THE ANATOMY OF AN HTTP REQUEST

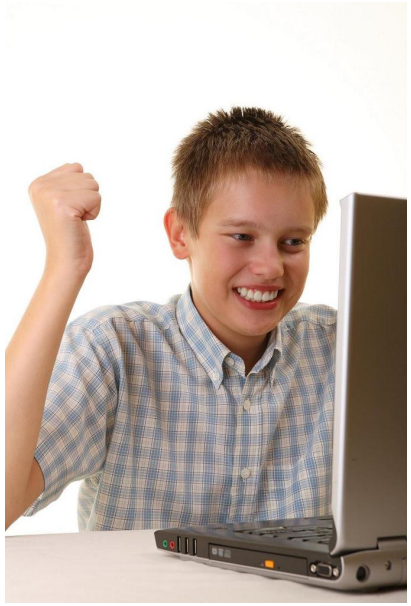
WHO AM I?

ADDISON HIGHAM

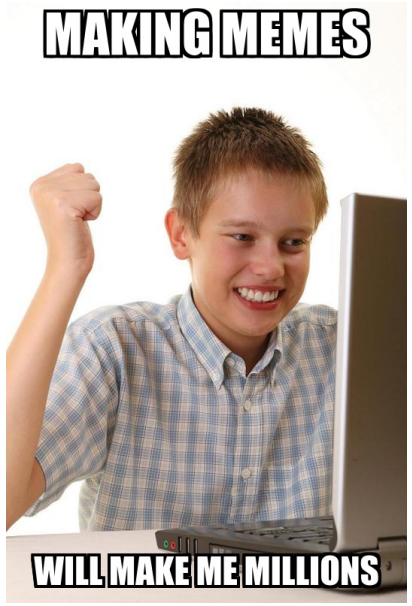
Shared services nomad

LET'S REALLY UNDERSTAND HOW HTTP WORKS

MEME GENERATION AS A SERVICE



+ Bottom Text + Top Text =



HTTP

HYPERTEXT TRANSFER PROTOCOL

WHAT HTTP IS

WHAT HTTP IS

- Gives us nice semantics for requests and responses

WHAT HTTP IS

- Gives us nice semantics for requests and responses
- Verbs, Paths and Headers give us a way of modeling APIs

WHAT A REQUEST LOOKS LIKE

```
POST /meme HTTP/1.1
```

This isn't *completely* accurate, HTTP uses a `\r\n` as a newline

WHAT A REQUEST LOOKS LIKE

```
POST /meme HTTP/1.1
Host: www.memegen.tld
Content-Type: application/json
Content-Length: 2048


```

This isn't *completely* accurate, HTTP uses a `\r\n` as a newline

WHAT A REQUEST LOOKS LIKE

```
POST /meme HTTP/1.1
Host: www.memegen.tld
Content-Type: application/json
Content-Length: 2048

{"image": "...", "topText": "My First Meme", "bottomText": "Isn't Really Good"}
```

This isn't *completely* accurate, HTTP uses a `\r\n` as a newline

AND A RESPONSE

```
HTTP/1.1 200 OK
```

AND A RESPONSE

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 2048
```

AND A RESPONSE

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 2048

/9j/4AAQSkZJRgABAQEAYABgAAD/2wBDAAgGBgcGBQgHBwcJCQgKDBQ...
```

IN CODE

```
const net = require('net')
const jsonBody = '{"image": ...}'
const headers = [ 'POST /meme HTTP/1.1', ...]
```

IN CODE

```
const net = require('net')
const jsonBody = '{"image": ...}'
const headers = [ 'POST /meme HTTP/1.1', ...]

const socket = net.connect(80, 'www.memegen.tld')
socket.on('connect', () => {
```

IN CODE

```
const net = require('net')
const jsonBody = '{"image": ...}'
const headers = [ 'POST /meme HTTP/1.1', ...]

const socket = net.connect(80, 'www.memegen.tld')
socket.on('connect', () => {
  headers.forEach((header) => socket.write(header + '\r\n')
  socket.write('\r\n')
  socket.write(jsonBody)
})
```

IN CODE

```
const net = require('net')
const jsonBody = '{"image": ...}'
const headers = [ 'POST /meme HTTP/1.1', ...]

const socket = net.connect(80, 'www.memegen.tld')
socket.on('connect', () => {
  headers.forEach((header) => socket.write(header + '\r\n')
  socket.write('\r\n')
  socket.write(jsonBody)
})
socket.on('data', (d) => console.log(d))
socket.on('close', () => console.log('\n\nfinished!'))
```

IN SUMMARY...

HTTP is a message orientied protocol that takes a verb, an action, headers, and a body as a request.

It sends that data to a server, and the server returns with a response that includes a status code, headers, and a body.

THAT'S IT!

HTTP IS SO SIMPLE!

RIGHT?

WELL... A FEW QUESTIONS STILL REMAIN

RIGHT?

WELL... A FEW QUESTIONS STILL REMAIN

- What is that `socket` objects and why do we wait for a connection?

RIGHT?

WELL... A FEW QUESTIONS STILL REMAIN

- What is that `socket` objects and why do we wait for a connection?
- How do we find the server we want to talk to?

RIGHT?

WELL... A FEW QUESTIONS STILL REMAIN

- What is that `socket` objects and why do we wait for a connection?
- How do we find the server we want to talk to?
- Can we really send very large images this way with no changes?

RIGHT?

WELL... A FEW QUESTIONS STILL REMAIN

- What is that `socket` objects and why do we wait for a connection?
- How do we find the server we want to talk to?
- Can we really send very large images this way with no changes?
- How about missed messages, errors, corrupted data, etc?

RIGHT?

WELL... A FEW QUESTIONS STILL REMAIN

- What is that `socket` objects and why do we wait for a connection?
- How do we find the server we want to talk to?
- Can we really send very large images this way with no changes?
- How about missed messages, errors, corrupted data, etc?
- How does the data actually get sent over the internet?

TCP

TRANSMISSION CONTROL PROTOCOL

HTTP ASSUMES, BUT DOES NOT DIRECTLY CARE ABOUT

- Messages actually arriviving
- Messages remaining free of corruption
- How messages get sent over the internet

WHAT TCP IS

WHAT TCP IS

- Provides a persistent connection that we can logically reason about

WHAT TCP IS

- Provides a persistent connection that we can logically reason about
- Doesn't care about messages or formats, it is just streams of bytes

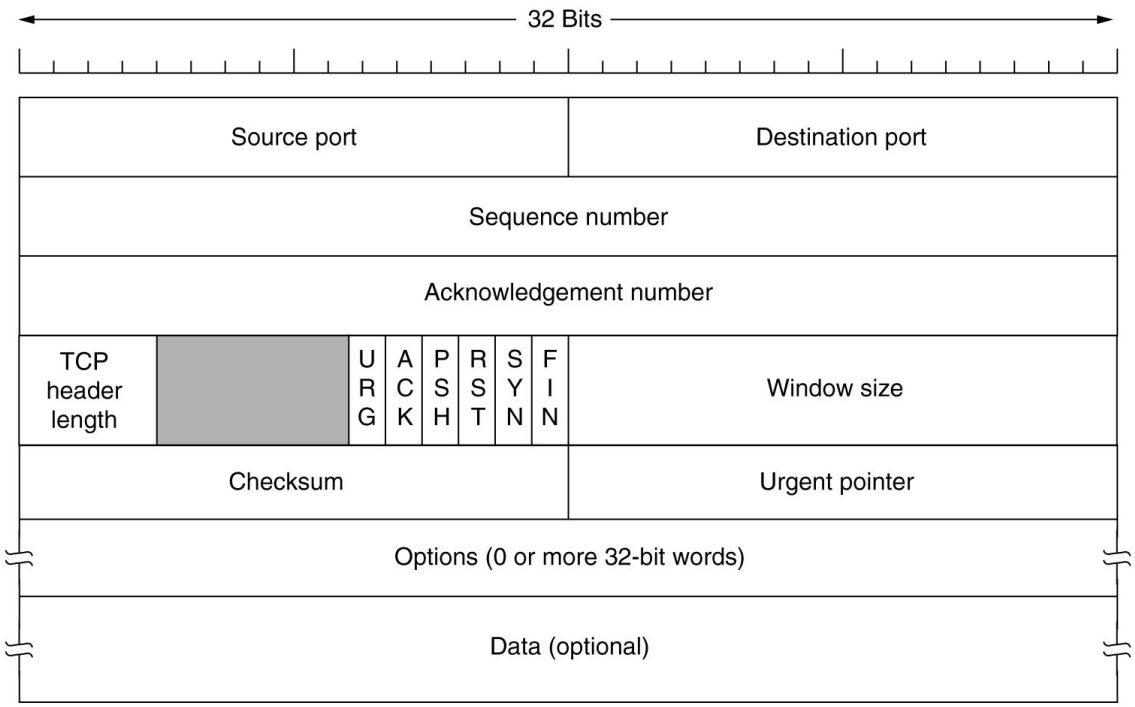
WHAT TCP IS

- Provides a persistent connection that we can logically reason about
- Doesn't care about messages or formats, it is just streams of bytes
- Ensures those streams of bytes all arrive in order and tries to detect corruption

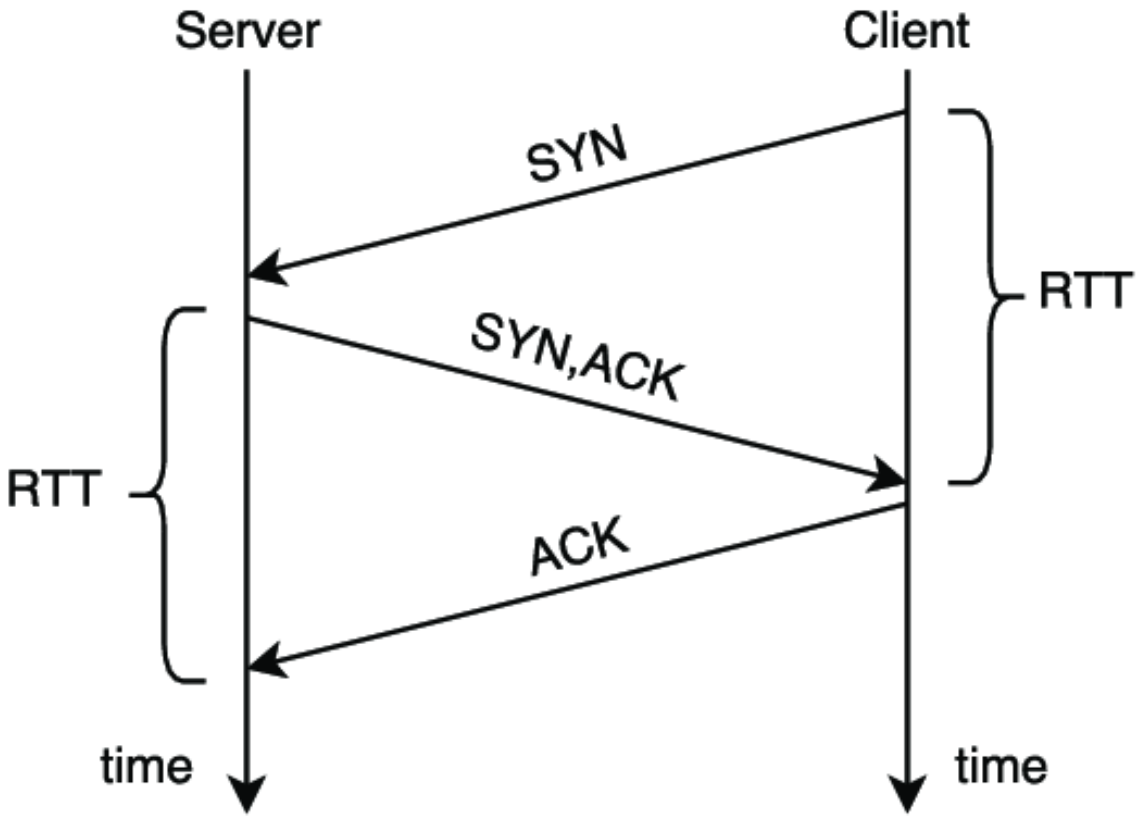
WHAT TCP IS

- Provides a persistent connection that we can logically reason about
- Doesn't care about messages or formats, it is just streams of bytes
- Ensures those streams of bytes all arrive in order and tries to detect corruption
- Doesn't care about who is talking and when (full-duplex)

