Computer Programming – CS 6011
Lecture 17: Web Socket Implementation

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

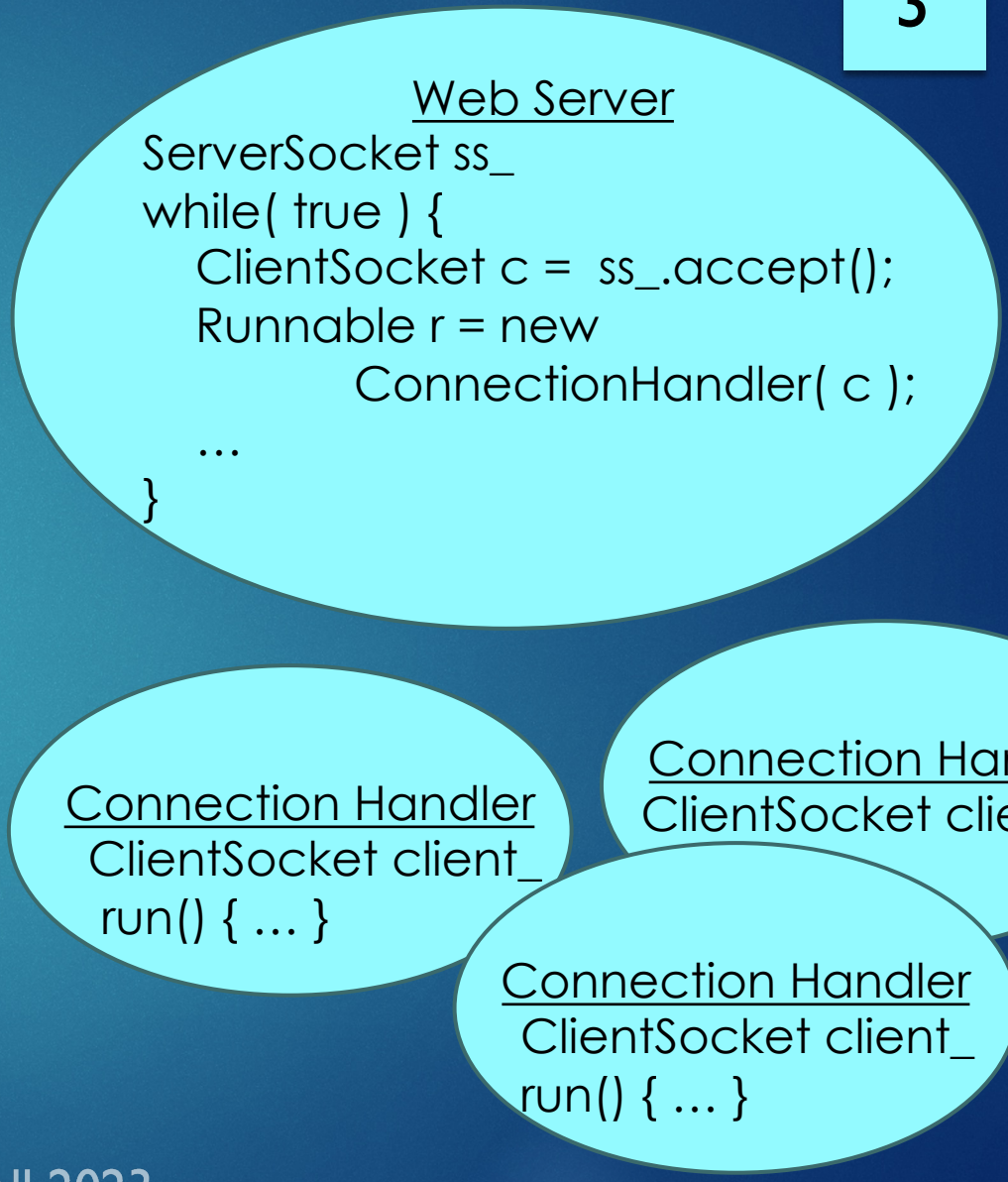
FALL 2023

Miscellaneous

- Passing Data to a Thread (Runnable)
  - Constructor
  - Global Variables
  - Starting / Joining
- Questions?

Web Server Classes

- How to organize the data / functionality of your web server? (Bubble Diagram!)
  - What are the main pieces?
    - The server itself.
    - A connection handler…
      - The ability to handle (parse) a request
      - The ability to send a response.
  - Given those pieces, what data does the server need?
    - A ServerSocket.
  - What data does a "connection handler" need?
    - The ability to read / write data to the client… so what data?
      - The Client Socket
  - Which of these objects should be the thread (or more specifically, the runnable)?
    - Connection Handler



Sharing Data Among Objects…

- Who creates (gets) the Client Socket?
  - The Server
- Who needs the client socket in order to do its job?
  - The Connection Handler (Runnable).
- Who creates a Connection Handler?
  - The Server
- How does the Connection Handler get the client socket?
  - The server gives the client socket to it… how?

```
class Server { // Pseudo Code
    private ServerSocket ss = …;
    public void main() {
        …
        // How does the server get the client socket?
        Socket client = ss.accept();
        // How does it give it to the connection Handler?
        ConnectionHandler ch = new ConnectionHandler( client );
        // Use the constructor ----------^^^^^^^^^^^^^^^^^^^^.
        // How to make this ConnectionHandler go do its thing?
        Thread t = new Thread( ch )
        t.start();
        // Shove it into a thread and start it running.
        …
    }
```

- Notice, no HttpRequest or HttpResponse objects in the above code… Why not?
  - The ConnectionHandler uses them, not the Server!

Connection Handler // Pseudo code


```
class ConnectionHandler  implements Runnable {
        // In order to read headers / send a response, what does the CH need?
        Socket client_;
        // How does the CH get the client socket?
        public ConnectionHandler( Socket client ) {
                // what code goes here?
                client_ = client;
        }
        // Methods?
        void run() {
                // We use client_ for the following:
                // what happens here (high level)?
                // read / parse request headers
                // send response headers / file to requestor
        }
```

```
// How do we read / parse headers in one line of code?
request = new HttpRequest(  client_ );
request.parse(); // ok, maybe 2 lines…
// How to send a response (again, with only one line)?
response = new HttpResponse( client_ ,  request.getFilename() ,  request.getHeaders() )
```

Lecture 17 – Topics

- Web Sockets
  - Implementing a WebSocket (Chat) *Server*
  - We'll be discussing this over the next two days…

WebSocket Protocol

- First, we need to understand the protocol.
  - HTTP Request followed by a HTTP Response…
    - This is called a *handshake*.
      - What header fields are used, and how are they set?
  - WebSockets then switch to using binary data packets.
    - What does this look like?
- Look at these documents… can you answer the above questions?
  - https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers
  - https://datatracker.ietf.org/doc/html/rfc6455

Some Key Points From The Docs…

- From: https://datatracker.ietf.org/doc/html/rfc6455
- 1.1 Background
  - Historically, creating an instant messenger chat client as a Web application has required an abuse of HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls.
- 1.3 Handshake
  - Important header fields?
    - Request
      - Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
    - Response:
      - Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
- 5.2 Base Framing Protocol
  - The specification of a WebSocket message (Header + Payload)

Starts with HTTP Protocol

- What is the first thing that happens when a client tries to establish a WebSocket connection to a server?
  - A Handshake occurs… in other words, the client:
  - Sends an HTTP Request… So we need to parse the request for the relevant information:
  - How do we (or did you) store the request fields?
    - Map
      - HashMap (the actual implementation of a Map) – using what type of Keys / Values
      - HashMap< String, String >
  - Which key(s) do we look at to decide what type of request this is?
    - Connection: Upgrade
    - Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==

```
GET /chat HTTP/1.1
Host: example.com:8000
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dG…ZQ==
Sec-WebSocket-Version: 13
```

