Chapter 1: Websockets

Websockets

Webscokets is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. It can work on HTTP which is a good thing because JavaScript in the browser can't do TCP or UDP.

HTTP

- Traditionally, you make a request and get a response
- Problem: Not nearly as flexible as UDP and TCP
- Workarounds:
  - Polling
  - Long-polling (Comet)
  - Other tricks

We want to send messages either way whenever, we want them to arrive in order, the primary solution is to **Websockets**. **WebRTC** and **EventSource** are also valid solutions but less common.

Technology Breakdown

- Websockets!
  - Messages
  - Full duplex
- WebRTC
  - Streaming Media
  - Messages (same API as Websockets)
- EventSource
  - Sever-generated events
  - Just HTTP

You can upgrade existing HTTP connections to a websockets by using the `Upgrade` header. This allows websockets to operate on the same port as HTTP(S) and allows them to be compatible with proxies and reverse proxies.

Why Websockets

It allows for full-duplex message-based communication

- Server can push whenever, client can push whenever
- Both sides can send data at the same time
- Connection stays open
- Don't need to poll
- Avoid big HTTP headers for each message
- Reuse existing technologies as much as possible

Websocket Handshake

First the client makes a request

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: soap, wamp
Sec-WebSocket-Version: 13
```

And the server responses with

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept:
↪  s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: wamp
```

Use GET to specify which Websocket, 1 webserver can service multiple websocket services. *wss://server.example.com/mysocket*. `Connection: Upgrade` is used to upgrade the connection to use websockets not `keep-alive` nor `close`. `Upgrade: websocket` specifiy the protocol we're upgrading to. An optional header can be sent, `Sec-WebSocket-Protocol: wamp`, this specify the sub-protocol on what kind of websockets the user agent want to use.

Key and Accept

It uses a kind of Brown M&M test. The client sends `Sec-WebSocket-Key` with a random string. The server responses with a `Sec-WebSocket-Accept` header. This header is generated by taking the client key and appends, `258EAFA5-E914-47DA-95CA-C5AB0DC85B11`. Then takes the SHA1 sum of the appended key. Then finally takes the base64-encoding of the SHA1 sum to generate the accept. After this the client knows for sure the server's ready for websocket.

Websocket Protocol

The client and server exchange "frames".

- The control frames (out of band): with the hex code `0x8` to close the connection, `0x9` for a ping and `0xA` for a pong.
- The data frame: These are the messages, updates, events, RPCs, whatever you're exchanging with the server.

Messages

Are sent in data frames, the messages could be binary or text, and the size if known ahead of time. This allows for efficient buffering. The messages could be fragmented into multiple data frames.

In Figure 1.1 shows the websocket format.

The first byte contains

- `FIN` bit
- 3 reserved bits
- 4-bit opcode

The second byte contains:

- A mask bit that enables XOR "encryption" mask

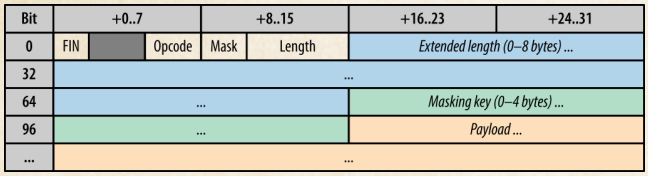

Figure 1.1: Websocket Format

- 7-bits for payload length.
- If the payload length = 126 then a 16-bit payload length follows
- If the payload length = 127 then a 64-bit payload length follows
- If masks is set then the masking key follows

Finally the data.

The smallest fragment is a one byte message, no mask equals 4 bytes. The largest possible header is 14 bytes. Each message just send fragments until FIN. Ordering is handled by TCP.

Opcodes

There are several opcodes for websockets

- `0x1` UTF-8 text (don't break characters)
- `0x2` binary
- `0x8` close
- `0x9` ping
- `0xa` pong

Example

`0x01 0x03 0x48 0x65 0x6c` => "Hel". The `0x01` tells the recipient, that the message is UTF-8. The `0x03` is the payload length of 3. `0x80 0x02 0x6c 0x6f` => "lo". The `0x80` sets the FIN bit and nothing else. The `0x02` is the payload length of 2.

Why the mask

The primary reason for the mask is to prevent accidentally being parsed as HTTP. The websocket protocol is supposed to work with existing infrastructure and maintainers were worried about cache poisoning by sending fake looking GET requests over websockets.

Masking encodes and garbles a frame with a mask so that you can't send a GET request in the plain. The mask is not there for privacy. This allows browsers to protect against malicious pages doing bad things they shouldn't. Of course we can't do anything about custom clients, which can send whatever they want over TCP.

Webscoket URIs

You can use the scheme `ws` to use websockets over a uri. For example, `ws://server.com:9090/path`. `wss:` is websocket secure, it inherits TLS from the HTTPS connection used initially. The format is the same as HTTP URI but only for the GET method.

Performance

The performance benefit for using websockets are as follows:

- Better 2-way communication
- Missing out on client side caching
- Reinvent the wheel by reusing TC/UDP
- Beat the firewall
- Doesn't fully replace XHR/fetch AJAX

Errors

During errors like a bad UTF-8 encoding this will lead to a close connection. There is no real prescript other than to close the connection. Closing is done by control frame, TLS, and TCP close.

Websockets in the Browser

JS code in the browser won't have access to fragments, masking, etc. Those are handled by the WebSocket API. The Simple browser API have the following actions

- Open
- Send and receive messages
- Close

The browser sanitizes everything, is in control to prevent malicious web pages from exploiting your browser to do things like poison proxies.

Websockets in JS

Here is an example usage for the websocket API in JavaScript.

```
var ws = new
↪  WebSocket("ws://www.example.com/socketserver");
var ws = new
↪  WebSocket("ws://www.example.com/socketserver",
↪  ["proto1", "proto2"]);

ws.send("A string");

var buffer = new ArrayBuffer(16);
var int32View = new Int32Array(buffer);
websocketInstance.send( int32View ); // send
↪  binary

ws.close();
```

You can send a plain string text or an array of binary information, your choice.