JAN
23

Writing Your Own WebSocket Server

The WebSocket protocol has applications beyond plain vanilla web development. I will explain how the protocol works, how to implement your own server and share some insights I had along the way. Before we get down and dirty, I will explain what I've been doing with it.

At this point I expect many of you are saying "I'm not working on a web game this doesn't seem relevant to me." Well, neither am I. I embed a WebSocket server into my game engine and with a local web application use the WebSocket protocol as a medium to control, configure and monitor my game engine. Some concrete examples of what I've done so far:

- Monitor memory allocation statistics
- Monitor performance of subsystems
- Set and query configuration variables
- Live edit the world
- Loaded asset preview

Soon, I will attempt to stream the display of the game to the web browser and stream mouse, as well as the keyboard data back to the game. In other words, remote desktop for the game engine. Another use case I would like to investigate is writing unit tests for the engine in javascript and drive it from the web browser. The possibilities are endless.

Another benefit of this approach is that your development UI is platform independent. This is nice when you are developing a title against many architectures. For example, consoles where the target does not usually have keyboard or mouse input are harder to interact with- by moving your UI to the web browser this problem is avoided.

Hang in there, this article is about the plumbing. My next article will be about the above applications.

WebSocket is a communication protocol that allows for bi-directional text and binary message passing. The client is a Javascript application running inside a web browser and, typically, the server is a web server. I say 'typically' because that is not how I use it.

WebSocket was developed because sending data between the web server and web application over HTTP was inefficient. The WebSocket protocol is very bandwidth efficient (message framing is at most 14 bytes) and the payloads are custom to the application. A WebSocket connection begins life as a regular HTTP connection. The connection is upgraded from HTTP to WebSocket. This upgrade is one way- you can't revert back to an HTTP connection. You can read more about WebSocket at WikiPedia and the complete specification is available. The WebSocket protocol was only recently finalized in December 2011.

Okay, let's dive into how WebSocket works. I will cover creating a connection, sending and receiving messages, and responding to pings.

Connecting

Creating a WebSocket connection is initiated by the client sending the following upgrade request:

```
    GET /servicename HTTP/1.1
    Host: server.example.com
    Upgrade: websocket
    Connection: Upgrade
    Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
    Origin: http://example.com
```

The server responds with:

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

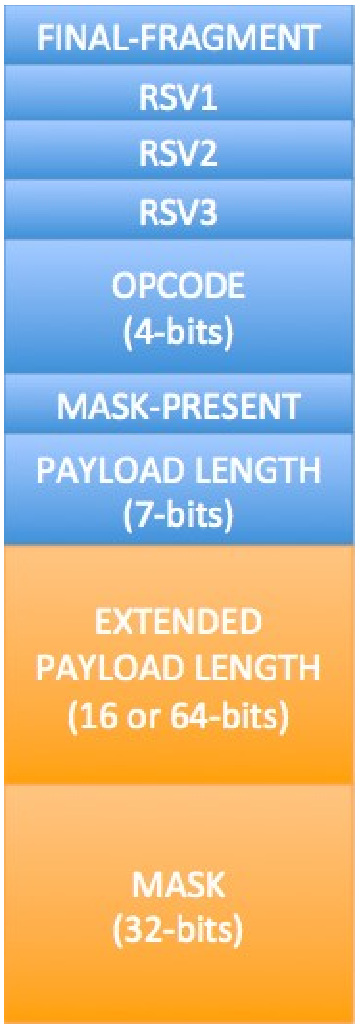
Most of these HTTP fields are self explanatory but not Sec-WebSocket-Key and Sec-WebSocket-Accept. Sec-WebSocket-Key is a string sent by the client as a challenge to the server. This leads to the question- how does the server calculate the value of Sec-WebSocket-Accept and complete the challenge? It is quite simple. The server first takes Sec-WebSocket-Key and concatenates it with a GUID string from the WebSocket specification. Then the SHA-1 hash of the resulting string is computed and, finally, Sec-WebSocket-Accept is the base64 encoding of the hash value. Let's work through an example:

```
SpecifcationGUID = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";
FullWebSocketKey = concatenate(Sec-WebSocket-Key, SpecifcationGUID);
-> dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CA-C5AB0DC85B11
KeyHash = SHA-1(FullWebSocketKey);
-> 0xb3 0x7a 0x4f 0x2c 0xc0 0x62 0x4f 0x16 0x90 0xf6 0x46 0x06 0xcf 0x38 0x59
0x45 0xb2 0xbe 0xc4 0xea
Sec-Websocket-Accept = base64(KeyHash)
-> s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

Transmission

WebSocket is a message based protocol. Each message begins with a header defining the length of the message, the type (text, binary or control) and other meta-data. The payload immediately follows the header. All incoming messages will include a 32-bit mask which must be applied to the entire payload with a XOR operation. Each message will have a different mask. The masking is used to guard against simple snooping.

The header begins with a 16-bit mask (blue) and up to 12-bytes of optional header (orange).



The header mask indicates whether this is the final fragment of a message (messages can be split into fragments), the op-code, and whether a mask is present. The payload length field plays double duty. For small messages (less than 125 bytes) it is the length of the message, but for messages that are longer, the payload length is used as a flag to indicate how large the extended payload length field is. The extended payload length follows immediately after the first 16-bits of the header (it comes before the mask). When payload length is equal to 126, the extended payload length is 16-bits and when it is equal to 127 the extended payload length is 64-bits.

WebSocket op-codes are split into three categories: continuation, non-control and control. Continuation and non-control op-codes indicate user messages and control frames are used to configure the protocol itself. Presently the following op-codes are defined:

Op-code	**Meaning**

0x0	Message continuation [*continuation*]
0x1	Text message [*non-control*]
0x2	Binary message [*non-control*]
0x8	Connection Close [*control*]
0x9	Ping [*control*]
0xA	Pong [*control*]

Once you have parsed the header, extracting the payload is trivial. Do not forget to XOR in the mask. Parsing the header is made interesting by the fact that its size and layout is variable and thus cannot be mapped directly to a C structure. Or can it?

