



POLITECHNIKA WARSZAWSKA  
WYDZIAŁ ELEKTRYCZNY

Instytut Elektrotechniki Teoretycznej  
i Systemów Informacyjno-Pomiarowych  
Zakład Elektrotechniki Teoretycznej  
i Informatyki Stosowanej

# PRACA DYPLOMOWA INŻYNIERSKA

na kierunku Informatyka  
w specjalności: Inżynieria oprogramowania

Wykorzystanie protokołu HTTP/2 do budowy szybkiej aplikacji  
internetowej

Piotr Szklanko  
nr albumu 244145

promotor  
mgr inż. Bartosz Chaber

Warszawa, 2017

# Wykorzystanie protokołu HTTP/2.0 do budowy szybkiej aplikacji internetowej

## Streszczenie

Praca składa się ze wstępu, w którym informuję o czym jest praca i dlaczego zdecydowałem się na taki temat. Opisuję też krótko wybrane technologie oraz biblioteki oraz powody, dla których się na nie zdecydowałem. Drugi rozdział to krótka historia protokołu HTTP/2 oraz opis elementów, które zostały do niego wprowadzone. Trzeci rozdział opisuje kluczowe elementy stworzonej aplikacji. Krok po kroku przedstawia ich implementację. W rozdziale czwartym przeprowadzam testy porównawcze obu wersji protokołu HTTP. Dodatkowo sprawdzam też kompatybilność protokołu HTTP/2 z najnowszymi przeglądarkami internetowymi. W ostatnim rozdziale podsumowuję wyniki swojej pracy, wyciągam wnioski oraz przedstawiam plany na przyszłość związane z tym projektem.

**Słowa kluczowe:** protokół, HTTP, HTTP/2, SERVER PUSH

## THESIS TITLE

## Abstract

**Keywords:** thesis, LaTeX, quality

Warszawa, 1 lutego 2017

POLITECHNIKA WARSZAWSKA  
WYDZIAŁ ELEKTRYCZNY

### OŚWIADCZENIE

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa inżynierska pt. Wykorzystanie protokołu HTTP/2 do budowy szybkiej aplikacji internetowej:

- została napisana przeze mnie samodzielnie,
- nie narusza niczyich praw autorskich,
- nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam, że przedłożona do obrony praca dyplomowa nie była wcześniej podstawą postępowania związanego z uzyskaniem dyplomu lub tytułu zawodowego w uczelni wyższej. Jestem świadom, że praca zawiera również rezultaty stanowiące własności intelektualne Politechniki Warszawskiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Władz Wydziału Elektrycznego.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Piotr Szklanko.....



# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>1</b>
<b>2</b>	<b>Opis wybranych elementów HTTP/2</b>	<b>3</b>
2.1	Historia . . . . .	3
2.2	Protokół binarny . . . . .	3
2.3	Multiplexing . . . . .	4
2.4	Pierwszeństwo . . . . .	5
2.5	Server push . . . . .	6
2.6	Kompresja nagłówków . . . . .	6
<b>3</b>	<b>Budowa aplikacji testowej</b>	<b>9</b>
3.1	node-spdy – konfiguracja serwera . . . . .	9
3.2	Server Push . . . . .	10
<b>4</b>	<b>Testy</b>	<b>14</b>
4.1	Chrome DevTools . . . . .	14
4.2	Środowisko testowe . . . . .	16
4.3	Przygotowanie testów . . . . .	18
4.4	Porównanie prędkości protokołu HTTP/2 i HTTP/1.1 . . . . .	21
4.5	Porównanie prędkości przy bezpiecznym połączeniu HTTP/1.1 . . . . .	22
4.6	Porównanie prędkości z włączonym CACHE . . . . .	25
4.7	Porównanie prędkości przy wykorzystaniu Server Push . . . . .	25
4.8	Kompatybilność popularnych przeglądarek internetowych z HTTP/2 . . . . .	31
<b>5</b>	<b>Wnioski</b>	<b>32</b>
	<b>Bibliografia</b>	<b>35</b>



# Rozdział 1

## Wstęp

Moim celem jest przeprowadzenie testów protokołu HTTP w najnowszej wersji – HTTP/2 (opisanej w standardzie RFC7540 [1]). Obecnie powszechnie stosowana jest wersja 1.1 (opisanej w standardzie RFC2616 [6]), która została wprowadzona w roku 1999. Jednakże szybki rozwój technologii internetowych sprawia, że wprowadzony osiemnaście lat temu protokół przestaje powoli spełniać swoje zadanie. w tym momencie bez wykorzystania takich środków jak:

1. wykorzystanie kieszeniowania przeglądarki, dzięki któremu nie musimy przysyłać wszystkich plików naszej aplikacji do użytkownika, który korzysta z niej kolejny raz. Wysyłamy jedynie to, co się zmieniło,
2. wielu połączeń TCP – wiele połączeń TCP oznacza straty związane z czasem wymaganym do nawiązania połączenia[5]. Jest to szczególnie widoczne przy pobieraniu małych zasobów, gdzie czas nawiązania połączenia jest duży w stosunku do czasu wykonania samego zapytania. Niestety w przypadku HTTP/1.1 jest to jedyny sposób na jednoczesne przesyłanie wielu zasobów,
3. łączenia zasobów – sposób na ograniczenie liczby połączeń. Dzięki wykorzystaniu narzędzi takich jak webpack możemy ograniczyć liczbę połączeń TCP poprzez łączenie wielu plików danego typu w jeden duży plik. Na przykład gdy mamy wiele plików z arkuszami stylów możemy je połączyć w jeden. Jest to tylko mała część możliwości tego pakietu, zainteresowanych odsyłam do strony internetowej[2].

nie jest możliwe stworzenie rozbudowanej aplikacji, która działałaby w sposób satysfakcjonujący użytkownika. Gdyby zmiany, które wprowadza protokół HTTP/2 faktycznie pozwalały zapomnieć o wspomnianych środkach, to życie wielu programistów stałoby się dużo łatwiejsze. Dzięki temu mogliby oni ten czas poświęcić na rozwój aplikacji.

Za pomocą samodzielnie zbudowanej aplikacji zamierzam przetestować wpływ funkcji oferowanych przez HTTP/2, w kontekście tworzenia aplikacji internetowych. Dodatkowo praca ta była dla mnie motywacją do lepszego poznania protokołu HTTP ogólnie, nie tylko jego najnowszej wersji.

Swoją aplikację stworzyłem wykorzystując zestaw oprogramowania MEAN[4] – MongoDB, Express.js, Angular i Node.js.

- MongoDB – baza danych NoSQL,
- Express.js – framework Node.js do tworzenia aplikacji sieciowych od strony serwera. Udostępnia on wiele metod ułatwiających obsługę zapytań HTTP, routing zapytań, renderowanie widoków HTML,
- Angular – framework JavaScript służący do budowy dynamicznych aplikacji internetowych od strony użytkownika,
- Node.js – środowisko uruchomieniowe języka JavaScript, które pozwala wystartować serwer.

Zdecydowałem się na to rozwiązanie z kilku powodów:

- po przejrzaniu dostępnych w sieci informacji doszedłem do wniosku, że implementacja protokołu HTTP/2 jest najlepiej opisana oraz wspierana przez środowisko związane z JavaScriptem,
- dobrej znajomości języka JavaScript oraz jednoczesna chęć rozwoju umiejętności tworzenia aplikacji w tym języku,
- chęci poszerzenia wiedzy dotyczącej budowania aplikacji internetowych za pomocą technologii javascriptowych,
- znaczenia i popularności tego rozwiązania na rynku pracy.

Dodatkowo, poza narzędziami składającymi się na zestaw MEAN, wykorzystałem następujące biblioteki:

1. node-spdy – zewnętrzny moduł do node.js, który umożliwia tworzenie serwerów wspierających HTTP/2 (repozytorium GitHub: [11]). Jest on kompatybilny z biblioteką Express.js, którą wykorzystuję w swoim projekcie. Dzięki temu modułowi możemy zaimplementować serwer HTTP/2 wraz z Server Push. Pomimo, że nie jest to oficjalny moduł node.js, to trwające obecnie prace, które mają na celu wdrożenie HTTP/2 oficjalnie do node.js, bazują na tej bibliotece,
2. mongoose – ułatwia modelowanie danych MongoDB, walidację oraz pisanie logiki biznesowej.



## Rozdział 2

# Opis wybranych elementów HTTP/2

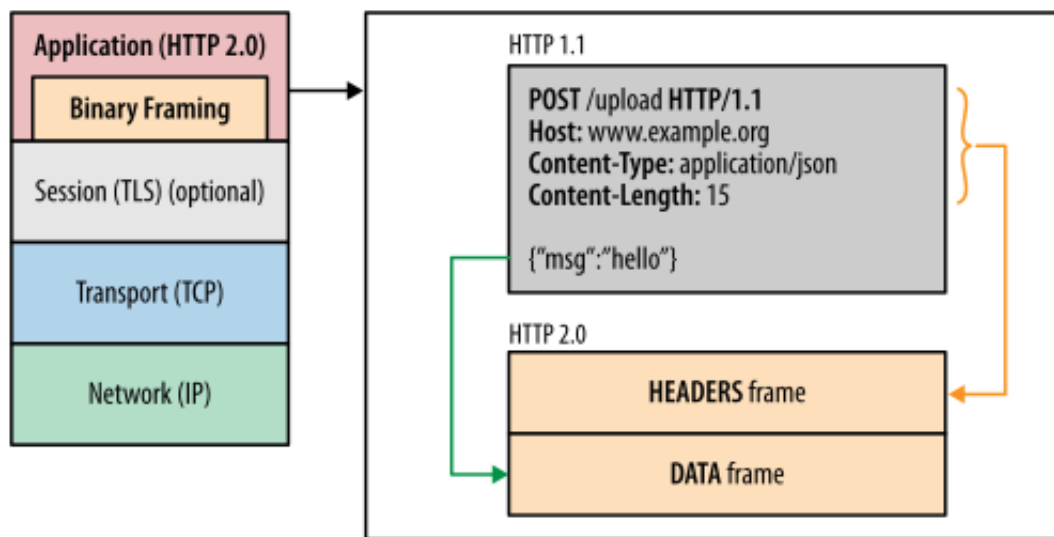
### 2.1 Historia

Pracę nad zmianami w protokole zapoczątkowała w 2009 roku firma Google ze swoim projektem SPDY. Zdecydowali się oni na stworzenie protokołu, który miał usprawnić działanie aplikacji oraz stron internetowych rozwiązując ograniczenia nałożone przez HTTP/1.1. Z biegiem czasu coraz więcej przeglądarek oraz stron internetowych, zarówno tych dużych jak i tych małych, zaczęło wspierać SPDY, co zainteresowało osoby pracujące nad protokołem HTTP. Zdecydowali się oni wykorzystać dokumentację protokołu SPDY jako początek prac nad własnym protokołem – HTTP/2. Od tego momentu aż do roku 2015, kiedy to standard HTTP/2 został oficjalnie zaakceptowany (specyfikacja protokołu: [1]), projekty były rozwijane równolegle. SPDY było wykorzystywane do testów nowych funkcji, które miały zostać wprowadzone do nowego protokołu HTTP. Niedługo po oficjalnym zaakceptowaniu HTTP/2 ogłoszono, że SPDY nie będzie dalej wspierane.