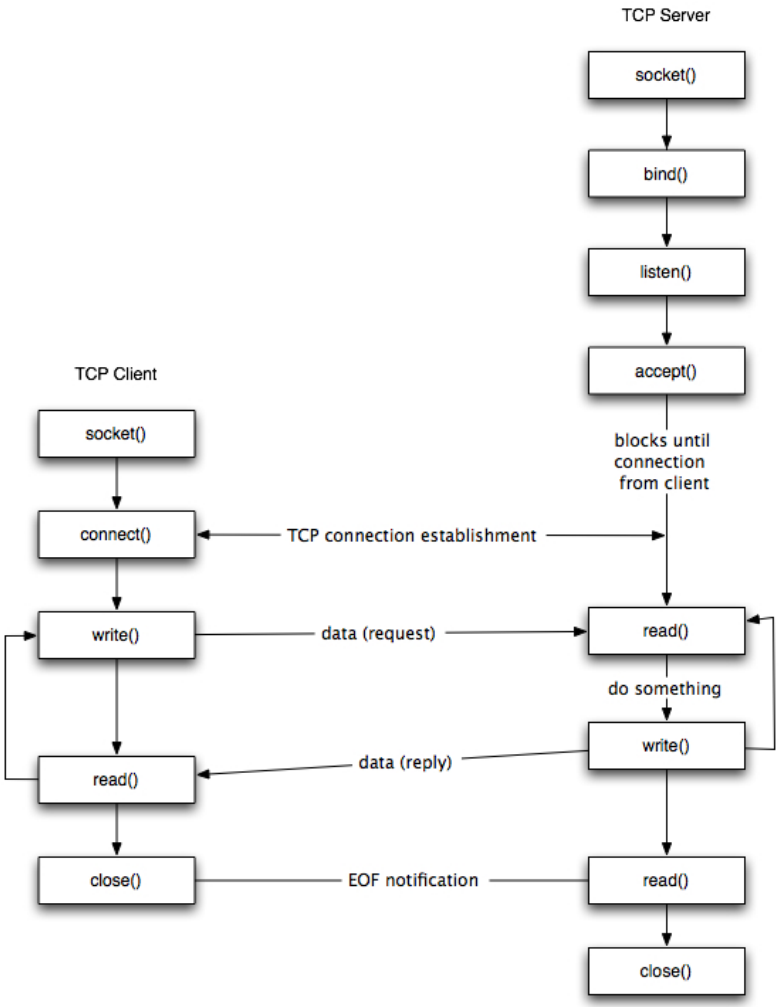
THE TCP DATAGRAM



THE TCP SESSION



THE TCP SOCKET



IN CODE

Well... at least the first packet of the handshake...

```
const raw = require('raw-socket')
const ip = require('ip')

function sendSyn(srcIp, srcPort, destIp, destPort) {
  const socket = raw.createSocket({protocol: raw.Protocol.TCP, addressFamily: raw.AddressFamily.IPv4})
```

IN CODE

Well... at least the first packet of the handshake...

```
const raw = require('raw-socket')
const ip = require('ip')

function sendSyn(srcIp, srcPort, destIp, destPort) {
  const socket = raw.createSocket({protocol: raw.Protocol.TCP, addressFamily: raw.AddressFamily.IPv4})
  const tcpBuffer = Buffer.from([
    0x00,0x00,              // TCP: src port (should be random)
    0x00,0x00,              // TCP: dst port (should be the port you want to connect to)
    0x00,0x00,0x00,0x00,    // TCP: sequence number (should be random)
    0x00,0x00,0x00,0x00,    // TCP: acquitment number (must be null because WE are intiating the SYN, static value)
    0x00,0x02,              // TCP: header length (data offset) && flags (fin=1,syn=2,rst=4,psh=8,ack=16,urg=32, static value)
    0x72,0x10,              // TCP: window
    0x00,0x00,              // TCP: checksum for TCP part of this packet)
    0x00,0x00,              // TCP: ptr urgent
    0x02,0x04,              // TCP: options
    0x05,0xb4,              // TCP: padding (mss=1460, static value)
    0x04,0x02,              // TCP: SACK Permitted (4) Option
    0x08,0x0a,              // TCP: TSval, Length
      0x01,0x75,0xdd,0xe8,  // value
      0x00,0x00,0x00,0x00,  // TSecr
    0x01,                   // TCP: NOP
    0x03,0x03,0x07          // TCP: Window scale
 ])

  tcpBuffer.writeUInt32BE(parseInt(Math.random()*0xffffffff), 4) // TCP: create random sequence number
  tcpBuffer.writeUInt8(tcpBuffer.length << 2, 12) // TCP: write Header Length
  tcpBuffer.writeUInt16BE(srcPort, 0) // TCP: save src port into the buffer
  tcpBuffer.writeUInt16BE(destPort, 2) // TCP: save dst port into the buffer
```

IN CODE

Well... at least the first packet of the handshake...

```
const raw = require('raw-socket')
const ip = require('ip')

function sendSyn(srcIp, srcPort, destIp, destPort) {
  const socket = raw.createSocket({protocol: raw.Protocol.TCP, addressFamily: raw.AddressFamily.IPv4})
  const tcpBuffer = Buffer.from([
    0x00,0x00,              // TCP: src port (should be random)
    0x00,0x00,              // TCP: dst port (should be the port you want to connect to)
    0x00,0x00,0x00,0x00,    // TCP: sequence number (should be random)
    0x00,0x00,0x00,0x00,    // TCP: acquitment number (must be null because WE are intiating the SYN, static value)
    0x00,0x02,              // TCP: header length (data offset) && flags (fin=1,syn=2,rst=4,psh=8,ack=16,urg=32, static value)
    0x72,0x10,              // TCP: window
    0x00,0x00,              // TCP: checksum for TCP part of this packet)
    0x00,0x00,              // TCP: ptr urgent
    0x02,0x04,              // TCP: options
    0x05,0xb4,              // TCP: padding (mss=1460, static value)
    0x04,0x02,              // TCP: SACK Permitted (4) Option
    0x08,0x0a,              // TCP: TSval, Length
      0x01,0x75,0xdd,0xe8,  // value
      0x00,0x00,0x00,0x00,  // TSecr
    0x01,                   // TCP: NOP
    0x03,0x03,0x07          // TCP: Window scale
 ])

  tcpBuffer.writeUInt32BE(parseInt(Math.random()*0xffffffff), 4) // TCP: create random sequence number
  tcpBuffer.writeUInt8(tcpBuffer.length << 2, 12) // TCP: write Header Length
  tcpBuffer.writeUInt16BE(srcPort, 0) // TCP: save src port into the buffer
  tcpBuffer.writeUInt16BE(destPort, 2) // TCP: save dst port into the buffer

  const pseudoBuffer = new Buffer.from([
    0x00,0x00,0x00,0x00,    // IP: ip src
    0x00,0x00,0x00,0x00,    // IP: ip dst
    0x00,
    0x06, // IP: protocol (ICMP=1, IGMP=2, TCP=6, UDP=17, static value)
    (tcpBuffer.length >>; 8) & 0xff, tcpBuffer.length & 0xff
  ])
  ip.toBuffer(srcIp, pseudoBuffer, 0) // IP: save ip src into the buffer
  ip.toBuffer(destIp, pseudoBuffer, 4) // IP: save ip dst into the buffer
  pseudoBuffer = Buffer.concat([pseudoBuffer, tcpBuffer])
```

IN CODE

Well... at least the first packet of the handshake...

```
const raw = require('raw-socket')
const ip = require('ip')

function sendSyn(srcIp, srcPort, destIp, destPort) {
  const socket = raw.createSocket({protocol: raw.Protocol.TCP, addressFamily: raw.AddressFamily.IPv4})
  const tcpBuffer = Buffer.from([
    0x00,0x00,              // TCP: src port (should be random)
    0x00,0x00,              // TCP: dst port (should be the port you want to connect to)
    0x00,0x00,0x00,0x00,    // TCP: sequence number (should be random)
    0x00,0x00,0x00,0x00,    // TCP: acquitment number (must be null because WE are intiating the SYN, static value)
    0x00,0x02,              // TCP: header length (data offset) && flags (fin=1,syn=2,rst=4,psh=8,ack=16,urg=32, static value)
    0x72,0x10,              // TCP: window
    0x00,0x00,              // TCP: checksum for TCP part of this packet)
    0x00,0x00,              // TCP: ptr urgent
    0x02,0x04,              // TCP: options
    0x05,0xb4,              // TCP: padding (mss=1460, static value)
    0x04,0x02,              // TCP: SACK Permitted (4) Option
    0x08,0x0a,              // TCP: TSval, Length
      0x01,0x75,0xdd,0xe8,  // value
      0x00,0x00,0x00,0x00,  // TSecr
    0x01,                   // TCP: NOP
    0x03,0x03,0x07          // TCP: Window scale
 ])

  tcpBuffer.writeUInt32BE(parseInt(Math.random()*0xffffffff), 4) // TCP: create random sequence number
  tcpBuffer.writeUInt8(tcpBuffer.length << 2, 12) // TCP: write Header Length
  tcpBuffer.writeUInt16BE(srcPort, 0) // TCP: save src port into the buffer
  tcpBuffer.writeUInt16BE(destPort, 2) // TCP: save dst port into the buffer

  const pseudoBuffer = new Buffer.from([
    0x00,0x00,0x00,0x00,    // IP: ip src
    0x00,0x00,0x00,0x00,    // IP: ip dst
    0x00,
    0x06, // IP: protocol (ICMP=1, IGMP=2, TCP=6, UDP=17, static value)
    (tcpBuffer.length >>; 8) & 0xff, tcpBuffer.length & 0xff
  ])
  ip.toBuffer(srcIp, pseudoBuffer, 0) // IP: save ip src into the buffer
  ip.toBuffer(destIp, pseudoBuffer, 4) // IP: save ip dst into the buffer
  pseudoBuffer = Buffer.concat([pseudoBuffer, tcpBuffer])

  raw.writeChecksum(tcpBuffer, 16, raw.createChecksum(pseudoBuffer))
  socket.send(tcpBuffer, 0, tcpBuffer.length, destIp)

  // and that was just to send the first packet...
}
```

WHERE DOES THAT GET US?

Now with a TCP session established we have:

- A way of reliably sending and recieving data
- Something that gurantees order and that we are reasonably confident hasn't been corrupted
- An API to program against that is easy to reason about

IN SUMMARY

HTTP is a message orientied protocol that is encapsulated by TCP, a connection-oriented protocol that allows for getting ordered streams of bytes.

TCP ensures that all messages arrive in order and can detect most corruption.

HTTP uses TCP to send messages that takes a verb, an action, headers, and a body as a request.

It sends that data to a server, and the server returns with a response that includes a status code, headers, and a body.

THAT'S IT!

HTTP AND TCP IS FAIRLY SIMPLE!

RIGHT?

WELL... WE STILL DON'T KNOW

RIGHT?

WELL... WE STILL DON'T KNOW

- How do we find the server we want to talk to?

RIGHT?

WELL... WE STILL DON'T KNOW

- How do we find the server we want to talk to?
- How does the data actually get sent over the internet?

IP

INTERNET PROTOCOL

TCP ASSUMES, BUT DOES NOT DIRECTLY CARE ABOUT

- Addressing
- Routing
- How TCP streams get sent over the internet

WHAT IP IS

WHAT IP IS

- Provides a mechanism for specifying the final destination of traffic

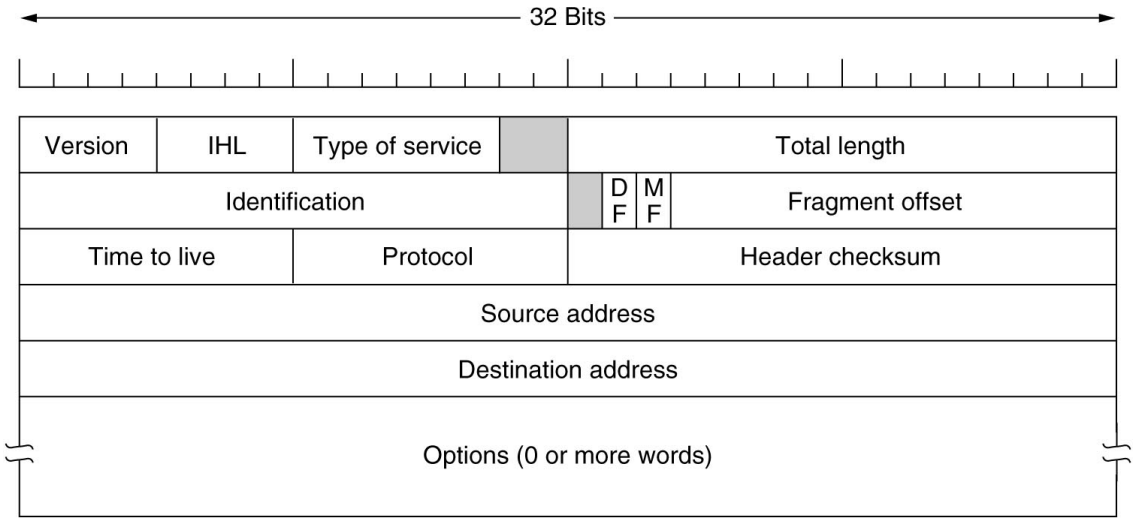
WHAT IP IS

- Provides a mechanism for specifying the final destination of traffic
- Gives us tools to reason about how to segment and route traffic

