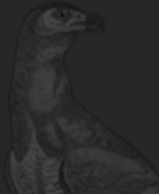




WebSocket

BROWSER APIS AND PROTOCOLS, CHAPTER 17



WebSocket enables bidirectional, message-oriented streaming of text and binary data between client and server. It is the closest API to a raw network socket in the browser. Except a WebSocket connection is also much more than a network socket, as the browser abstracts all the complexity behind a simple API and provides a number of additional services:

- Connection negotiation and same-origin policy enforcement
- Interoperability with existing HTTP infrastructure
- Message-oriented communication and efficient message framing
- Subprotocol negotiation and extensibility

WebSocket is one of the most versatile and flexible transports available in the browser. The simple and minimal API enables us to layer and deliver arbitrary application protocols between client and server—anything from simple JSON payloads to custom binary message formats—in a streaming fashion, where either side can send data at any time.

However, the trade-off with custom protocols is that they are, well, custom. The application must account for missing state management, compression, caching, and other services otherwise provided by the browser. There are always design constraints and performance trade-offs, and leveraging WebSocket is no exception. In short, WebSocket is not a replacement for HTTP, XHR, or SSE, and for best performance it is critical that we leverage the strengths of each transport.

Note

WebSocket is a set of multiple standards: the WebSocket API is defined by the W3C, and the WebSocket protocol (RFC 6455) and its extensions are defined by the HyBi Working Group (IETF).

WebSocket API



The WebSocket API provided by the browser is remarkably small and simple. Once again, all the low-level details of connection management and message processing are taken care of by the browser. To initiate a new connection, we need the URL of a WebSocket resource and a few application callbacks:



```
ws.onerror = function (error) { ... } ❷  
ws.onclose = function () { ... } ❸  
  
ws.onopen = function () { ❹  
  ws.send("Connection established. Hello server!"); ❺  
}  
  
ws.onmessage = function(msg) { ❻  
  if(msg.data instanceof Blob) { ❼  
    processBlob(msg.data);  
  } else {  
    processText(msg.data);  
  }  
}
```

- ❶ Open a new secure WebSocket connection (wss)
- ❷ Optional callback, invoked if a connection error has occurred
- ❸ Optional callback, invoked when the connection is terminated
- ❹ Optional callback, invoked when a WebSocket connection is established
- ❺ Client-initiated message to the server
- ❻ A callback function invoked for each new message from the server
- ❼ Invoke binary or text processing logic for the received message

The API speaks for itself. In fact, it should look very similar to the EventSource API we saw in the preceding chapter. This is intentional, as WebSocket offers similar and extended functionality. Having said that, there are a number of important differences as well. Let's take a look at them one by one.

Emulating WebSocket



WebSocket protocol has undergone a number of revisions, implementation rollbacks, and security investigations. However, the good news is that the latest version (v13) defined by RFC6455 is now supported by all modern browsers. The only notable omission is the Android browser. For the latest status, see <http://caniuse.com/websockets> .

Similar to the SSE polyfill strategy ([Emulating EventSource with Custom JavaScript](#)), the WebSocket browser API can be emulated via an optional JavaScript library. However, the hard part with emulating WebSockets is not the API, but the transport! As a result, the choice of the polyfill library and its fallback transport (XHR polling, EventSource, iframe polling, etc.) will have significant impact on the performance of an emulated WebSocket session.

To simplify cross-browser deployment, popular libraries such as SockJS provide an implementation of WebSocket-like object in the browser but also go one step further by providing a custom server that implements support for WebSocket and a variety of alternative transports. The



Other libraries, such as Socket.IO, go even further by implementing additional features, such as heartbeats, timeouts, support for automatic reconnects, and more, in addition to a multitransport fallback functionality.

When considering a polyfill library or a "real-time framework," such as Socket.IO, pay close attention to the underlying implementation and configuration of the client and server: always leverage the native WebSocket interface for best performance, and ensure that fallback transports meet your performance goals.

WS and WSS URL Schemes



The WebSocket resource URL uses its own custom scheme: `ws` for plain-text communication (e.g., `ws://example.com/socket`), and `wss` when an encrypted channel (TCP+TLS) is required. Why the custom scheme, instead of the familiar `http`?

The primary use case for the WebSocket protocol is to provide an optimized, bi-directional communication channel between applications running in the browser and the server. However, the WebSocket wire protocol can be used outside the browser and could be negotiated via a non-HTTP exchange. As a result, the HyBi Working Group chose to adopt a custom URL scheme.

Note

Despite the non-HTTP negotiation option enabled by the custom scheme, in practice there are no existing standards for alternative handshake mechanisms for establishing a WebSocket session.

Receiving Text and Binary Data



WebSocket communication consists of messages and application code and does not need to worry about buffering, parsing, and reconstructing received data. For example, if the server sends a 1 MB payload, the application's `onmessage` callback will be called only when the entire message is available on the client.