W kilku poniższych akapitach postaram się przybliżyć zmiany, które zostały wprowadzone do protokołu HTTP.

### 2.2 Protokół binarny

Kluczową zmianą, która determinuje brak wstecznej kompatybilności z HTTP/1.1, jest przejście na kodowanie binarne przesyłanych wiadomości. Przykładowa ramka widoczna jest na rysunku 2.1. Jest to rozwiązanie dużo bardziej kompaktowe i łatwiejsze w implementacji, niż przesyłanie zwykłego tekstu. Dzięki temu zabiegowi w ramach jednego połączenia TCP z serwerem

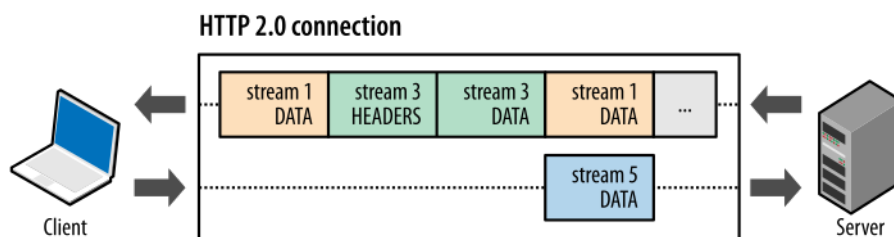


Rysunek 2.1: Schemat ramki protokołu HTTP/2 (źródło: [3])

może zostać utworzonych wiele dwukierunkowych strumieni danych przesyłających wiadomości HTTP. Taka wiadomość to w rzeczywistości zapytanie od klienta lub odpowiedź serwera składające się z ramek. Każda ramka natomiast musi posiadać przynajmniej nagłówek z informacją, do którego strumienia danych należy. Kodowanie binarne nie ma wpływu na składnię zawartości ramki – wszystkie nagłówki czy zapytania HTTP/1.1 pozostawiono bez zmian.

## 2.3 Multiplexing

W poprzedniej wersji protokołu, pomimo, że istniała możliwość przesyłania wielu zapytań w ramach jednego połączenia, nie można było wykonywać ich równolegle. Każde zapytanie musiało być rozpatrywane i odesłane przez serwer do klienta zgodnie z kolejnością nadania, co powodowało efekt head-of-line blocking (problem polegający na head-of-line blocking, dokładniej opisany w [7]). Aby wykonywać zapytania równolegle należało utworzyć kilka zapytań TCP, co obciążało serwer oraz było czasochłonne. Protokół HTTP/2 umożliwia przesyłanie oraz odbieranie wielu wiadomości jednocześnie, co pokazuje schemat na rysunku 2.2. Są one rozbijane na pojedyncze ramki, przesyłane, a następnie odczytywane i składane z powrotem w całość po stronie odbiorcy. Dzięki temu nie jest już konieczne uciekanie się do takich



Rysunek 2.2: Schemat wykorzystania multiplexingu w HTTP/2 (źródło: [3])

zabiegów jak:

- scalanie plików (na przykład webpack),
- wykorzystywanie duszków CSS (więcej o użyciu duszków CSS w kontekście aplikacji internetowych: [8]),
- domain sharding – rozdzielanie zasobów strony internetowej na wiele domen. Dzięki temu klient może jednocześnie pobierać większą liczbę zasobów, ponieważ liczba aktywnych połączeń z jedną domeną jest ograniczona. Zazwyczaj wykorzystywane do pobierania zewnętrznych bibliotek takich jak na przykład jQuery czy Bootstrap (dla zainteresowanych: [9]).

To wszystko sprawia, że aplikacje stają się szybsze oraz prostsze.

## 2.4 Pierwszeństwo

Po przeczytaniu poprzedniej sekcji możemy dojść do wniosku, że multiplexing nie ma prawa działać, bo przecież zazwyczaj kolejność otrzymanych informacji jednak ma znaczenie. Z tego powodu HTTP/2 wprowadza system wag zapytań. Umożliwia on:

1. przypisanie wagi w postaci liczby naturalnej od 1 do 256 każdemu strumieniowi danych,
2. ustalenie zależności danego strumienia od innych.

Informacja o pierwszeństwie umieszczana jest jako odpowiednie pola w nagłówku strumienia. Gdy zajdzie potrzeba zmiany priorytetów możemy wysłać również odpowiednią ramkę, która zawiera jedynie informacje o pierwszeństwie danego strumienia. W obu przypadkach przekazujemy informacje o zależności danego strumienia od innych strumieni oraz o jego wadze. W

wypadku, gdy klient nie ustawi tych wartości, to rodzicem strumienia staje się nieistniejący strumień 0 (0x0). Zostaje on jednocześnie korzeniem drzewa zależności strumieni, od którego zaczynamy rozpatrywanie wszystkich strumieni. Standardowa waga pakietu, w przypadku, gdy nie została ustalona ta wartość, to 16. Jeżeli dwa strumienie posiadają tego samego rodzica, to serwer dokonuje podziału zasobów (procesor, pamięć) proporcjonalnie do wag odpowiednich strumieni. Po przetworzeniu strumienia i przygotowaniu odpowiedzi przepustowość pasma również jest proporcjonalna do wagi przypisanej zapytaniu.

Dzięki temu zabiegowi serwer potrafi rozdysponować zasoby (np. CPU czy pamięć) tak, aby w optymalny sposób wykonać prośby nadesłane przez klienta.

## 2.5 Server push

Wykorzystując protokół HTTP/1.1 nie mamy możliwości otrzymania zasobu, o który nie poprosiliśmy wysyłając zapytanie. Powoduje to opóźnienia na przykład podczas ładowania strony internetowej. Zanim otrzymamy skrypty czy arkusze stylów, które wykorzystuje nasza strona musi ona o nie poprosić. Zapytanie do serwera wysyłane jest gdy w kodzie pliku HTML napotkamy na taki kod (przykład z mojego projektu 3.2):

Listing 2.1: Zasoby przesyłane z wykorzystaniem server push

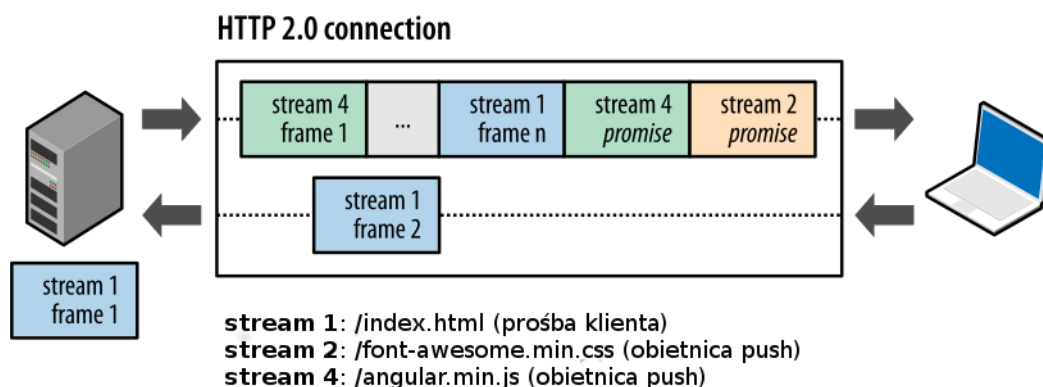
```
<link rel="stylesheet"
      href="libs/bootstrap/dist/css/bootstrap.min.css">
<link rel="stylesheet"
      href="libs/font-awesome/css/font-awesome.min.css">

<script src="libs/angular/angular.min.js"></script>
```

Takie rozwiązanie, chociaż w wielu przypadkach jest pożądane, tutaj jedynie spowalnia działanie aplikacji. Jeżeli mamy pewność, że użytkownik będzie potrzebował danych zasobów 2.3 możemy mu je od razu udostępnić, co zdecydowanie skraca czas ładowania aplikacji i dzięki temu unikam niechcianego efektu, gdy strona się załaduje, ale na przykład bez pliku zawierającego style, który jest dopiero przesyłany.

## 2.6 Kompresja nagłówków

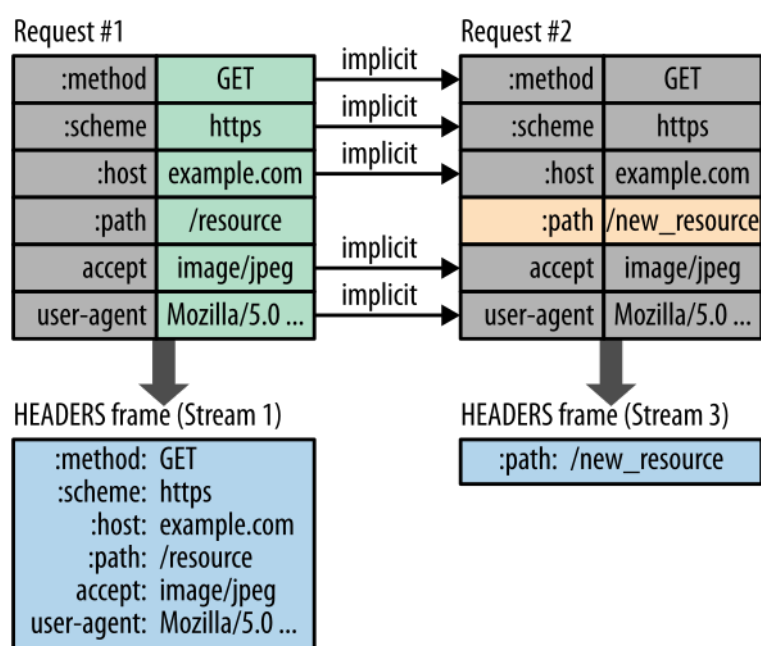
Kolejną ważną, chociaż pozornie niezauważalną zmianą jest kompresja nagłówków. Pomimo, że problem może wydawać się marginalny, to te kilka



Rysunek 2.3: Schemat Server push HTTP/2 (źródło: [3])

bajtów w każdym zapytaniu potrafi dość mocno wydłużyć czas komunikacji z serwerem. HTTP/2 wykorzystuje format kompresji zwany HPACK (opis standardu: [10]), który dzięki wykorzystaniu dwóch technik – kodowania oraz indeksowania, jest w stanie znacznie usprawnić komunikację ze względu na nagłówki. Po pierwsze kodowanie Huffmana znacznie kompresuje przesyłane dane. Dodatkowo klient oraz serwer przechowują listę widzianych wcześniej nagłówków i nie ma potrzeby wysyłania za każdym razem całego nagłówka. Przesyłamy jedynie pola, które się zmieniły (patrz rysunek 2.4).