WHAT IP IS

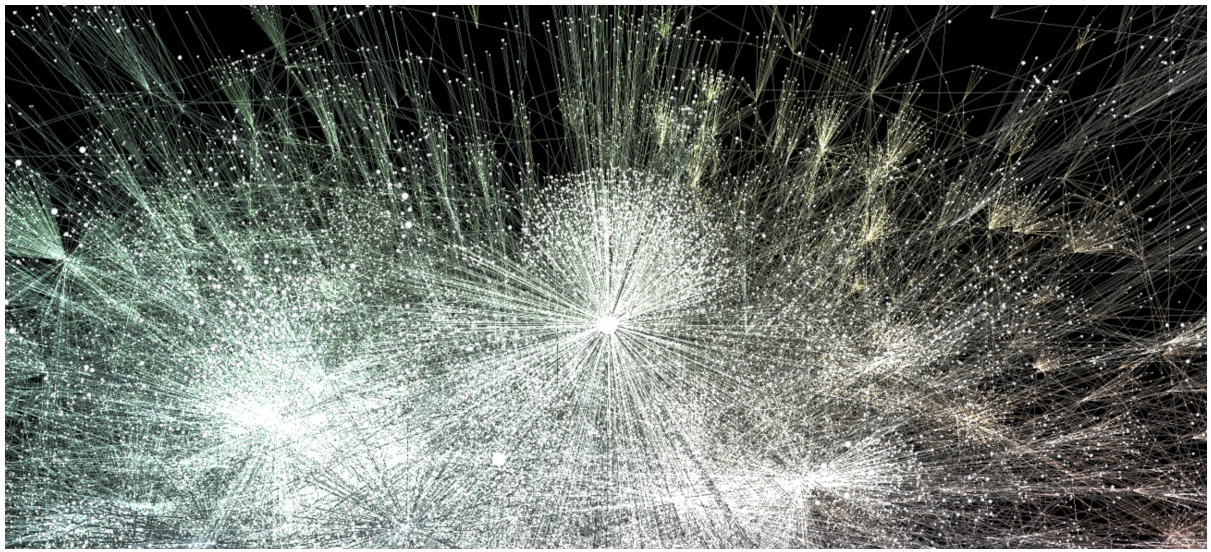
- Provides a mechanism for specifying the final destination of traffic
- Gives us tools to reason about how to segment and route traffic
- Does not care about connections or streams, you just send messages

THE IP PACKET



WHAT IS THE INTERNET ANYWAY?

We often think of the internet as a *network* but in reality, it is a recursive network, with networks inside networks inside networks



ADDRESSING

This network of networks has private networks, which share address space, and public networks that must have unique addresses

IP addresses give us a destination, but by themselves, they aren't enough to give us a path to get there

ASIDE: DNS

DNS is critical to the operation of the internet, but in some respects, it is a totally seperate system

If the internet is a library, then DNS is a card catalog

ROUTE TABLES

Provide information on the next hop

```
$ ip route
default via 10.0.8.1 dev enx00e11f0005f3 proto dhcp metric 100
default via 10.0.8.1 dev wlp2s0 proto dhcp metric 600
10.0.8.0/21 dev enx00e11f0005f3 proto kernel scope link src 10.0.13.191
10.0.8.0/21 dev wlp2s0 proto kernel scope link src 10.0.13.209 metric 60
169.254.0.0/16 dev enx00e11f0005f3 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdow
172.18.0.0/16 dev br-aaec6a8da6fc proto kernel scope link src 172.18.0.1
```

ROUTE TABLES

Provide information on the next hop

```
$ ip route
default via 10.0.8.1 dev enx00e11f0005f3 proto dhcp metric 100
default via 10.0.8.1 dev wlp2s0 proto dhcp metric 600
10.0.8.0/21 dev enx00e11f0005f3 proto kernel scope link src 10.0.13.191
10.0.8.0/21 dev wlp2s0 proto kernel scope link src 10.0.13.209 metric 60
169.254.0.0/16 dev enx00e11f0005f3 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdow
172.18.0.0/16 dev br-aaec6a8da6fc proto kernel scope link src 172.18.0.1
```

ROUTE TABLES

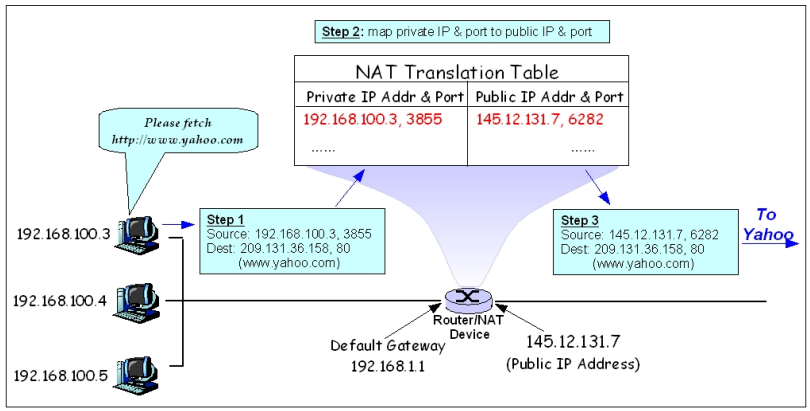
Provide information on the next hop

```
$ ip route
default via 10.0.8.1 dev enx00e11f0005f3 proto dhcp metric 100
default via 10.0.8.1 dev wlp2s0 proto dhcp metric 600
10.0.8.0/21 dev enx00e11f0005f3 proto kernel scope link src 10.0.13.191
10.0.8.0/21 dev wlp2s0 proto kernel scope link src 10.0.13.209 metric 60
169.254.0.0/16 dev enx00e11f0005f3 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdow
172.18.0.0/16 dev br-aaec6a8da6fc proto kernel scope link src 172.18.0.1
```

ASIDE: NAT

In most IPv4 networks, NAT is used to translate between local and remote networks.

It rewrites IP packets to be the correct source/destination IPs and tracks the state of TCP flows to do this.



IN CODE (WITH SOME DETAILS REMOVED...)

Still for just the first packet... and with more details removed...

```
const raw = require('raw-socket')
const ip = require('ip')

function genTcp(srcIp, srcPort, destIp, destPort) {...}

function sendSyn(srcIp, srcPort, destIp, destPort) {
  const socket = raw.createSocket({protocol: raw.Protocol.TCP, addressFamily: raw.AddressFamily.IPv4})
```

IN CODE (WITH SOME DETAILS REMOVED...)

Still for just the first packet... and with more details removed...

```
const raw = require('raw-socket')
const ip = require('ip')

function genTcp(srcIp, srcPort, destIp, destPort) {...}

function sendSyn(srcIp, srcPort, destIp, destPort) {
  const socket = raw.createSocket({protocol: raw.Protocol.TCP, addressFamily: raw.AddressFamily.IPv4})
  const ipBuffer = Buffer.from([
    0x45,                   // IP: Version (0x45 is IPv4)
    0x00,                   // IP: Differentiated Services Field
    0x00,0x3c,              // IP: Total Length
    0x00,0x00,              // IP: Identification
    0x40,                   // IP: Flags (0x20 Don't Fragment)
    0x00,                   // IP: Fragment Offset
    0x40,                   // IP: TTL (0x40 is 64)
    0x06,                   // IP: protocol (ICMP=1, IGMP=2, TCP=6, UDP=17, static value)
    0x00,0x00,              // IP: checksum for IP part of this packet
    0x00,0x00,0x00,0x00,    // IP: ip src
    0x00,0x00,0x00,0x00,    // IP: ip dst
  ])

  ipBuffer.writeUInt16BE(parseInt(Math.random()*0xffff), 4) // IP: set identification
  ip.toBuffer(src_ip, ipBuffer, 12) // IP: save ip src (src_ip var) into the buffer
  ip.toBuffer(dst_ip, ipBuffer, 16) // IP: save ip dst (dst_ip var) into the buffer
  raw.writeChecksum(ipBuffer, 10, raw.createChecksum(ipBuffer))
```

IN CODE (WITH SOME DETAILS REMOVED...)

Still for just the first packet... and with more details removed...

```
const raw = require('raw-socket')
const ip = require('ip')

function genTcp(srcIp, srcPort, destIp, destPort) {...}

function sendSyn(srcIp, srcPort, destIp, destPort) {
  const socket = raw.createSocket({protocol: raw.Protocol.TCP, addressFamily: raw.AddressFamily.IPv4})
  const ipBuffer = Buffer.from([
    0x45,                   // IP: Version (0x45 is IPv4)
    0x00,                   // IP: Differentiated Services Field
    0x00,0x3c,              // IP: Total Length
    0x00,0x00,              // IP: Identification
    0x40,                   // IP: Flags (0x20 Don't Fragment)
    0x00,                   // IP: Fragment Offset
    0x40,                   // IP: TTL (0x40 is 64)
    0x06,                   // IP: protocol (ICMP=1, IGMP=2, TCP=6, UDP=17, static value)
    0x00,0x00,              // IP: checksum for IP part of this packet
    0x00,0x00,0x00,0x00,    // IP: ip src
    0x00,0x00,0x00,0x00,    // IP: ip dst
  ])

  ipBuffer.writeUInt16BE(parseInt(Math.random()*0xffff), 4) // IP: set identification
  ip.toBuffer(src_ip, ipBuffer, 12) // IP: save ip src (src_ip var) into the buffer
  ip.toBuffer(dst_ip, ipBuffer, 16) // IP: save ip dst (dst_ip var) into the buffer
  raw.writeChecksum(ipBuffer, 10, raw.createChecksum(ipBuffer))

  function beforeSend() {
    socket.setOption(
      raw.SocketLevel.IPPROTO_IP,
      raw.SocketOption.IP_HDRINCL,
      new Buffer ([0x00, 0x00, 0x00, 0x01]),
      4
    )
  }
  const tcpBuffer = genTcp(srcIp, srcPort, destIp, destPort)
  const buffer = Buffer.concat([ipBuffer, tcpBuffer])
  socket.send(buffer, 0, buffer.length, destIp, beforeSend);

  // and that was just to send the first packet...
}
```

IN SUMMARY (PART 1)

HTTP is a message orientied protocol that is encapsulated by TCP, a connection-oriented protocol that allows for getting ordered streams of bytes, which is encapsulated by IP, a message orientied protocol.

IP gives us a mechanism for addressing and some routing and is corruption resistant.

IN SUMMARY (PART 2)

TCP ensures that all messages arrive in order and can detect most corruption.

HTTP uses TCP to send messages that takes a verb, an action, headers, and a body as a request.

It sends that data to a server, and the server returns with a response that includes a status code, headers, and a body.

The messages are directed to their final destination by IP packets with routing tables helping to find paths.

Other things, like DNS and NAT help do all this.

THAT'S IT!

HTTP, TCP AND IP ARE COMPLEX, BUT UNDERSTANDABLE

RIGHT?

WELL... WE STILL DON'T KNOW

- How does data get sent in between intermediate machines?
- How does the data actually get sent over the internet?

