bSocket Framing: Masking, Fragmentation and More

29 May 22

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-------+-+-------------+-------------------------------+
 |F|R|R|R| opcode|M| Payload len |    Extended payload length    |
 |I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
 |N|V|V|V|       |S|             |   (if payload len==126/127)   |
 | |1|2|3|       |K|             |                               |
 +-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
 |     Extended payload length continued, if payload len == 127  |
 + - - - - - - - - - - - - - - - +-------------------------------+
 |                               |Masking-key, if MASK set to 1  |
 +-------------------------------+-------------------------------+
 | Masking-key (continued)       |          Payload Data         |
 +-------------------------------- - - - - - - - - - - - - - - - +
 :                     Payload Data continued ...                :
 + - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
 |                     Payload Data continued ...                |
 +---------------------------------------------------------------+
```

The above diagram, taken from RFC 6455 The WebSocket Protocol, describes a WebSocket frame.

The smallest valid **full** WebSocket message is two bytes, such as this close message sent from the server with no payload: `138, 0`. Yet the longest possible **header** is 14 bytes, which would represent a message sent from the client to the server with a payload greater then 64KB.

Sending "over9000" from the client to the server will be 14 bytes with a seemingly random last 12 bytes, something like `129, 136, 136, 35, 93, 205, 231, 85, 56, 191, 177, 19, 109, 253`. Yet that same message sent from the server to the client will always be these exact 10 bytes: `129, 8, 118, 101, 114, 57, 48, 48, 48`.

Why is the longest possible header longer than the shortest possible message? Why does a message sent from the client to the server look so different than the same message sent from the server to the client? And, are there any other WebSocket oddities?

Payload Length

As a stream-oriented protocol, TCP has no concept of messages (or framing). It's just a sequence of bytes. If one side uses the `write` function four times, the other side might get those bytes across 1-N calls to `read`, where N will be the total number of bytes sent (i.e. where each read only reads a single byte). Solutions that sit on top of TCP, like WebSockets, have to provide their own framing. One common solution is to length prefix every message. For example, if we wanted to send two messages, `get power` and `set power 9000`, using a 2-byte length prefix, we'd send:

`0, 9, 'g', 'e', 't', ' ','p','o','w','e','r'`

and,

`0, 14, 's', 'e', 't', ' ','p','o','w','e','r', ' ', '9', '0', '0', '0'`.

WebSocket uses the first 7 bits of its 2nd byte to support a variable-length length prefix. When those bits are equal or less than 125, then the length of the payload is this value itself. When the bits equal 126, then the next 2 bytes indicate the length of the payload. When the bits equal 127, the the next 8 bytes indicate the length of the payload.

For example, if we wanted to send "hello", then the 2nd byte would contain all the length data needed, namely: `byte2 & 127 == 5`. But if we wanted to send a payload that was 300 bytes long, then we'd get `byte2 & 127 == 126` which would tell us to look at the next two bytes to get the length of the payload, in this case, they'd be `1, 44` (e.g. 256 + 44).

The benefit of having this variable length length-prefix is that messages with payloads that are 125 bytes or less only require a single byte. However, messages greater than 125 bytes and less than 64K will require 3 bytes (like the

300 byte example we just saw: `126, 1, 44`). Larger messages require 9 bytes.

I'd prefer a fixed 4-byte length prefix. I think it's safe to assume that most WebSocket messages are less than 16K, so this would mean most messages would be 1 byte larger. For small message, it would be 3 bytes longer. But a fixed-length length would be easier to deal with and have fewer error cases. I can't help but wonder if that 1 byte is worth those extra if statements and error cases.

Masking

Following the length portion of our frame is a 4-byte mask. From the specification: "The masking key needs to be unpredictable; thus, the masking key MUST be derived from a strong source of entropy, and the masking key for a given frame MUST NOT make it simple for a server/proxy to predict the masking key for a subsequent frame".

Masking is a security feature that's meant to thwart malicious client-side code from having control over the exact sequence of bytes which appear on the wire. Section 10.3 has more details.

The mask and payload are XOR together before being sent from the client to the server, and thus the server must reverse this. The byte value of "hello" (in ASCII/UTF-8) is `104, 101, 108, 108, 111`. But masked with `1, 2, 3, 4`, it becomes:

```
105, 103, 111, 104, 110
```

or, if you prefer:

```
'h' ^ 1, 'e' ^ 2, 'l' ^ 3, 'l' ^ 4, 'o' ^ 1
```

Since the 4-byte mask is sent as part of every client-initiated message, the server just reverses the process to get the unmasked payload (`('e' ^ 2) ^ 2) == 'e'`).

I have no idea of how important masking is in 2022. Maybe the threat is more serious than ever. Maybe browser security or WebSocket support have changed that WebSocket security should be re-evaluated. I don't know. But if you control the client and the server, and you're not concerned about malicious client-side code (like a desktop app), you could break the cryptographic requirements of the specification, while keeping the WebSocket frame valid, with a mask of `0, 0, 0, 0`. The server could detect this mask and skip the unmasking step. But make sure you know what you're doing - I don't.

A bit more annoying is that ALL messages from the client to the server MUST include a mask and the payloads must be masked (even messages with no payload must include the mask). And all messages from the server to the client MUST NOT be masked. Yet despite these strict requirements, the most-significant bit of our 2nd byte (remember, that length-byte which only used the 7 first bits?) is used to indicate whether or not a mask is present. This seems 100% redundant to me and only serves to introduce error cases.

Message Type

A WebSocket frame can be one of 6 types: `text`, `binary`, `ping`, `pong`, `close` and `continuation`. Furthermore, every frame is either a `fin` frame or not. The first byte of each frame is used to represent the type of frame (known as the op code) as well as whether or not it's a `fin` frame.

We'll talk more about `fin` and `continuation` next.

The difference between `text` and `binary` is that text must be valid UTF-8. I don't care for this distinction at a protocol level. It's wasted processing. If you care about UTF-8 validity, you'll probably check it again within your application (like when you try to decode the payload as JSON). So, use `binary` where possible just to avoid that check (especially on those 2 exabyte messages!).

`ping` and `pong` are also, in my opinion, unnecessary. Like `text`, it's better to handle this directly in application.

I do like the `close` type though. It has specific requirements around the payload (if a payload is present). Specifically, it requires the first two bytes of the payload to be a close code. It's useful for debugging.

Fragmentation

WebSocket has support for frame-fragmentation. One message can be sent across multiple frames. Given that WebSocket support 8-byte length prefixes (or exabyte-sized message), you might be wondering what this is for. I believe it exists for 3 reason, none of which you'll see very often.

The first case has to do with streaming, where the server doesn't have the full payload, doesn't know the final length, but still wants to send some data. The 2nd case has to do with being able to interrupt a large message with special control frames, such as ping (more on this later). The 3rd, a meta-reason I think, is that proxies are free to fragment messages for whatever reason they want (e.g. having smaller memory buffers), so long as they respect the rules around fragmentation (and every other part of the WebSocket specification).

The `continuation` and `fin` parts of the first byte are used to control fragmentation. The first fragment will have a `text` or `binary` type but `fin` will not be set. You'll then get 0 or more `continuation` frames where `fin` is still not set. Finally, you'll get 1 `continuation` frame with `fin` set. The payloads of each frame are concatenated together to form the final message.

I'm not a fan of this feature, at all. It makes the implementation much more difficult. There are a couple rules that make our lives a little easier. Only `text` and `binary` frames can be fragmented. Only control frames (`ping`, `pong` and `close`) can be inserted between fragmented frames. In other words, we're only ever dealing with a single fragmented message at a time.

However, even these restrictions on fragmentation mean that the life cycle for a frame gets more complicated. We no longer get one frame and pass it to the application. We have to accumulate frames, dealing with interleaved control frames, to create the message. It not only introduces a number of error cases, it makes memory management more complicated (it's hard to deal with fragmentation and interleaving without allocation more memory).

Autobahn

I need to send a shout-out to the open source Autobahn Testsuite project. It has a comprehensive number of test cases that validate the correctness of both client and server implementations. I think it's a model that every protocol should aspire to.

Conclusion

The shortest possible WebSocket frame of 2-bytes is a server-to-client message with no payload. Since there's no payload, the length is 0, and since it's server-to-client, there's no mask. The longest possible header is 14 bytes for a client-to-server message with a payload larger than 16KB: 8+1 bytes for the length and 4 bytes for the mask (plus the first fin/type byte).

"over9000" sent from the client to the server is longer and unpredictable because of the masking. Since the server to client message doesn't have a mask, it's always 4 bytes shorter and won't be random.

I'd love to better understand why specific parts of the protocol were designed the way they were. Why a variable-length length, and does it still make sense? Why both text and binary type? Why fragmentation?

Fragmentation in particular doesn't seem useful and, if removed, could still be implemented at the library/application level. I feel the same way about ping and pong. As I look at Slack's network traffic, I notice that they've implemented their own ping and pong logic.

This article was inspired by a recent Zig WebSocket library that I wrote. It isn't the first WebSocket server that I've written, and thus it isn't the first time that I've had these thoughts. If you want to see what parsing a WebSocket frame looks like, you can jump directly into the code.