W powyższych akapitach opisałem techniki wprowadzone w HTTP/2. W swojej pracy wykorzystam je do zbudowania własnej aplikacji. Następnie, wykorzystując stworzoną aplikację, zamierzam zbadać wydajność oraz poprawność ich działania.



Rysunek 2.4: Kompresja nagłówków w HTTP/2 (źródło: [3])

## Rozdział 3

# Budowa aplikacji testowej

### 3.1 node-spdy – konfiguracja serwera

Najważniejszym elementem aplikacji jest konfiguracja serwera, który będzie w stanie obsłużyć zapytania HTTP/2. Jak pokazuje listing 3.1 na początku należy ustawić opcje naszego serwera:

1. `plain` – jeśli opcja jest ustawiona na `true`, to serwer wykorzystuje protokół HTTP/1.1,
2. `ssl` – ustawienia zabezpieczeń połączenia – jeżeli ustawione na `false` oraz opcja `plain` ustawiona na `true`, to serwer będzie wykorzystywał HTTP/1.1 bez zabezpieczania połączenia za pomocą SSL,
3. `protocols` – lista protokołów, z których możemy korzystać,
4. `key` – klucz prywatny do połączeń SSL,
5. `cert` – certyfikat serwera do połączenia SSL.

Następnie uruchamiamy nasz serwer HTTP za pomocą polecenia `createServer` i podajemy mu naszą konfigurację oraz informację o tym, jak ma się zachować, gdy otrzyma zapytanie od klienta. W tym wypadku jest to moja aplikacja, więc za każdym razem, gdy serwer otrzyma zapytanie, będzie korzystał ze stworzonych przeze mnie funkcji.

Na koniec musimy ustawić nasz serwer tak, aby nasz serwer nasłuchiwał przychodzących połączeń. Osiągniemy to przy pomocy funkcji `listen()`, do której przekazujemy port, na którym nasza aplikacja ma działać. Jeśli wszystko zakończy się pomyślnie otrzymamy informację o tym, że aplikacja działa na wybranym przez nas porcie. Jeśli nie, to zostanie zwrócony komunikat o błędzie.

Listing 3.1: Konfiguracja serwera

```
var spdy          = require('spdy');
var port          = 8080;
var fs            = require('fs');
var app           = express();

const options = {
  spdy: {
    plain: true,
    ssl: false,
    protocols: ['h2', 'http/1.1'],
  },
  key: fs.readFileSync(__dirname + '/server.key'),
  cert: fs.readFileSync(__dirname + '/server.crt')
};

spdy
  .createServer(options, app)
  .listen(port, (error) => {
    if (error) {
      console.error(error);
      return process.exit(1);
    } else {
      console.log('Listening on port ' + port + '.');
    }
  });
```

Jak widać podstawowa konfiguracja serwera HTTP/2 z wykorzystaniem biblioteki node-spdy [11] jest dość prostym zadaniem. Należy jedynie pamiętać o wszystkich ustawieniach oraz wygenerowaniu kluczy dla połączenia SSL, ponieważ bez konfiguracji serwer się nie uruchomi.

## 3.2 Server Push

Najbardziej wymagającą częścią projektu było zaimplementowanie funkcji server push. Ostatecznie udało się stworzyć podstronę, która do pełnego działania wymaga biblioteki jQuery. Normalnie biblioteka ta byłaby wysłana dopiero po odczytaniu pliku push.html przez klienta i wysłaniu zapytania z prośbą o przesłanie jQuery. Jest to rozwiązanie, w którym tracimy czas na wysłanie zapytania o zasób, który mógł być wysłany od razu. Server Push zaimplementowany w HTTP/2 daje nam taką możliwość.

Przy pomocy listingu 3.2 postaram się przybliżyć sposób, w jaki udało mi się zaimplementować tę funkcję.

Najpierw tworzymy routing dla ścieżki /push. W nim, w zależności od tego co chcemy przesłać, scenariusz będzie trochę inny. W moim przypadku



przesyłam zawartość strony w postaci kodu HTML oraz bibliotekę jquery. Najpierw wczytujemy plik i zapisujemy jego zawartość. Jeśli wszystko zostało wykonane pomyślnie, to zapisujemy zawartość pliku HTML do wysłania jako odpowiedź na zapytanie użytkownika. Następnie wczytujemy zawartość pliku z biblioteką jQuery i umieszczamy ją w strumieniu danych wysyłanym w ramach funkcji Push. Ustawiamy nagłówki symulowanej prośby, typ odpowiedzi oraz dołączamy zawartość pliku, którą chcemy przesłać. Zamykamy strumień danych i kończymy odpowiedź komendą `res.end()`. Oczywiście w ramach jednego zapytania możemy przesłać wiele zasobów. Tworzymy po prostu kolejne strumienie danych analogicznie do pierwszego, który jest przedstawiony poniżej.

Server Push pomimo ogromnych usprawnień, których wyniki przybliżę w rozdziale 4.7, W związku z tym, że klient nie prosił o dane zasoby musimy koniecznie wypełnić nagłówek przesyłanego zasobu, aby przeglądarka mogła odpowiednio zareagować na odebrany zasób. Na przykład po prostu go odrzucić, gdy znajduje się on już w pamięci podręcznej. Jest to rozwiązane za pomocą `PUSH_PROMISE`, które jest wysyłane do klienta przed faktycznym przesłaniem całego strumienia. Jeśli klient zechce, to może odrzucić dany strumień wysyłając ramkę `RST_STREAM`.

Listing 3.2: Konfiguracja server push

```
app.get('/push', function(req, res) {
  fs.readFile('/push.html', function read(err, data) {
    if(err) {
      throw err;
    }
    content = data;
    res.write(content)
  })

  fs.readFile('/jquery.js', function read(err, data) {
    if(err) {
      throw err;
    }
    content = data;
    var stream = res.push('/libs/jquery/dist/jquery.js', {
      status: 200, // optional
      method: 'GET', // optional
      request: { accept: '*/*' },
      response: { 'content-type': 'application/javascript' }
    })
    stream.on('error', function(err) {
      console.log(err);
    })
    stream.end(content)
```

```

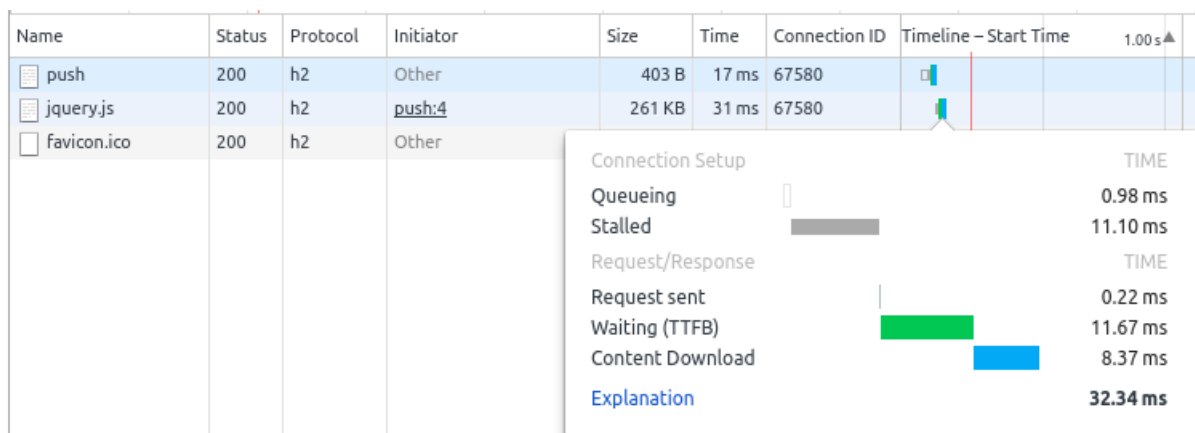
        res.end();
    })
})

```

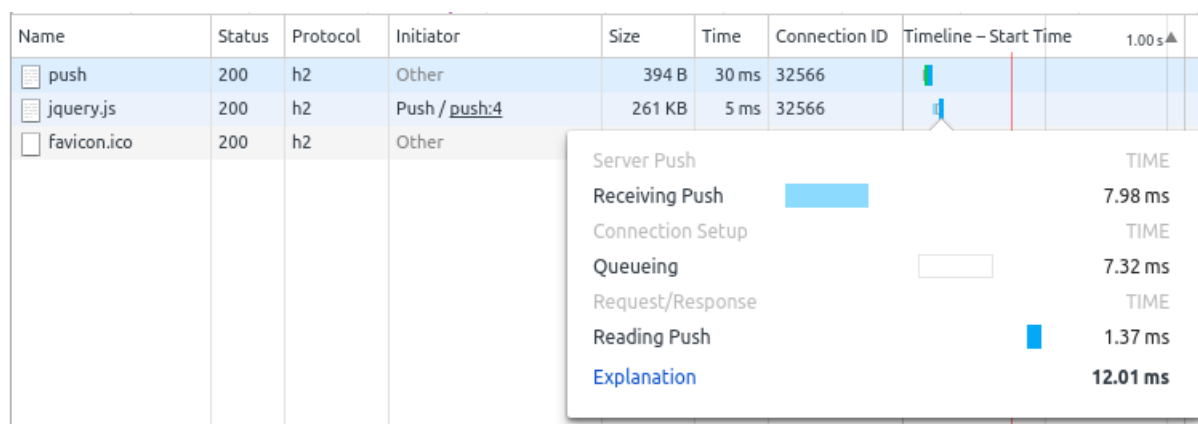
Na rysunku 3.1 widzimy przykład zapytania bez wykorzystania server push. Znajdują się tutaj typowe informacje zapytania HTTP:

1. sekcja **Connection Setup** dotycząca ustanowienia połączenia:
  - (a) **Queueing** – pakiet zostaje zakolejkowany, jeżeli jest zapytanie o wyższym priorytecie lub osiągnięto maksymalną liczbę połączeń TCP dla jednego klienta (możemy mieć otwartych jednocześnie 6 połączeń TCP),
  - (b) **Stalled** – czas na jaki pakiet utknął z któregoś z powodów wymienionych wyżej,
2. sekcja **Request/Response** dotycząca czasów przesyłania danych:
  - (a) **Request sent** – czas wysyłania zapytania,
  - (b) **Waiting (TTFB) – Time To First Byte** jest to czas oczekiwania na otrzymanie pierwszego bajtu odpowiedzi wysłanej przez serwer,
  - (c) **Content Download** – czas pobierania odpowiedzi przez przeglądarkę.