ETHERNET

IP ASSUMES, BUT DOES NOT DIRECTLY CARE ABOUT

- How messages are sent from one node to the next
- How messages are actually sent over a wire

WHAT ETHERNET IS

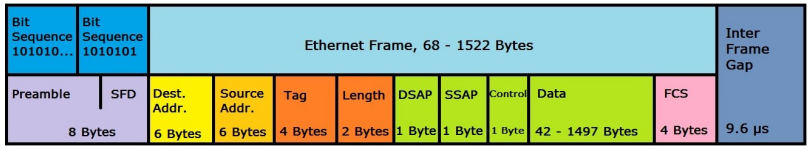
WHAT ETHERNET IS

- Provides a mechanism for communicating with the next node

WHAT ETHERNET IS

- Provides a mechanism for communicating with the next node
- Provides the basics for we send over a wire

THE ETHERNET FRAME



YET ANOTHER ADDRESS

IP addresses work great for the address of the internet... our final destination, but they don't tell us how to get to the next node in our path. For that, we need another address, the MAC address

```
ip addr show wlp2s0
2: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group de
    link/ether 00:28:f8:b1:d9:e2 brd ff:ff:ff:ff:ff:ff
    inet 10.0.13.209/21 brd 10.0.15.255 scope global dynamic noprefixroute wlp2s
       valid_lft 71880sec preferred_lft 71880sec
    inet6 fe80::88df:f80a:5f95:9df9/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

YET ANOTHER ADDRESS

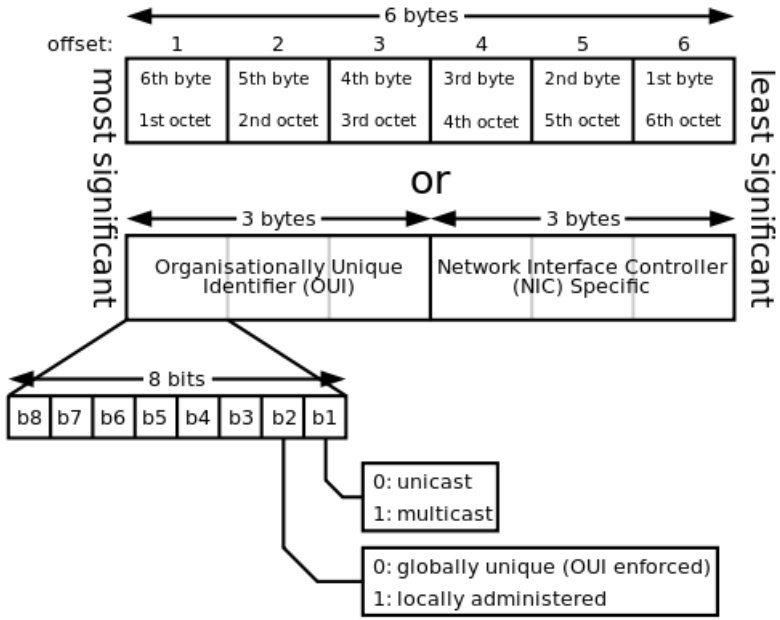
IP addresses work great for the address of the internet... our final destination, but they don't tell us how to get to the next node in our path. For that, we need another address, the MAC address

```
ip addr show wlp2s0
2: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group de
    link/ether 00:28:f8:b1:d9:e2 brd ff:ff:ff:ff:ff:ff
    inet 10.0.13.209/21 brd 10.0.15.255 scope global dynamic noprefixroute wlp2s
       valid_lft 71880sec preferred_lft 71880sec
    inet6 fe80::88df:f80a:5f95:9df9/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

YET ANOTHER ADDRESS

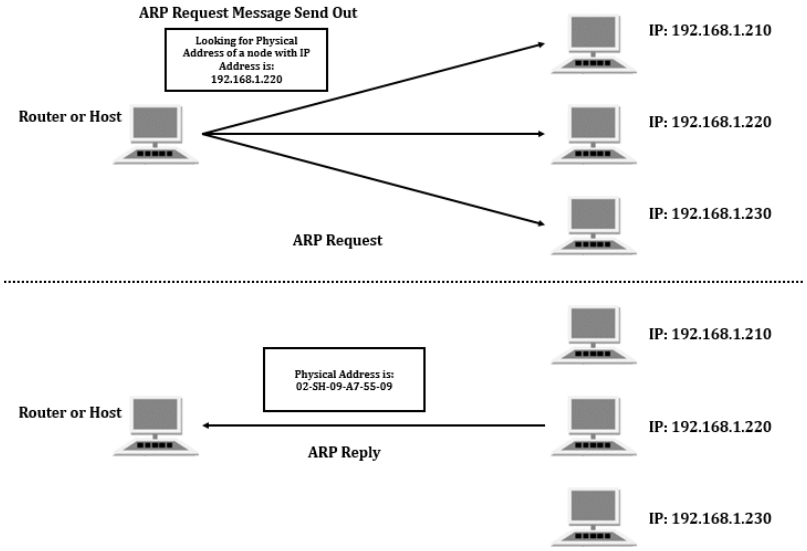
IP addresses work great for the address of the internet... our final destination, but they don't tell us how to get to the next node in our path. For that, we need another address, the MAC address

```
ip addr show wlp2s0
2: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group de
    link/ether 00:28:f8:b1:d9:e2 brd ff:ff:ff:ff:ff:ff
    inet 10.0.13.209/21 brd 10.0.15.255 scope global dynamic noprefixroute wlp2s
       valid_lft 71880sec preferred_lft 71880sec
    inet6 fe80::88df:f80a:5f95:9df9/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```


6 bytes
offset:
1
2
3
4
5
6
6th byte
1st octet
5th byte
2nd octet
4th byte
3rd octet
3rd byte
4th octet
2nd byte
5th octet
1st byte
6th octet
most significant
least significant
or
3 bytes
3 bytes
Organisationally Unique Identifier (OUI)
Network Interface Controller (NIC) Specific
8 bits
b8
b7
b6
b5
b4
b3
b2
b1
0: unicast
1: multicast
0: globally unique (OUI enforced)
1: locally administered

ASIDE: ARP

ARP is a protocol that broadcasts on a network and asks for information on which node knows which IP address is associated with which MAC address



IN CODE

magic code

```
const raw = require('raw-socket')
const ip = require('ip')

function genTcp(srcIp, srcPort, destIp, destPort) {...}
function genIp(srcIp, srcPort, destIp, destPort) {...}

function sendSyn(srcIp, srcPort, destIp, destPort) {
```

IN CODE

magic code

```
const raw = require('raw-socket')
const ip = require('ip')

function genTcp(srcIp, srcPort, destIp, destPort) {...}
function genIp(srcIp, srcPort, destIp, destPort) {...}

function sendSyn(srcIp, srcPort, destIp, destPort) {
```

IN CODE

magic code

```
const raw = require('raw-socket')
const ip = require('ip')

function genTcp(srcIp, srcPort, destIp, destPort) {...}
function genIp(srcIp, srcPort, destIp, destPort) {...}

function sendSyn(srcIp, srcPort, destIp, destPort) {


  //          _,'',_                        _
  //         /////\\                   _ |\ | \|\  _
  //        // e e \\                 / `|~\|_/|~\|_) _
  //       ("   >   ")      _         \_,           |_)|_)
  //        \\.=-=.//   \/ )     _ |\ /               | \|\
  //      \--\\\7///--/ /\/   _ |_)|~\                  |~\
  //      /`--`"""`--'\/ /   |_)| \
  //     /       :   \   / |\ |_)
  //     \  '-.,_      `-'  |~\
  //      `-._/__)======_____,
  //                  ((=====))
  //          _        |`"""`|     _//
  //       `\,_('>     |=     |    (')
  //          \~_)     |=     |     ( '-.
  //          -'=      `-...-`    _/-_(_)o


  const ipBuffer = genIp(srcIp, srcPort, destIp, destPort)
  const tcpBuffer = genTcp(srcIp, srcPort, destIp, destPort)
  const buffer = Buffer.concat([ethBuffer, ipBuffer, tcpBuffer, ethTrail])
  socket.send(buffer, 0, buffer.length, destIp, beforeSend);

  // and that was just to send the first packet...
}
```

IN SUMMARY (PART 1)

HTTP is a message orientied protocol that is encapsulated by TCP, a connection-oriented protocol that allows for getting ordered streams of bytes, which is encapsulated IP, a message orientied protocol. IP is in turn encapsulated by Ethernet, which is another message oriented protocol.

Together, this whole structure defines something that can be sent over a network.

Ethernet gives us a mechanism for physically finding the next node to send to in a network, and also has some checksums to ensure valid data.

IP gives us a mechanism for addressing and some routing, while also ensuring it's headers aren't corrupted.

IN SUMMARY (PART 2)

TCP ensures that all messages arrive in order and can detect most corruption.

HTTP uses TCP to send messages that takes a verb, an action, headers, and a body as a request.

It sends that data to a server, and the server returns with a response that includes a status code, headers, and a body.

The messages are directed to their final destination by IP packets, with ethernet frames being used for point to point communication between the origin address and the destination address.

Other things, like DNS, NAT and ARP make this work.

THAT'S IT!

HTTP, TCP, IP AND ETHERNET ARE FAIRLY COMPLEX!

RIGHT?

WELL... WE STILL DON'T KNOW

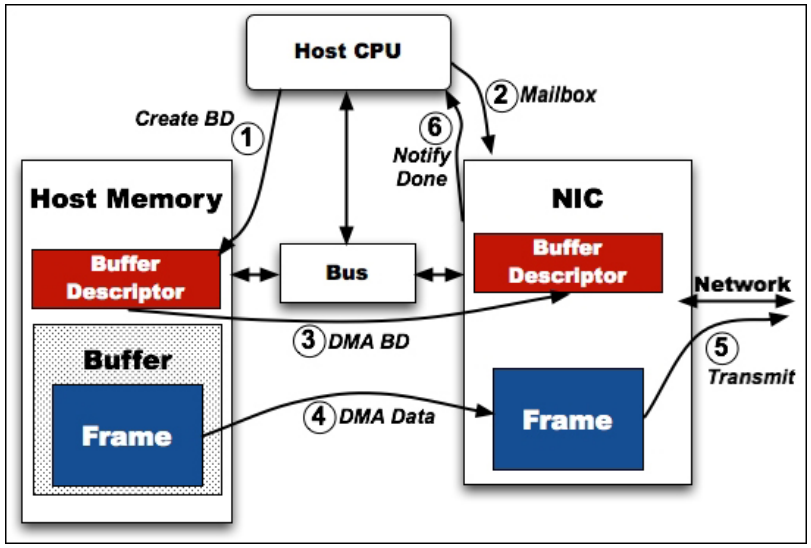
- How does that data even make it to a wire?

NETWORK CARDS

Now that we have this whole packet structure... how does it actually get communicated to a network card?

DMA

DIRECT MEMORY ACCESS



IN CODE

First we init our NIC...

```
...
// allocated for each tx queue, keeps state for the transmit function
struct ixgbe_tx_queue {
    volatile union ixgbe_adv_tx_desc* descriptors;
    uint16_t num_entries;
    // position to clean up descriptors that where sent out by the nic
    uint16_t clean_index;
    // position to insert packets for transmission
    uint16_t tx_index;
    // virtual addresses to map descriptors back to their mbuf for freeing
    void* virtual_addresses[];
};

// see section 4.6.4
static void init_link(struct ixgbe_device* dev) {
    // should already be set by the eeprom config, maybe we shouldn't override it here to support weirdo nics?
    set_reg32(dev->addr, IXGBE_AUTOC, (get_reg32(dev->addr, IXGBE_AUTOC) & ~IXGBE_AUTOC_LMS_MASK) | IXGBE_AUTOC_LMS_10G_SERIAL);
    set_reg32(dev->addr, IXGBE_AUTOC, (get_reg32(dev->addr, IXGBE_AUTOC) & ~IXGBE_AUTOC_10G_PMA_PMD_MASK) | IXGBE_AUTOC_10G_XAUI);
    // negotiate link
    set_flags32(dev->addr, IXGBE_AUTOC, IXGBE_AUTOC_AN_RESTART);
    // datasheet wants us to wait for the link here, but we can continue and wait afterwards
}
...

// it looks quite complicated in the data sheet, but it's actually really easy because we don't need fancy features
// see section 4.6.8
static void init_tx(struct ixgbe_device* dev) {
    // crc offload and small packet padding
    set_flags32(dev->addr, IXGBE_HLREG0, IXGBE_HLREG0_TXCRCEN | IXGBE_HLREG0_TXPADEN);

    // set default buffer size allocations
    // see also: section 4.6.11.3.4, no fancy features like DCB and VTd
    set_reg32(dev->addr, IXGBE_TXPBSIZE(0), IXGBE_TXPBSIZE_40KB);
    for (int i = 1; i < 8; i++) {
        set_reg32(dev->addr, IXGBE_TXPBSIZE(i), 0);
    }
    // required when not using DCB/VTd
    set_reg32(dev->addr, IXGBE_DTXMXSZRQ, 0xFFFF);
    clear_flags32(dev->addr, IXGBE_RTTDCS, IXGBE_RTTDCS_ARBDIS);

    // per-queue config for all queues
    for (uint16_t i = 0; i < dev->ixy.num_tx_queues; i++) {
        debug("initializing tx queue %d", i);

        // setup descriptor ring, see section 7.1.9
        uint32_t ring_size_bytes = NUM_TX_QUEUE_ENTRIES * sizeof(union ixgbe_adv_tx_desc);
        struct dma_memory mem = memory_allocate_dma(ring_size_bytes, true);
        memset(mem.virt, -1, ring_size_bytes);
        // tell the device where it can write to (its iova, so its view)
        set_reg32(dev->addr, IXGBE_TDBAL(i), (uint32_t) (mem.phy & 0xFFFFFFFFull));
        set_reg32(dev->addr, IXGBE_TDBAH(i), (uint32_t) (mem.phy >> 32));
        set_reg32(dev->addr, IXGBE_TDLEN(i), ring_size_bytes);
        debug("tx ring %d phy addr:  0x%012lX", i, mem.phy);
        debug("tx ring %d virt addr: 0x%012lX", i, (uintptr_t) mem.virt);

        // descriptor writeback magic values, important to get good performance and low PCIe overhead
        // see 7.2.3.4.1 and 7.2.3.5 for an explanation of these values and how to find good ones
        // we just use the defaults from DPDK here, but this is a potentially interesting point for optimizations
        uint32_t txdctl = get_reg32(dev->addr, IXGBE_TXDCTL(i));
        // there are no defines for this in ixgbe_type.h for some reason
        // pthresh: 6:0, hthresh: 14:8, wthresh: 22:16
        txdctl &= ~(0x3F | (0x3F << 8) | (0x3F << 16)); // clear bits
        txdctl |= (36 | (8 << 8) | (4 << 16)); // from DPDK
        set_reg32(dev->addr, IXGBE_TXDCTL(i), txdctl);

        // private data for the driver, 0-initialized
        struct ixgbe_tx_queue* queue = ((struct ixgbe_tx_queue*)(dev->tx_queues)) + i;
        queue->num_entries = NUM_TX_QUEUE_ENTRIES;
        queue->descriptors = (union ixgbe_adv_tx_desc*) mem.virt;
    }
    // final step: enable DMA
    set_reg32(dev->addr, IXGBE_DMATXCTL, IXGBE_DMATXCTL_TE);
}
```

IN CODE

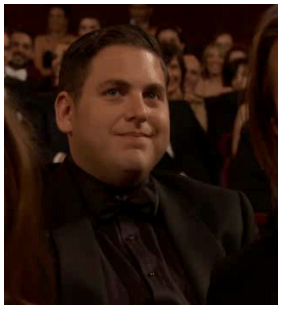
First we init our NIC...