Further, the WebSocket protocol makes no assumptions and places no constraints on the application payload: both text and binary data are fair game. Internally, the protocol tracks only two pieces of information about the message: the length of payload as a variable-length field and the type of payload to distinguish UTF-8 from binary transfers.

When a new message is received by the browser, it is automatically converted to a `DOMString` object for text-based data, or a `Blob` object for binary data, and then passed directly to the



client, is to tell the browser to convert the received binary data to an `ArrayBuffer` instead of `Blob`.

```
var ws = new WebSocket('wss://example.com/socket');
ws.binaryType = "arraybuffer"; ❶

ws.onmessage = function(msg) {
  if(msg.data instanceof ArrayBuffer) {
    processArrayBuffer(msg.data);
  } else {
    processText(msg.data);
  }
}
```

❶ Force an `ArrayBuffer` conversion when a binary message is received

“User agents can use this as a hint for how to handle incoming binary data: if the attribute is set to `"blob"`, it is safe to spool it to disk, and if it is set to `"arraybuffer"`, it is likely more efficient to keep the data in memory. Naturally, user agents are encouraged to use more subtle heuristics to decide whether to keep incoming data in memory or not...

The WebSocket API, W3C Candidate Recommendation

A `Blob` object represents a file-like object of immutable, raw data. If you do not need to modify the data and do not need to slice it into smaller chunks, then it is the optimal format—e.g., you can pass the entire `Blob` object to an image tag (see the example in [Downloading Data with XHR](#)). On the other hand, if you need to perform additional processing on the binary data, then `ArrayBuffer` is likely the better fit.

Decoding Binary Data with JavaScript



An `ArrayBuffer` is a generic, fixed-length binary data buffer. However, an `ArrayBuffer` can be used to create one or more `ArrayBufferView` objects, each of which can present the contents of the buffer in a specific format. For example, let's assume we have the following C-like binary data structure:

```
struct someStruct {
  char username[16];
  unsigned short id;
  float scores[32];
};
```

Given an `ArrayBuffer` object of this type, we can create multiple views into the same buffer, each with its own offset and data type:

```
var buffer = msg.data;

var usernameView = new Uint8Array(buffer, 0, 16);
```



```
console.log("ID: " + idView[0] + " username: " + usernameView[0]);  
for (var j = 0; j < 32; j++) { console.log(scoresView[j]) }
```

Each view takes the parent buffer, starting byte offset, and number of elements to process—the offset is calculated based on the size of the preceding fields. As a result, `ArrayBuffer` and `WebSocket` give our applications all the necessary tools to stream and process binary data within the browser.

Sending Text and Binary Data



Once a `WebSocket` connection is established, the client can send and receive UTF-8 and binary messages at will. `WebSocket` offers a bidirectional communication channel, which allows message delivery in both directions over the same TCP connection:

```
var ws = new WebSocket('wss://example.com/socket');  
  
ws.onopen = function () {  
  socket.send("Hello server!"); ❶  
  socket.send(JSON.stringify({'msg': 'payload'})); ❷  
  
  var buffer = new ArrayBuffer(128);  
  socket.send(buffer); ❸  
  
  var intview = new Uint32Array(buffer);  
  socket.send(intview); ❹  
  
  var blob = new Blob([buffer]);  
  socket.send(blob); ❺  
}
```

- ❶ Send a UTF-8 encoded text message
- ❷ Send a UTF-8 encoded JSON payload
- ❸ Send the `ArrayBuffer` contents as binary payload
- ❹ Send the `ArrayBufferView` contents as binary payload
- ❺ Send the `Blob` contents as binary payload

The `WebSocket` API accepts a `DOMString` object, which is encoded as UTF-8 on the wire, or one of `ArrayBuffer`, `ArrayBufferView`, or `Blob` objects for binary transfers. However, note that the latter binary options are simply an API convenience: on the wire, a `WebSocket` frame is either marked as binary or text via a single bit. Hence, if the application, or the server, need other content-type information about the payload, then they must use an additional mechanism to communicate this data.



returns immediately. As a result, especially when transferring large payloads, do not mistake the fast return for a signal that the data has been sent! To monitor the amount of data queued by the browser, the application can query the `bufferedAmount` attribute on the socket:

```
var ws = new WebSocket('wss://example.com/socket');

ws.onopen = function () {
  subscribeToApplicationUpdates(function(evt) { ❶
    if (ws.bufferedAmount == 0) ❷
      ws.send(evt.data); ❸
  });
};
```

- ❶ Subscribe to application updates (e.g., game state changes)
- ❷ Check the amount of buffered data on the client
- ❸ Send the next update if the buffer is empty

The preceding example attempts to send application updates to the server, but only if the previous messages have been drained from the client's buffer. Why bother with such checks? All WebSocket messages are delivered in the exact order in which they are queued by the client. As a result, a large backlog of queued messages, or even a single large message, will delay delivery of messages queued behind it—head-of-line blocking!

To work around this problem, the application can split large messages into smaller chunks, monitor the `bufferedAmount` value carefully to avoid head-of-line blocking, and even implement its own priority queue for pending messages instead of blindly queuing them all on the socket.