Responding to a Request

- Remember, HTTP Requests coming to your server can be normal requests (like what we have handled already), or a WebSocket request.
  - Once we have determined which type of request it is (previous slide) – if it is a WebSocket…
- Handshake Request we will need to send back the required response (as a normal HTTP Response)… [See Handshake on next slide]
  - and then…
- Switch to the WebSocket protocol for all future messages.
  - From plain text (ASCII) Request / Response, switch to a binary protocol.

```
void run() {
  // We use client_ for the following:
  // what happens here (high level)?
  // read / parse request headers
  // send response headers / file to requestor
  // After handing a normal http request,
  // what is the last thing we actually need to
  // do here?
  client_.close();
}
```

WebSocket Handshake

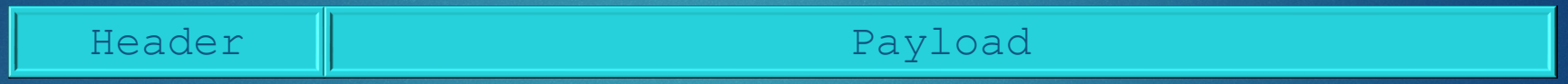
- What response header (key) do we use to respond to the WS request?
  - `Sec-WebSocket-Accept: <value>`
- What `<value>` will it contain (ie, what value do we put in this field)?
  - Concatenate the client's Sec-WebSocketKey and the magic string:
    - "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"
  - Then use:
    - SHA-1 Hash and Base64 encoding
    - MessageDigest class will do the SHA-1 hashing
    - Base64 class will encode the output of the MessageDigest into a string.
- How to test that the Handshake worked?
  - Wireshark can show that the server is sending packets back.
  - But much easier… how does the browser know when a WebSocket handshake has completed?
    - It runs the `ws.onopen` callback.

GET /chat HTTP/1.1
Host: example.com:8000
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dG…ZQ==
Sec-WebSocket-Version: 13

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3p…o=

WebSocket Messages

- Server doesn't close the client socket, it continues to use it to send data…
- However, the messages now look like: (because this is a binary format)



- On client (web browser / JavaScript), it's easy:
  - `ws.send( "my message" );` // Client library formats message for us.
- On the server, it's (a little?) more complicated…
  - First, why do we need the `Header` for this message (mostly)?
    - It has the length of the payload. (This is common to almost all binary formats.)
    - Also a few other fields of interest…

Example WebSocket Message

- Data going across the network (representing a WebSocket message) looks like:

```
<-  0x82 0xFE 0x0100 0xABCD …
```

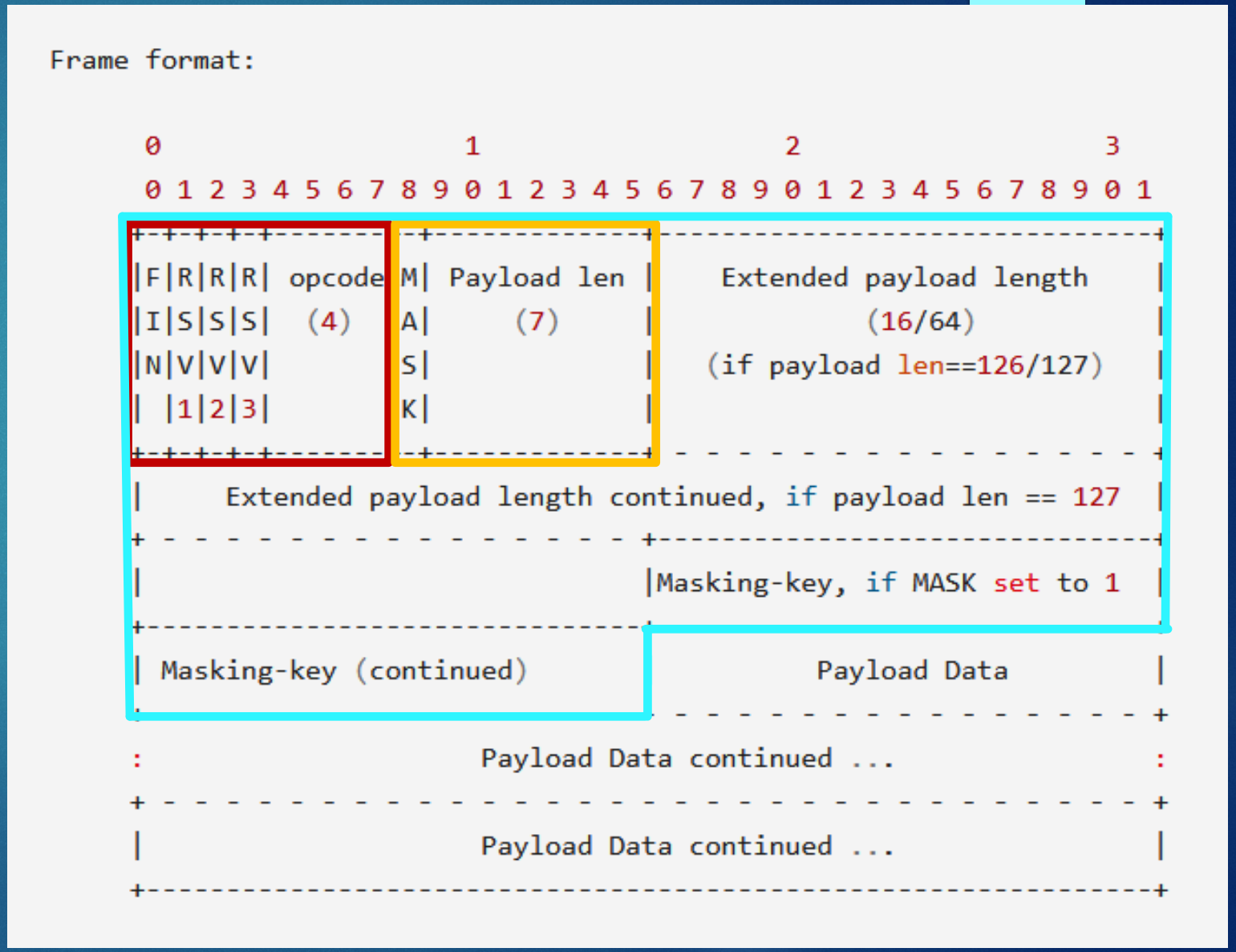
- In Binary this looks like:

```
<-  1000 0010   1111 1110   0000 0001 0000 0000   1010 1011  1100 1101   ……….
```

  - What do each of those bits mean?
    - Let's figure that out starting on the next slide…

