LWIP WebSocket

Wojciech Mrzygłód, Jan Wołowiec

Table of Contents

Opis protokołu

WebSocket handshake

W celu ustanowienia połączenia z klientem należy stworzyć nowe połączenie TCP i uruchomić nasłuchiwanie połączeń przychodzących na wybranym porcie. W LwIP są to funkcje tcp_new(), tcp_bind (struct tcp_pcb *pcb, const ip_addr_t *ipaddr, u16_t port) i tcp_listen(struct tcp_pcb *pcb). Przykładowy kod jak otworzyć połączenie nasłuchujące tcp na danym porcie korzystając z biblioteki LwIP znajduje się poniżej.

Stworzenie połączenia TCP w C

```
websocket_t *new_websocket (ip_addr_t * ip, u16_t port)
{
	websocket_t *websocket = (websocket_t *)malloc(sizeof(websocket_t));
	if (websocket == NULL) return NULL;
	memset(websocket, 0x00, sizeof(websocket_t));

	websocket->ip = ip;
	websocket->port = port;
	websocket->status = websocket_status_closed;
	websocket->listen_conn = tcp_new();

	if (websocket->listen_conn == NULL) return 0;

	if (tcp_bind(websocket->listen_conn, ip, port) == ERR_OK)
	{
		websocket->listen_conn = tcp_listen( websocket->listen_conn);
		tcp_arg(websocket->listen_conn, websocket);
		tcp_accept(websocket->listen_conn, websocket_accept);
		return websocket;
	}
	else
	{
		free(websocket);
		return NULL;
	}
}
```

Szczegóły połączenia negocjowane są za pomocą protokołu HTTP. Podczas pierwszego zapytania przeglądarka wysyła do serwera zapytanie GET informujące serwer o zmianie protokołu komunikacji. Minimalny zestaw informacji w zapytaniu wysłany przez klienta wygląda następująco:

```
GET /chat HTTP/1.1
Host: example.com:8000
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

Powyższe zapytanie zawiera żądanie zmiany protokołu na websocket, wersję protokołu (według standardu RFC 6455 zawsze v13) i WebSocket-Key, który zostanie opisany później.

Na takie zapytanie serwer odpowiada kodem 101 Switching Protocols, co potwierdza powodzenie zmiany protokołu komunikacji. Minimalna odpowiedź serwera przedstawiona jest poniżej.

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

Najważniejszym elementem odpowiedzi jest Sec-WebSocket-Accept. Jest to hash SHA-1 Sec-WebSocket-Key otrzymany w zapytaniu HTTP klienta połączony z websocket GUID (258EAFA5-E914-47DA-95CA-C5AB0DC85B11), a następnie zakodowany base64, w celu sprawdzenia integralności przesłanych danych.

Wygenerowanie odpowiedzi HTTP Switching Protocols w C

Aby wygenerować websocket-key w C potrzebujemy funkcji skrótu sha1 i base64. Takie funkcje dostępne są na przykład w bibliotece mbed_tls. Ich użycie do wygenerowania odpowiedzi na zapytanie otwierające połączenie websocket przedstawia poniższy kod.

```
strlcpy((char *)&websocket->client_key[24], guid, 64);
SHA1((char*)websocket->encoded_key,(char*)websocket->client_key,sizeof(guid)+23);
char *base64_key = base64_encode((char*)websocket->encoded_key);
memcpy(websocket->encoded_key, base64_key, 28);
if (base64_key) free(base64_key);
websocket->encoded_key[28] = '\0';
```

Następnie zostaje wysłana odpowiedź HTTP Switching protocol z uzupełnionym polem Sec-WebSocket-Accept poprzed dodanie wcześniej obliczonego klucza

```
const char http_response[] = "HTTP/1.1 101 Switching Protocols\r\n"
                             "Upgrade: websocket\r\n"
                             "Connection: Upgrade\r\n"
                             "Sec-WebSocket-Accept: %s\r\n\r\n";
```

Do powyższego stringa klucz może zostać dopisany za pomocą funkcji sprintf z biblioteki standardowej, a następnie wysłany za pomocą funkcji tcp_write z biblioteki LwIP

```
sprintf(buffer, response, websocket->encoded_key);
tcp_write(tpcb, buffer, strlen(buffer), TCP_WRITE_FLAG_COPY);
```

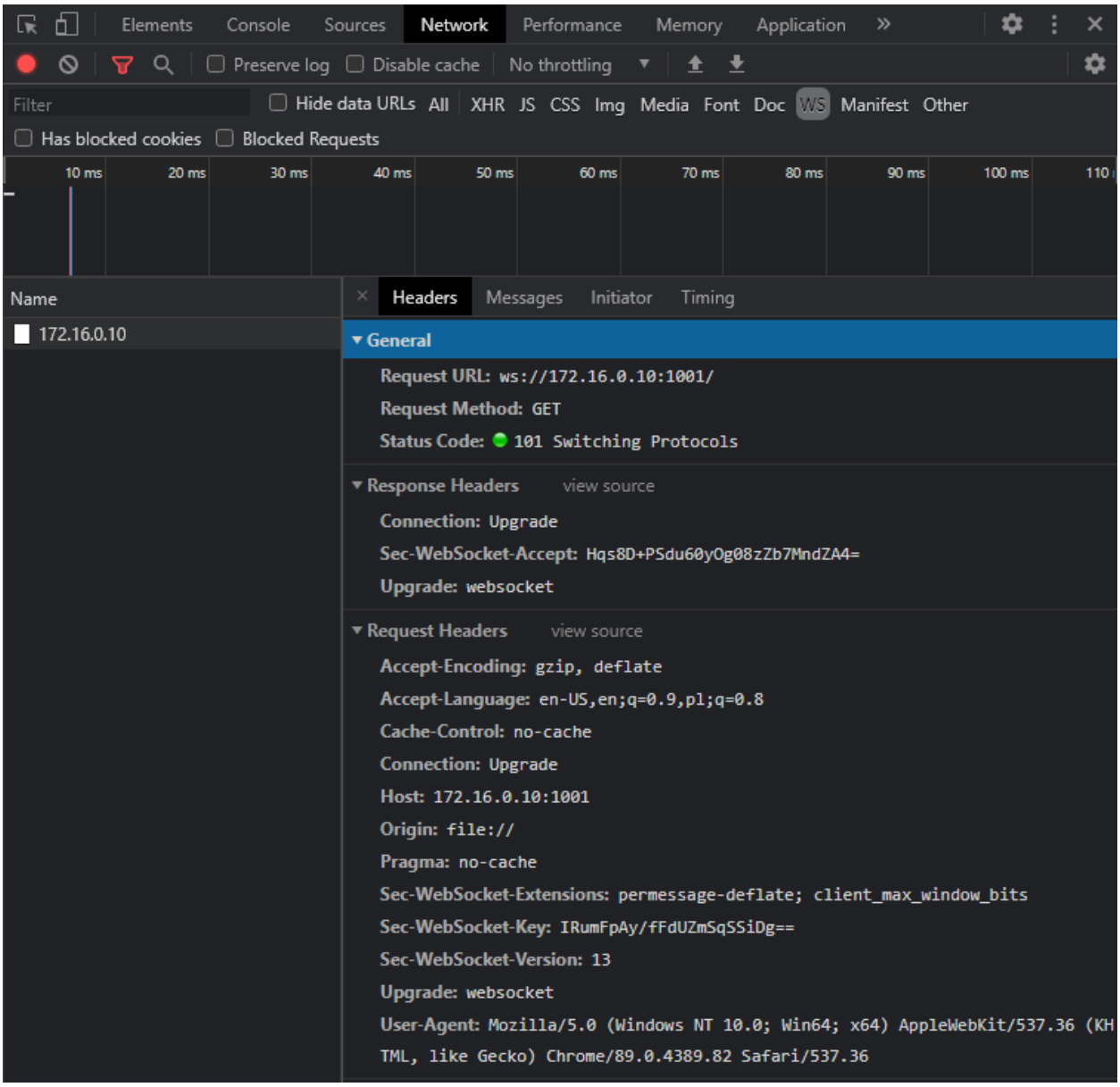
Rozpoczęcie połączenia w JavaScript

JavaScript natywnie obsługuje połączenia websocket. Połączenie rozpoczyna się tworząc nowy obiekt typu WebSocket z odpowiednim adresem IP i portem.