Note

Many applications generate multiple classes of messages: high-priority updates, such as control traffic, and low-priority updates, such as background transfers. To optimize delivery, the application should pay close attention to how and when each type of message is queued on the socket!

Subprotocol Negotiation



WebSocket protocol makes no assumptions about the format of each message: a single bit tracks whether the message contains text or binary data, such that it can be efficiently decoded by the client and server, but otherwise the message contents are opaque.

Further, unlike HTTP or XHR requests, which communicate additional metadata via HTTP headers of each request and response, there is no such equivalent mechanism for a WebSocket



server must agree to implement their own subprotocol to communicate this data.

- The client and server can agree on a fixed message format upfront—e.g., all communication will be done via JSON-encoded messages or a custom binary format, and necessary message metadata will be part of the encoded structure.
- If the client and server need to transfer different data types, then they can agree on a consistent message header, which can be used to communicate the instructions to decode the remainder of the payload.
- A mix of text and binary messages can be used to communicate the payload and metadata information—e.g., a text message can communicate an equivalent of HTTP headers, followed by a binary message with the application payload.

This list is just a small sample of possible strategies. The flexibility and low overhead of a WebSocket message come at the cost of extra application logic. However, message serialization and management of metadata are only part of the problem! Once we determine the serialization format for our messages, how do we ensure that both client and server understand each other, and how do we keep them in sync?

Thankfully, WebSocket provides a simple and convenient *subprotocol negotiation* API to address the second problem. The client can advertise which protocols it supports to the server as part of its initial connection handshake:

```
var ws = new WebSocket('wss://example.com/socket',
    ['appProtocol', 'appProtocol-v2']); ❶

ws.onopen = function () {
    if (ws.protocol == 'appProtocol-v2') { ❷
        ...
    } else {
        ...
    }
}
```

❶ Array of subprotocols to advertise during WebSocket handshake

❷ Check the subprotocol chosen by the server

As the preceding example illustrates, the WebSocket constructor accepts an optional array of subprotocol names, which allows the client to advertise the list of protocols it understands or is willing to use for this connection. The specified list is sent to the server, and the server is allowed to pick one of the protocols advertised by the client.

If the subprotocol negotiation is successful, then the `onopen` callback is fired on the client, and the application can query the `protocol` attribute on the WebSocket object to determine the chosen protocol. On the other hand, if the server does not support any of the client protocols



invoked, and the connection is terminated.

Note

The subprotocol names are defined by the application and are sent as specified to the server during the initial HTTP handshake. Other than that, the specified subprotocol has no effect on the core WebSocket API.

WebSocket Protocol



The WebSocket wire protocol (RFC 6455) developed by the HyBi Working Group consists of two high-level components: the opening HTTP handshake used to negotiate the parameters of the connection and a binary message framing mechanism to allow for low overhead, message-based delivery of both text and binary data.

“ *The WebSocket Protocol attempts to address the goals of existing bidirectional HTTP technologies in the context of the existing HTTP infrastructure; as such, it is designed to work over HTTP ports 80 and 443... However, the design does not limit WebSocket to HTTP, and future implementations could use a simpler handshake over a dedicated port without reinventing the entire protocol.*

WebSocket Protocol, RFC 6455

WebSocket protocol is a fully functional, standalone protocol that can be used outside the browser. Having said that, its primary application is as a bidirectional transport for browser-based applications.

Binary Framing Layer



Client and server WebSocket applications communicate via a message-oriented API: the sender provides an arbitrary UTF-8 or binary payload, and the receiver is notified of its delivery when the entire message is available. To enable this, WebSocket uses a custom binary framing format (Figure 17-1), which splits each application message into one or more *frames*, transports them to the destination, reassembles them, and finally notifies the receiver once the entire message has been received.

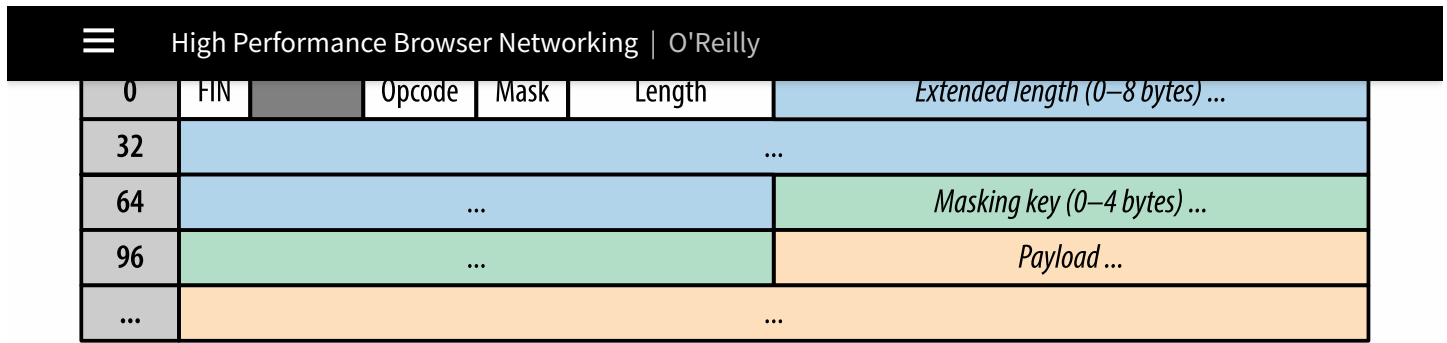


Figure 17-1. WebSocket frame: 2–14 bytes + payload

Frame

The smallest unit of communication, each containing a variable-length frame header and a payload that may carry all or part of the application message.