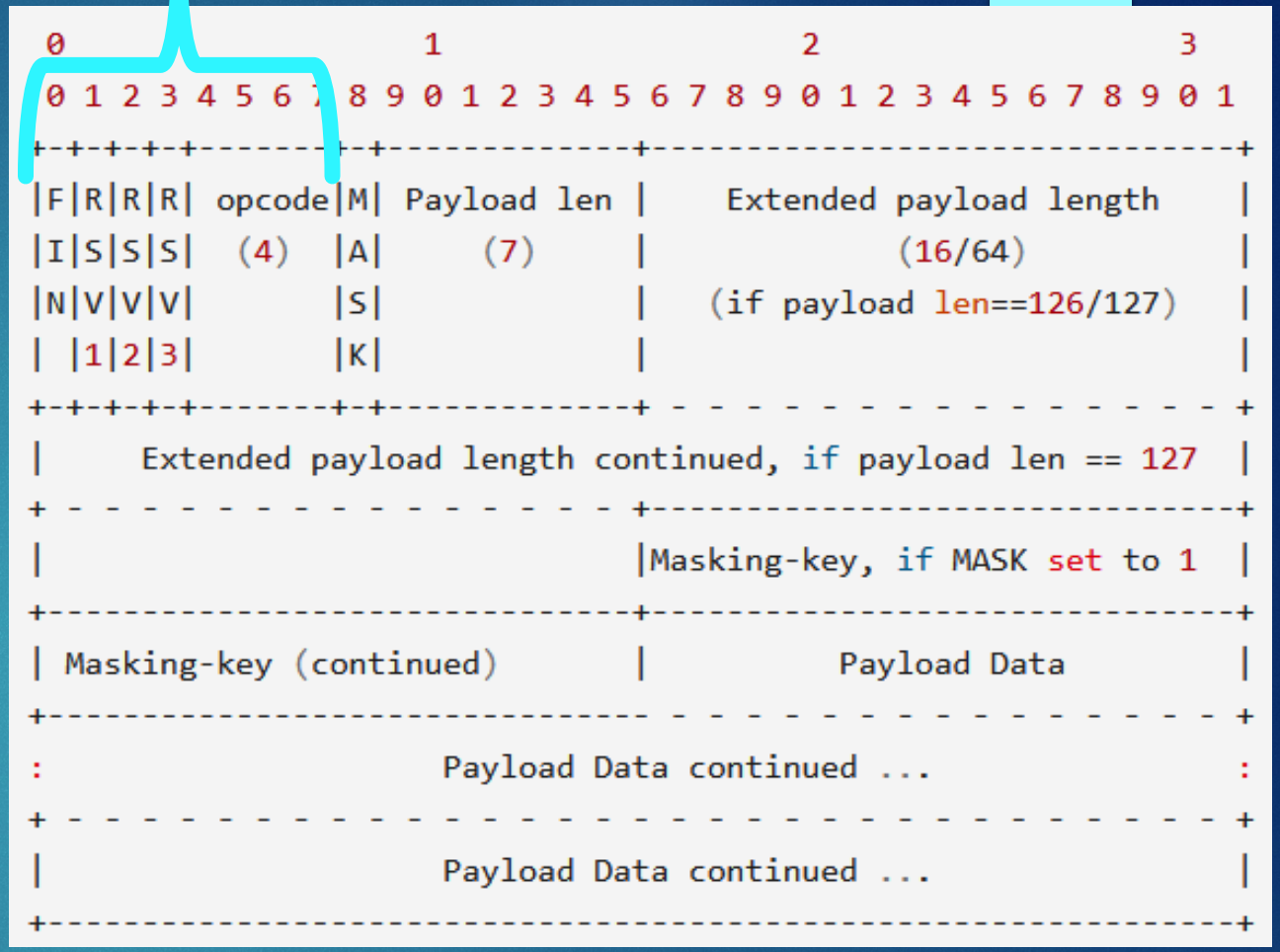
WebSocket Header

- Bytes are shown left to right.
- How big is the header?
  - Often 2 bytes, but…
  - Length data can be in byte 1 (2nd byte), or be the next 2 (or 8) bytes
- Byte 0 (Shown In Red)
  - Opcode?
    - `b0 & 0x0F;`
- Byte 1 (Shown in Orange)
  - Masked?
    - `(b1 & 0x80) != 0;`
  - Also contains the length*.



WebSocket Header

- Max size of header?
  - 14 bytes
- Header Size:
  - If Payload is 0-125 bytes, with a client to server message:
    - Header size: 6 bytes (2 + mask)
- …

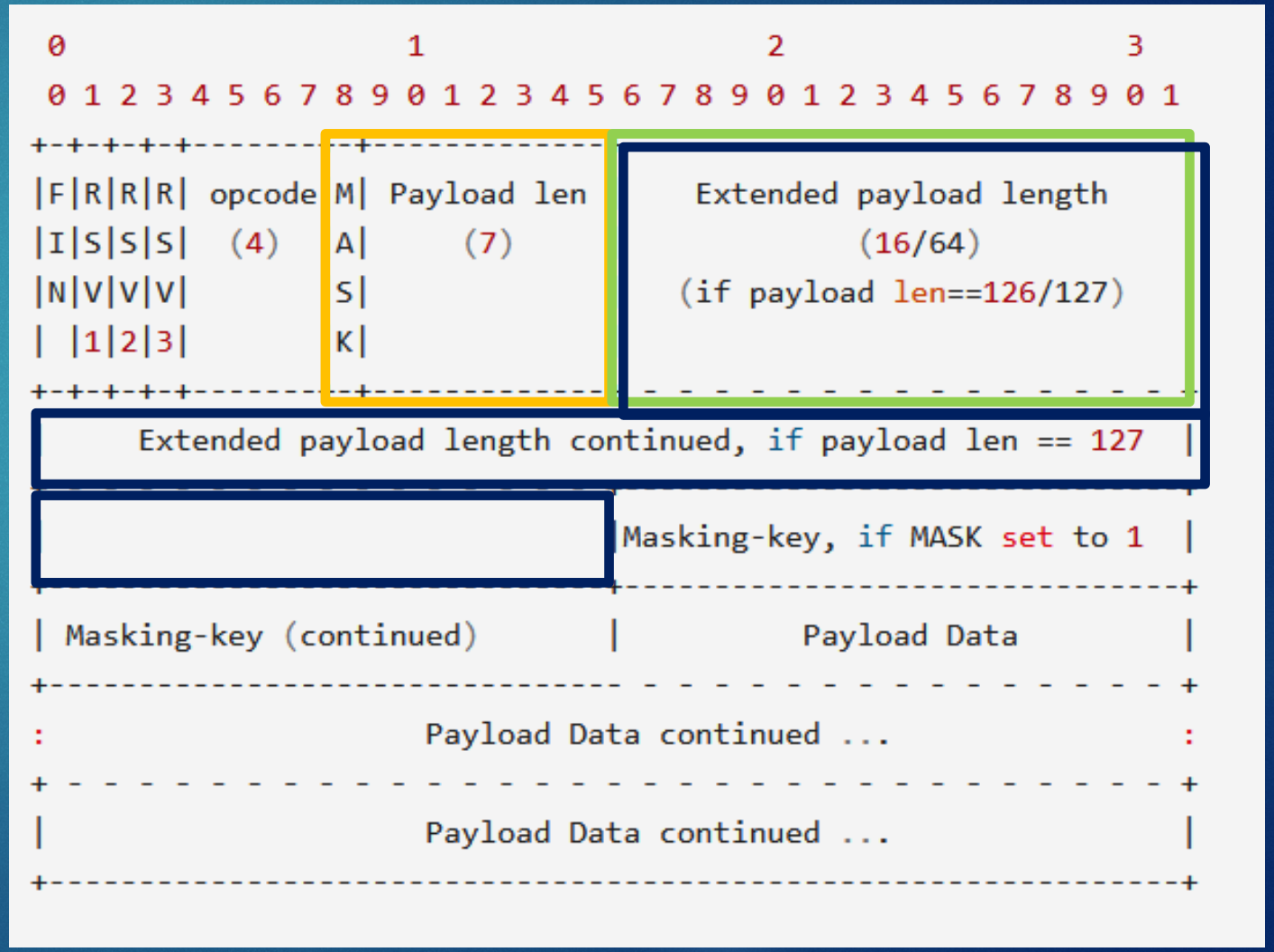


0	1	2	3	4	5	6	7	8	9	10	11	12	13	…	…
f op	m len	[opt	len]	[more	opt	ional	len	bytes	…]	[opt	ional	…	mask]	[Payl	oad..