```
var WebsocketConnection = new WebSocket("ws://172.16.0.10:1001");
```

No.	Time	Source	Destination	Protocol	Length	Info
499	*REF*	172.16.0.104	172.16.0.10	TCP	66	49907 → 1001 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
500	0.000880	172.16.0.10	172.16.0.104	TCP	60	1001 → 49907 [SYN, ACK] Seq=0 Ack=1 Win=2048 Len=0 MSS=1446
501	0.000935	172.16.0.104	172.16.0.10	TCP	54	49907 → 1001 [ACK] Seq=1 Ack=1 Win=64240 Len=0
502	0.001088	172.16.0.104	172.16.0.10	HTTP	541	GET / HTTP/1.1
503	0.003963	172.16.0.10	172.16.0.104	HTTP	183	HTTP/1.1 101 Switching Protocols
504	0.048546	172.16.0.104	172.16.0.10	TCP	54	49907 → 1001 [ACK] Seq=488 Ack=130 Win=64111 Len=0

Rysunek 1 Wymiana pakietów między klientem a serwerem podczas inicjacji połączenia

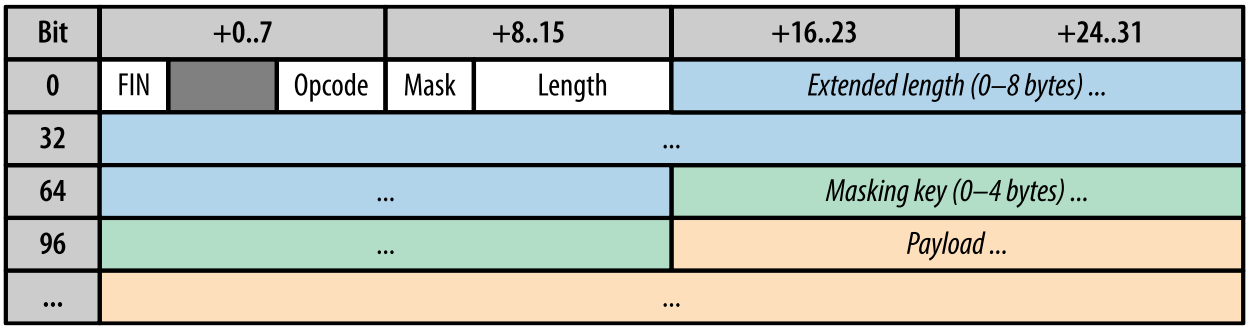


Rysunek 2 Poprawne rozpoczęcie połączenia przedstawia poniższy zrzut ekranu z narzędzia web inspector przeglądarki.

Transmisja danych

Po ustanowieniu połączenia poprzez utworzony kanał komunikacyjny można przesyłać ramki websocket po otwartym połączeniu TCP. Jest to największa różnica w porównaniu do http, gdzie połączenie po uzyskaniu odpowiedzi za każdym razem jest zamykane. W protokole websocket połączenie jest cały czas otwarte co umożliwia asynchroniczną wymianę danych między klientem a serwerem i zmniejsza narzut protokołu TCP na transmisję.

Struktura ramki websocket wygląda następująco



Rysunek 3 Struktura ramki Websocket

Pierwszym polem ramki jest bit FIN, który oznacza koniec transmisji w przypadku, kiedy pojedynczy pakiet danych składa się z wielu ramek.

Opcode zawiera typ ramki. W przypadku transmisji tekstowej jest to 1, a w przypadku ramki binarnej 2.

Kolejnym polem jest bit oznaczający włączone maskowanie danych w ramce. Ze względów bezpieczeństwa dane od klienta do serwera muszą być zamaskowane. Maskowanie polega na wykonaniu operacji XOR na danych z Masking key zawartym w nagłówku ramki. W przypadku, kiedy serwer otrzyma od klienta niezamaskowane dane powinien od razu zamknąć połączenie.

Pole Length zawiera informacje o długości danych w ramce. Kiedy zawiera wartość pomiędzy 0 a 125, jest to bezpośrednio ilość danych. Kiedy wartość tego pola jest równa 126, to kolejne pole Extended length ma 2 bajty i zawiera ilość danych w postaci 16 bitowej liczby, a kiedy Length równa się 127, to pole Extended length ma 64 bity.

Na końcu ramki znajdują się przesyłane dane (Payload).

Transmisja danych w JavaScript

Utworzony wcześniej obiekt typu WebSocket zawiera metodę.send() do wysyłania danych. Typ ramki dostosowuje się automatycznie do danych wejściowych.

```
WebsocketConnection.send(“test”); // ramka tekstowa
WebsocketConnection.send(new Uint32Array([0x13371337])); // ramka binarna
```

Odbieranie danych w JavaScript

W celu odbierania danych należy zdefiniować EventHandler .onmessage obiektu typu WebSocket. Poniższy przykład przedstawia wypisanie odebranych danych na konsolę debugową.