Message

A complete sequence of frames that map to a logical application message.


The decision to fragment an application message into multiple frames is made by the underlying implementation of the client and server framing code. Hence, the applications remain blissfully unaware of the individual WebSocket frames or how the framing is performed. Having said that, it is still useful to understand the highlights of how each WebSocket frame is represented on the wire:

- The first bit of each frame (FIN) indicates whether the frame is a final fragment of a message. A message may consist of just a single frame.
- The opcode (4 bits) indicates type of transferred frame: text (1) or binary (2) for transferring application data or a control frame such as connection close (8), ping (9), and pong (10) for connection liveness checks.
- The mask bit indicates whether the payload is masked (for messages sent from the client to the server only).
- Payload length is represented as a variable-length field:
 - If 0–125, then that is the payload length.
 - If 126, then the following 2 bytes represent a 16-bit unsigned integer indicating the frame length.
 - If 127, then the following 8 bytes represent a 64-bit unsigned integer indicating the frame length.
- Masking key contains a 32-bit value used to mask the payload.
- Payload contains the application data and custom extension data if the client and server negotiated an extension when the connection was established.

Note

The payload of all client-initiated frames is masked using the value specified in the frame header: this prevents malicious scripts executing on the client from performing a cache



Full details of this attack, refer to [Talking to Yourself for Fun and Profit](#) , presented at W2S1 2011.

As a result, each server-sent WebSocket frame incurs 2–10 bytes of framing overhead. The client must also send a masking key, which adds an extra 4 bytes to the header, resulting in 6–14 bytes over overhead. No other metadata, such as header fields or other information about the payload, is available: all WebSocket communication is performed by exchanging frames that treat the payload as an opaque blob of application data.

WebSocket Multiplexing and Head-of-Line Blocking



WebSocket is susceptible to head-of-line blocking: messages can be split into one or more frames, but frames from different messages can't be interleaved, as there is no equivalent to a "stream ID" found in the HTTP/2 framing mechanism; see [Streams, Messages, and Frames](#)).

As a result, a large message, even when split into multiple WebSocket frames, will block the delivery of frames associated with other messages. If your application is delivering latency-sensitive data, be careful about the payload size of each message and consider splitting large messages into multiple application messages!

The lack of multiplexing in core WebSocket specification also means that each WebSocket connection requires a dedicated TCP connection, which may become a potential problem for HTTP/1.x deployments due to a restricted number of connections per origin maintained by the browser; see [Exhausting Client and Server Resources](#).

On the bright side, the new "Multiplexing Extension for WebSockets" developed by the HyBi Working Group addresses the latter limitation:

“ With this extension, one TCP connection can provide multiple virtual WebSocket connections by encapsulating frames tagged with a channel ID... The multiplexing extension maintains separate logical channels, each of which provides fully the logical equivalent of an independent WebSocket connection, including separate handshake headers.

WebSocket Multiplexing (Draft 10)

With this extension in place, multiple WebSocket connections (channels) can be multiplexed over the same TCP connection. However, each individual channel is still susceptible to head-of-line blocking! Hence, one potential workaround is to use different channels, or dedicated TCP connections, to multiplex multiple messages in parallel.

Finally, note that the preceding extension is necessary only for HTTP/1.x connections. While no official specification is yet available for transporting WebSocket frames with HTTP/2, doing so would be much easier: HTTP/2 has built-in stream multiplexing, and multiple WebSocket



Protocol Extensions



WebSocket specification allows for protocol extensions: the wire format and the semantics of the WebSocket protocol can be extended with new opcodes and data fields. While somewhat unusual, this is a very powerful feature, as it allows the client and server to implement additional functionality on top of the base WebSocket framing layer without requiring any intervention or cooperation from the application code.

What are some examples of WebSocket protocol extensions? The HyBi Working Group, which is responsible for the development of the WebSocket specification, lists two official extensions in development:

"A Multiplexing Extension for WebSockets"

This extension provides a way for separate logical WebSocket connections to share an underlying transport connection.

"Compression Extensions for WebSocket"

A framework for creating WebSocket extensions that add compression functionality to the WebSocket Protocol.

As we noted earlier, each WebSocket connection requires a dedicated TCP connection, which is inefficient. Multiplexing extension addresses this problem by extending each WebSocket frame with an additional "channel ID" to allow multiple virtual WebSocket channels to share a single TCP connection.