First byte: Fin / Rsv / Rsv / Rsv / Opcode

WebSocket Header

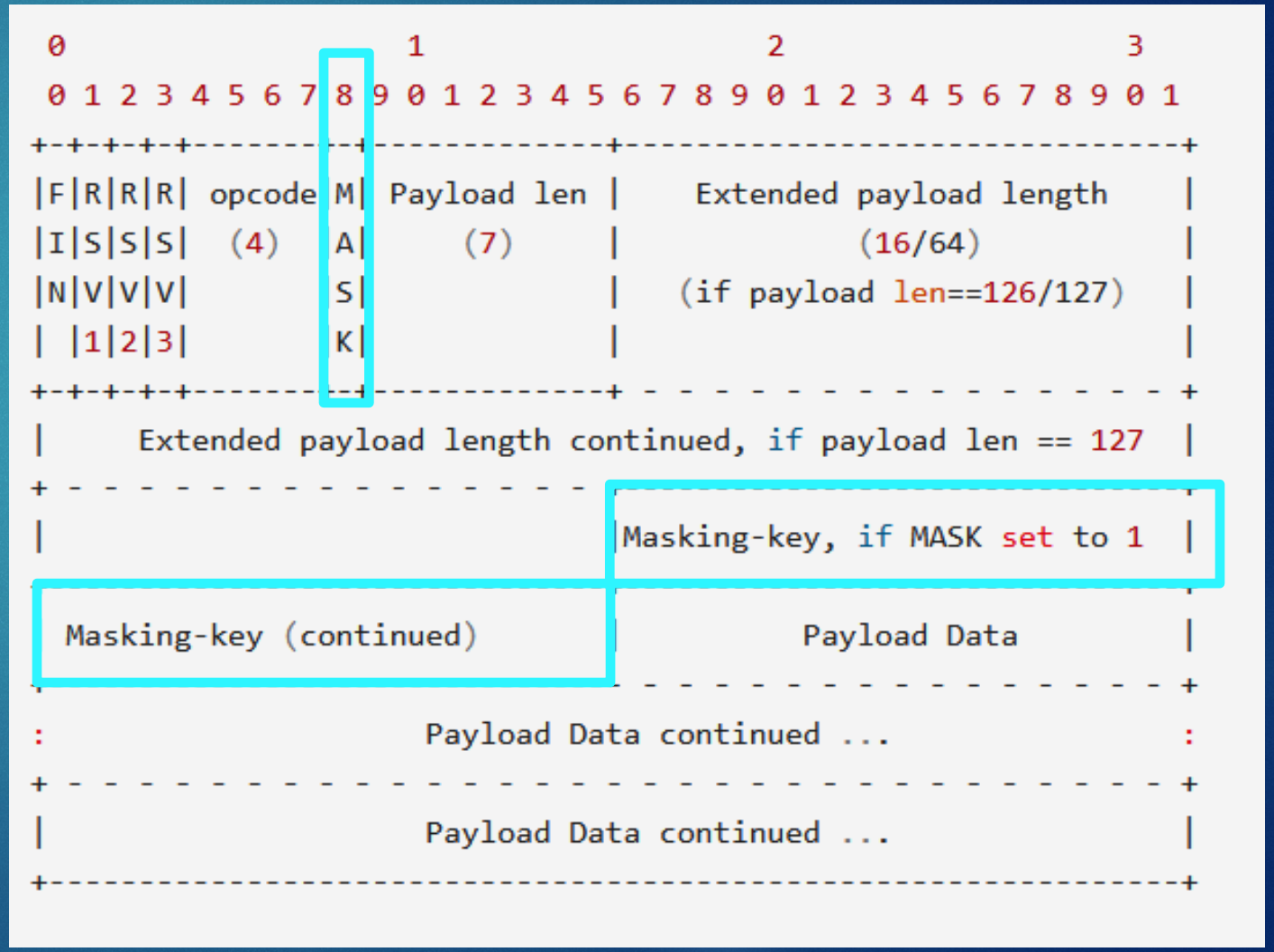
- Header size changes based on?
  - Length of payload
- If payload length is 125 bytes or less, then the length is found in b1 (remember to disregard the mask bit).
  - `lenGuess = b1 & 0x7F`
- If the value in b1 is 126, then the length of the payload is found in bytes b2-b3.
  - Type? Range?
    - *Unsigned* Short… why?
- If the value in b1 is 127, then the length of the payload is found in bytes b2-b9.
  - Type?
    - Long (Will this ever happen?)



WebSocket Header

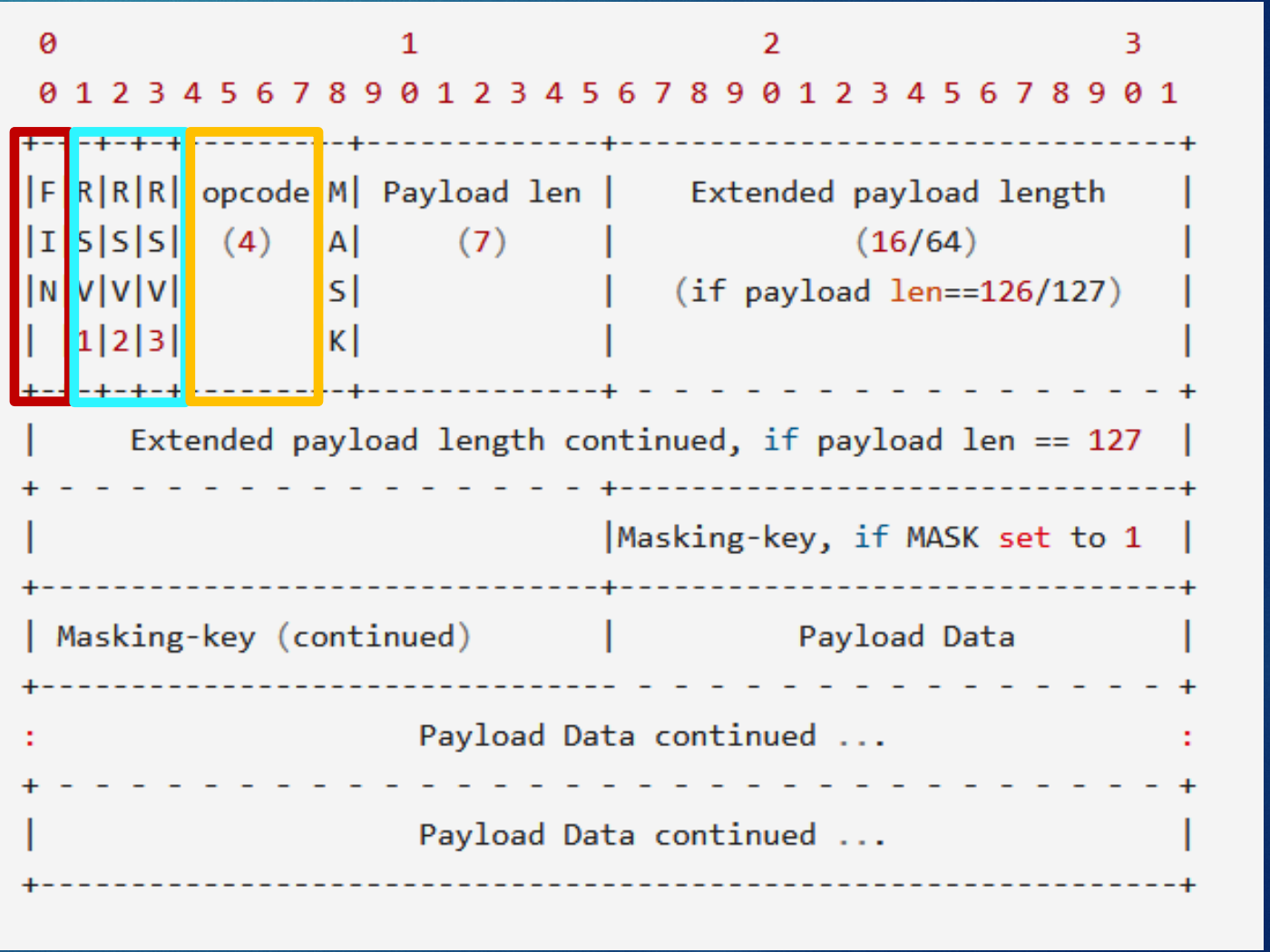
- Mask is only used for messages that go from? [RTFM*]
  - *Read the manual. ☺
  - Client to server.
- For these messages, the mask bit will be set to 1, and the masking-key will be 4 bytes.
- Conversely, messages sent from the server to the client are not masked, and there is NO masking-key.
- Code for unmasking is straightforward (and right out of the manual)…

```
var DECODED = "";
for (var i = 0; i < ENCODED.length; i++) {
    DECODED[i] = ENCODED[i] ^ MASK[i % 4];
```