Natomiast na rysunku 3.2 nie ma informacji świadczących o tym, że zostało wysłane jakiegokolwiek zapytanie – **Request sent**, **Waiting (TTFB)** i **Content Download**. Z tego wynika, że klient nie wysłał prośby o dany plik, a mimo to został on wysłany. Dodatkowym potwierdzeniem, że korzystamy z server push jest informacja, że inicjatorem zapytania jest **Push/push:4** widoczna w kolumnie **Initiator**. Widać też, że czas zapytania to prawie 3 razy mniej (32.34 ms zamiast 12.01 ms) dla danych przesłanych z wykorzystaniem możliwości HTTP/2. Widać tutaj jak dużo czasu trwa oczekiwanie na zasoby, które mogłyby zostać wysłane od razu.



Rysunek 3.1: Szczegóły zapytania wysłanego bez wykorzystania Server Push



Rysunek 3.2: Szczegóły zapytania wysłanego z wykorzystaniem Server Push

# Rozdział 4

## Testy

### 4.1 Chrome DevTools

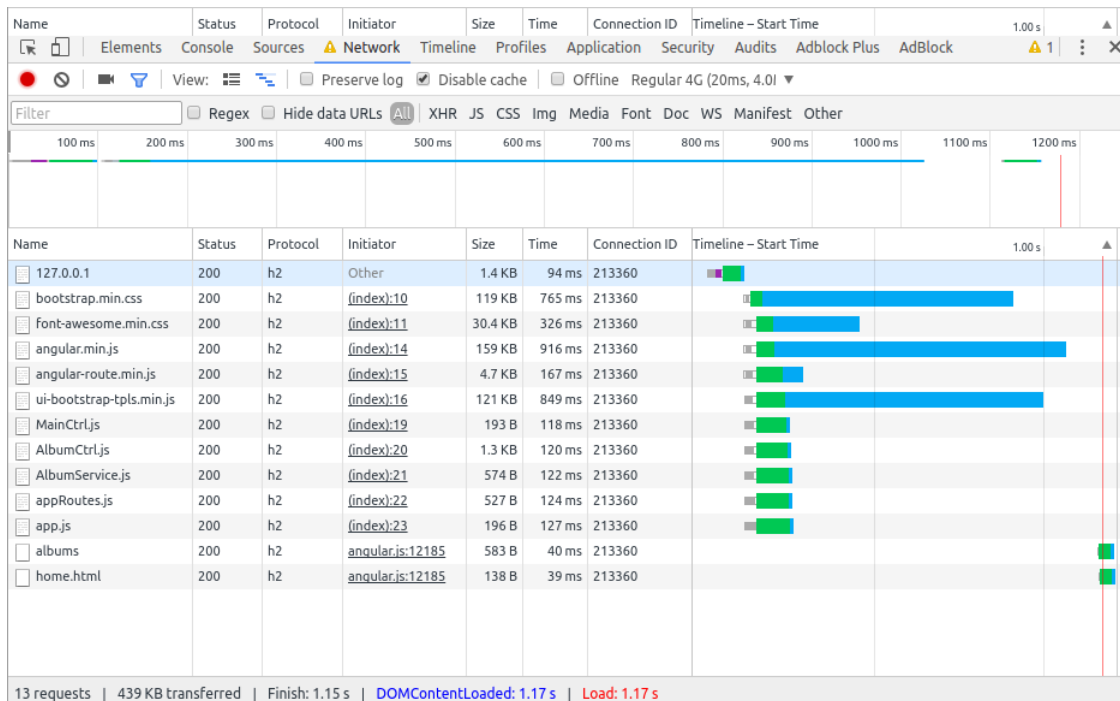
Do pomiaru prędkości oraz uzyskania innych ważnych informacji wykorzystałem narzędzie Chrome DevTools (więcej informacji o tym narzędziu [12]). Opiszę tutaj pokrótce co i jak mierzyłem za pomocą tego oprogramowania.

Po uruchomieniu konsoli przeglądarki przechodzimy do zakładki Network i naszym oczom ukazuje się okno jak na rysunku 4.1.

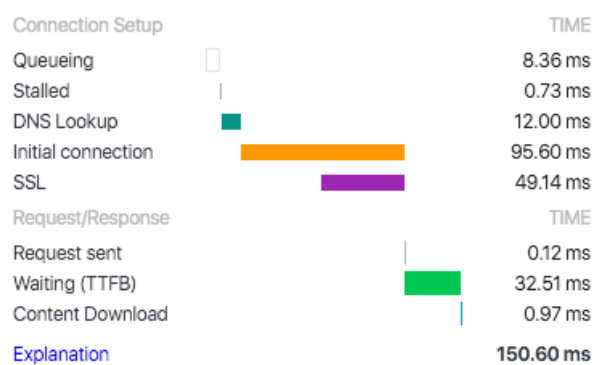
Widoczne okno składa się z pięciu głównych elementów:

1. paska kontroli – umożliwia on między innymi edycję wyglądu panelu sieciowego,
2. paska filtrów – pozwala na stworzenie reguł i wybór tylko tych pakietów, które nas interesują,
3. paska przeglądu – ukazuje nam oś czasu, która daje nam obraz tego, jak przesyłane były pakiety danych,
4. tabeli zapytań – zawiera szczegółowe informacje na temat każdego zapytania
5. podsumowania – zawiera informacje o łącznej liczbie zapytań, przesłanych danych oraz czasie trwania.

W moich badaniach najczęściej korzystałem z informacji zawartych w tabeli zapytań, a szczególnie z przedstawionej w niej osi czasu. Po najechaniu kursorem na którykolwiek pasek na osi otrzymujemy szczegółowe informacje o czasie każdego z etapów zapytania jak na rysunku 4.2.



Rysunek 4.1: Wygląd okna Chrome DevTools



Rysunek 4.2: Szczegółowe informacje na temat czasu zapytania

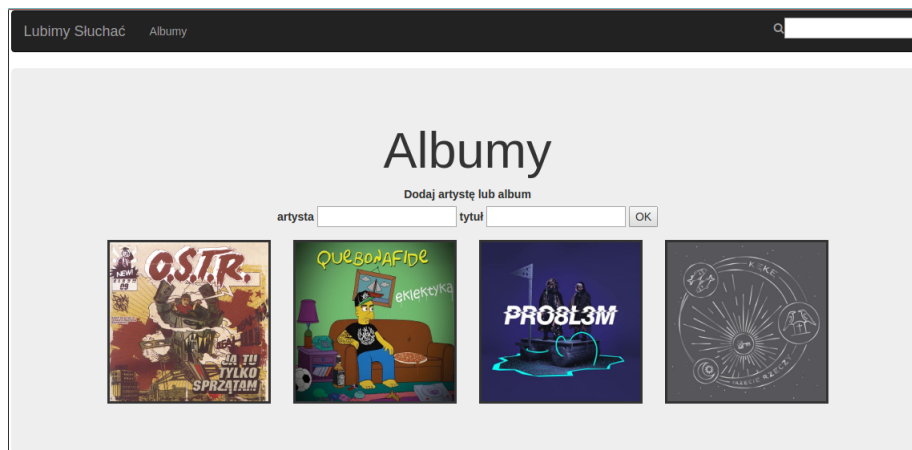
## 4.2 Środowisko testowe

Aplikacja, którą stworzyłem nosi nazwę *LubimySłuchać*. Jest to aplikacja, w której użytkownik może zarządzać swoją wirtualną kolekcją albumów muzycznych. W obecnym stadium aplikacja pozwala na:

1. przeglądanie listy albumów,
2. wyszukiwanie albumu po nazwie lub wykonawcy,
3. dodanie albumu,
4. edycję albumu,
5. usunięcie albumu.

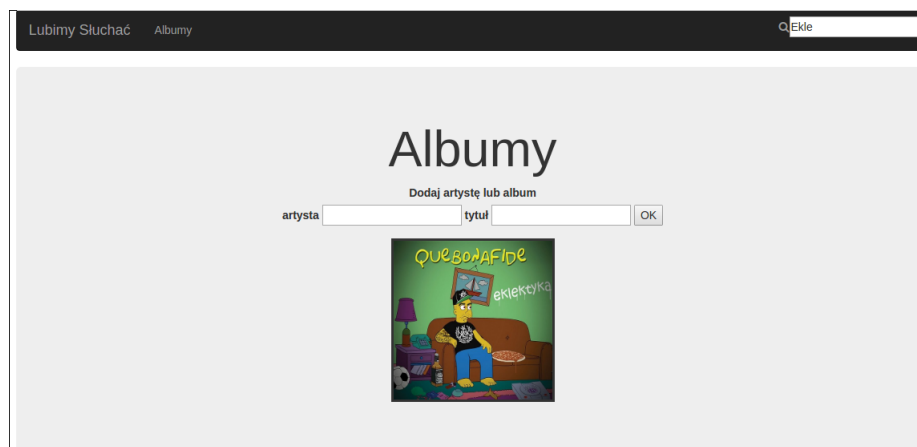
Są to podstawowe funkcje, dzięki którym mogłem przetestować protokół HTTP/2. W przyszłości planuję rozwój aplikacji.