```
WebsocketConnection.onmessage = (event) => {console.log(event);};
```

Zrzut ekranu z konsoli po odebraniu ramki tekstowej o treści „test”

Rysunek 4 Odebrana ramka websocket

Transmisja danych w C

Aby wysłać ramkę websocket należy utowrzyć bufor do którego zostanie wpisana struktura, taka jak na Rysunku 1. Następnie za pomocą funkcji tcp_write z biblioteki LwIP taka ramka może zostać wysłana do klienta.

Odbieranie danych w C

Aby obebrać ramkę websocket należy stworzyć funkcję która będzie wywoływana wtedy kiedy na otwartym porcie TCP zostaną odebrane dane. Następnie należy sprawdzić czy format ramki zgadza się z opisem w dokumentacji protokołu. Następnie można skopiować dane użytkownika z ramki i przekazać je dalej do analizy w programie.

	Time	Source	Destination	Protocol	Length	Info
506	*REF*	172.16.0.104	172.16.0.10	WebSocket	64	WebSocket Binary [FIN] [MASKED]
507	0.046102	172.16.0.10	172.16.0.104	TCP	60	1001 → 51606 [ACK] Seq=130 Ack=498 Win=1551 Len=0

Rysunek 5 Transmisja pojedynczej ramki o standardowej długości

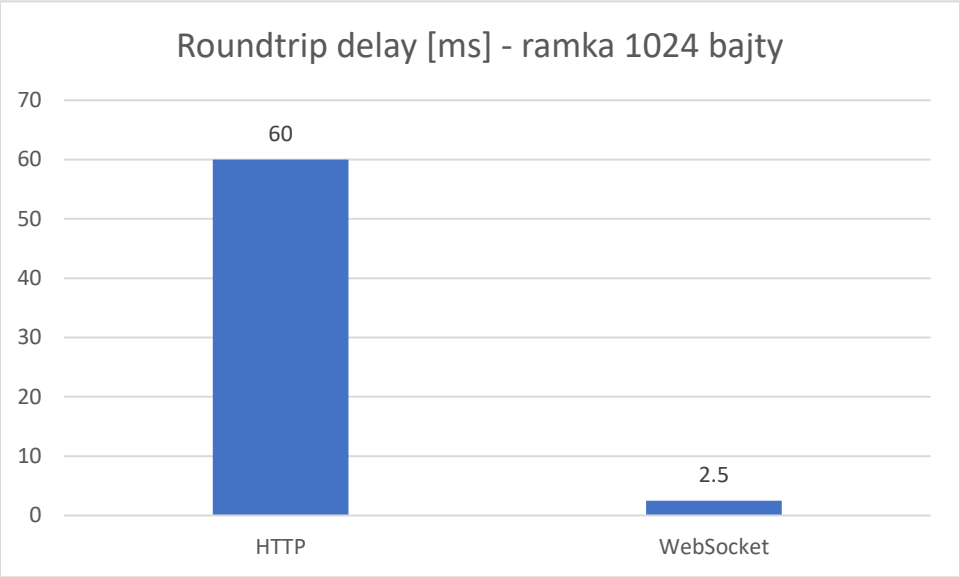
Zakończenie połączenia

W celu zakończenia połączenia serwer lub klient wysyła odpowiednią ramkę oznaczającą koniec połączenia. Wtedy też połączenie TCP jest zamykane. Ramka kończąca transmisję składa się z ciągu czterech bajtów: 0x88, 0x02, 0x03, 0xe8. Druga strona musi potwierdzić zamknięcie połączenia odsyłając taką samą ramkę.

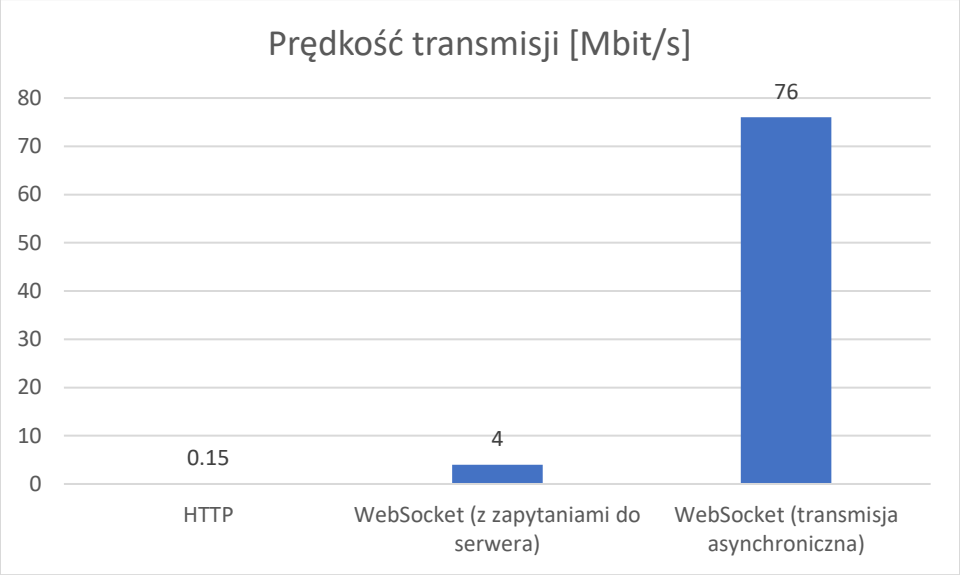
No.	Time	Source	Destination	Protocol	Length	Info
104	*REF*	172.16.0.104	172.16.0.10	WebSocket	60	WebSocket Connection Close [FIN] [MASKED]
105	0.001849	172.16.0.10	172.16.0.104	WebSocket	60	WebSocket Connection Close [FIN]
106	0.043828	172.16.0.104	172.16.0.10	TCP	54	53831 → 1001 [ACK] Seq=494 Ack=134 Win=64107 Len=0
107	0.044746	172.16.0.10	172.16.0.104	TCP	60	1001 → 53831 [FIN, ACK] Seq=134 Ack=494 Win=1555 Len=0
108	0.044783	172.16.0.104	172.16.0.10	TCP	54	53831 → 1001 [ACK] Seq=494 Ack=135 Win=64107 Len=0
109	0.044916	172.16.0.104	172.16.0.10	TCP	54	53831 → 1001 [FIN, ACK] Seq=494 Ack=135 Win=64107 Len=0
110	0.046917	172.16.0.10	172.16.0.104	TCP	60	1001 → 53831 [ACK] Seq=135 Ack=495 Win=1554 Len=0

Rysunek 6 Wymiana pakietów między klientem a serwerem podczas zamykania połączenia

Testy i porównanie z HTTP



Rysunek 7 Porównanie opóźnienia transmisji



Rysunek 8 Porównianie prędkości transmisji

HTTP

Time	Source	Destination	Protocol	Length	Info
REF	172.16.0.104	172.16.0.10	TCP	66	55909 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
0.004629	172.16.0.10	172.16.0.104	TCP	60	80 → 55909 [SYN, ACK] Seq=0 Ack=1 Win=2048 Len=0 MSS=1446
0.004690	172.16.0.104	172.16.0.10	TCP	54	55909 → 80 [ACK] Seq=1 Ack=1 Win=64240 Len=0
0.004822	172.16.0.104	172.16.0.10	HTTP	352	GET /xhr_test HTTP/1.1
0.007812	172.16.0.10	172.16.0.104	TCP	1097	80 → 55909 [PSH, ACK] Seq=1 Ack=299 Win=1750 Len=1043 [TCP segment of a reassembled PDU]
0.049040	172.16.0.104	172.16.0.10	TCP	54	55909 → 80 [ACK] Seq=299 Ack=1044 Win=65070 Len=0
0.049934	172.16.0.10	172.16.0.104	HTTP	60	HTTP/1.1 200 OK (text/html)
0.049963	172.16.0.104	172.16.0.10	TCP	54	55909 → 80 [ACK] Seq=299 Ack=1045 Win=65070 Len=0
0.050051	172.16.0.104	172.16.0.10	TCP	54	55909 → 80 [FIN, ACK] Seq=299 Ack=1045 Win=65070 Len=0
0.051075	172.16.0.10	172.16.0.104	TCP	60	80 → 55909 [ACK] Seq=1045 Ack=300 Win=1749 Len=0

Rysunek 9 Transmisja ramki 1024B przez HTTP

WebSocket

Time	Source	Destination	Protocol	Length	Info
REF	172.16.0.104	172.16.0.10	WebSocket	61	WebSocket Text [FIN] [MASKED]
0.005842	172.16.0.10	172.16.0.104	WebSocket	1025	WebSocket Text [FIN]