```
struct WebSocketMessageHeader {
 union {
   struct {
     unsigned int OP_CODE : 4;
     unsigned int RSV1 : 1;
     unsigned int RSV2 : 1;
     unsigned int RSV3 : 1;
     unsigned int FIN : 1;
     unsigned int PAYLOAD : 7;
     unsigned int MASK : 1;
   } bits;
   uint16_t short_header;
 };

 size_t GetMessageLength() const;
 size_t GetPayloadOffset() const;
 size_t GetPayloadLength() const;
 uint32_t GetMask() const;
 uint8_t GetOpCode() const;
 bool IsFinal() const;
 bool IsMasked() const;

 ...

};
```

A **WebSocketMessageHeader** will always be at least 16-bits long, so the only data element defined inside the struct is short_header. Accessing the mask, extended payload lengths, or the payload is done with an offset from &short_header. When I want to parse a header, I simply do this:

```
WebSocketMessageHeader* header = &incoming_buffer[read_index];
```

I found this to be a very clean approach and is generally useful when dealing with structures that do not have a fixed length or layout.

Messages can be split into multiple fragments. When this happens the FINAL-FRAGMENT bit will be zero until the final message. The first fragment will have the op-code indicating either a text (**0x1**) or binary (**0x2**) message and the rest of the fragments will have the op-code of continuation (**0x0**).

Ping Pong

The protocol supports ping (**0x9**) and pong (**0xA**) messages. When a ping message has a payload, the resulting pong message must have an identical payload. You are only required to pong the most recent ping if more than one arrive.

Server Design

Finally, I want to describe the high level design of my WebSocket server. My server uses three buffers. One buffer for incoming WebSocket data, one for outgoing WebSocket data and one to store fully parsed incoming messages. An outline of the API:

```
class WebSocketServer {
public:
 WebSocketServer();

 int AcceptConnection(TcpListener* listener);
 int CloseConnection();

 void Update();

 int SendTextMessage(const char* msg);
 int SendTextMessage(const char* msg, size_t msg_length);

 uint64_t PendingMessageCount() const;
 void ProcessMessages(OnMessageDelegate del, void* userdata);
 void ClearMessages();
};
```

NOTE: I've trimmed a bunch of trivial methods from the outline and only left the ones worth discussing.

Connection Handling

It is important to decouple listening for a connection over TCP from the WebSocket server itself. Each instance of **WebSocketServer** is responsible for only one client. This keeps the code and resource allocation simple. A higher level system should manage multiple connection requests and multiple live WebSocket connections.

Updating

My WebSocket server has a single **Update** method. This method pumps the connection, it is responsible for sending any pending messages, receiving any new messages (ultimately moving them to the message buffer), and updating status flags (connection opened, connection closed, connection error).

Message Processing

Complete incoming messages are stored in their own buffer. When the engine system is ready to process incoming messages, a call to **ProcessMessages** is made and a delegate function is passed in. The **WebSocketServer** will iterate over all messages in the buffer and call this delegate for each one. When the engine is done with the messages they must be cleared by calling **ClearMessages**.

Conclusion

Hopefully, you are still with me and have a clear grasp on how WebSocket protocol works and how I designed my WebSocket server. In my next article, I will take this a step further- using WebSocket inside my engine as a remote procedure call medium and controlling my engine using a web browser. Next month I will be speaking at AltDevConf on this subject. Hope to see you there.