Na rysunku 4.3 widzimy główny widok mojej aplikacji. Na górze znajduje się pasek nawigacyjny, po jego prawej stronie jest okno wyszukiwania albumów. W głównej części mamy podgląd wszystkich dodanych przez użytkownika albumów oraz formularz umożliwiający dodanie kolejnej pozycji do listy.



Rysunek 4.3: Główne okno aplikacji *LubimySłuchać*

Po wpisaniu w okno wyszukiwarki nazwy albumu lub wykonawcy lista jest odpowiednio filtrowana i wyświetlane są tylko pasujące albumy. Jak widać na rysunku 4.4 wystarczy wpisać fragment nazwy, a aplikacja dynamicznie filtruje listę albumów.



Rysunek 4.4: Widok listy albumów po wprowadzeniu fragmentu nazwy poszukiwanego albumu

Takie dynamiczne wyszukiwanie jest możliwe do realizacji w bardzo prosty sposób, dzięki funkcjom udostępnianym przez framework AngularJS. Na listingu 4.1 widzimy kod pola `input` odpowiedzialnego za filtrowanie albumów. Tekst wprowadzony w to pole jest automatycznie przypisywane zmiennej `searchAlbum`. Na listingu 4.2 widzimy, że zmienna ta jest wykorzystywana do filtrowania wszystkich albumów: `ng-repeat=album in albums | filter:searchText` – ten fragment jest odpowiedzialny za wyświetlanie kolejnych albumów pobranych z serwera, które spełniają kryteria wyszukiwania.

Listing 4.1: Kod filtra w pasku nawigacyjnym aplikacji

```
<ul class="nav navbar-form navbar-right">
  <li>
    <i class="fa fa-search"></i><input ng-model="searchAlbum"
      >
    </li>
</ul>
```

Listing 4.2: Kod odpowiedzialny za wyświetlanie albumów

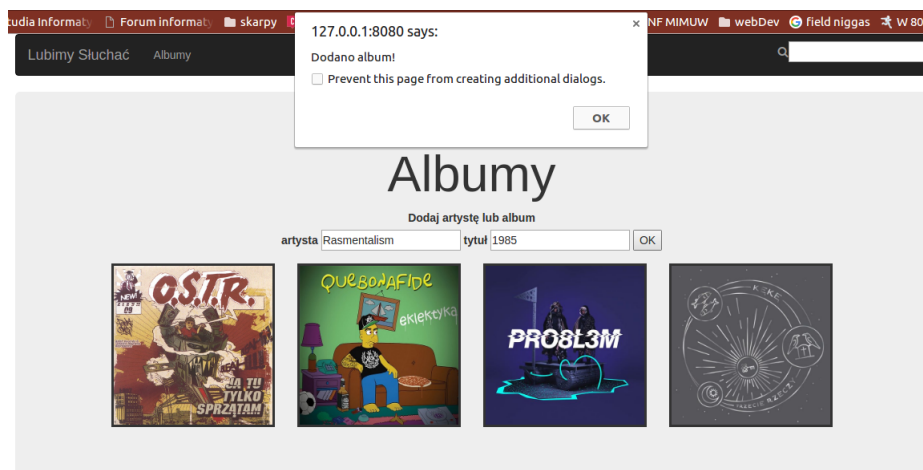
```
<div class="box2" ng-repeat="album in albums | filter:
  searchText"
  ng-mouseenter="active=true"
  ng-mouseleave="active=false"
  ng-click="editWindow=!editWindow">
  <div class="content">
    <p>{{album.artist}}</p>
    <p>{{album.title}}</p>
```

```

        <i class="fa fa-trash" aria-hidden="true" ng-show="
            active" ng-click="deleteAlbum(album._id)"></i>
        <i class="fa fa-pencil" aria-hidden="true" ng-show="
            active" ng-click="editAlbum(album)"></i>
    </div>
    
</div>

```

Użytkownik ma możliwość dodania, edycji oraz usunięcia albumu ze swojej listy. Aby dodać album wystarczy wprowadzić artystę oraz tytuł albumu i zostanie on automatycznie dodany. O sukcesie operacji zostaniemy poinformowani stosownym komunikatem widocznym na rysunku 4.5.



Rysunek 4.5: Widok aplikacji po wprowadzeniu danych nowego albumu i kliknięciu przycisku OK

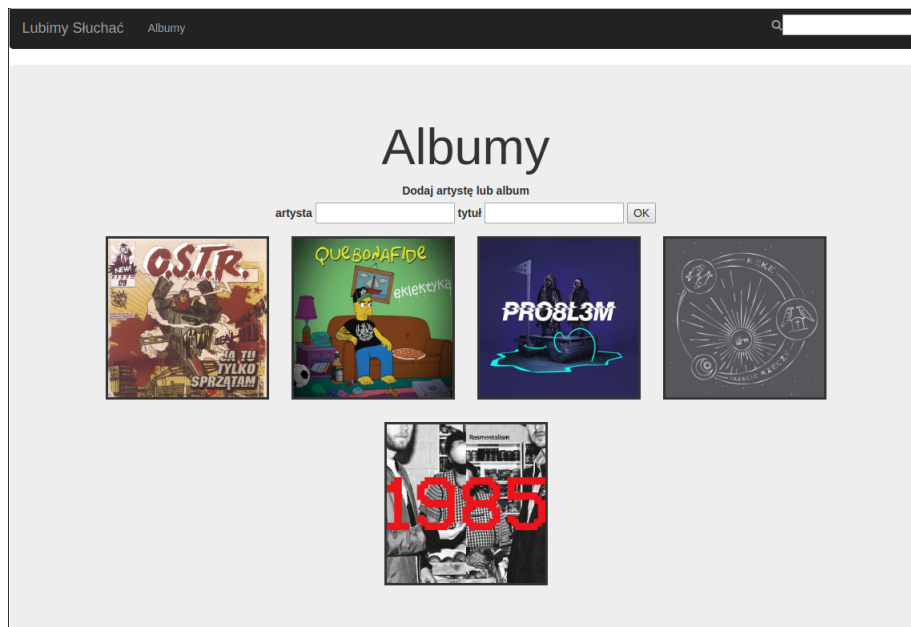
Jak widać na rysunku 4.6 album natychmiast pojawia się na liście.

## 4.3 Przygotowanie testów

Przed przejściem do porównania prędkości działania obu wersji protokołu chciałem zaprezentować pierwsze efekty implementacji protokołu HTTP/2, które przedstawia rysunek 4.7.

Widzimy tutaj dwie rzeczy, które powinny nas zainteresować. W sekcji Protocol oznaczonej na rysunku 4.7 numerem 1 widnieje napis h2 przy każdym zapytaniu. Jest to informacja, że do komunikacji z serwerem wykorzystana została najnowsza wersja protokołu HTTP. Dodatkowo w sekcji



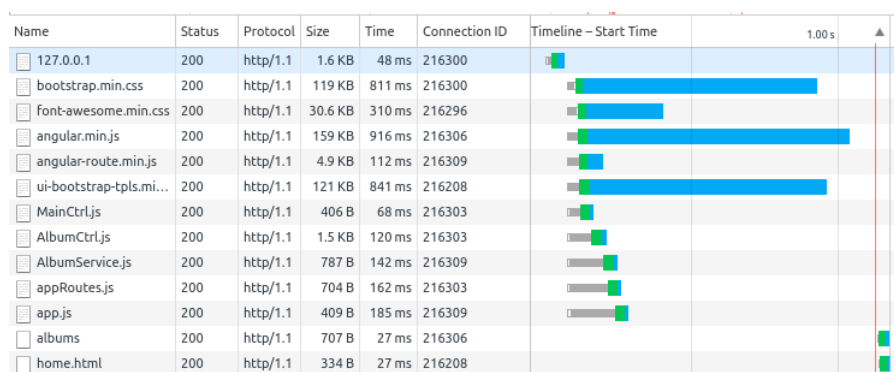


Rysunek 4.6: Widok aplikacji po udanym wprowadzeniu nazwy oraz wykonawcy albumu

Name	Status	Protocol	Size	Time	Connection ID	Timeline – Start Time	1.00 s
127.0.0.1	200	h2	1.4 KB	41 ms	214114		
bootstrap.min.css	200	h2	119 KB	794 ms	214114		
font-awesome.min...	200	h2	30.4 KB	282 ms	214114		
angular.min.js	200	h2	159 KB	924 ms	214114		
angular-route.min.js	200	h2	4.7 KB	189 ms	214114		
ui-bootstrap-tpls.m...	200	h2	121 KB	860 ms	214114		
MainCtrl.js	200	h2	229 B	146 ms	214114		
AlbumCtrl.js	200	h2	1.3 KB	145 ms	214114		
AlbumService.js	200	h2	574 B	105 ms	214114		
appRoutes.js	200	h2	491 B	104 ms	214114		
app.js	200	h2	196 B	105 ms	214114		
albums	200	h2	583 B	24 ms	214114		
home.html	200	h2	138 B	29 ms	214114		

Rysunek 4.7: Dowód działania protokołu HTTP/2

Connection ID (na rysunku 4.7 jest to numer 2) widzimy, że wszystkie zapytania zostały wykonane z wykorzystaniem tego samego połączenia TCP. Nie mogłoby to mieć miejsca, gdybyśmy wykorzystali HTTP/1.1, co pokazuje rysunek 4.8.



Rysunek 4.8: Przykładowe linia czasu dla zapytań wykonanych w HTTP/1.1

Na rysunku widzimy, że protokół z jakiego korzystamy to HTTP/1.1, a wszystkie zapytania, które są wykonywane równolegle przesyłane są w ramach różnych połączeń TCP. Świadczą o tym wartości w kolumnie 'Connection ID'.



Jeszcze jedną ciekawą rzeczą są wielkości nagłówków. Dzięki przejściu na kodowanie binarne nagłówki w protokole HTTP/2 są o wiele mniejsze (w moim przykładzie jest to 20 B dla HTTP/2 i 136 B dla HTTP/1.1) niż w tekstowym HTTP/1.1. W swojej pracy zaimplementowałem możliwość wysłania zapytania, które w odpowiedzi dostaje odpowiedź w postaci samego nagłówka. Dzięki temu można porównać wielkość nagłówków HTTP na konkretnym przykładzie przedstawionym na listingu 4.3. Na początku tworzymy nagłówek odpowiedzi ze statusem 200, który informuje, że zapytanie powiodło się. Następnie za pomocą `res.end('')` tworzymy pustą odpowiedź i odsyłamy wszystko do użytkownika.