0.050700	172.16.0.104	172.16.0.10	TCP	54	56313 → 1001 [ACK] Seq=495 Ack=1101 Win=65070 Len=0

Rysunek 10 Tansmisja ramki 1024B przez websocket

Jak widać transmisja HTTP jest wolniejsza i ma większe opóźnienie z powodu potrzeby wymiana dużo większej ilości pakietów TCP pomiędzy klientem a serwerem - w tym przypadku 10. Dodatkowo HTTP jest protokołem tekstowym co powoduje większy narzut protokołu. W przypadku websocketa są to 3 ramki TCP, dodatkowo ramki websocket są binarne, co zmniejsza narzut protokołu na prędkość transmisji.

Teoretyczne porównanie sposobów komunikacji między serwerem a klientem

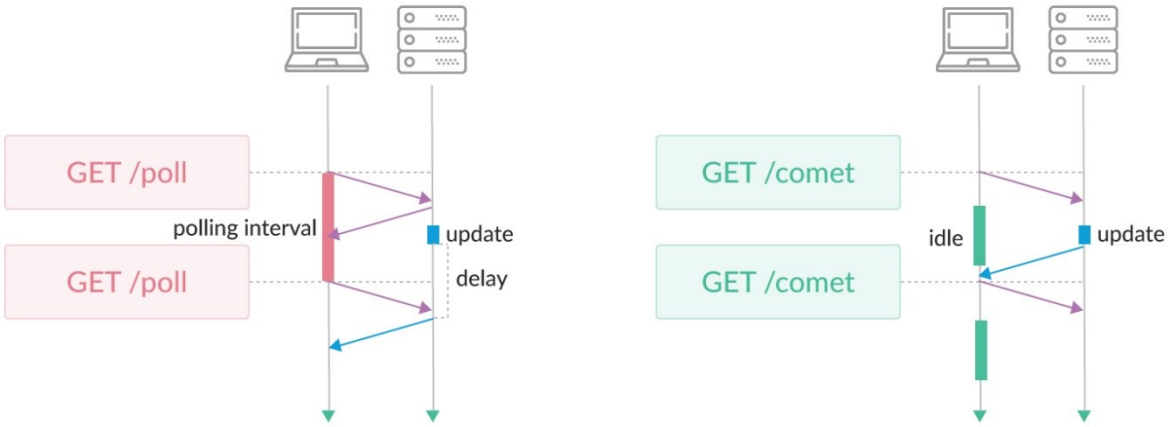
Long polling i short polling

Opis

W przypadku tradycyjnego podejścia zwanego “short polling”, klient wysyła regularne zapytania do serwera, aby sprawdzić czy dostępne są nowe dane. Jeśli nowe dane nie są dostępne serwer zwraca pustą odpowiedź i klient czeka pewien czas na wykonanie kolejnego zapytania. Wadą takiego rozwiązania jest duże zużycie zasobów i znaczące opóźnienia, zależne od częstotliwość odpytywania serwera.

Aby zminimalizować te problemy stosuje się tak zwany “long polling”, minimalizujący opóźnienie i zużycie zasobów. Efekt ten jest uzyskiwany dzięki temu, że serwer odpowiada na zapytanie dopiero wtedy, gdy dostępne są nowe dane. Klient po otrzymaniu danych zazwyczaj natychmiast wysyła kolejne zapytanie, powoduje to, że połączenie jest otwarte cały czas i pozwala na niejako asynchroniczną komunikację serwera.

Na rysunku poniżej został przedstawiony graficznie sposób komunikacji między serwerem, a klientem w przypadku kolejno short pollingu, jak i long pollingu:



Główne problemy long pollingu:

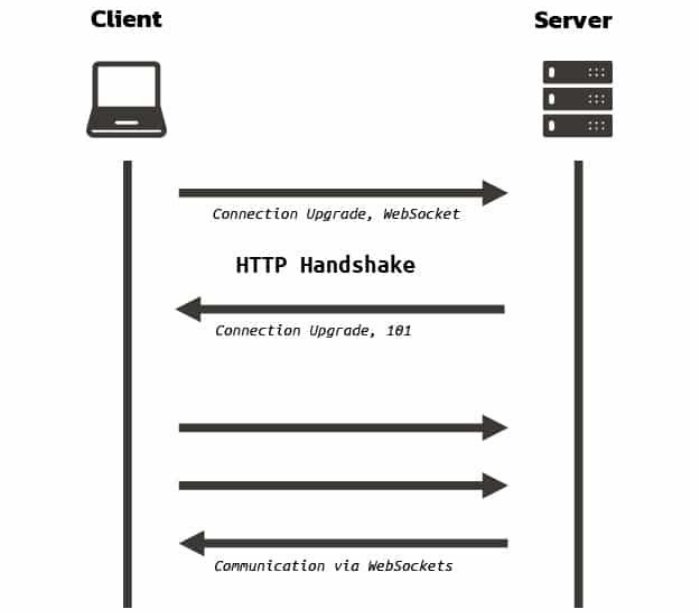
- Overhead protokołu HTTP – każde zapytanie i odpowiedź musza się składać z pełnej ramki HTTP, wskutek czego przesyłane są zbędne informacje
- Opóźnienie – pomiędzy odebraniem kolejnych odpowiedzi opóźnienie będzie wynosić zawsze więcej niż czas 3 transakcji (odpowiedź serwera, zapytanie klienta, odpowiedź serwera), dodatkowo należy uwzględnić możliwość wystąpienia retransmisji
- Wykorzystanie zasobów – otwarte połączenia TCP/IP w trakcie czekania na odpowiedź serwera, zwiększają zużycie zasobów

WebSocket

Opis

Protokół WebSocket pozwala na przesyłanie informacji między klientem, a serwerem dwukierunkowo oraz w dowolnym momencie. Połączenie jest otwierane na początku i kanał jest utrzymywany do wysyłania i odbierania danych. Jest to podejście przeciwne do standardu HTTP, który zakłada, że klient wysyła zapytanie i po otrzymaniu odpowiedzi połączenie jest zamykane.

WebSocket wykorzystuje HTTP tylko do otwarcia połączenia, dalsze dane są przesyłane bezpośrednio poprzez TCP/IP.



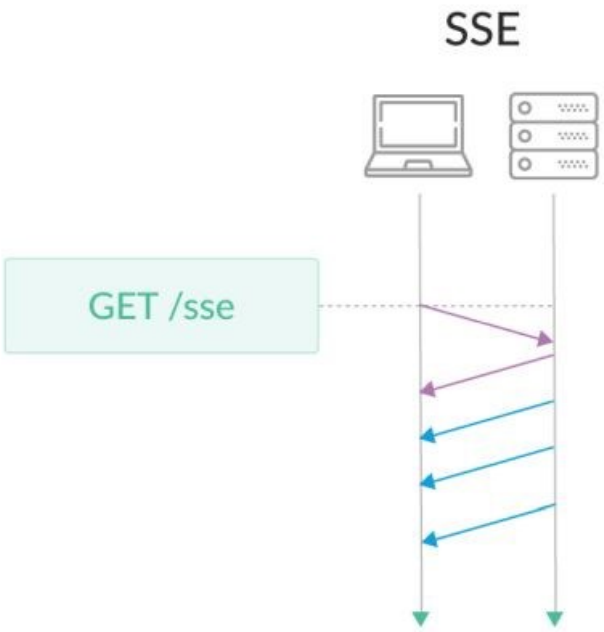
Główne problemy WebSocketów:

- Wykorzystanie zasobów – połączenie między klientem, a serwerem jest utrzymywane przez cały czas, kiedy strona jest otwarta
- Ilość połączeń – możliwa duża ilość połączeń jest teoretycznie zaletą websocketu, lecz w praktyce może to powodować problemy z ich obsługą. Z racji, że websocket domyślnie nie jest multipleksowany przez protokół HTTP/2, należałoby w takim wypadku zaimplementować własne rozwiązanie pozwalające na multipleksowanie połączeń websocket
- Load balancing – jest ono utrudnione w przypadku połączeń websocket, nie jest możliwe przeniesienie połączenia w przypadku wysokiego obciążenia serwera. Musi ono zostać najpierw zamknięte, a następnie otwarte.
- Brak mechanizmu ponownego połączenia – konieczna jest implementacja własnego rozwiązania