```
...
// allocated for each tx queue, keeps state for the transmit function
struct ixgbe_tx_queue {
    volatile union ixgbe_adv_tx_desc* descriptors;
    uint16_t num_entries;
    // position to clean up descriptors that where sent out by the nic
    uint16_t clean_index;
    // position to insert packets for transmission
    uint16_t tx_index;
    // virtual addresses to map descriptors back to their mbuf for freeing
    void* virtual_addresses[];
};

// see section 4.6.4
static void init_link(struct ixgbe_device* dev) {
    // should already be set by the eeprom config, maybe we shouldn't override it here to support weirdo nics?
    set_reg32(dev->addr, IXGBE_AUTOC, (get_reg32(dev->addr, IXGBE_AUTOC) & ~IXGBE_AUTOC_LMS_MASK) | IXGBE_AUTOC_LMS_10G_SERIAL);
    set_reg32(dev->addr, IXGBE_AUTOC, (get_reg32(dev->addr, IXGBE_AUTOC) & ~IXGBE_AUTOC_10G_PMA_PMD_MASK) | IXGBE_AUTOC_10G_XAUI);
    // negotiate link
    set_flags32(dev->addr, IXGBE_AUTOC, IXGBE_AUTOC_AN_RESTART);
    // datasheet wants us to wait for the link here, but we can continue and wait afterwards
}
...

// it looks quite complicated in the data sheet, but it's actually really easy because we don't need fancy features
// see section 4.6.8
static void init_tx(struct ixgbe_device* dev) {
    // crc offload and small packet padding
    set_flags32(dev->addr, IXGBE_HLREG0, IXGBE_HLREG0_TXCRCEN | IXGBE_HLREG0_TXPADEN);

    // set default buffer size allocations
    // see also: section 4.6.11.3.4, no fancy features like DCB and VTd
    set_reg32(dev->addr, IXGBE_TXPBSIZE(0), IXGBE_TXPBSIZE_40KB);
    for (int i = 1; i < 8; i++) {
        set_reg32(dev->addr, IXGBE_TXPBSIZE(i), 0);
    }
    // required when not using DCB/VTd
    set_reg32(dev->addr, IXGBE_DTXMXSZRQ, 0xFFFF);
    clear_flags32(dev->addr, IXGBE_RTTDCS, IXGBE_RTTDCS_ARBDIS);

    // per-queue config for all queues
    for (uint16_t i = 0; i < dev->ixy.num_tx_queues; i++) {
        debug("initializing tx queue %d", i);

        // setup descriptor ring, see section 7.1.9
        uint32_t ring_size_bytes = NUM_TX_QUEUE_ENTRIES * sizeof(union ixgbe_adv_tx_desc);
        struct dma_memory mem = memory_allocate_dma(ring_size_bytes, true);
        memset(mem.virt, -1, ring_size_bytes);
        // tell the device where it can write to (its iova, so its view)
        set_reg32(dev->addr, IXGBE_TDBAL(i), (uint32_t) (mem.phy & 0xFFFFFFFFull));
        set_reg32(dev->addr, IXGBE_TDBAH(i), (uint32_t) (mem.phy >> 32));
        set_reg32(dev->addr, IXGBE_TDLEN(i), ring_size_bytes);
        debug("tx ring %d phy addr:  0x%012lX", i, mem.phy);
        debug("tx ring %d virt addr: 0x%012lX", i, (uintptr_t) mem.virt);

        // descriptor writeback magic values, important to get good performance and low PCIe overhead
        // see 7.2.3.4.1 and 7.2.3.5 for an explanation of these values and how to find good ones
        // we just use the defaults from DPDK here, but this is a potentially interesting point for optimizations
        uint32_t txdctl = get_reg32(dev->addr, IXGBE_TXDCTL(i));
        // there are no defines for this in ixgbe_type.h for some reason
        // pthresh: 6:0, hthresh: 14:8, wthresh: 22:16
        txdctl &= ~(0x3F | (0x3F << 8) | (0x3F << 16)); // clear bits
        txdctl |= (36 | (8 << 8) | (4 << 16)); // from DPDK
        set_reg32(dev->addr, IXGBE_TXDCTL(i), txdctl);

        // private data for the driver, 0-initialized
        struct ixgbe_tx_queue* queue = ((struct ixgbe_tx_queue*)(dev->tx_queues)) + i;
        queue->num_entries = NUM_TX_QUEUE_ENTRIES;
        queue->descriptors = (union ixgbe_adv_tx_desc*) mem.virt;
    }
    // final step: enable DMA
    set_reg32(dev->addr, IXGBE_DMATXCTL, IXGBE_DMATXCTL_TE);
}
```

MORE CODE

Just a bit more init...

```
// see section 4.6.3
static void reset_and_init(struct ixgbe_device* dev) {
    info("Resetting device %s", dev->ixy.pci_addr);
    // section 4.6.3.1 - disable all interrupts
    set_reg32(dev->addr, IXGBE_EIMC, 0x7FFFFFFF);

    // section 4.6.3.2
    set_reg32(dev->addr, IXGBE_CTRL, IXGBE_CTRL_RST_MASK);
    wait_clear_reg32(dev->addr, IXGBE_CTRL, IXGBE_CTRL_RST_MASK);
    usleep(10000);

    // section 4.6.3.1 - disable interrupts again after reset
    set_reg32(dev->addr, IXGBE_EIMC, 0x7FFFFFFF);

    info("Initializing device %s", dev->ixy.pci_addr);

    // section 4.6.3 - Wait for EEPROM auto read completion
    wait_set_reg32(dev->addr, IXGBE_EEC, IXGBE_EEC_ARD);

    // section 4.6.3 - Wait for DMA initialization done (RDRXCTL.DMAIDONE)
    wait_set_reg32(dev->addr, IXGBE_RDRXCTL, IXGBE_RDRXCTL_DMAIDONE);

    // section 4.6.4 - initialize link (auto negotiation)
    init_link(dev);

    // section 4.6.5 - statistical counters
    // reset-on-read registers, just read them once
    ixgbe_read_stats(&dev->ixy, NULL);

    // section 4.6.7 - init rx
    init_rx(dev);

    // section 4.6.8 - init tx
    init_tx(dev);

    // enables queues after initializing everything
    for (uint16_t i = 0; i < dev->ixy.num_rx_queues; i++) {
        start_rx_queue(dev, i);
    }
    for (uint16_t i = 0; i < dev->ixy.num_tx_queues; i++) {
        start_tx_queue(dev, i);
    }

    // skip last step from 4.6.3 - don't want interrupts
    // finally, enable promisc mode by default, it makes testing less annoying
    ixgbe_set_promisc(&dev->ixy, true);

    // wait for some time for the link to come up
    wait_for_link(dev);
}

struct ixy_device* ixgbe_init(const char* pci_addr, uint16_t rx_queues, uint16_t tx_queues) {
    if (getuid()) {
        warn("Not running as root, this will probably fail");
    }
    if (rx_queues > MAX_QUEUES) {
        error("cannot configure %d rx queues: limit is %d", rx_queues, MAX_QUEUES);
    }
    if (tx_queues > MAX_QUEUES) {
        error("cannot configure %d tx queues: limit is %d", tx_queues, MAX_QUEUES);
    }

    // Allocate memory for the ixgbe device that will be returned
    struct ixgbe_device* dev = (struct ixgbe_device*) malloc(sizeof(struct ixgbe_device));
    dev->ixy.pci_addr = strdup(pci_addr);

    // Check if we want the VFIO stuff
    // This is done by checking if the device is in an IOMMU group.
    char path[PATH_MAX];
    snprintf(path, PATH_MAX, "/sys/bus/pci/devices/%s/iommu_group", pci_addr);
    struct stat buffer;
    dev->ixy.vfio = stat(path, &buffer) == 0;
    if (dev->ixy.vfio) {
        // initialize the IOMMU for this device
        dev->ixy.vfio_fd = vfio_init(pci_addr);
        if (dev->ixy.vfio_fd < 0) {
            error("could not initialize the IOMMU for device %s", pci_addr);
        }
    }
    dev->ixy.driver_name = driver_name;
    dev->ixy.num_rx_queues = rx_queues;
```



MORE CODE

Just a bit more init...

```
// see section 4.6.3
static void reset_and_init(struct ixgbe_device* dev) {
    info("Resetting device %s", dev->ixy.pci_addr);
    // section 4.6.3.1 - disable all interrupts
    set_reg32(dev->addr, IXGBE_EIMC, 0x7FFFFFFF);

    // section 4.6.3.2
    set_reg32(dev->addr, IXGBE_CTRL, IXGBE_CTRL_RST_MASK);
    wait_clear_reg32(dev->addr, IXGBE_CTRL, IXGBE_CTRL_RST_MASK);
    usleep(10000);

    // section 4.6.3.1 - disable interrupts again after reset
    set_reg32(dev->addr, IXGBE_EIMC, 0x7FFFFFFF);

    info("Initializing device %s", dev->ixy.pci_addr);

    // section 4.6.3 - Wait for EEPROM auto read completion
    wait_set_reg32(dev->addr, IXGBE_EEC, IXGBE_EEC_ARD);

    // section 4.6.3 - Wait for DMA initialization done (RDRXCTL.DMAIDONE)
    wait_set_reg32(dev->addr, IXGBE_RDRXCTL, IXGBE_RDRXCTL_DMAIDONE);

    // section 4.6.4 - initialize link (auto negotiation)
    init_link(dev);

    // section 4.6.5 - statistical counters
    // reset-on-read registers, just read them once
    ixgbe_read_stats(&dev->ixy, NULL);

    // section 4.6.7 - init rx
    init_rx(dev);

    // section 4.6.8 - init tx
    init_tx(dev);

    // enables queues after initializing everything
    for (uint16_t i = 0; i < dev->ixy.num_rx_queues; i++) {
        start_rx_queue(dev, i);
    }
    for (uint16_t i = 0; i < dev->ixy.num_tx_queues; i++) {
        start_tx_queue(dev, i);
    }

    // skip last step from 4.6.3 - don't want interrupts
    // finally, enable promisc mode by default, it makes testing less annoying
    ixgbe_set_promisc(&dev->ixy, true);

    // wait for some time for the link to come up
    wait_for_link(dev);
}

struct ixy_device* ixgbe_init(const char* pci_addr, uint16_t rx_queues, uint16_t tx_queues) {
    if (getuid()) {
        warn("Not running as root, this will probably fail");
    }
    if (rx_queues > MAX_QUEUES) {
        error("cannot configure %d rx queues: limit is %d", rx_queues, MAX_QUEUES);
    }
    if (tx_queues > MAX_QUEUES) {
        error("cannot configure %d tx queues: limit is %d", tx_queues, MAX_QUEUES);
    }

    // Allocate memory for the ixgbe device that will be returned
    struct ixgbe_device* dev = (struct ixgbe_device*) malloc(sizeof(struct ixgbe_device));
    dev->ixy.pci_addr = strdup(pci_addr);

    // Check if we want the VFIO stuff
    // This is done by checking if the device is in an IOMMU group.
    char path[PATH_MAX];
    snprintf(path, PATH_MAX, "/sys/bus/pci/devices/%s/iommu_group", pci_addr);
    struct stat buffer;
    dev->ixy.vfio = stat(path, &buffer) == 0;
    if (dev->ixy.vfio) {
        // initialize the IOMMU for this device
        dev->ixy.vfio_fd = vfio_init(pci_addr);
        if (dev->ixy.vfio_fd < 0) {
            error("could not initialize the IOMMU for device %s", pci_addr);
        }
    }
    dev->ixy.driver_name = driver_name;
    dev->ixy.num_rx_queues = rx_queues;
```

A BIT MORE...

Do our real work...