Similarly, the base WebSocket specification provides no mechanism or provisions for compression of transferred data: each frame carries payload data as provided by the application. As a result, while this may not be a problem for optimized binary data structures, this can result in high byte transfer overhead unless the application implements its own data compression and decompression logic. In effect, compression extension enables an equivalent of transfer-encoding negotiation provided by HTTP.

To enable one or more extensions, the client must advertise them in the initial Upgrade handshake, and the server must select and acknowledge the extensions that will be used for the lifetime of the negotiated connection. For a hands-on example, let's now take a closer look at the Upgrade sequence.

WebSocket Multiplexing and Compression in the Wild



As of mid-2013, WebSocket multiplexing is not yet supported by any popular browser. Similarly, there is limited support for compression: Google Chrome and the latest WebKit browsers may



As the name implies, per-frame compresses the payload contents on a frame-by-frame basis, which is suboptimal for large messages that may be split between multiple frames. As a result, latest revisions of the compression extension have switched to per-message compression—that's the good news. The bad news is per-message compression is still experimental and is not yet available in any popular browser.

As a result, the application should pay close attention to the content-type of transferred data and apply its own compression where applicable. That is, at least until native WebSocket compression support is available widely across all the popular browsers. This is especially important for mobile applications, where each unnecessary byte carries high costs to the user.

HTTP Upgrade Negotiation



The WebSocket protocol delivers a lot of powerful features: message-oriented communication, its own binary framing layer, subprotocol negotiation, optional protocol extensions, and more. As a result, before any messages can be exchanged, the client and server must negotiate the appropriate parameters to establish the connection.

Leveraging HTTP to perform the handshake offers several advantages. First, it makes WebSockets compatible with existing HTTP infrastructure: WebSocket servers can run on port 80 and 443, which are frequently the only open ports for the client. Second, it allows us to reuse and extend the HTTP Upgrade flow with custom WebSocket headers to perform the negotiation:

Sec-WebSocket-Version

Sent by the client to indicate version ("13" for RFC6455) of the WebSocket protocol it wants to use. If the server does not support the client version, then it must reply with a list of supported versions.

Sec-WebSocket-Key

An auto-generated key sent by the client, which acts as a "challenge" to the server to prove that the server supports the requested version of the protocol.

Sec-WebSocket-Accept

Server response that contains signed value of Sec-WebSocket-Key, proving that it understands the requested protocol version.

Sec-WebSocket-Protocol

Used to negotiate the application subprotocol: client advertises the list of supported protocols; server must reply with a single protocol name.

Sec-WebSocket-Extensions

Used to negotiate WebSocket extensions to be used for this connection: client advertises supported extensions, and the server confirms one or more extensions by returning the same



With that, we now have all the necessary pieces to perform an HTTP Upgrade and negotiate a new WebSocket connection between the client and server:

```
GET /socket HTTP/1.1
Host: thirdparty.com
Origin: http://example.com
Connection: Upgrade
Upgrade: websocket ❶
Sec-WebSocket-Version: 13 ❷
Sec-WebSocket-Key: dGh1IHhnbXBsZSBub25jZQ== ❸
Sec-WebSocket-Protocol: appProtocol, appProtocol-v2 ❹
Sec-WebSocket-Extensions: x-webkit-deflate-message, x-custom-extension ❺
```

- ❶ Request to perform an upgrade to the WebSocket protocol
- ❷ WebSocket protocol version used by the client
- ❸ Auto-generated key to verify server protocol support
- ❹ Optional list of subprotocols specified by the application
- ❺ Optional list of protocol extensions supported by the client

Just like any other client-initiated connection in the browser, WebSocket requests are subject to the same-origin policy: the browser automatically appends the Origin header to the upgrade handshake, and the remote server can use CORS to accept or deny the cross origin request; see [Cross-Origin Resource Sharing \(CORS\)](#). To complete the handshake, the server must return a successful "Switching Protocols" response and confirm the selected options advertised by the client:

```
HTTP/1.1 101 Switching Protocols ❶
Upgrade: websocket
Connection: Upgrade
Access-Control-Allow-Origin: http://example.com ❷
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+x0o= ❸
Sec-WebSocket-Protocol: appProtocol-v2 ❹
Sec-WebSocket-Extensions: x-custom-extension ❺
```

- ❶ 101 response code confirming WebSocket upgrade
- ❷ CORS header indicating opt-in for cross-origin connection
- ❸ Signed Key value proving protocol support
- ❹ Application subprotocol selected by the server
- ❺ List of WebSocket extensions selected by the server



the client challenge: the contents of the Sec-WebSocket-Key are concatenated with a unique GUID string defined in the standard, a SHA1 hash is computed, and the resulting string is base-64 encoded and sent back to the client.