- Podatność na ataki DoS

SSE

Opis

HTTP dostarcza mechanizm przesyłania informacji z serwera do klienta za pomocą Server-Sent Events. Mechanizm ten pozwala na asynchroniczne przesyłanie danych przez serwer do klienta, po ustanowieniu między nimi połączenia.



Główne właściwości SSE

- Dane mogą być przesyłane jedynie od serwera do klienta
- W przypadku jeśli nie korzystamy z HTTP/2 wadą jest ograniczenie do 6 równoczesnych połączeń na serwer, lecz multipleksowanie połączeń w HTTP/2 znosi to ograniczenie.
- Dzięki wykorzystaniu HTTP mamy domyślne wsparcie load balancerów oraz proxies.
- W przeciwieństwie do long pollingu, który otwiera połączenie za każdym pullem, jest ono otwierane tylko raz, co powoduje, że nie mamy do czynienia m.in. z problemami overheadu i opóźnienia.
- Ważną zaletą jest również wbudowana funkcjonalność wznawiania połączenia w przypadku jego przerwania
- Jedną z wad jest brak wsparcia dla przesyłania danych w formie binarnej

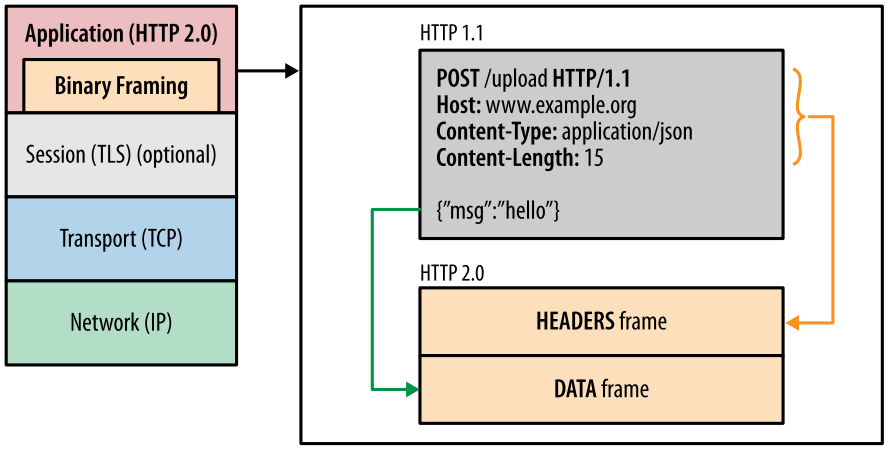
HTTP/2

Głównym celem wprowadzenia standardu HTTP/2 zwiększenie wydajności komunikacji pomiędzy klientem, a serwerem, przy jednoczesnym zachowaniu maksymalnej kompatybilności z obecnie działającymi aplikacjami.

Poniżej przedstawiono najważniejsze funkcjonalności wprowadzane przez protokół HTTP/2

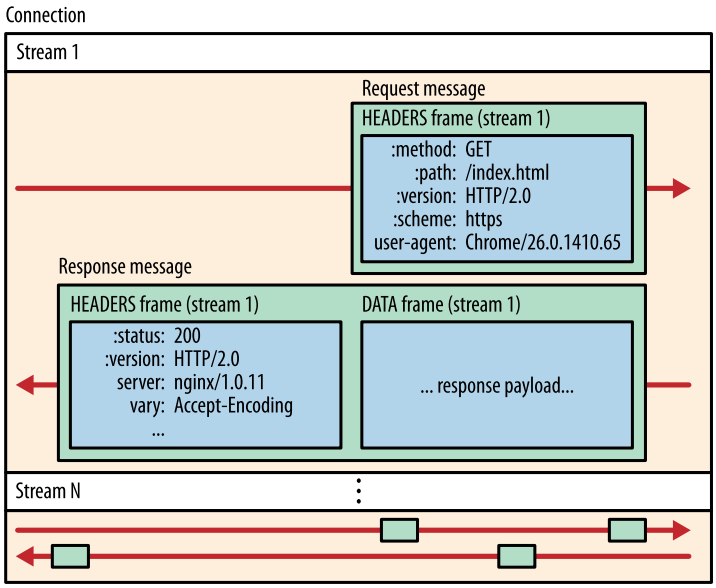
Binarny format komunikacji

Protokół HTTP/1.x wykorzystuje w komunikacji tekst oddzielony znakami nowej linii, w przeciwieństwie do niego komunikacja HTTP/2 jest dzielona na mniejsze części oraz enkodowana w formacie binarnym, celem zwiększenia efektywności.



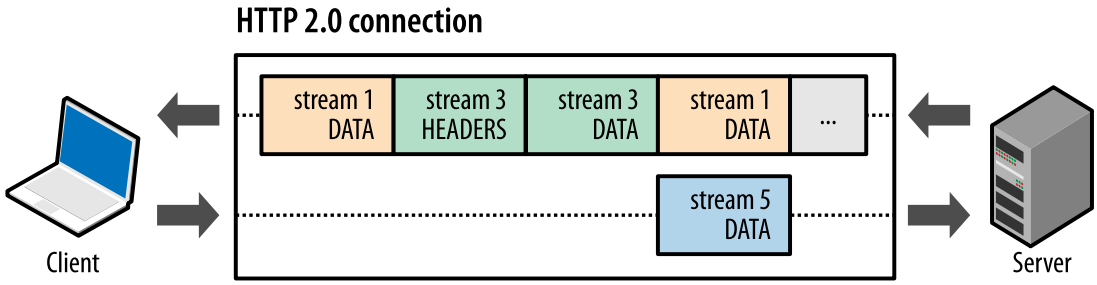
Streamy, wiadomości i ramki

HTTP/2 dzieli komunikację odpowiednio na streamy, wiadomości oraz ramki. Ramka jest to najmniejsza jednostka danych, która może być przesłana. Wiadomość z kolei jest to sekwencja ramek składająca się w logiczną całość. Wiadomości te organizowane są w streamy, określające dwukierunkowy przepływ danych.



Multipleksowanie komunikacji

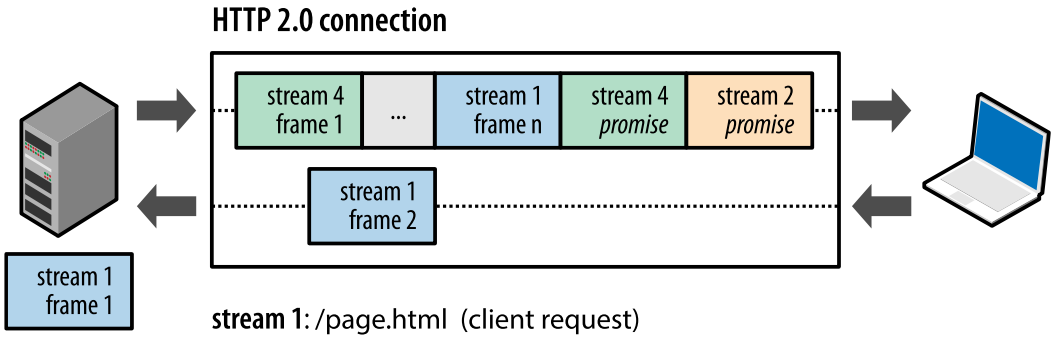
W przypadku HTTP/1.x jeśli klient chciał wykonać kilka równoległych zapytań, celem zwiększenia wydajności, wymagało to również użycia kilku połączeń TCP, co wynikało ze specyfikacji protokołu.



Zastosowanie multipleksowania w HTTP/2 pozwala na swobodne przesyłanie wielu zapytań i odpowiedzi za pomocą jednego połączenia w dowolnej kolejności, co znacznie usprawnia komunikację oraz eliminuje wiele problemów. Przykładowo pozwala to na ominięcie ograniczenia 6 jednoczesnych połączeń HTTP dla SSE z racji, że wykorzystanie HTTP/2 pozwala na przesyłanie dużej ilości streamów jednym połączeniem.

Server push

Mechanizm ten pozwala na przesłanie przez serwer do klienta dodatkowych zasobów, bez konieczności ze strony klienta na wysyłanie zapytań kolejno o każdy z nich osobno. Wykorzystując fakt, że serwer już wie czego będzie potrzebował klient, pozwala to skrócić czas ładowania stron. Warto jednak zwrócić uwagę, że mechanizm ten dotyczy interakcji przeglądarki z serwerem, lecz nie udostępnia on API dla aplikacji uruchamianych przez przeglądarkę. Oznacza to, że nie jest on zastępstwem dla protokołów takich jak np. WebSocket.



stream 1: /page.html (client request)
stream 2: /script.js (push promise)
stream 4: /style.css (push promise)

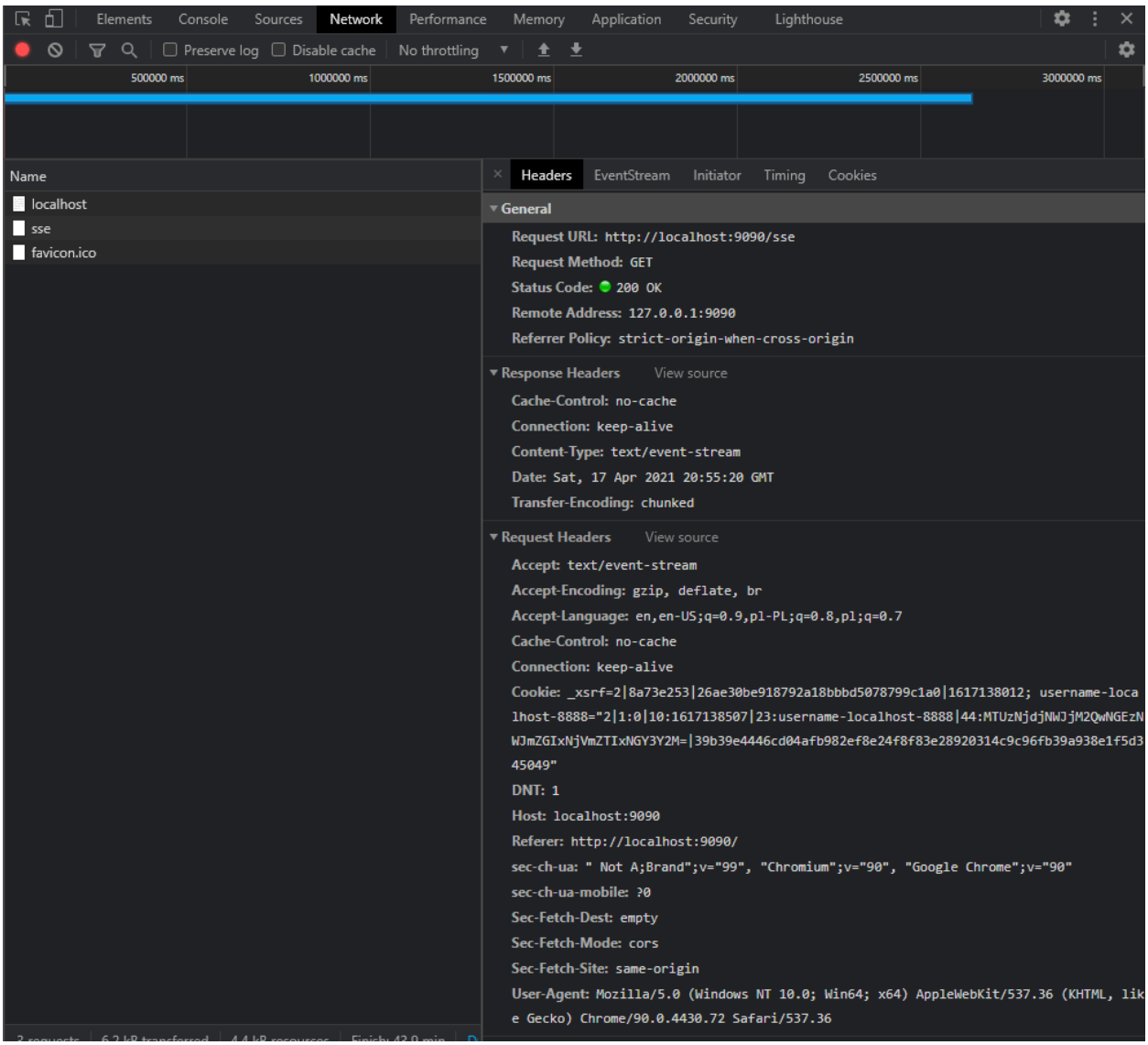
SSE w praktyce

Otwarcie połączenia

Aby nawiązać połączenie za pomocą SSE, na początek należy stworzyć obiekt **EventSource** z racji że JavaScript natywnie udostępnia API do obsługi server-sent events.