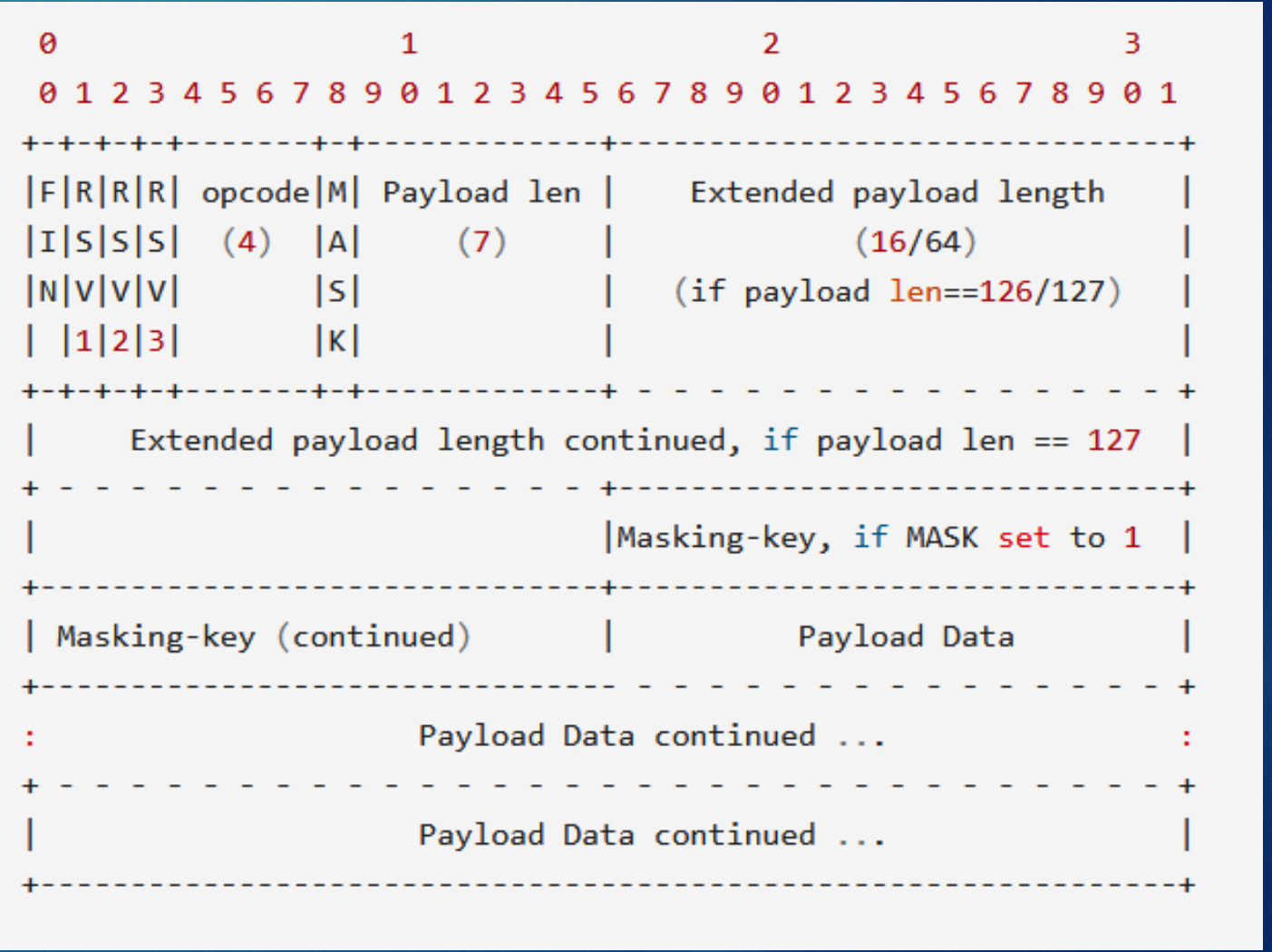
Other Header Fields

- FIN bit?
  - WebSocket messages can be sent in parts, though we probably won't see these…
  - FIN of 1 means this "series of messages" is over… Since we will only have one message in each series, this should always be 1.
- RSV – Reserved
  - Bits reserved for future use if the standard is updated
- Opcode
  - Is it a text message, binary, etc?
  - For the most part we really don't care about the opcode, but it would be good to check to make sure that it is a 0x1 (ie: a text message). [Note: An opcode of 0x2 means a binary message).]
  - Note, opcode of 0x8 means the client is about to close the connection…
    - Ie, the browser window is closing



