Casual Explanation of the WebSocket Protocol

Milind Luthra

Refer to the actual *Internet Standards Track document* first.

This will help you get a brief idea of the nature and ordering of the steps, after which this document will help you understand them more intuitively.

Also refer to the relevant git repository on *github.com/milindl/learning-websockets*

Handshake

This is the first step. The browser initiates the WebSocket connection by sending a request typically structured like so:

```
Host: location_of_server
User-Agent: Mozilla/5.0
Accept: text/html
Sec-WebSocket-Version: 13
Origin: location_of_client
Sec-WebSocket-Protocol: arbitrarily_named_protocol
Sec-WebSocket-Key: n6SorMqVHOdlOyQvgBYeRg==
Connection: keep-alive, Upgrade
Upgrade: websocket
```

This connection is supposed to be an HTTP 1.1 "Upgrade". All this means is that since this is based on HTTP, you need to send an HTTP request to initiate the whole thing.

The `Sec-WebSocket-Protocol` is also important since the server has to reply with the same protocol. The `Sec-WebSocket-Key` is also important.

You need get SHA1 digest the `Sec-WebSocket-Key` to which a constant `258EAFA5-E914-47DA-95CA-C5AB0DC85B11` is appended, then encode it in base 64, then send it back. Apparently this is to make sure that the client isn't tricked by a nefarious server or whatever.

The response looks like so:

```
Connection: Upgrade
Sec-WebSocket-Accept: 8hq6y22P1VArkWL7LKm/dYtZ+NU=
Sec-WebSocket-Protocol: same_arbitrarily_named_protocol
```

`Upgrade: websocket`

It's fairly clear from the explanation above. Look how weird the `Sec-WebSocket-Accept` thing is. It's basically

```
base64(sha1(Sec-WebSocket-Key + 258EAFA5-E914-47DA-95CA-C5AB0DC85B11))
```

Sending: Frames

Since HTTP Clients and Servers don't use streams, we need to use something called frames. A frame is a bit of data with certain parameters appended in front of it to tell us how to interpret that data.

You should refer to the *actual document* for a better (and a more confusing) image of the frame, but I'll try to give a brief idea here. A frame looks like so -

```
(fin-1)(rsv-3)(opcode-4)(mask-1)(payload_size-7)
[optional - extended_payload_size]
actual payload data
```

The number suffix denotes the number of bits the information takes.

- `fin` : denotes whether the frame is self-contained(1 if it is), i.e. if it's the complete message you're sending. You can split your message across several frames. I have never needed to till now, since the size of a frame can be massive.
- `rsv` : Three reserved bits. Best leave alone.
- `opcode` : denotes the kind of message. The most relevant ones are 0b0001 (text), 0b0010 (binary) and 0b1000 (close). They're frequently used in my code.
- `mask` : denotes whether the data is masked or not. When you send, this is usually 0, but data from the client *will be masked*. More in the next section.
- `payload_size` : Size of the data you are sending. Write the actual length here(in bytes). 2^7 gives us 127B as our maximum size, which isn't too much. However, it works slightly differently. If your size is between 0-125B, you write the size directly. If it is between 127B to 216B, you write 126 in the `payload_size` field and utilize the `extended_payload_size` field. By using 127 instead of 126, you can go up to 2^64B, which is usually sufficient. Refer to *this site* for a nice explanation.

Receiving: Unmasking

The client will also send you frames. Unlike the server side, client sided messages have to be masked with a masking key that's given in the frame.

The masking key is a 4-byte key. The unmasking looks like so:

Yeah. It has a MOD and a bitwise XOR in it. Confusing. Refer to the *this answer* for a very nice explanation of masking.

Closing

Closing is simple, however it's structured as a handshake.

This sends a frame to the server with the opcode 0b1000. So if you get that, the you have to respond appropriately.

You need to send frame with the opcode 0b1000 back. The frame will look like so:

`(1)(000)(1000)(0)(00000010) status_code_in_binary`

They should be fairly obvious(look at the format I've given above).

- The opcode is 0b1000, the length of the message is 2 bytes.
- The message itself is an integer, a status code from 1000 to 1004, which denotes how the websocket was closed. 1000 is normal closure. Note than you need to "pack" the integer into two separate bytes, since the status code is in binary.