```
var ev = new EventSource('/sse');
```

Jako argument jest tu przekazywany relatywny adres pod który zostanie wysłane zapytanie otwierające połączenie SSE.



Rysunek 11 Otwarcie połaczenia SSE

Rozpoczęcie połączenia przedstawia powyższy zrzut ekranu z narzędzia web inspector przeglądarki.

Najważniejsze jest aby w zapytaniu znajdowało się pole:

```
Accept: text/event-stream
```

Pozwala ono na zidentyfikowanie, że mamy tu do czynienia z transmisją **event-stream**. Kolejne odpowiedzi przesyłane przez serwer będą również posiadały pole:

```
Content-Type: text/event-stream
```

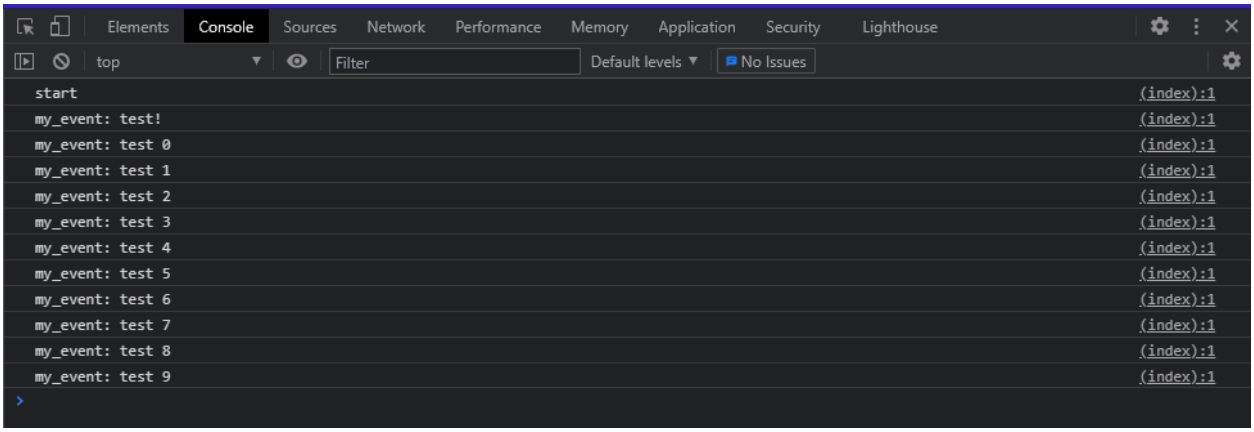
Odbieranie danych

Aby być w stanie odbierać dane należy rozpocząć nasłuchiwanie wiadomości. W tym celu można dodać handler onmessage, który będzie obsługiwał wszystkie przychodzące wiadomości, które nie są przypisane do żadnego eventu.

Dodatkowo możliwe jest nasłuchiwanie określonych eventów, które nas interesują. W tym celu wykorzystuje się funkcję "**addEventListener()**" pozwalającą skonfigurować event oraz skorelowany z nim callback, wywoływany po otrzymaniu tego eventu.