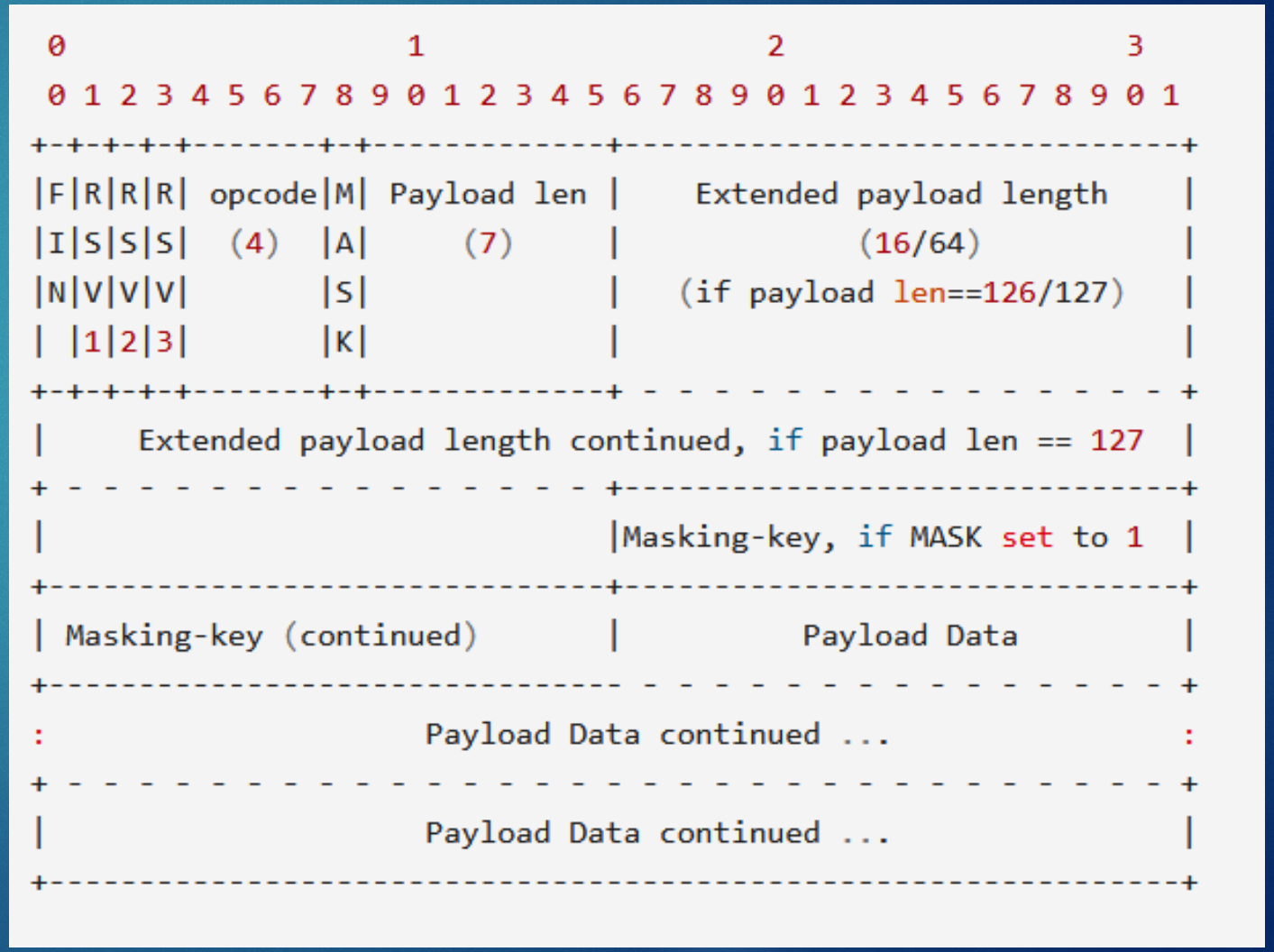
Steps to read the packet…

- How big is this entire message? How many bytes can we read from the socket?
  - We don't know… yet.
- How big is the header?
  - Could be several sizes…
- How much of the header do we know will always be the same size?
  - First 2 bytes.
- Once we have the first 2 bytes, how much more "header" do we read?
  - Depends on the length field.
  - Read 0, or 2, or 8 bytes more.
- Are we at the payload yet?
  - Depends on the mask bit.
  - If it is set to 1, read how much more?
    - 4 bytes.
- Now we are at the payload. ☺



In Java:

- Get the InputStream from the socket…
- DataInputStream wraps a normal InputStream (like a Scanner does).
- What does the Scanner do for us?
  - Makes it easy for us read strings, ints, etc.
- DataInputStream makes it easy for us to read groups of bytes (binary data).
  - readNBytes(X)
  - readFully(byte [])
  - readShort()
  - readLong()
  - readInt()



Incoming WebSocket Message

- We have finished the handshake and now it is time to read a real WebSocket message from the client socket.. Message looks like this…

```
<-  0x82 0xFE 0x0100 0xABCD …
```

Remember, http packets look like this:

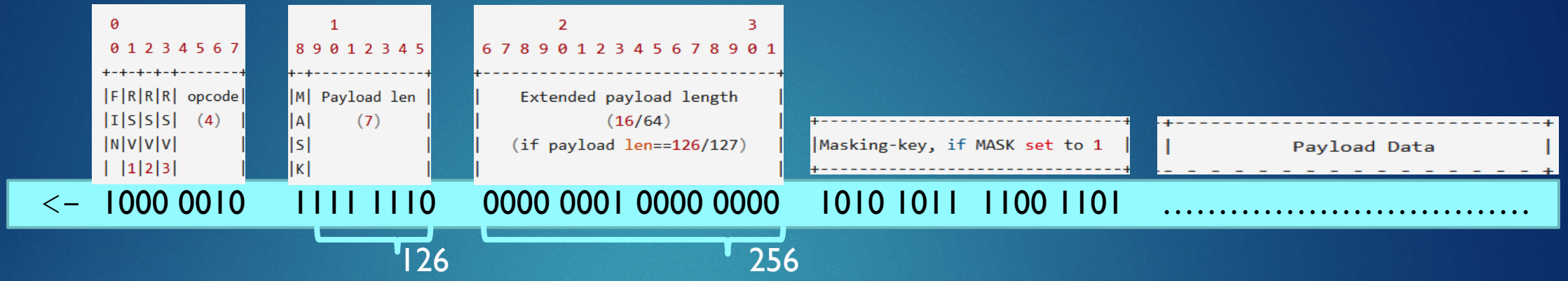
```
<-  GET /file HTTP/1.1\r\nHost: google.com\r\n\r\n
```

- What does this mean?
  - Perhaps it is easier if we look at it in binary?

```
<-  1000 0010   1111 1110   0000 0001 0000 0000   1010 1011  1100 1101   ……….
```

  - What do each of those bits mean? Let's look at them one byte at a time…
    - But first, do we know how long the entire message is?
      - Not yet! So how many bytes can we safely read to get started understanding / parsing this message?
        - 2

