WebSocket Connections

meh

March 2023

1 Introduction

WebSockets are a full-duplex communication system through streams and frames over the TCP/IP bridge. They maintain efficiency and low latency by utilizing a compact binary framing format to reduce the overhead of transmissions. Unlike HTTP requests, which are unidirectional and stateless, websockets allow for bidirectional communication between the client and server. Applications which require real-time updates from the server at anytime (without the client's request) must use a websocket connection to transmit the data. Common usecases of websockets are multiplayer games, social medias, and real time analytics.

2 Procedure

2.1 Header Exchange

WebSockets are an upgrade of an HTTP connection, so a websocket server is an extension of a HTTP server and the websocket connection is an extension of a GET request to that HTTP server. Ideally, the server should send a 426 (Upgrade Required) Status Code if a GET request with improper headers are sent. A proper websocket connection starts off with a HTTP GET request to a server with the following headers required for a connection:

```
{
    "Connection": "Upgrade",
    "Upgrade": "websocket",
    // Random 16 bytes converted to Base64
    "Sec-WebSocket-Key": crypto.randomBytes(16).toString("base64"),
    "Sec-WebSocket-Version": "13"
}
```

The server then creates a SHA-1 hash of Sec-WebSocket-Key and the default magic GUID of a websocket, digested into a Base64 string. The server then sends a 101 (Switching Protocols) Status Code, approving the upgrade, along with the following response headers:

```
{
    "Connection": "Upgrade",
    "Upgrade": "websocket",
    "Sec-WebSocket-Accept": sha1.update(req_ws_key + magic).digest("base64")
}
```

The client receives the server's response, verifies the "Sec-WebSocket-Accept" header by computing its own response key using the same method as the server, and compares the two response keys. If they match, the client acknowledges the successful upgrade by sending a confirmation message in a WebSocket frame. After this, the connection is fully opened and binary frames can be transmitted.

2.2 Low Level Functions

```
/**
* Creates an endpoint for communication. [SERVER + CLIENT]
* @param domain Communication domain (e.g. IPv4/IPv6).
* @param type The socket type (e.g. TCP/UDP).
* @param protocol The protocol for communication.
*/
int socket(int domain, int type, int protocol);

/**
* Binds the server to a port to listen for incoming connections. [SERVER]
* @param sockfd The socket file descriptor (created by invoking socket()).
* @param addr A pointer to a sockaddr struct of the server containing networking information.
* @param addrlen The size of the addr struct.
*/
int bind(int sockfd, const struct sockaddr* addr, socklen_t addrlen);



/**
* Enables listening for incoming client connection. [SERVER]
* @param sockfd The socket file descriptor
* @param max_conn Maximum amount of pending connections (in queue, before acceptance).
*/
int listen(int sockfd, int backlog);

/**
* Accept an incoming client connection. [SERVER]
* @param sockfd The socket file descriptor (created by invoking socket()).
* @param addr A pointer to the sockaddr struct of the client containing networking information.
* @param addrlen The size of the addr struct.
*/
int accept(int sockfd, const struct sockaddr* addr, socklen_t addrlen);

/**
* Reads from the socket file descriptor. [SERVER + CLIENT]
* @param fd The file descriptor of the socket to read from.
* @param buffer A pointer to the buffer to write the incoming message to.
* @param count The maximum amount of bytes to write to the buffer.
* @param flags Flags for managing behavior of reading. [recv() only].
*/
```

```c
ssize_t read(int fd, void* buffer, size_t count); // General file-descriptor reader.
ssize_t recv(int sockfd, void* buffer, size_t count, int flags); // Socket-specific file-descriptor reader.

/**
* Write to the socket file descriptor. [SERVER + CLIENT]
* @param fd The file descriptor of the socket to write to.
* @param buffer A pointer to the buffer with the data to be sent.
* @param count The maximum amount of bytes to send through the socket.
* @param flags Flags for managing behavior of reading. [recv() only].
*/
ssize_t write(int fd, void* buffer, size_t count); // General file-descriptor writer.
ssize_t send(int sockfd, void* buffer, size_t count, int flags); // Socket-specific file-descriptor writer.

/**
* Closes the connection.
* @param fd The file descriptor of the socket to terminate.
*/
int close(int fd);

/**
* Creates a connection.
* @param sockfd The socket file descriptor (created by invoking socket()).
* @param addr A pointer to the sockaddr struct of the server containing networking information.
* @param addrlen The size of the addr struct.
*/
int connect(int sockfd, const struct sockaddr* addr, socklen_t addrlen);
```