

- Client and Server establishes connection 1, server sends an encrypted cookie
- Client stores the TFO cookie.
- Client want to create another connection
- Client sends SYN, data and TFO cookie in TCP options
- Server authenticate the cookie and sends response + SYN/ACK



TCP Fast Open (TFO)

- TFO is enabled by default in linux 3.13 >
- You can enable TFO in curl --tcp-fastopen
- Goes without saying, you still get TCP Slow start with TCP Fast open
- You can take advantage of this feature to send early data

Listening Server

Understanding what to listen on

Listening

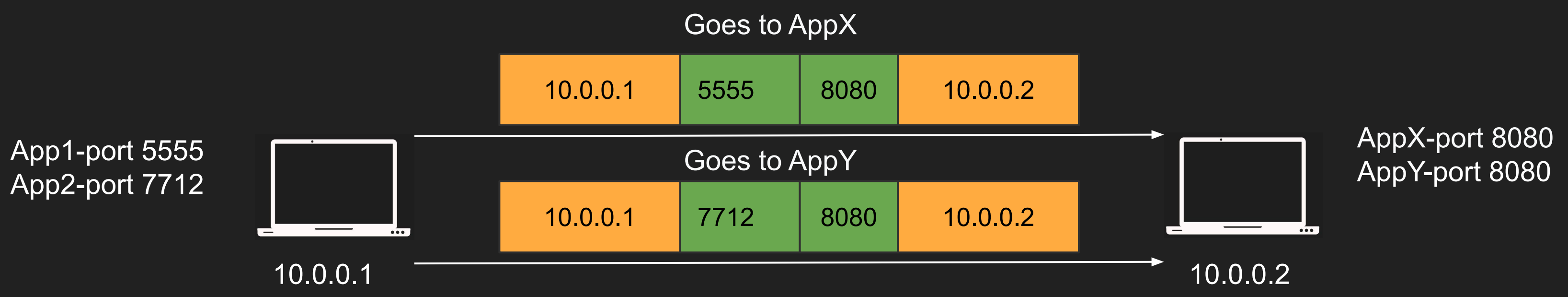
- You create a server by listening on a port on a specific ip address
- Your machine might have multiple interfaces with multiple IP address
- listen(127.0.0.1, 8080) -> listens on the local host ipv4 interface on port 8080
- listen(::1, 8080) -> listens on localhost ipv6 interface on port 8080
- listen(192.168.1.2, 8080) -> listens on 192.168.1.2 on port 8080
- listen(0.0.0.0, 8080) -> listens on all interfaces on port 8080 (can be dangerous)

Listening

- You can only have one process in a host listening on IP/Port
- No two processes can listen on the same port
- P1->Listen(127.0.0.1,8080)
- P2->Listen(127.0.0.1,8080) error

There is always an exception

- There is a configuration that allows more than one process to listen on the same port
- SO_PORTREUSE
- Operating systems balance segments among processes
- OS creates a hash source ip/source port/dest ip/ dest port
- Guarantees always go to the same process if the pair match

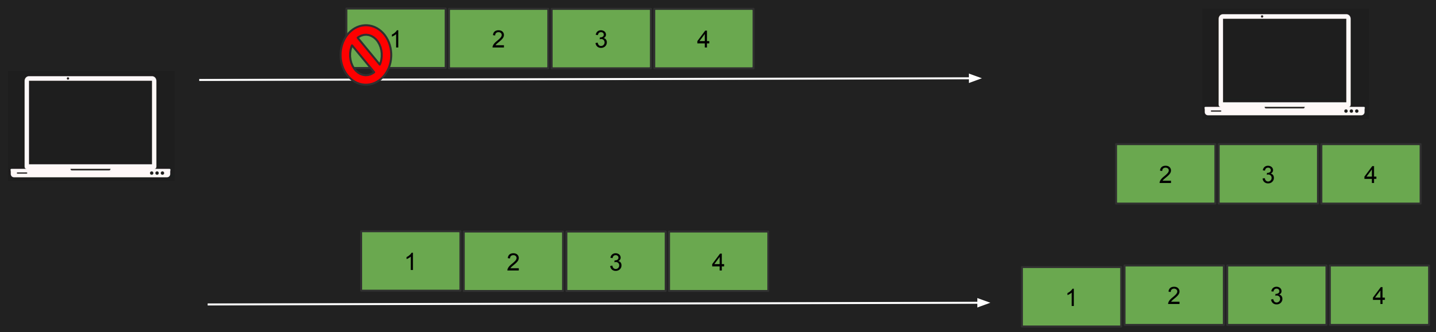


TCP HOL

Head of line blocking

TCP head of line blocking

- TCP orders packets in the order they are sent
- The segments are not acknowledged or delivered to the app until they are in order
- This is great! But what if multiple clients are using the same connection



TCP head of line blocking

- HTTP requests may use the same connection to send multiple requests
- Request 1 is segments 1,2
- Request 2 is segments 3,4
- Segments 2,3,4 arrive but 1 is lost?
- Request 2 technically was delivered but TCP is blocking it
- Huge latency in apps, big problem in HTTP/2 with streams
- QUIC solves this