Incoming WebSocket Message

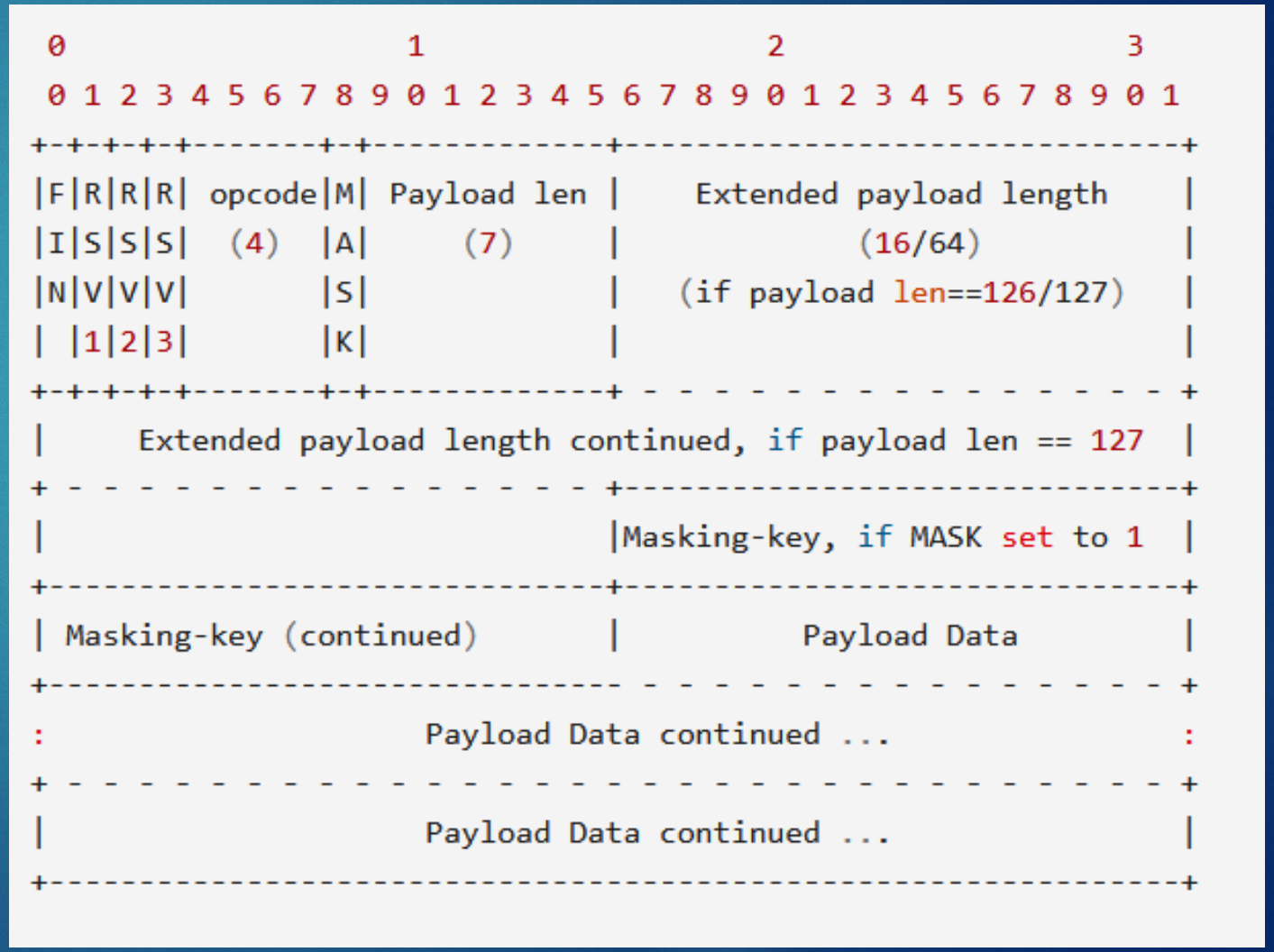


- Read 2 bytes… what does the 1st byte tell us?
  - Does looking at the spec help?
  - Fin bit is true – this is the only packet in this msg.
  - Opcode of 2 – means binary message.
- What does the 2nd byte tell us?
  - Spec?
  - Message will be masked.
  - Length is 126… which means?
  - Spec?
  - The next two bytes are the real length of the payload.
  - So the real length is?
- 256 bytes.
- What's next after the length fields?
  - Spec?
  - The mask… which is?
  - 0 x A B C D
- Finally, read the rest of the message… What is this called?
  - Payload – how many bytes?
    - 256

Responding

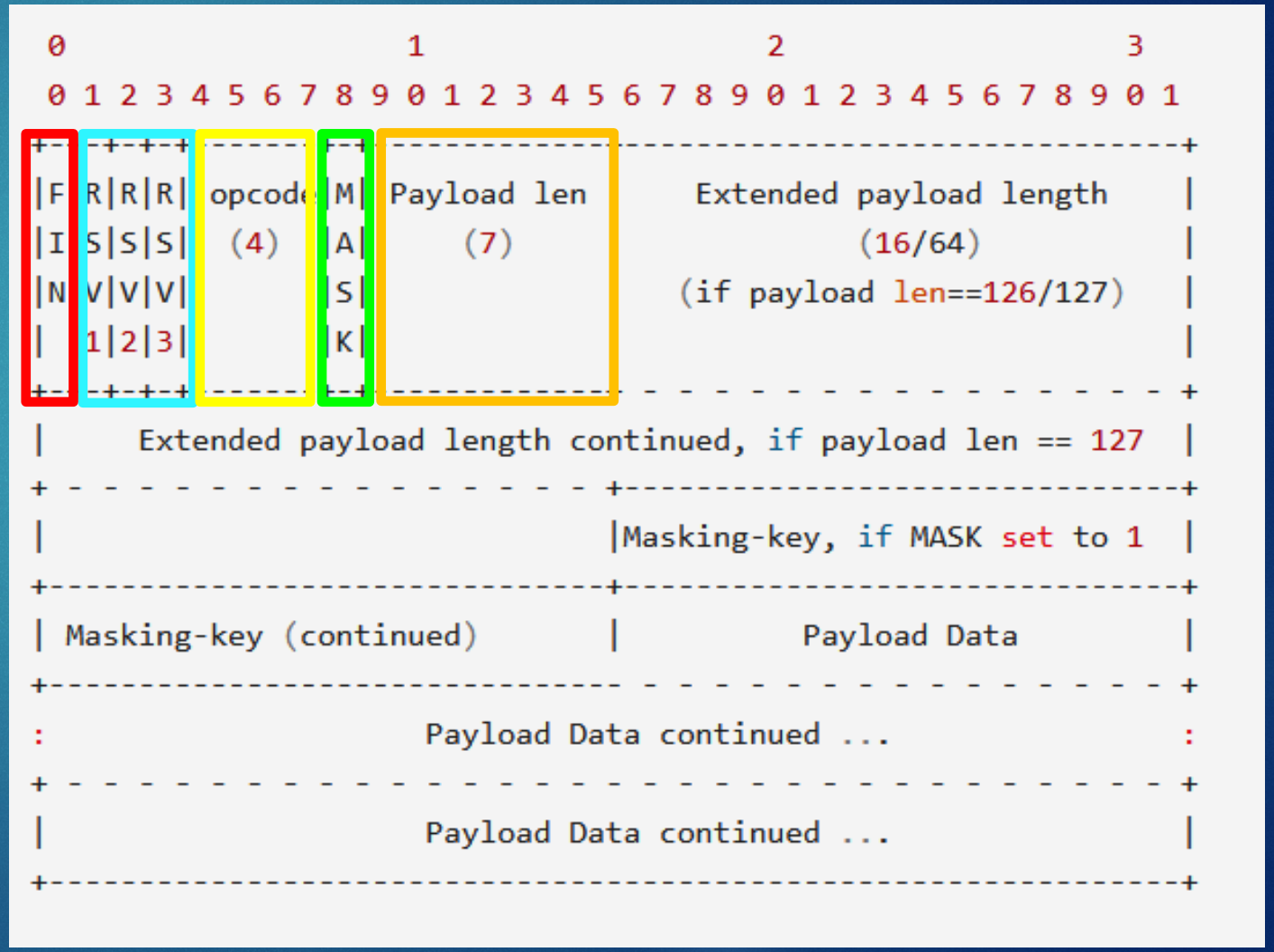
- What does the server do after it receives (and parses) a message (over a WebSocket)?
  - Sends back a* response. (more later)
- Let's say the payload is:
  - "Davison How are you?"
- What message does the Server send?

  '{ "type" : "message", "user" :"Davison", "room" :"testroom", "message" :"How are you? "} '
- How is this message sent?
  - As a WebSocket message… which means?