```
// section 1.8.1 and 7.2
// we control the tail, hardware the head
// huge performance gains possible here by sending packets in batches - writing to TDT for every packet is not efficient
// returns the number of packets transmitted, will not block when the queue is full
uint32_t ixgbe_tx_batch(struct ixy_device* ixy, uint16_t queue_id, struct pkt_buf* bufs[], uint32_t num_bufs) {
    struct ixgbe_device* dev = IXY_TO_IXGBE(ixy);
    struct ixgbe_tx_queue* queue = ((struct ixgbe_tx_queue*)(dev->tx_queues)) + queue_id;
    // the descriptor is explained in section 7.2.3.2.4
    // we just use a struct copy & pasted from intel, but it basically has two formats (hence a union):
    // 1. the write-back format which is written by the NIC once sending it is finished this is used in step 1
    // 2. the read format which is read by the NIC and written by us, this is used in step 2

    uint16_t clean_index = queue->clean_index; // next descriptor to clean up

    // step 1: clean up descriptors that were sent out by the hardware and return them to the mempool
    // start by reading step 2 which is done first for each packet
    // cleaning up must be done in batches for performance reasons, so this is unfortunately somewhat complicated
    while (true) {
        // figure out how many descriptors can be cleaned up
        int32_t cleanable = queue->tx_index - clean_index; // tx_index is always ahead of clean (invariant of our queue)
        if (cleanable < 0) { // handle wrap-around
            cleanable = queue->num_entries + cleanable;
        }
        if (cleanable < TX_CLEAN_BATCH) {
            break;
        }
        // calculcate the index of the last transcriptor in the clean batch
        // we can't check all descriptors for performance reasons
        int32_t cleanup_to = clean_index + TX_CLEAN_BATCH - 1;
        if (cleanup_to >= queue->num_entries) {
            cleanup_to -= queue->num_entries;
        }
        volatile union ixgbe_adv_tx_desc* txd = queue->descriptors + cleanup_to;
        uint32_t status = txd->wb.status;
        // hardware sets this flag as soon as it's sent out, we can give back all bufs in the batch back to the mempool
        if (status & IXGBE_ADVTXD_STAT_DD) {
            int32_t i = clean_index;
            while (true) {
                struct pkt_buf* buf = queue->virtual_addresses[i];
                pkt_buf_free(buf);
                if (i == cleanup_to) {
                    break;
                }
                i = wrap_ring(i, queue->num_entries);
            }
            // next descriptor to be cleaned up is one after the one we just cleaned
            clean_index = wrap_ring(cleanup_to, queue->num_entries);
        } else {
            // clean the whole batch or nothing; yes, this leaves some packets in
            // the queue forever if you stop transmitting, but that's not a real concern
            break;
        }
    }
    queue->clean_index = clean_index;

    // step 2: send out as many of our packets as possible
    uint32_t sent;
    for (sent = 0; sent < num_bufs; sent++) {
        uint32_t next_index = wrap_ring(queue->tx_index, queue->num_entries);
        // we are full if the next index is the one we are trying to reclaim
        if (clean_index == next_index) {
            break;
        }
        struct pkt_buf* buf = bufs[sent];
        // remember virtual address to clean it up later
        queue->virtual_addresses[queue->tx_index] = (void*) buf;
        volatile union ixgbe_adv_tx_desc* txd = queue->descriptors + queue->tx_index;
        queue->tx_index = next_index;
        // NIC reads from here
        txd->read.buffer_addr = buf->buf_addr_phy + offsetof(struct pkt_buf, data);
        // always the same flags: one buffer (EOP), advanced data descriptor, CRC offload, data length
        txd->read.cmd_type_len =
            IXGBE_ADVTXD_DCMD_EOP | IXGBE_ADVTXD_DCMD_RS | IXGBE_ADVTXD_DCMD_IFCS | IXGBE_ADVTXD_DCMD_DEXT | IXGBE_ADVTXD_DTYP_DATA | buf->size;
        // no fancy offloading stuff - only the total payload length
        // implement offloading flags here:
        //     * ip checksum offloading is trivial: just set the offset
        //     * tcp/udp checksum offloading is more annoying, you have to precalculate the pseudo-header checksum
        txd->read.olinfo_status = buf->size << IXGBE_ADVTXD_PAYLEN_SHIFT;
    }
    // send out by advancing tail, i.e., pass control of the bufs to the nic
    // this seems like a textbook case for a release memory order, but Intel's driver doesn't even use a compiler barrier here
```

A BIT MORE...

Do our real work...

```
// section 1.8.1 and 7.2
// we control the tail, hardware the head
// huge performance gains possible here by sending packets in batches - writing to TDT for every packet is not efficient
// returns the number of packets transmitted, will not block when the queue is full
uint32_t ixgbe_tx_batch(struct ixy_device* ixy, uint16_t queue_id, struct pkt_buf* bufs[], uint32_t num_bufs) {
    struct ixgbe_device* dev = IXY_TO_IXGBE(ixy);
    struct ixgbe_tx_queue* queue = ((struct ixgbe_tx_queue*)(dev->tx_queues)) + queue_id;
    // the descriptor is explained in section 7.2.3.2.4
    // we just use a struct copy & pasted from intel, but it basically has two formats (hence a union):
    // 1. the write-back format which is written by the NIC once sending it is finished this is used in step 1
    // 2. the read format which is read by the NIC and written by us, this is used in step 2

    uint16_t clean_index = queue->clean_index; // next descriptor to clean up

    // step 1: clean up descriptors that were sent out by the hardware and return them to the mempool
    // start by reading step 2 which is done first for each packet
    // cleaning up must be done in batches for performance reasons, so this is unfortunately somewhat complicated
    while (true) {
        // figure out how many descriptors can be cleaned up
        int32_t cleanable = queue->tx_index - clean_index; // tx_index is always ahead of clean (invariant of our queue)
        if (cleanable < 0) { // handle wrap-around
            cleanable = queue->num_entries + cleanable;
        }
        if (cleanable < TX_CLEAN_BATCH) {
            break;
        }
        // calculcate the index of the last transcriptor in the clean batch
        // we can't check all descriptors for performance reasons
        int32_t cleanup_to = clean_index + TX_CLEAN_BATCH - 1;
        if (cleanup_to >= queue->num_entries) {
            cleanup_to -= queue->num_entries;
        }
        volatile union ixgbe_adv_tx_desc* txd = queue->descriptors + cleanup_to;
        uint32_t status = txd->wb.status;
        // hardware sets this flag as soon as it's sent out, we can give back all bufs in the batch back to the mempool
        if (status & IXGBE_ADVTXD_STAT_DD) {
            int32_t i = clean_index;
            while (true) {
                struct pkt_buf* buf = queue->virtual_addresses[i];
                pkt_buf_free(buf);
                if (i == cleanup_to) {
                    break;
                }
                i = wrap_ring(i, queue->num_entries);
            }
            // next descriptor to be cleaned up is one after the one we just cleaned
            clean_index = wrap_ring(cleanup_to, queue->num_entries);
        } else {
            // clean the whole batch or nothing; yes, this leaves some packets in
            // the queue forever if you stop transmitting, but that's not a real concern
            break;
        }
    }
    queue->clean_index = clean_index;