At a minimum, a successful WebSocket handshake must contain the protocol version and an auto-generated challenge value sent by the client, followed by a 101 HTTP response code (Switching Protocols) from the server with a hashed challenge-response to confirm the selected protocol version:

- Client must send Sec-WebSocket-Version and Sec-WebSocket-Key.
- Server must confirm the protocol by returning Sec-WebSocket-Accept.
- Client may send a list of application subprotocols via Sec-WebSocket-Protocol.
- Server must select one of the advertised subprotocols and return it via Sec-WebSocket-Protocol. If the server does not support any, then the connection is aborted.
- Client may send a list of protocol extensions in Sec-WebSocket-Extensions.
- Server may confirm one or more selected extensions via Sec-WebSocket-Extensions. If no extensions are provided, then the connection proceeds without them.

Finally, once the preceding handshake is complete, and if the handshake is successful, the connection can now be used as a two-way communication channel for exchanging WebSocket messages. From here on, there is no other explicit HTTP communication between the client and server, and the WebSocket protocol takes over.

Proxies, Intermediaries, and WebSockets



In practice, for security and policy reasons, many users have a restricted set of open ports—specifically port 80 (HTTP), and port 443 (HTTPS). As a result, WebSocket negotiation is performed via the HTTP Upgrade flow to ensure the best compatibility with existing network policies and infrastructure.

However, as we noted earlier in [Proxies, Intermediaries, TLS, and New Protocols on the Web](#), many existing HTTP intermediaries may not understand the new WebSocket protocol, which can lead to a variety of failure cases: blind connection upgrades, unintended buffering of WebSocket frames, content modification without understanding of the protocol, misclassification of WebSocket traffic as compromised HTTP connections, and so on.

The WebSocket Key and Accept handshake addresses some of these problems: it is a security policy against servers and intermediaries that may blindly "upgrade" the connection without actually understanding the WebSocket protocol. However, while this precaution addresses some deployment issues with explicit proxies, it is nonetheless insufficient for "transparent proxies," which may analyze and modify the data on the wire without notice.



tunnel, which resolves all of the previously listed concerns. This is especially true for mobile clients, whose traffic often passes through a variety of proxy services that may not play well with WebSocket.

WebSocket Use Cases and Performance



WebSocket API provides a simple interface for bidirectional, message-oriented streaming of text and binary data between client and server: pass in a WebSocket URL to the constructor, set up a few JavaScript callback functions, and we are up and running—the rest is handled by the browser. Add to that the WebSocket protocol, which offers binary framing, extensibility, and subprotocol negotiation, and WebSocket becomes a perfect fit for delivering custom application protocols in the browser.

However, just as with any discussion on performance, while the implementation complexity of the WebSocket protocol is hidden from the application, it nonetheless has important performance implications for how and when WebSocket should be used. WebSocket is not a replacement for XHR or SSE, and for best performance it is critical that we leverage the strengths of each transport!

Note

Refer to [XHR Use Cases and Performance](#) and [SSE Use Cases and Performance](#) for a review of the performance characteristics of each transport.

Request and Response Streaming



WebSocket is the only transport that allows bidirectional communication over the same TCP connection ([Figure 17-2](#)): the client and server can exchange messages at will. As a result, WebSocket provides low latency delivery of text and binary application data in both directions.