```
ev.onmessage = function(event) {
   console.log(event.data)
};
ev.addEventListener('my_event', function(event) {
   console.log(event.type + ': ' + event.data)
})
```

Poniżej przedstawiono wynik konsoli przy odbieraniu danych przesyłanych przez serwer.



Rysunek 12 Odebrane wiadomości

Format danych

Każda otrzymana wiadomość, może rozpoczynać się od jednego z 4 możliwych pól:

- event – określa typ otrzymanego eventu, który później może zostać obsłużony za pomocą "**addEventListener**"
- data – zawiera dane przesyłane od serwera, jeśli zostanie odebranych kilka kolejnych linii są one łączone w jedną wiadomość
- id – wartość last event ID może posłużyć do rozpoznania w którym momencie zostało przerwane połączenie z serwerem.
- retry – określa czas pomiędzy kolejnymi próbami połączenia w ms.

Poniżej zostały przedstawione przykładowe wiadomości przesyłane stream'em SSE:

```
event: userconnect
data: {"username": "bobby", "time": "02:33:48"}

data: Here's a system message of some kind that will get used
data: to accomplish some task.

event: usermessage
data: {"username": "bobby", "time": "02:34:11", "text": "Hi everyone."}
```

Przesyłane wiadomości zawsze rozdzielone są dodatkowym znakiem nowej linii, w innym przypadku są one traktowane jako jedna wiadomość. W tym przypadku otrzymamy 3 wiadomości, pierwsza i trzecia z nich będą obsłużone jako eventy jeśli były skonfigurowane w funkcji "**addEventListener**", zostanie również do nich przypisany obiekt JSON przesyłany w polu "**data:**". Z kolei w przypadku wiadomości numer dwa odbierzemy oba stringi połączone ze sobą jako "*Here's a system message of some kind that will get used\nto accomplish some task.*". Z racji, że nie posiadają one przypisanego eventu zostaną one obsłużone w handlerze "**onmessage**".

Przykładowe implementacje serwerów

- https://github.com/manuel-di-iorio/sse-node
- https://github.com/kammce/esp32-webserver-sse

Za bazę do testów posłużył serwer zaimplementowany w Node.js, wykorzystujący natywnie wspierany serwer HTTP do odpowiedniej obsługi zapytań oraz odsyłania danych w postaci opisanej w poprzednich punktach.

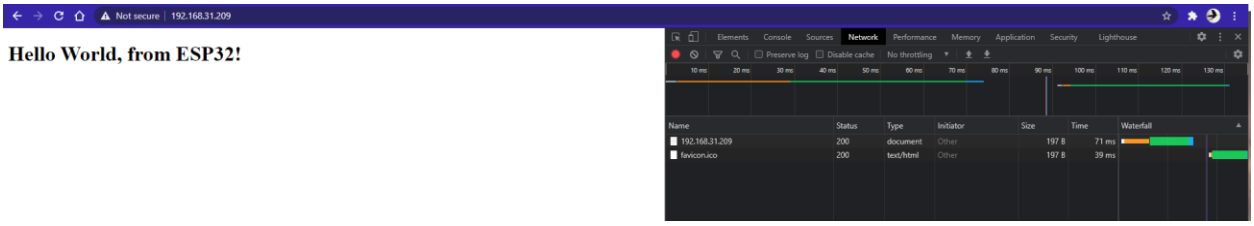
Test SSE na ESP32

Wykonano fork przykładowej implementacji z ESP32 podanej w poprzednim punkcie, aby uruchomić kod na nowej wersji ESP-IDF oraz wprowadzić pewne usprawnienia.
https://github.com/NVi5/esp32-webserver-sse

Aby połączyć się z siecią, należy jedynie wprowadzić odpowiednie SSID oraz hasło pozwalające się zalogować do sieci WIFI.

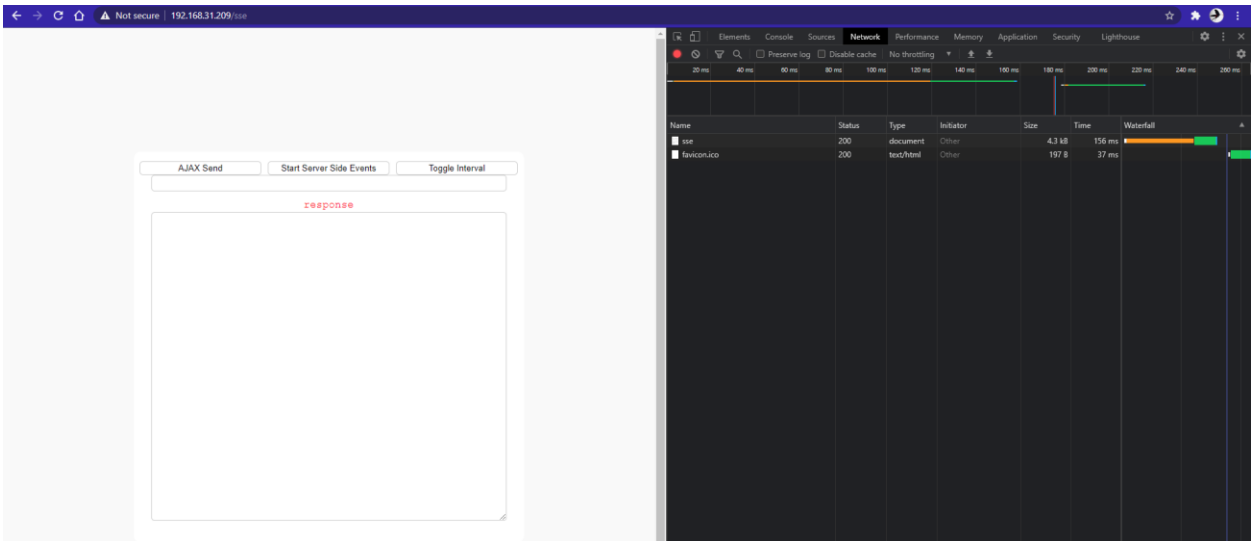
Działanie serwera