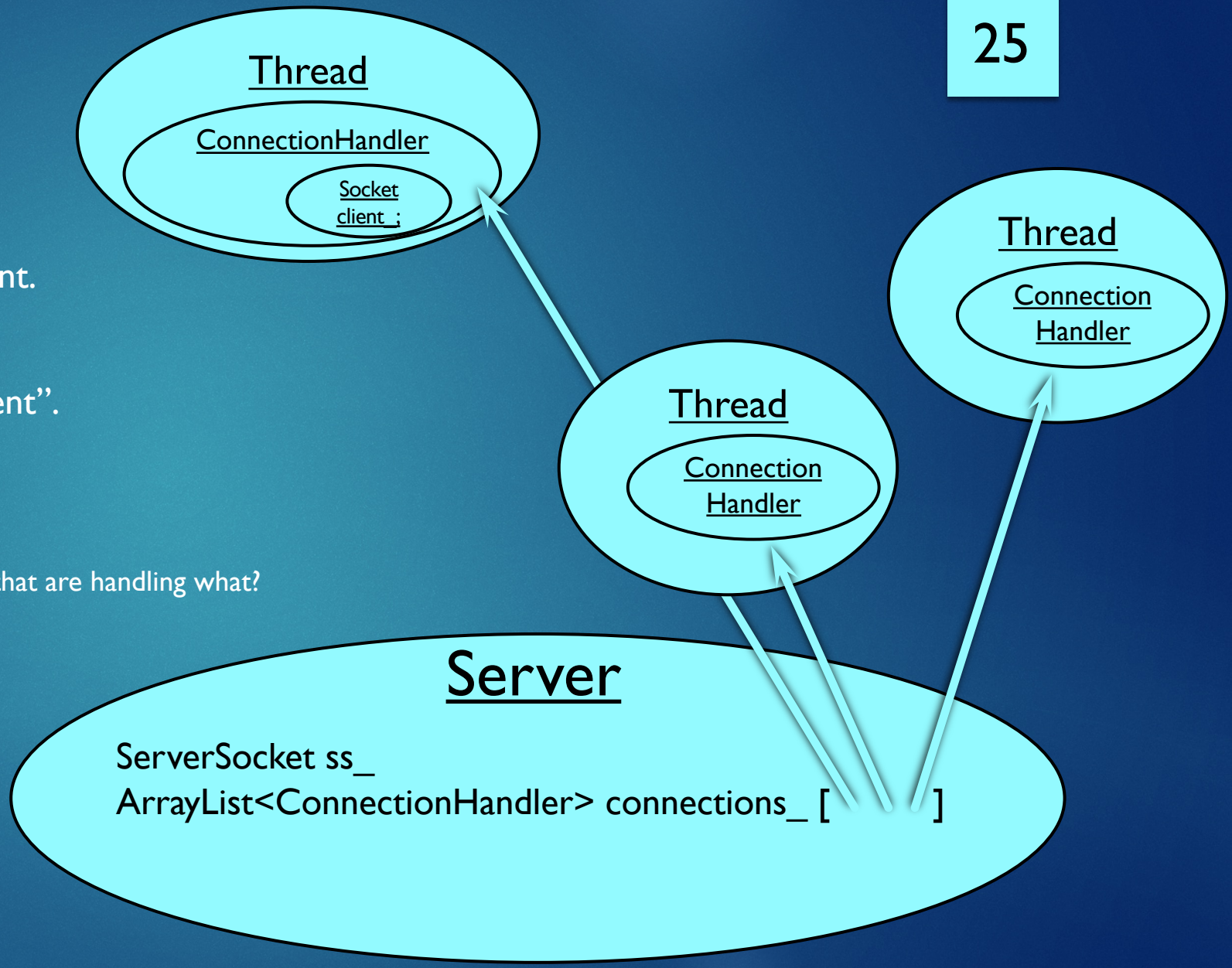
Responding

- Fill in the information need on based on the specification to the right:
- 1000 0001 0000 1100 <payload>
- What do all those 1s and 0s mean?
- 1000 0001 0000 1100 <payload>
- 1 – Final packet in this message.
- 0001 – Opcode. Value? Means?
  - 1, Text Message
- 0 – No masking… why?
  - Spec says messages from server to client are not masked.
- 010 1100 – Length. Value?
  - 122 – '{ "type" : "message", "user" :"Davison", "room" :"testroom", "message" :"How are you? "}
- Payload?
  - Just ASCII '{ "type" : "message", "user" :"Davison", "room" :"testroom", "message" :"How are you? "}

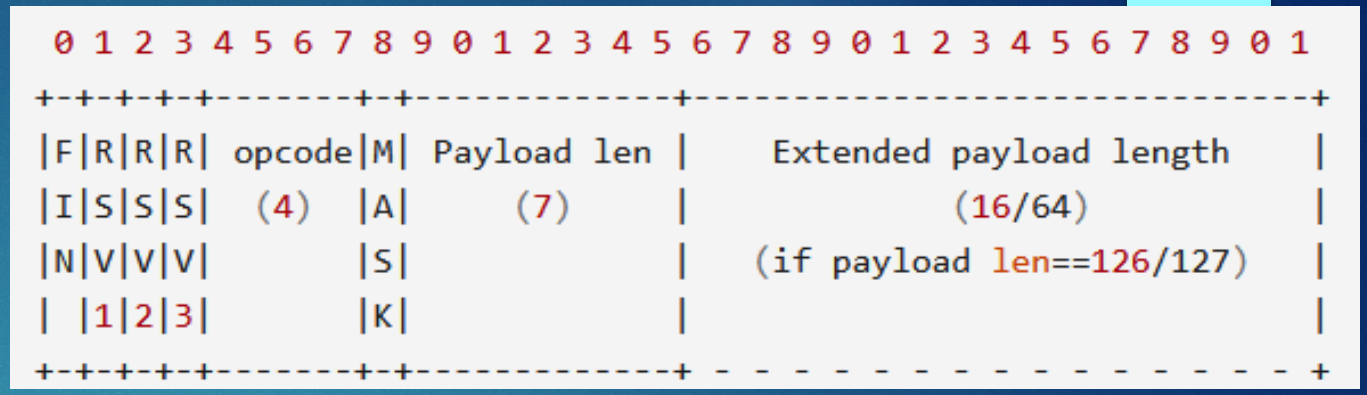


Responding

- Sends back a* response… ???
  - Only one?
  - A message to every connected client.
    - For now.
  - What determines a "connected client".
    - The ConnectionHandler Thread…
    - Always?
      - No – only ConnectionHandlers that are handling what?
        - WebSocket requests
  - Who keeps track of all the ConnectionHandlers?
    - The Server – How? Draw a bubble diagram of the program.



Testing



- You can use the Spec – Section 5.7:
  - Unmasked text message:
    - 0x81 0x05 0x48 0x65 0x6c 0x6c 0x6f
      - contains a body of "Hello“
  - Masked text message:
    - 0x81 0x85 0x37 0xfa 0x21 0x3d 0x7f 0x9f 0x4d 0x51 0x58
      - contains a body of "Hello"
  - 256 bytes binary message in a single unmasked frame
    - 0x82 0x7E 0x0100 [256 bytes of binary data]
  - 64 KiB binary message in a single unmasked frame
    - 0x82 0x7F 0x0000000000010000 [65536 bytes of binary data]
- Can use your client to send a (WebSocket) message to the server and decode it.

Echo Server

- Client sends a WebSocket message
- Server reads it, decodes it, and sends it back

- Why are we using a Threaded server?
  - Because the WebSocket stays open "forever" on the server sitting in a loop:

```
while( true ) {
    // Read (WebSocket) message.
    // Respond to message… with a WebSocket message.
}
```

- Note, the server should continue to serve normal HTTP requests also.

Wednesday Assignments

- Code Review – Multithreaded Web Server
- Assignment – WebSocket Echoes

~ Fin ~