Listing 4.3: Wygląd funkcji wykonywanej po wykonaniu zapytania GET na adres `/empty`



```
app.get('/empty', function(req, res) {
  res.writeHead(200);
  res.end('');
});
```

Wyniki przedstawiają rysunki 4.9 i 4.10.

Od razu widać, że nagłówki w najnowszej wersji protokołu uległy znacznej kompresji.

Name	Status	Protocol	Size	Time	Connection ID	Timeline – Start Time
 empty	200	h2	20 B	29 ms	216048	

Rysunek 4.9: Pusta odpowiedź w HTTP/2

Name	Status	Protocol	Size	Time	Connection ID	Timeline – Start Time
 empty	200	http/1.1	136 B	33 ms	216774	

Rysunek 4.10: Pusta odpowiedź w HTTP/1.1

## 4.4 Porównanie prędkości protokołu HTTP/2 i HTTP/1.1

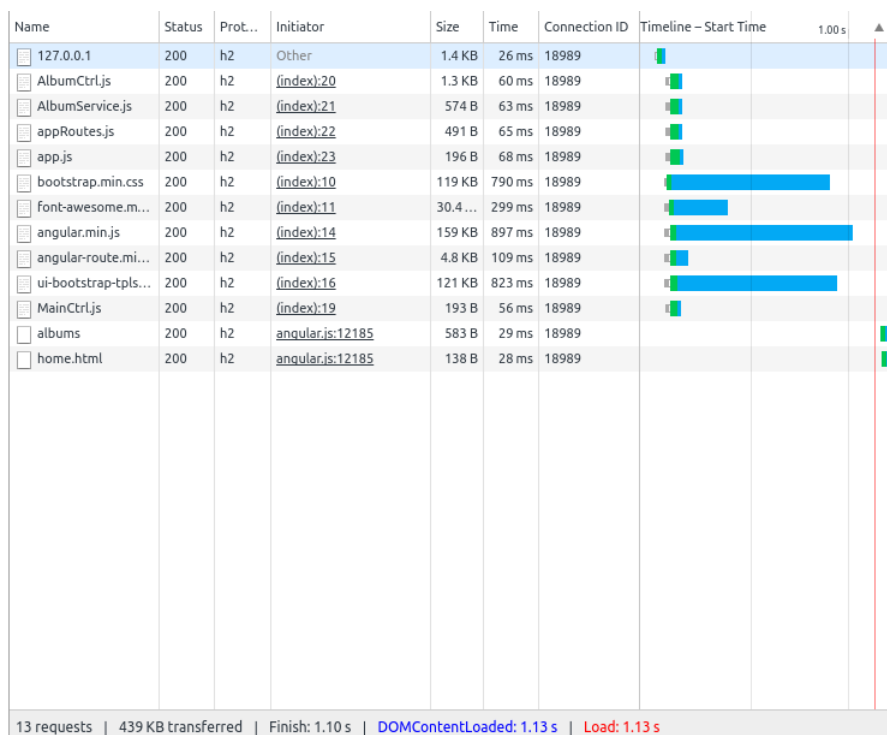
W ramach tego testu sprawdzałem prędkość ładowania wszystkich zasobów strony z wyłączonym CACHE przeglądarki oraz z ustawioną prędkością Regular 4G (20ms, 4.0 Mb/s, 3.0 Mb/s). Dane z wykorzystaniem protokołu HTTP/1.1 są przesyłane za pomocą niezabezpieczonego połączenia. HTTP/2 do utworzenia połączenia wymaga zabezpieczenia SSL. Taki test nie jest miarodajny, ponieważ czas nawiązania połączenia zabezpieczonego zawsze będzie dłuższy. Jest to związane ze sposobem nawiązywania bezpiecznego połączenia – przed rozpoczęciem przesyłania jakichkolwiek informacji serwer musi przesłać do użytkownika certyfikat SSL, który musi zostać zweryfikowany po stronie klienta. Następnie, jeśli certyfikat jest godny zaufania, klient przesyła odpowiednią informację do serwera. Sytuację, kiedy korzystamy z zabezpieczonego połączenia HTTP/1.1 przedstawię w następnej sekcji.

Próba	HTTP/1.1	HTTP/2
1	1.23s	1.26s
2	1.49s	1.12s
3	1.28s	1.39s
4	1.20s	1.09s
5	1.22s	1.09s
6	1.25s	1.26s
7	1.21s	1.12s
8	1.25s	1.11s
9	1.21s	1.16s
10	1.26s	1.10s
ŚREDNIA	1.26s	1.17s

Tabela ukazuje wyniki wykonanych pomiarów. Jak widać protokół HTTP/2 przyspiesza czas ładowania strony o około 15%. W dużej mierze jest to czas

zaoszczędzony na nawiązywanie nowych połączeń TCP. Sporą oszczędność utrzymujemy też dzięki zmniejszeniu objętości nagłówków odpowiedzi.

Na rysunkach 4.11 oraz 4.12 widoczny jest przykładowy wynik testu. W sekcji podsumowania widoczna jest całkowita objętość przesłanych danych. Są to odpowiednio 443 KB i 439 KB dla protokołu HTTP/1.1 i HTTP/2. Możemy też porównać objętości oraz czasy przesyłania poszczególnych plików.



Rysunek 4.11: Czas ładowania strony dla HTTP/2

Wykres przedstawiony na rysunku 4.13 ukazuje porównanie wyników w każdej z prób oraz wynik średni dla obu protokołów.

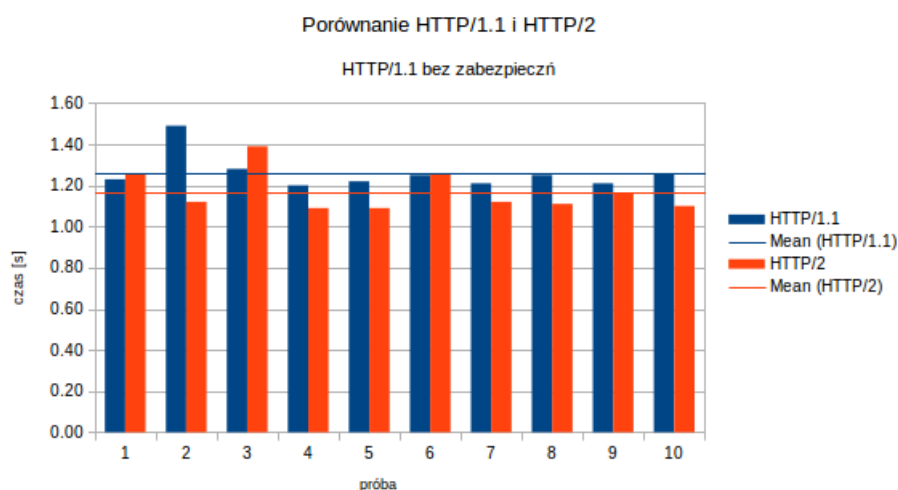
## 4.5 Porównanie prędkości przy bezpiecznym połączeniu HTTP/1.1

W tym teście dane przesyłane za pomocą protokołu HTTP/1.1 są zabezpieczone SSL.

Name	Status	Protocol	Initiator	Size	Time	Connection ID	Timeline – Start Time	
127.0.0.1	200	http/1.1	Other	1.6 KB	34 ms	21718		
bootstrap.min.css	200	http/1.1	(index):10	119 ...	819 ms	21718		
font-awesome.m...	200	http/1.1	(index):11	30.6...	304 ms	21755		
angular.min.js	200	http/1.1	(index):14	159 ...	910 ms	21764		
angular-route.mi...	200	http/1.1	(index):15	4.9 KB	95 ms	21760		
ui-bootstrap-tpls...	200	http/1.1	(index):16	121 ...	834 ms	21768		
MainCtrl.js	200	http/1.1	(index):19	406 B	54 ms	21751		
AlbumCtrl.js	200	http/1.1	(index):20	1.5 KB	108 ms	21751		
AlbumService.js	200	http/1.1	(index):21	787 B	125 ms	21760		
appRoutes.js	200	http/1.1	(index):22	704 B	140 ms	21751		
app.js	200	http/1.1	(index):23	409 B	168 ms	21760		
albums	200	http/1.1	angular.js:12185	707 B	28 ms	21764		
home.html	200	http/1.1	angular.js:12185	334 B	36 ms	21768		
favicon.ico	200	http/1.1	Other	1.6 KB	43 ms	21768		

14 requests | 443 KB transferred | Finish: 1.26 s | DOMContentLoaded: 1.17 s | Load: 1.16 s

Rysunek 4.12: Czas ładowania strony dla HTTP/1.1 bez SSL

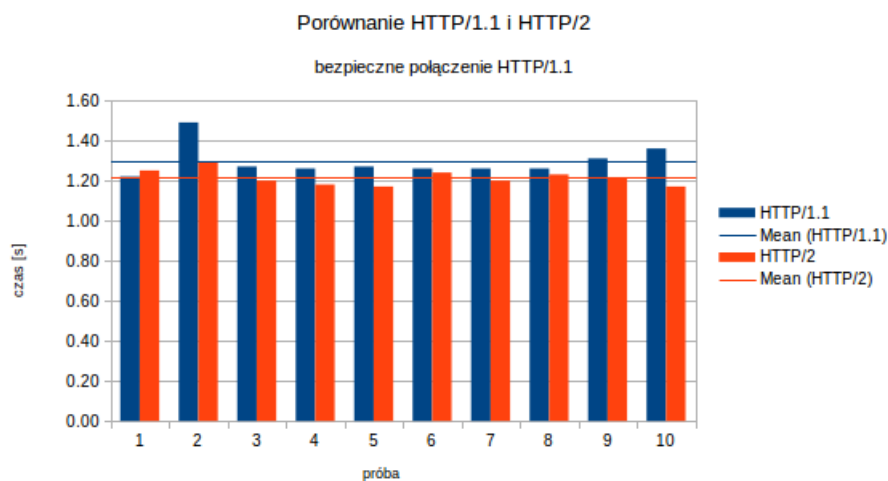


Rysunek 4.13: Wykres podsumowujący

Próba	HTTP/1.1	HTTP/2
1	1.22s	1.25s
2	1.49s	1.29s
3	1.27s	1.20s
4	1.26s	1.18s
5	1.27s	1.17s
6	1.26s	1.24s
7	1.26s	1.20s
8	1.26s	1.23s
9	1.31s	1.21s
10	1.36s	1.17s
ŚREDNIA	1.29s	1.21s

W powyższym teście protokół HTTP/2 uzyskał wyniki lepsze średnio o 6%. Na podstawie tego testu widzimy też, że w przypadku wykorzystania zabezpieczonego połączenia czasy dla HTTP/1.1 są minimalnie większe, niż bez SSL.