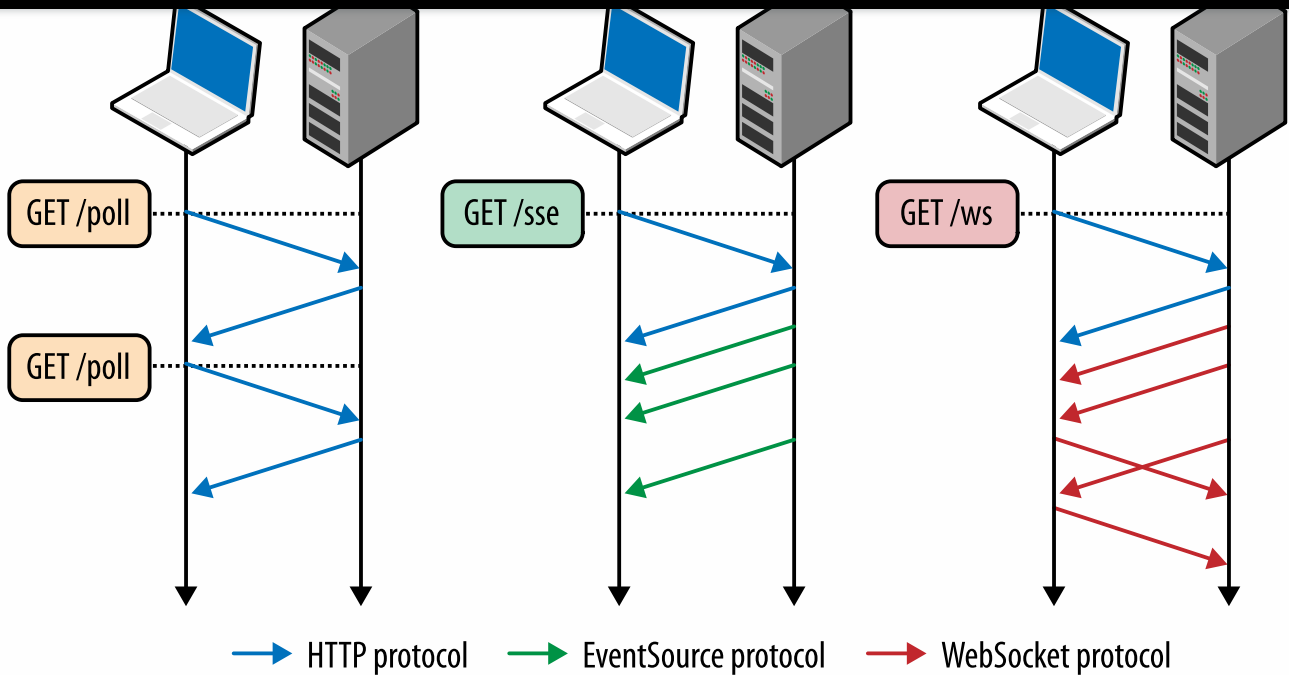


Figure 17-2. Communication flow of XHR, SSE, and WebSocket

- XHR is optimized for "transactional" request-response communication: the client sends the full, well-formed HTTP request to the server, and the server responds with a full response. There is no support for request streaming, and until the Streams API is available, no reliable cross-browser response streaming API.
- SSE enables efficient, low-latency server-to-client streaming of text-based data: the client initiates the SSE connection, and the server uses the event source protocol to stream updates to the client. The client can't send any data to the server after the initial handshake.

Propagation and Queuing Latency

§

Switching transports from XHR to SSE or WebSocket does not decrease the roundtrip between client and server! Regardless of the transport, the *propagation latency* of the data packets is the same. However, aside from propagation latency, there is also the *queuing latency*: the time the message has to wait on the client or server before it can be routed to the other party.

In the case of XHR polling, the queuing latency is a function of the client polling interval: the message may be available on the server, but it cannot be sent until the next client XHR request; see [Modeling Performance of XHR Polling](#). By contrast, both SSE and WebSocket use a persistent connection, which allows the server to dispatch the message (and client, in the case of WebSocket), the moment it becomes available.

As a result, "low-latency delivery" for SSE and WebSocket is specifically referring to the elimination of message queuing latency. We have not yet figured out how to make WebSocket data packets travel faster than the speed of light!



Once a WebSocket connection is established, the client and server exchange data via the WebSocket protocol: application messages are split into one or more frames, each of which adds from 2 to 14 bytes of overhead. Further, because the framing is done via a custom binary format, both UTF-8 and binary application data can be efficiently encoded via the same mechanism. How does that compare with XHR and SSE?

- SSE adds as little as 5 bytes per message but is restricted to UTF-8 content only; see [Event Stream Protocol](#).
- HTTP/1.x requests (XHR or otherwise) will carry an additional 500–800 bytes of HTTP metadata, plus cookies; see [Measuring and Controlling Protocol Overhead](#).
- HTTP/2 compresses the HTTP metadata, which significantly reduces the overhead; see [Header Compression](#). In fact, if the headers do not change between requests, the overhead can be as low as 8 bytes!

Note

Keep in mind that these overhead numbers do not include the overhead of IP, TCP, and TLS framing, which add 60–100 bytes of combined overhead per message, regardless of the application protocol; see [Optimize TLS Record Size](#).

Data Efficiency and Compression



Every XHR request can negotiate the optimal transfer encoding format (e.g., gzip for text-based data), via regular HTTP negotiation. Similarly, because SSE is restricted to UTF-8-only transfers, the event stream data can be efficiently compressed by applying gzip across the entire session.

With WebSocket, the situation is more complex: WebSocket can transfer both text and binary data, and as a result it doesn't make sense to compress the entire session. The binary payloads may be compressed already! As a result, WebSocket must implement its own compression mechanism and selectively apply it to each message.

The good news is the HyBi working group is developing the per-message compression extension for the WebSocket protocol. However, it is not yet available in any of the browsers. As a result, unless the application implements its own compression logic by carefully optimizing its binary payloads (see [Decoding Binary Data with JavaScript](#)) and implementing its own compression logic for text-based messages, it may incur high byte overhead on the transferred data!



of the compression extension to the WebSocket protocol; see [WebSocket Multiplexing and Compression in the Wild](#).

Custom Application Protocols



The browser is optimized for HTTP data transfers: it understands the protocol, and it provides a wide array of services, such as authentication, caching, compression, and much more. As a result, XHR requests inherit all of this functionality for free.

By contrast, streaming allows us to deliver custom protocols between client and server, but at the cost of bypassing many of the services provided by the browser: the initial HTTP handshake may be able to perform some negotiation of the parameters of the connection, but once the session is established, all further data streamed between the client and server is opaque to the browser. As a result, the flexibility of delivering a custom protocol also has its downsides, and the application may have to implement its own logic to fill in the missing gaps: caching, state management, delivery of message metadata, and so on!