    // step 2: send out as many of our packets as possible
    uint32_t sent;
    for (sent = 0; sent < num_bufs; sent++) {
        uint32_t next_index = wrap_ring(queue->tx_index, queue->num_entries);
        // we are full if the next index is the one we are trying to reclaim
        if (clean_index == next_index) {
            break;
        }
        struct pkt_buf* buf = bufs[sent];
        // remember virtual address to clean it up later
        queue->virtual_addresses[queue->tx_index] = (void*) buf;
        volatile union ixgbe_adv_tx_desc* txd = queue->descriptors + queue->tx_index;
        queue->tx_index = next_index;
        // NIC reads from here
        txd->read.buffer_addr = buf->buf_addr_phy + offsetof(struct pkt_buf, data);
        // always the same flags: one buffer (EOP), advanced data descriptor, CRC offload, data length
        txd->read.cmd_type_len =
            IXGBE_ADVTXD_DCMD_EOP | IXGBE_ADVTXD_DCMD_RS | IXGBE_ADVTXD_DCMD_IFCS | IXGBE_ADVTXD_DCMD_DEXT | IXGBE_ADVTXD_DTYP_DATA | buf->size;
        // no fancy offloading stuff - only the total payload length
        // implement offloading flags here:
        //     * ip checksum offloading is trivial: just set the offset
        //     * tcp/udp checksum offloading is more annoying, you have to precalculate the pseudo-header checksum
        txd->read.olinfo_status = buf->size << IXGBE_ADVTXD_PAYLEN_SHIFT;
    }
    // send out by advancing tail, i.e., pass control of the bufs to the nic
    // this seems like a textbook case for a release memory order, but Intel's driver doesn't even use a compiler barrier here
```

IN SUMMARY (PART 1)

HTTP is a message orientied protocol that is encapsulated by TCP, a connection-oriented protocol that allows for getting ordered streams of bytes, which is encapsulated IP, a message orientied protocol. IP is in turn encapsulated by Ethernet, which is another message oriented protocol.

Together, this whole structure defines something that can be sent over a network, via a kernel network driver, which builds ring buffers of packets to be sent, which are by a NIC using DMA, which then sends the data on to the actual network by writing signals to a wire.

IN SUMMARY (PART 2)

Ethernet gives us a mechanism for physically finding the next node to send to in a network, and also has some checksums to ensure valid data.

IP gives us a mechanism for addressing and some routing, while also ensuring it's headers aren't corrupted.

TCP ensures that all messages arrive in order and can detect most corruption.

IN SUMMARY (PART 3)

HTTP uses TCP to send messages that takes a verb, an action, headers, and a body as a request.

It sends that data to a server, and the server returns with a response that includes a status code, headers, and a body.

The messages are directed to their final destination by IP packets, with ethernet frames being used for point to point communication between the origin address and the destination address.

Other things, like DNS, NAT and ARP make this work.

THAT'S IT!

HTTP, TCP, IP, ETHERNET AND NICS ARE A HUGE SYSTEM TO DO SOMETHING WE SEE AS "SIMPLE"

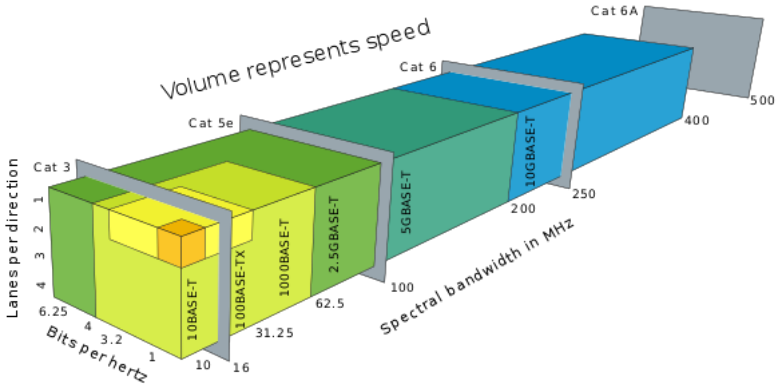
HOORAY! WE KNOW THINGS WORK...

HOORAY! WE KNOW THINGS WORK...

SORTA

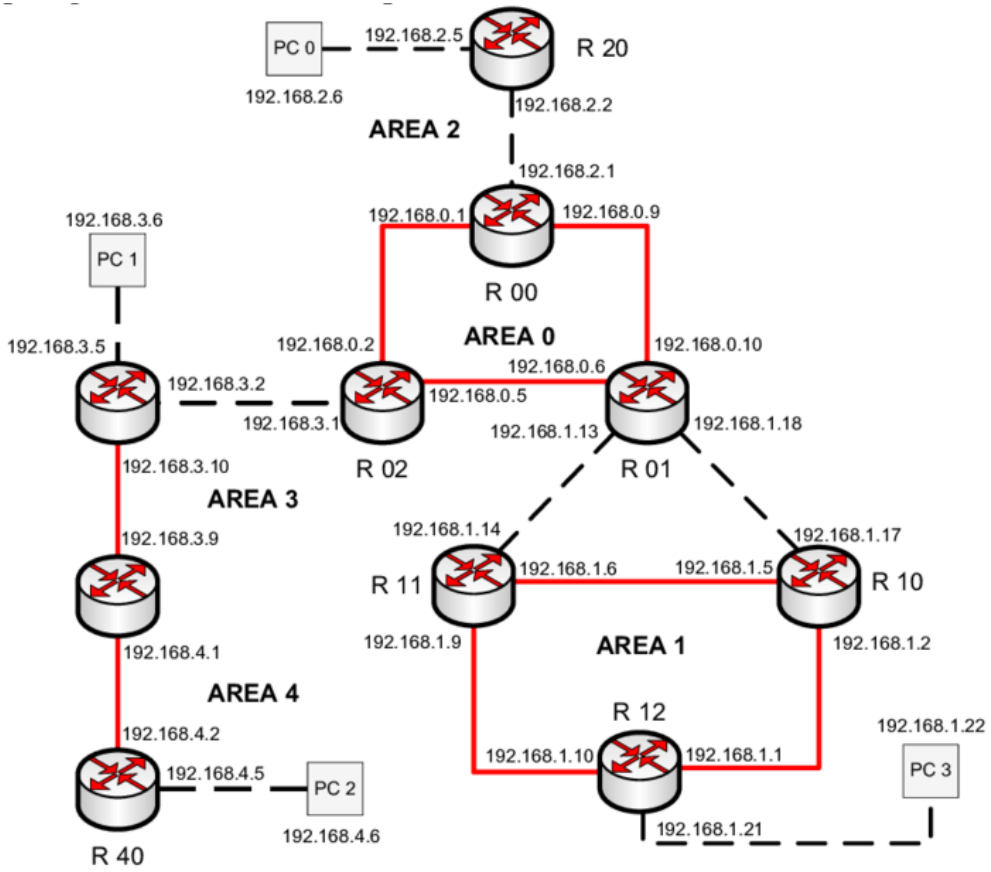