Aby połączyć się z serwerem za pomocą przeglądarki należy wpisać adres przypisany urządzeniu przez router. Zostanie wtedy wyświetlona prosta strona z powitaniem.



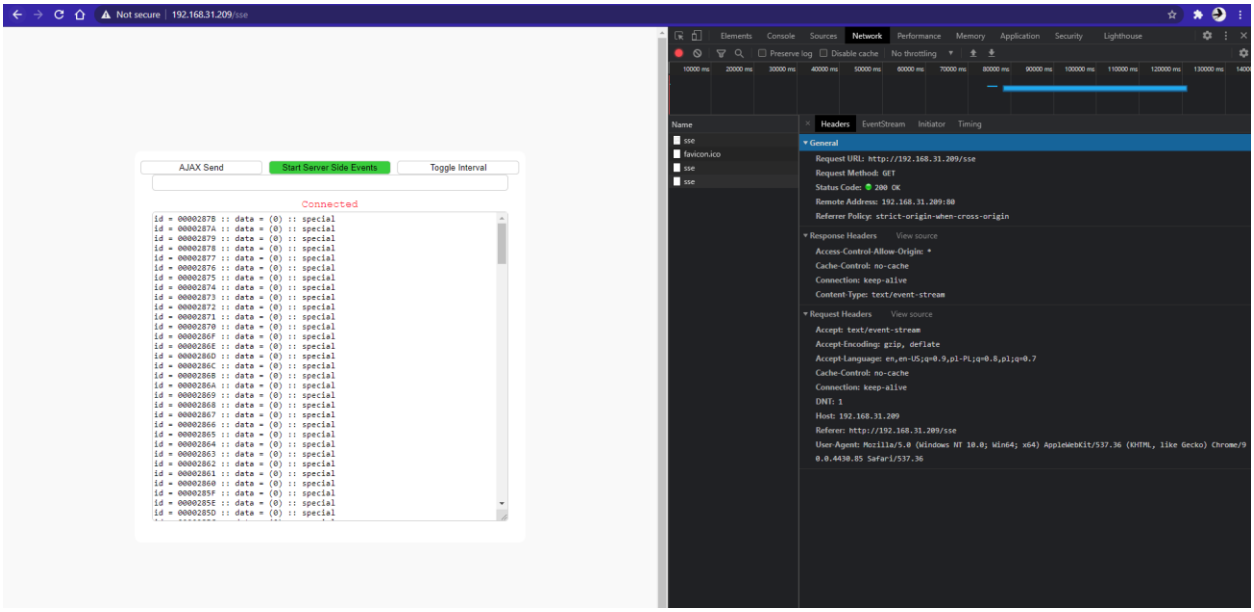
Rysunek 13 Strona z powitaniem

Aby przejść do testów SSE, należy udać się pod adres /sse, jak pokazano poniżej.



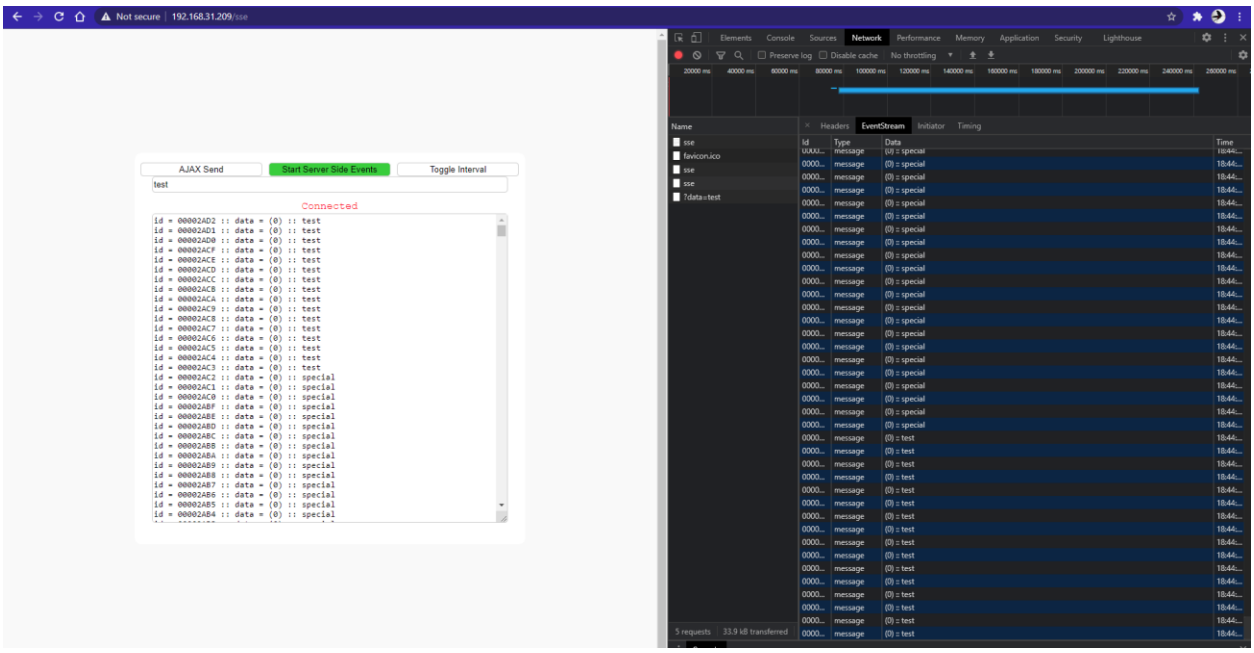
Rysunek 14 Strona do testowania SSE

Po kliknięciu przycisku Start Server Side Events, zostanie otwarte połączenie SSE i serwer zacznie transmitować dane.



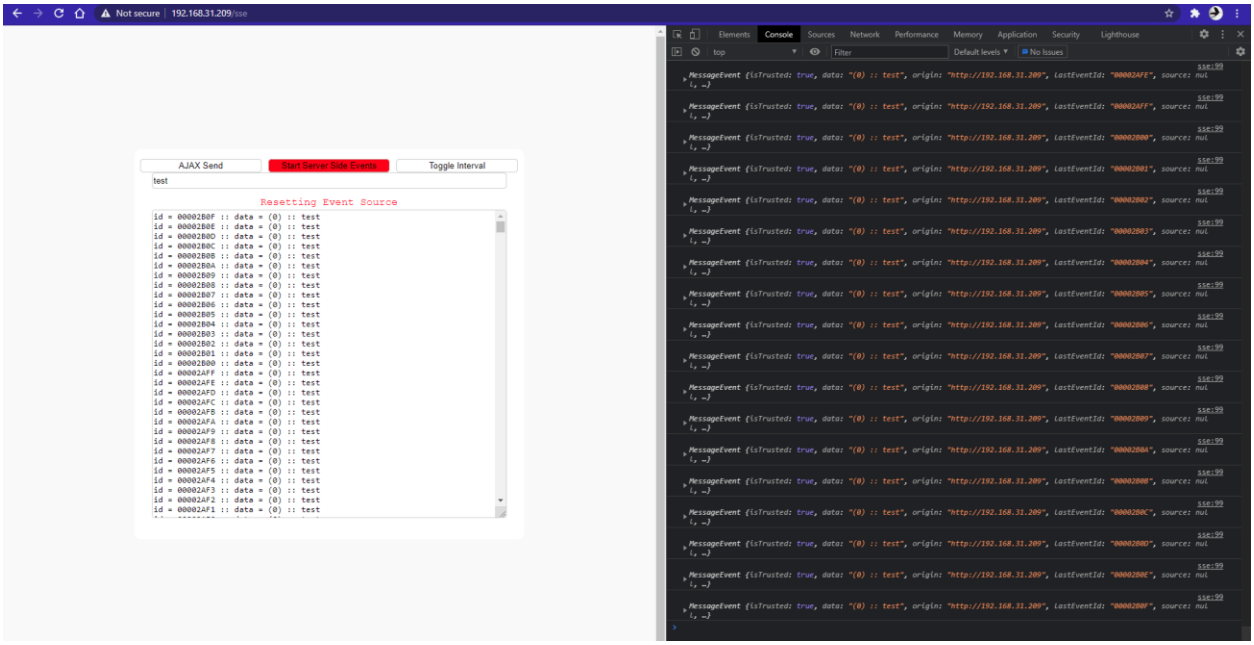
Rysunek 15 Widok transmisji danych w zakładce Network

Dodatkowo za pomocą przycisku AJAX Send możemy wysłać string który będzie później do nas odsyłany przez SSE.



Rysunek 16 Widok odebranych pakietów EventStream

Warto też zwrócić uwagę na widok pakietów w zakładce EventStream z zakładki Network, która pokazuje otrzymane wiadomości. Dodatkowe informacje są również wyświetlane w oknie konsoli.



Rysunek 17 Widok konsoli

Analiza kodu

Z perspektywy kodu można wyszczególnić 2 najważniejsze części, task „handle_sse” oraz task „http_server”.

handle_sse

Task ten okresowo iteruje po liście klientów SSE i jeśli dany klient istnieje, to znaczy nie jest nullem, następuje próba transmisji pewnej przygotowanej wiadomości. W przypadku gdy taka transmisja się nie powiedzie dany klient jest usuwany.

http_server

Task oczekuje na połączenie przychodzące i w razie otrzymania takowego tworzy osobne taski „http_server_netconn_serve”, które następnie sprawdzają jakiego typu request otrzymaliśmy.

Kolejność sprawdzania jest następująca:

1. Jeśli request przychodzący zawiera pole „Accept: text/event-stream” spróbuj dodać kolejnego klienta SSE obsługiwanego przez handle_sse, w przeciwnym wypadku przejdź dalej.
2. Jeśli request przekazuje dane w postaci „/?data=???” spróbuj odczytać dane, w przeciwnym wypadku przejdź dalej.
3. Jeśli request jest kierowany pod adres „/sse” odeślij stronę służącą do testowania funkcjonalności SSE, w przeciwnym wypadku przejdź dalej.
4. Jeśli request jest kierowany pod adres „/” odeślij stronę powitalną.

Jeśli jakieś string zostanie przekazany do serwera np. w postaci„/?data=test”, będzie on później odsyłany w takiej formie taskiem handle_sse.

Źródła

- https://developers.google.com/web/fundamentals/performance/http2
- https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers
- https://www.smashingmagazine.com/2018/02/sse-websockets-data-flow-http2/
- https://tools.ietf.org/html/rfc6202
- https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events
- https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events