Note

The initial HTTP Upgrade handshake does allow the server to leverage the existing HTTP cookie mechanism to validate the user. If the validation fails, the server can decline the WebSocket upgrade.

Leveraging Browser and Intermediary Caches



Using regular HTTP has significant advantages. Ask yourself a simple question: would the client benefit from caching the received data? Or could an intermediary optimize the delivery of the asset if it could cache it?

For example, WebSocket supports binary transfers, which allows the application to stream arbitrary image formats with no overhead—nice win! However, the fact that the image is delivered within a custom protocol means that it won't be cached by the browser cache, or any intermediary (e.g., a CDN). As a result, you may incur unnecessary transfers to the client and much higher traffic to the origin servers. The same logic applies to all other data formats: video, text, and so on.

As a result, make sure you choose the right transport for the job! A simple but effective strategy to address these concerns could be to use the WebSocket session to deliver non-cacheable data, such as real-time updates and application "control" messages, which can trigger XHR requests to fetch other assets via the HTTP protocol.



HTTP is optimized for short and bursty transfers. As a result, many of the servers, proxies, and other intermediaries are often configured to aggressively timeout idle HTTP connections, which, of course, is exactly what we don't want to see for long-lived WebSocket sessions. To address this, there are three pieces to consider:

- Routers, load-balancers, and proxies within own network
- Transparent and explicit proxies in external network (e.g., ISP and carrier proxies)
- Routers, firewalls, and proxies within the client's network

We have no control over the policy of the client's network. In fact, some networks may block WebSocket traffic entirely, which is why you may need a fallback strategy. Similarly, we don't have control over the proxies on the external network. However, this is where TLS may help! By tunneling over a secure end-to-end connection, WebSocket traffic can bypass all the intermediate proxies.

Note

Using TLS does not prevent the intermediary from timing out an idle TCP connection. However, in practice, it significantly increases the success rate of negotiating the WebSocket session and often also helps to extend the connection timeout intervals.

Finally, there is the infrastructure that we deploy and manage ourselves, which also often requires attention and tuning. As easy as it is to blame the client or external networks, all too often the problem is close to home. Each load-balancer, router, proxy, and web server in the serving path must be tuned to allow long-lived connections.

For example, Nginx 1.3.13+ can proxy WebSocket traffic, but defaults to aggressive 60-second timeouts! To increase the limit, we must explicitly define the longer timeouts:

```
location /websocket {
    proxy_pass http://backend;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_read_timeout 3600; ①
    proxy_send_timeout 3600; ②
}
```

- ① Set 60-minute timeout between reads
- ② Set 60-minute timeout between writes



Nginx servers. Not surprisingly, we need to apply similar explicit configuration here as well – e.g., for HAProxy:

```
defaults http
  timeout connect 30s
  timeout client 30s
  timeout server 30s
  timeout tunnel 1h 1
```

1 60-minute inactivity timeout for tunnels

The gotcha with the preceding example is the extra "tunnel" timeout. In HAProxy the `connect`, `client`, and `server` timeouts are applied only to the initial HTTP Upgrade handshake, but once the upgrade is complete, the timeout is controlled by the `tunnel` value.

Nginx and HAProxy are just two of hundreds of different servers, proxies, and load balancers running in our data centers. We can't enumerate all the configuration possibilities in these pages. The previous examples are just an illustration that most infrastructure requires custom configuration to handle long-lived sessions. Hence, before implementing application keepalives, double-check your infrastructure first.

Note

Long-lived and idle sessions occupy memory and socket resources on all the intermediate servers. Hence, short timeouts are often justified as a security, resource, and operational precaution. Deploying WebSocket, SSE, and HTTP/2, each of which relies on long-lived sessions, brings its own class of new operational challenges.

Performance Checklist



Deploying a high-performance WebSocket service requires careful tuning and consideration, both on the client and on the server. A short list of criteria to put on the agenda:

- Use secure WebSocket (WSS over TLS) for reliable deployments.
- Pay close attention to polyfill performance (if necessary).
- Leverage subprotocol negotiation to determine the application protocol.
- Optimize binary payloads to minimize transfer size.
- Consider compressing UTF-8 content to minimize transfer size.
- Set the right binary type for received binary payloads.
- Monitor the amount of buffered data on the client.
- Split large application messages to avoid head-of-line blocking.



Last, but definitely not least, optimize for mobile! Real-time push can be a costly performance anti-pattern on mobile handsets, where battery life is always at a premium. That's not to say that WebSocket should not be used on mobile. To the contrary, it can be a highly efficient transport, but make sure to account for its requirements:

- [Preserve Battery Power](#)
- [Eliminate Periodic and Inefficient Data Transfers](#)
- [Nagle and Efficient Server Push](#)
- [Eliminate Unnecessary Application Keepalives](#)

[« Back to the Table of Contents](#)

Copyright © 2013 [Ilya Grigorik](#). Published by O'Reilly Media, Inc. Licensed under [CC BY-NC-ND 4.0](#).