Wykres przedstawiony na schemacie 4.14 ukazuje porównanie wyników w każdej z prób oraz wynik średni dla obu protokołów.



Rysunek 4.14: Porównanie czasów pobierania zasobów w kolejnych próbach

## 4.6 Porównanie prędkości z włączonym CACHE

Próba	HTTP/1.1	HTTP/2
1	709ms	534ms
2	555ms	483ms
3	622ms	527ms
4	625ms	511ms
5	550ms	518ms
6	546ms	555ms
7	603ms	557ms
8	517ms	496ms
9	584ms	531ms
10	521ms	532ms
ŚREDNIA	583ms	524ms

Kolejny test pokazuje, że średnio czas ładowania strony z wykorzystaniem HTTP/2 jest około 11% krótszy. Jak można było się spodziewać czasy z wykorzystaniem pamięci podręcznej przeglądarki będą znacznie mniejsze, niż gdy ta pamięć jest wyłączona. Pokazuje to jak ważnym elementem pracy twórcy aplikacji internetowych jest rozsądne zarządzanie zasobami, które mogą być przechowywane w pamięci podręcznej po stronie klienta i nie muszą one być za każdym razem wysyłane ponownie.

Obrazują to poniższe rysunki. Na rysunku 4.16 widzimy, co się dzieje, gdy pierwszy raz wchodzimy na daną stronę lub mamy wyłączoną pamięć podręczną przeglądarki. Czas pobierania niektórych zasobów (niebieska część paska) jest elementem, który zabiera najwięcej czasu. Natomiast, gdy spojrzymy na rysunek 4.15, to praktycznie nie widzimy niebieskiej części paska. Taka sytuacja ma miejsce, gdy wchodzimy na stronę kolejny raz. Na rysunku 4.15 w sekcji Status widzimy status 304. Takim statusem opatrzona jest odpowiedź serwera w wypadku, gdy od czasu ostatniego razu, gdy pobieraliśmy dany zasób nie zmienił się on i nie ma potrzeby jego ponownego wysyłania.

Wykres przedstawiony na schemacie 4.17 ukazuje porównanie wyników w każdej z prób oraz wynik średni dla obu protokołów.

## 4.7 Porównanie prędkości przy wykorzystaniu Server Push

Ostatni test chciałem przeprowadzić dla zapytania wykorzystującego wprowadzony przez HTTP/2 Server Push, który, z punktu widzenia programisty aplikacji internetowych, wydaje się być najciekawszą nowością wprowadzoną do protokołu HTTP.

Name	Status	Protocol	Initiator	Size	Time	Connection ID	Timeline – Start	Time
127.0.0.1	200	h2	Other	1.3 KB	25 ms	80921		
bootstrap.min.css	304	h2	(index):10	64 B	45 ms	80921		
Font-awesome.m...	304	h2	(index):11	64 B	45 ms	80921		
angular.min.js	304	h2	(index):14	65 B	44 ms	80921		
angular-route.mi...	304	h2	(index):15	64 B	51 ms	80921		
ui-bootstrap-tpls...	304	h2	(index):16	64 B	51 ms	80921		
MainCtrl.js	304	h2	(index):19	63 B	52 ms	80921		
AlbumCtrl.js	304	h2	(index):20	63 B	63 ms	80921		
AlbumService.js	304	h2	(index):21	63 B	66 ms	80921		
appRoutes.js	304	h2	(index):22	63 B	68 ms	80921		
app.js	304	h2	(index):23	99 B	71 ms	80921		
albums	304	h2	angular.js:12185	47 B	36 ms	80921		
home.html	304	h2	angular.js:12185	62 B	33 ms	80921		
favicon.ico	200	h2	Other	1.4 KB	35 ms	80921		
14 requests   3.5 KB transferred   Finish: 571 ms   DOMContentLoaded: 349 ms   Load: 348 ms								

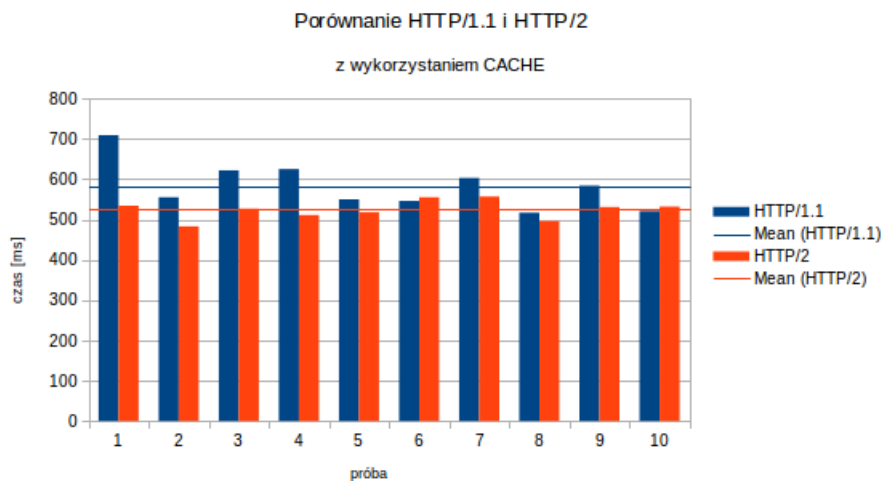
Rysunek 4.15: Przykład zapytania z wykorzystaniem kieszeniowania



Name	Status	Protocol	Initiator	Size	Time	Connection ID	Timeline - Start Time	
127.0.0.1	200	h2	Other	1.3 KB	27 ms	80566		
ui-bootstrap-tpls...	200	h2	(index):16	121 KB	819 ms	80566		
AlbumCtrl.js	200	h2	(index):20	1.3 KB	111 ms	80566		
AlbumService.js	200	h2	(index):21	574 B	111 ms	80566		
appRoutes.js	200	h2	(index):22	527 B	112 ms	80566		
app.js	200	h2	(index):23	196 B	112 ms	80566		
bootstrap.min.css	200	h2	(index):10	119 KB	712 ms	80566		
font-awesome.m...	200	h2	(index):11	30.4 KB	334 ms	80566		
angular.min.js	200	h2	(index):14	159 KB	895 ms	80566		
angular-route.mi...	200	h2	(index):15	4.7 KB	146 ms	80566		
MainCtrl.js	200	h2	(index):19	193 B	88 ms	80566		
albums	200	h2	angular.js:12185	583 B	27 ms	80566		
home.html	200	h2	angular.js:12185	138 B	31 ms	80566		
favicon.ico	200	h2	Other	1.4 KB	28 ms	80566		

14 requests | 440 KB transferred | Finish: 1.17 s | DOMContentLoaded: 1.08 s | Load: 1.08 s







Rysunek 4.16: Przykład zapytania bez wykorzystania kieszeniowania



Rysunek 4.17: Czasy pobierania zasobów z wykorzystaniem kieszeniowania w kolejnych próbach

Próba	HTTP/1.1	HTTP/2	HTTP/2 PUSH
1	444ms	454ms	387ms
2	456ms	441ms	372ms
3	386ms	515ms	351ms
4	380ms	436ms	471ms
5	407ms	342ms	415ms
6	404ms	438ms	420ms
7	481ms	385ms	470ms
8	471ms	450ms	253ms
9	567ms	481ms	269ms
10	418ms	413ms	409ms
ŚREDNIA	441ms	436ms	382ms







Poniższe rysunki (4.18, 4.19 i 4.20) ukazują porównanie przykładowych zapytań z powyższego testu.

Name	Sta...	Protocol	Initiator	Size	Time	Connecti...	Timeline – Start Time
 push	200	http/1.1	Other	618 B	37 ms	154410	
 jquery.js	200	http/1.1	push<4	261 KB	9 ms	154410	
 favicon.ico	200	http/1.1	Other	1.6 KB	12 ms	154410	







3 requests | 263 KB transferred | Finish: 418 ms | DOMContentLoaded: 217 ms | Load: 216 ms

Rysunek 4.18: Przykład zapytania dla HTTP/1.1

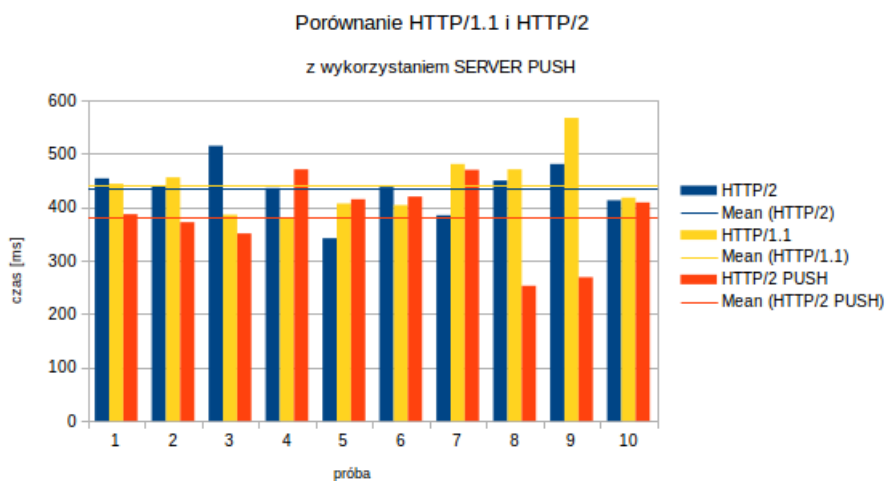
Wykres przedstawiony na schemacie 4.21 ukazuje porównanie wyników w każdej z prób oraz wynik średni dla wszystkich przypadków.