STILL A FEW THINGS TO UNDERSTAND THOUGH

SPEEDING UP ETHERNET

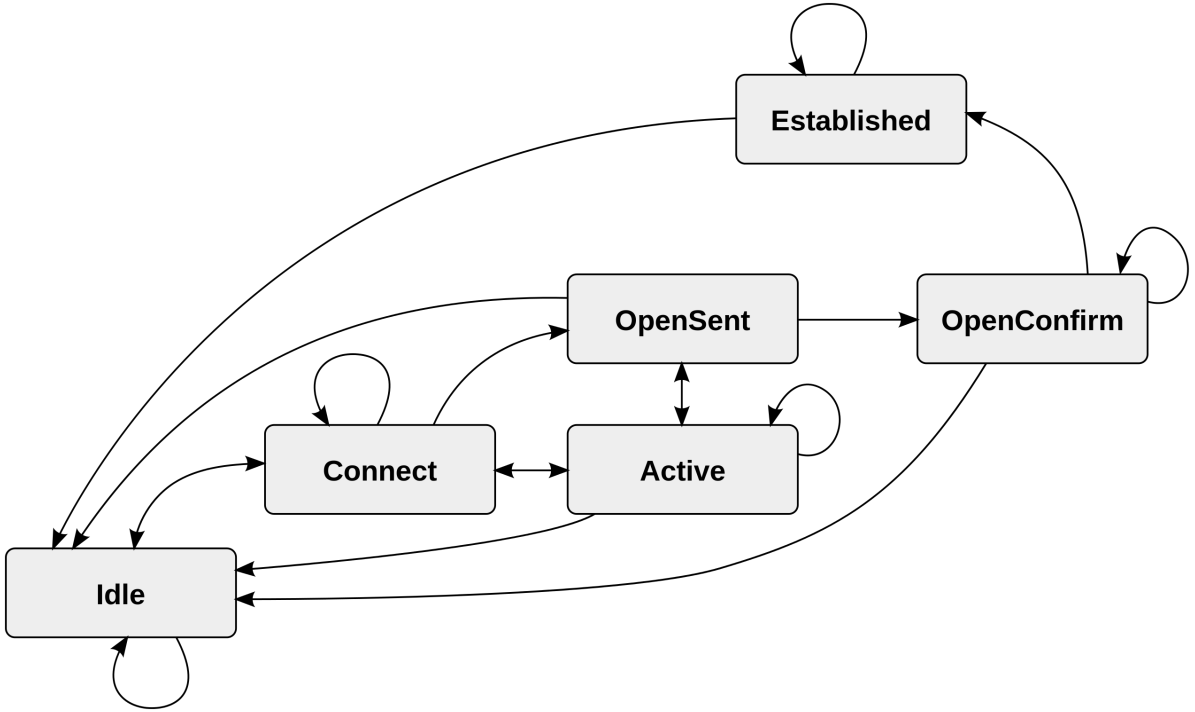


The 802.3an standard specifies the wire-level modulation for 10GBASE-T to use Tomlinson-Harashima precoding (THP) and pulse-amplitude modulation with 16 discrete levels (PAM-16), encoded in a two-dimensional checkerboard pattern known as DSQ128 sent on the line at 800 Msymbols/sec.[52][53] Prior to precoding, forward error correction (FEC) coding is performed using a $[2048,1723]_2$ low-density parity-check code on 1723 bits, with the parity check matrix construction based on a generalized Reed–Solomon [32,2,31] code over GF(2^6).[53] Another 1536 bits are uncoded. Within each 1723+1536 block, there are 1+50+8+1 signaling and error detection bits and 3200 data bits (and occupying 320 ns on the line). By contrast PAM-5 is the modulation technique used in 1000BASE-T Gigabit Ethernet.

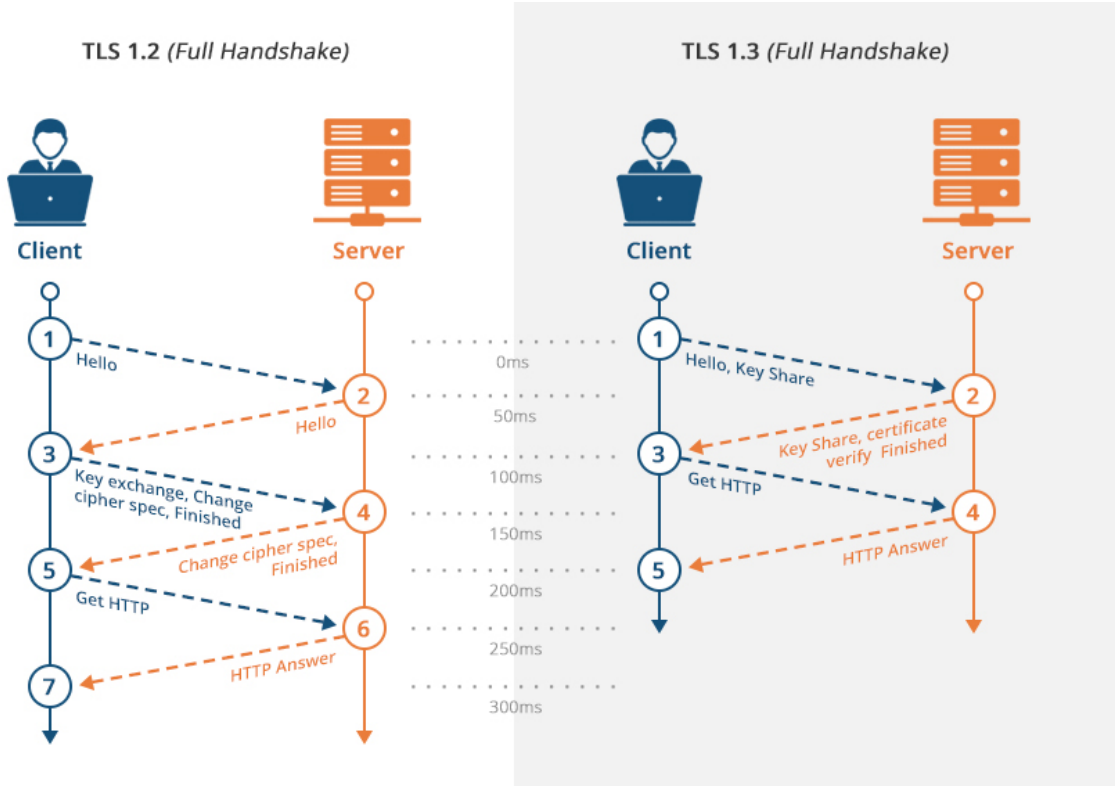
MAKE SURE WE UNDERSTAND OSPF REAL QUICK



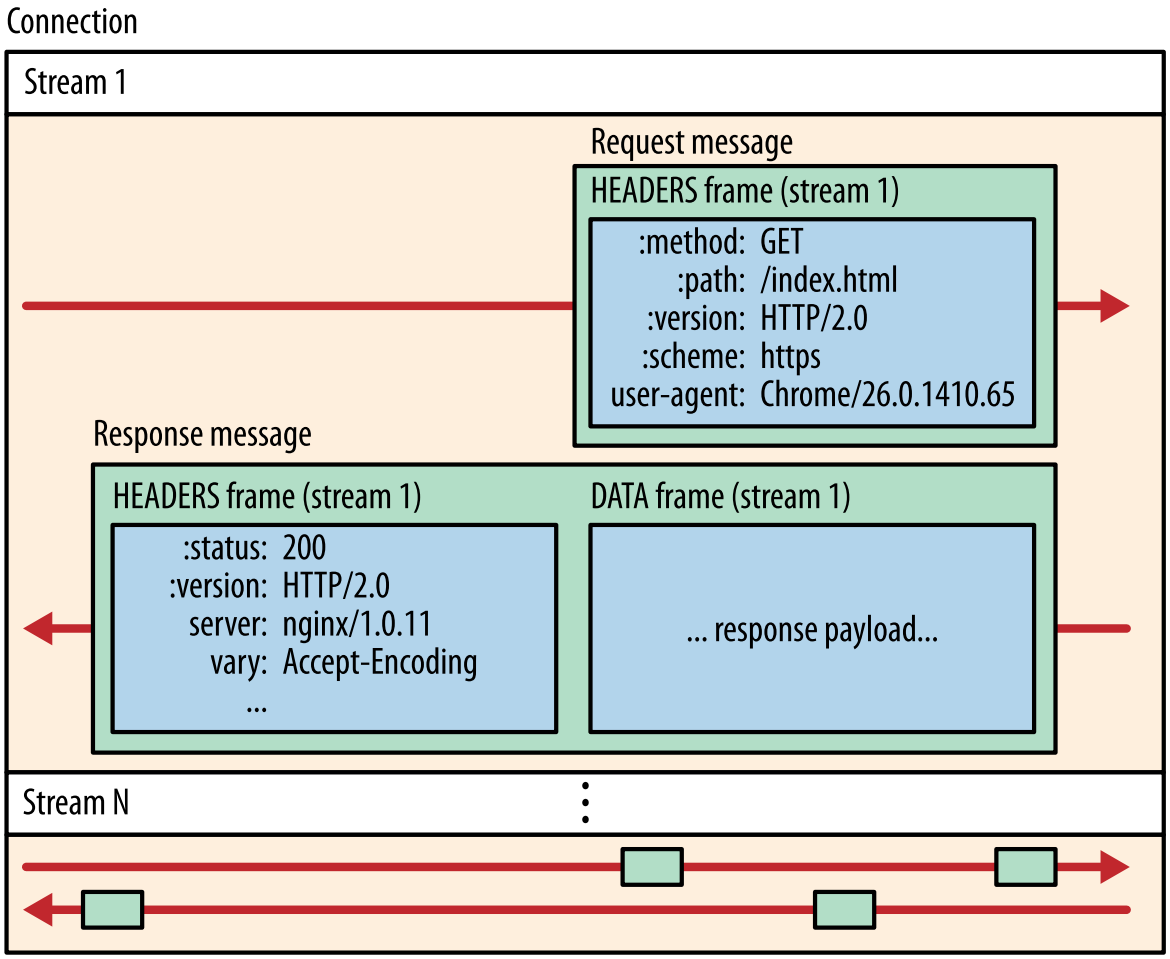
A BIT OF BGP



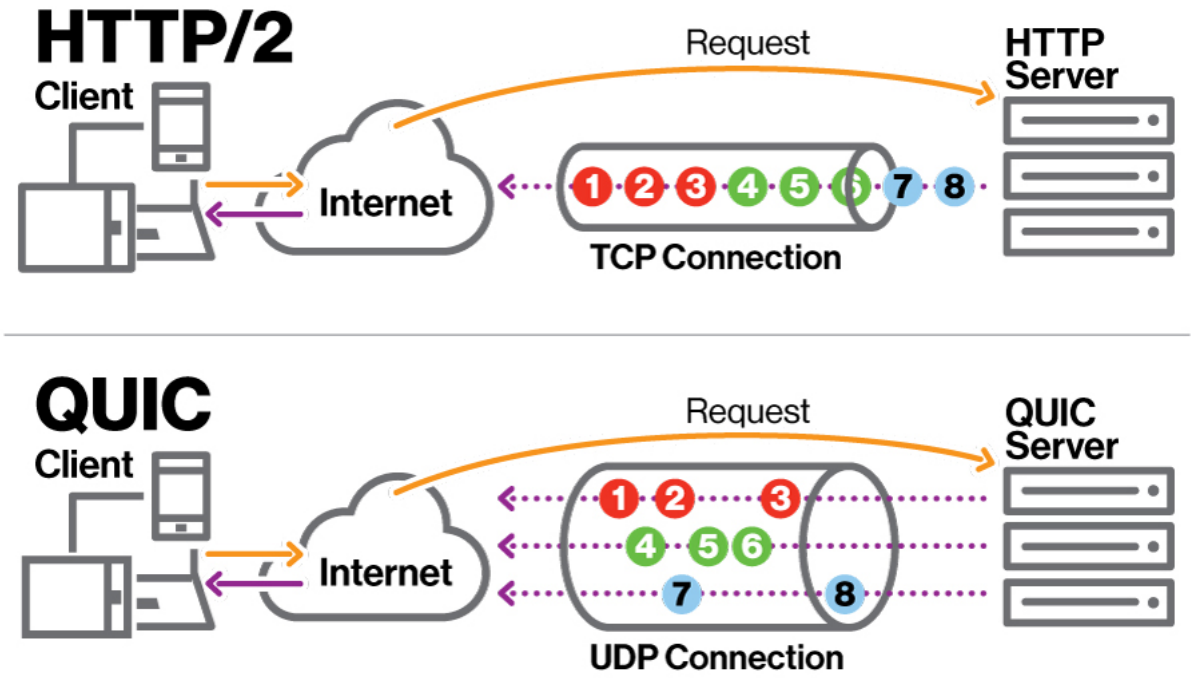
TLS IS SORTA IMPORTANT



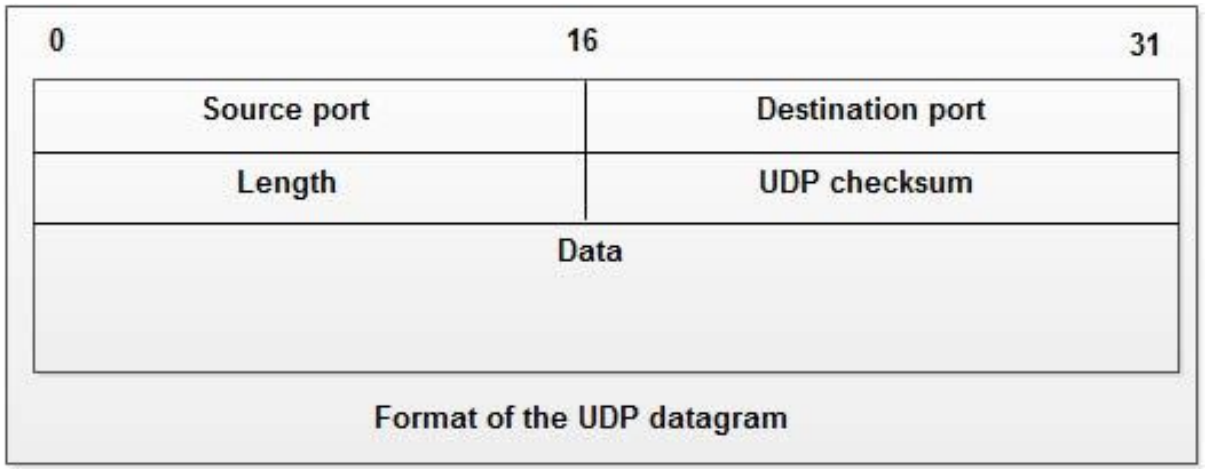
HTTP/2 IS THE FUTURE



HTTP/3 IS THE FUTURE-FUTURE

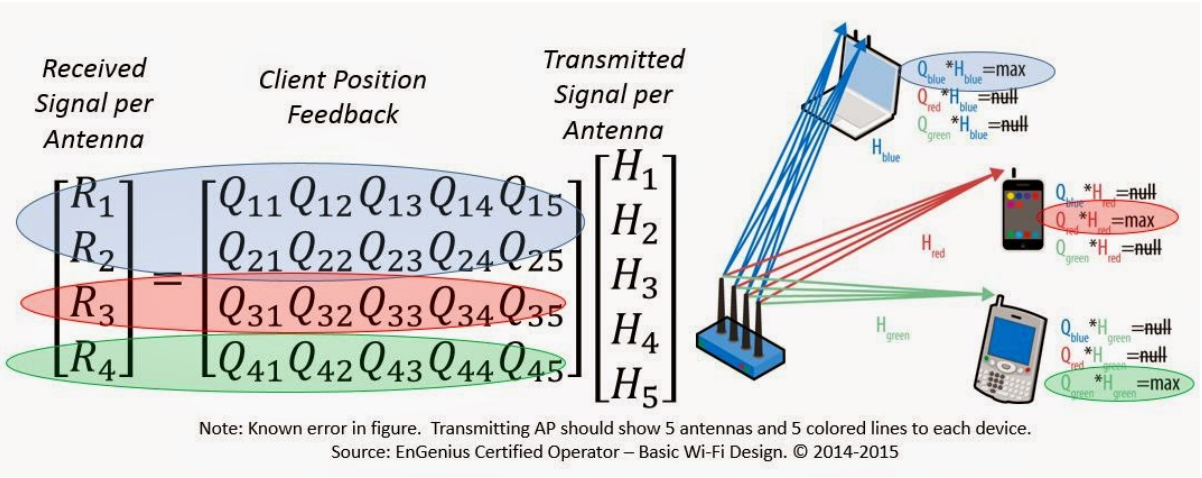


ENTER UDP



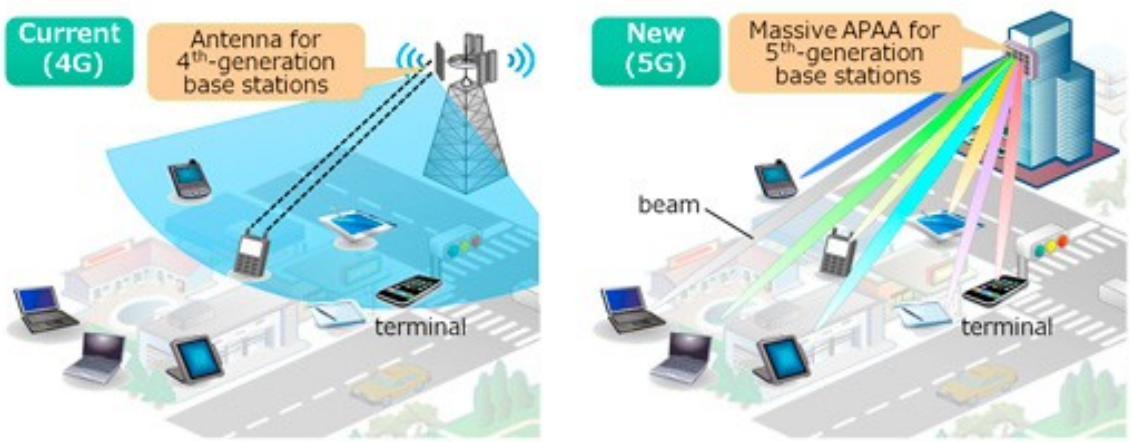
Format of the UDP datagram

WI-FI IS NICE TO HAVE

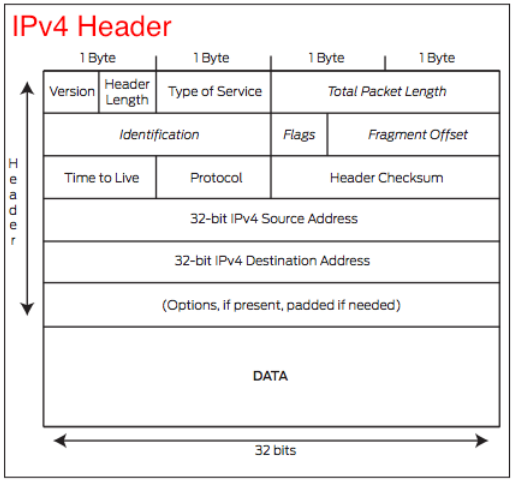


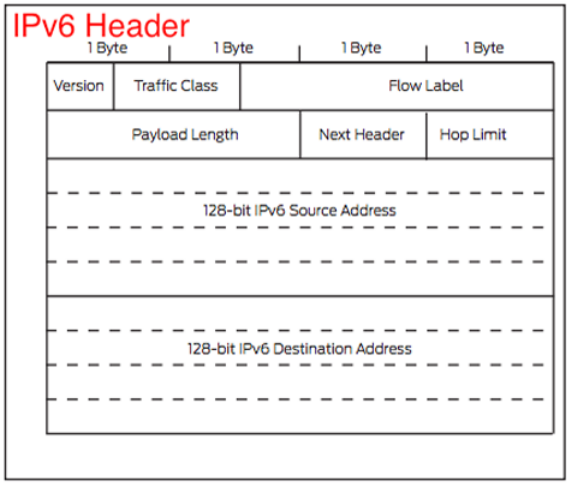
Note: Known error in figure. Transmitting AP should show 5 antennas and 5 colored lines to each device.
Source: EnGenius Certified Operator – Basic Wi-Fi Design. © 2014-2015

LTE AND 5G ARE PRETTY POPULAR



IPV6 IS IMPORTANT





NOT TO MENTION A MILLION OTHER CONCERNS

- Latency
- Performance
- Security
- Predictability

IN SUMMARY


IF YOU WANT TO BAKE AN APPLE PIE FROM SCRATCH
YOU MUST FIRST CREATE THE UNIVERSE
quickmeme.com

IT IS A WONDER ANYTHING EVER WORKS AT ALL

IT IS A WONDER ANYTHING EVER WORKS AT ALL

BUT IT DOES

HOW DO WE BUILD SYSTEMS THAT ARE AS AMBITIOUS AS THE INTERNET IS?

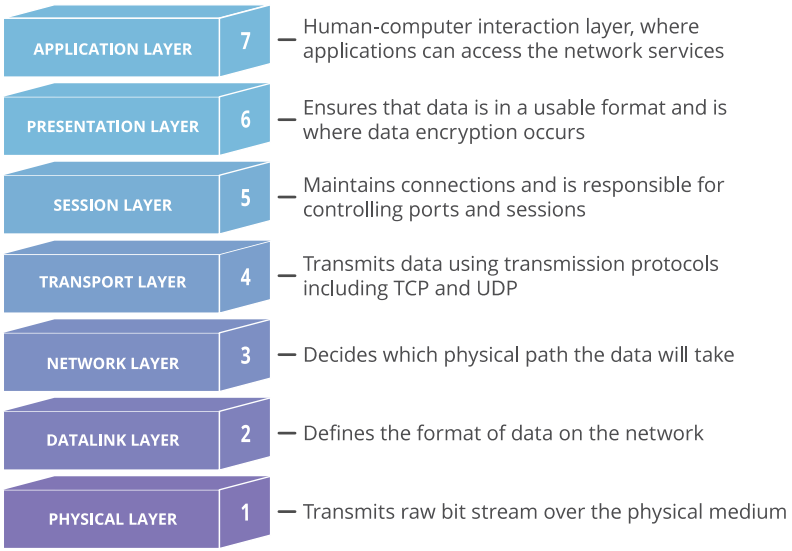
HOW DO WE DEAL WITH COMPLEXITY?

HOW DO WE DEAL WITH COMPLEXITY?

¯\(ツ)/¯

THERE MIGHT BE A FEW LESSONS WE COULD LEARN THOUGH

HOW TO MODEL


APPLICATION LAYER
7
Human-computer interaction layer, where applications can access the network services
PRESENTATION LAYER
6
Ensures that data is in a usable format and is where data encryption occurs
SESSION LAYER
5
Maintains connections and is responsible for controlling ports and sessions
TRANSPORT LAYER
4
Transmits data using transmission protocols including TCP and UDP
NETWORK LAYER
3
Decides which physical path the data will take
DATALINK LAYER
2
Defines the format of data on the network
PHYSICAL LAYER
1
Transmits raw bit stream over the physical medium

HOW TO LAYER

Not all of these abstractions are completely opaque

HOW TO GROUP AND SEPERATE CONCERNS

Some problems are best solved in one place and one place only (routing) others are stratified throughout the entire stack (error correction)

PATIENCE TO EVOLVE

It took us 30 years to get where we are today, how do we give more technologies the chance to evolve

RESOURCES TO MAKE CHANGE

We still have things we need to fix, continued investment is critical

THANKS!