Name	Status	Protocol	Initiator	Size	Time	Connection ID	Timeline – Start Time
 push	200	h2	Other	403 B	21 ms	33498	
 jquery.js	200	h2	push:4	261 KB	43 ms	33498	
 favicon.ico	200	h2	Other	1.4 KB	10 ms	33498	
3 requests   263 KB transferred   Finish: 459 ms   DOMContentLoaded: 267 ms   Load: 263 ms							

Rysunek 4.19: Przykład zapytania bez wykorzystania Server Push

Name	Status	Protocol	Initiator	Size	Time	Connection ID	Timeline – Start Time
 push	200	h2	Other	435 B	105 ms	31756	
 jquery.js	200	h2	Push / <u>push</u> :4	261 KB	15 ms	31756	
 favicon.ico	200	h2	Other	1.4 KB	18 ms	31756	
3 requests   263 KB transferred   Finish: 435 ms   DOMContentLoaded: 326 ms   Load: 319 ms							

Rysunek 4.20: Przykład zapytania wykorzystującego Server Push



Rysunek 4.21: Wykres podsumowujący

## 4.8 Kompatybilność popularnych przeglądarek internetowych z HTTP/2

Ważną cechą poza szybkością aplikacji jest również jej niezawodność oraz uniwersalność. Aby aplikacja była uniwersalna użytkownik musi mieć możliwość jej uruchomienia w różnych środowiskach. Dlatego przeprowadziłem testy wsparcia HTTP/2 przez popularne przeglądarki. Testy przegląderek Firefox, Chrome oraz Opera przeprowadziłem na Ubuntu 16.04. Mobilną wersję przeglądarki Safari przetestowałem na telefonie iPhone 6 z wersją oprogramowania 10.2. Testy Edge oraz Internet Explorer zostały przeprowadzone na maszynie wirtualnej z zainstalowanym Windows 8. Wyniki moich badań przedstawiam w poniższej tabeli.

Lp	Firefox	Chrome	Opera	Safari	Internet Explorer	Edge
1.	50	55	42			14
2.	49	54	41	10.2	11	13
3.	48	53	40			12

Jak widać protokół HTTP/2 jest prawie w pełni wspierany przez najpopularniejsze przeglądarki internetowe w najnowszych wersjach. Daje nam to pewność, że aplikacja stworzona z wykorzystaniem tej technologii będzie sprawnie działać u większości użytkowników. Jak widać tylko najnowsza wersja przeglądarki Internet Explorer nie wspiera HTTP/2. Należy zaznaczyć, że ostatnia aktualizacja IE miała miejsce pod koniec roku 2015, więc nie jest to już wspierany produkt i większość użytkowników odchodzi od niego.

# Rozdział 5

## Wnioski

W obecnych czasach, gdy technologia tak szybko porusza się do przodu, korzystanie z rozwiązań, które nie zmieniły się od ponad 15 lat wydaje się niewłaściwe. HTTP/1.1 coraz mocniej ogranicza możliwości programistów i nie pozwala rozsądnie i w pełni wykorzystać zasobów, którymi dzisiaj dysponujemy. Po przeczytaniu, że numer dwa przy wersji protokołu, to właściwie tylko informacja o tym, że nie jest on wstecznie kompatybilny zacząłem się zastanawiać czy nie jest to swego rodzaju uspokojenie pokładanych w tym protokole oczekiwań. Postanowiłem więc przeprowadzić testy, których wynikiem jest niniejsza praca.

Patrząc na wyniki testów widać przyśpieszenie, względem starszej wersji protokołu. Pokazuje to, że wprowadzone udoskonalenia mogą w przyszłości przyczynić się do znacznego przyspieszenia aplikacji internetowych.

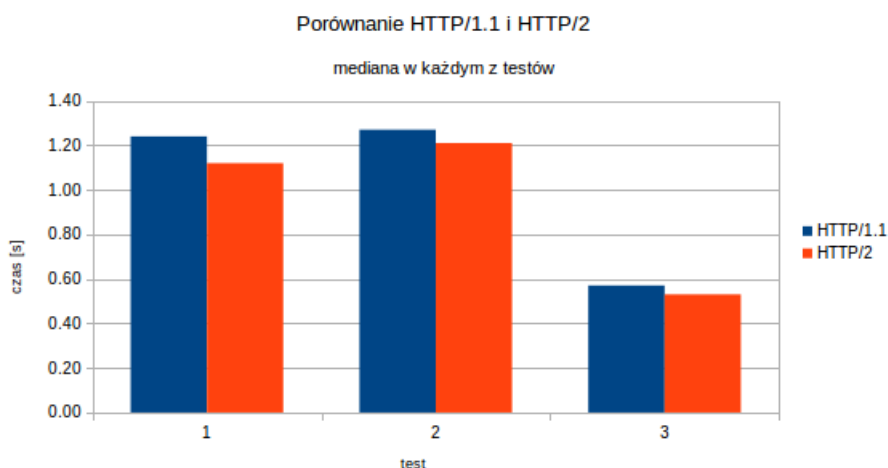
Dodatkowo protokół HTTP/2 to nie tylko szybkość, ale też jakość połączenia i jego bezpieczeństwo. Bardzo dobrym krokiem jest wymuszanie na twórcach korzystanie z certyfikatów bezpieczeństwa. Na pewno sprawi to, że sieć będzie się stawać coraz bezpieczniejszym miejscem i o wiele trudniej będzie przestępcy wykorzystać zwykłego użytkownika.

Wykorzystanie multiplexingu rozwiązało ostatecznie problem HOL blocking, który w niektórych przypadkach potrafił bardzo mocno spowolnić działanie aplikacji. Oczywiście były sposoby, żeby sobie z tym poradzić. Przykładem jest wykorzystywanie wielu połączeń TCP do przesyłania informacji równolegle. Takie rozwiązanie jednak znacznie bardziej obciąża serwer niepotrzebnymi prośbami o połączenie. Multiplexing pozwala na przesyłanie wielu informacji w ramach jednego połączenia, a dodatkowo kolejność przesyłanych informacji nie ma znaczenia.

Kolejny element, czyli kompresja nagłówków. Może nie jest on tak znaczący dla zasobów o bardzo dużej objętości. Możemy dojść do takiego wniosku na pierwszy rzut oka, jednak podsumowując powtarzającą się część na-

główek, widać, że ma to niemałe przełożenie na ilość przesyłanych danych.

Na koniec chciałem przedstawić wykresy przedstawiające mediane wyników pierwszych trzech testów (rysunek 5.1) oraz oddzielnie dla ostatniego testu dotyczącego SERVER PUSH (rysunek 5.2). Należy pamiętać, że ostatni test został przeprowadzony na innym zestawie pobieranych danych. Co ciekawe, przeciętna wartość dla testu HTTP/2 bez SERVER PUSH jest niższa, niż z wykorzystaniem tej technologii.



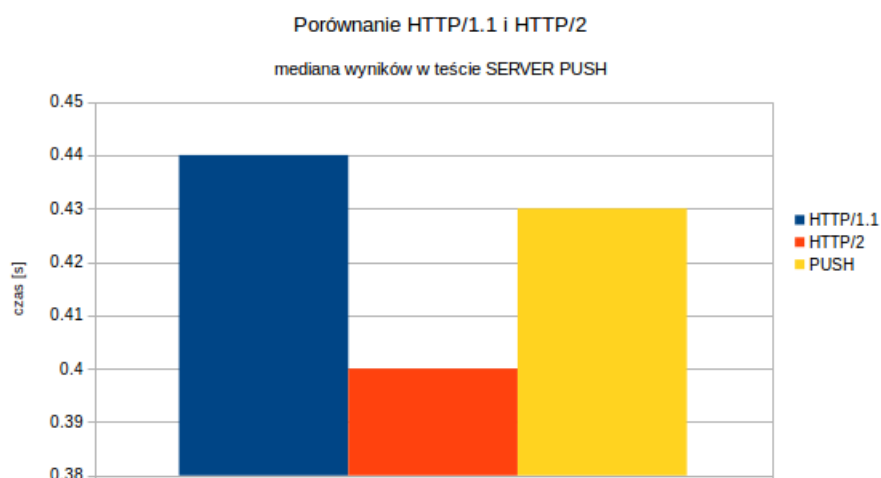
Rysunek 5.1: Mediany wyników trzech pierwszych testów

Jak widać, chociaż HTTP/2 jest stosunkowo nową technologią, to jego potencjał w tworzeniu wydajnych aplikacji internetowych jest ogromny. Wyniki moich badań napawają pozytywnie na przyszłość. Warto obserwować dalszy rozwój bibliotek i technologii wykorzystujących ten protokół.

W przyszłości chciałbym przede wszystkim dużo dokładniej przetestować możliwości Server Push oraz kompresji nagłówków. Można to osiągnąć poprzez znaczne rozbudowanie aplikacji tak, aby lepiej przypominała ona dzisiejsze aplikacje, które są bardzo rozbudowane pod względem zasobów.

Ciekawym badaniem mogłoby być również prześledzenie czasu, który jest wykorzystywany tylko i wyłącznie na nawiązanie połączenia TCP. Jak wspominałem protokół HTTP/1.1 nawiązuje kilka połączeń w ramach komunikacji klienta z serwerem. Chciałbym się przekonać, czy faktycznie zmiany wprowadzone przez HTTP/2 są na tym poziomie bardzo widoczne. Niestety ograniczone zasoby czasowe nie pozwoliły mi zająć się tym problemem i jest to mój cel na najbliższą przyszłość.

Podsumowując, przeprowadzone testy wskazują, że HTTP/2 ma bardzo



Rysunek 5.2: Mediany wyników testu SERVER PUSH

duży potencjał na przyszłość. Z czasem będą się pojawiać biblioteki, które dużo lepiej zaczną wykorzystywać jej możliwości. Na tę chwilę ciekawym pomysłem może być śledzenie prac, które są wykonywane, aby oficjalnie wdrożyć ten protokół w node.js (repozytorium: [13]).



# Bibliografia

- [1] M. Belshe, R. Peon, M. Thomson, „Hypertext Transfer Protocol Version 2 (HTTP/2)“, 2015.  
<https://tools.ietf.org/html/rfc7540>
- [2] założenia webpack  
<https://webpack.js.org/concepts/>
- [3] podstawy HTTP/2  
<https://developers.google.com/web/fundamentals/performance/http2/>
- [4] Haviv A., „MEAN Web Development”, Birmingham 2014, ISBN 978-1-78398-328-5
- [5] Totty B., Sayer M., Reddy S., Aggarwal A., Gourley D., „HTTP: The Definitive Guide“, 2002, rozdział 4
- [6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, „Hypertext Transfer Protocol – HTTP/1.1“, 1999.  
<https://tools.ietf.org/html/rfc2616>
- [7] I. Grigorik, „High Performance Browser Networking“, 2013  
rozdział: <https://hpbrowser.com/building-blocks-of-tcp/#head-of-line-blocking>
- [8] R. Rendle, „Spriting with <img>“, 2015.  
<https://css-tricks.com/spriting-img/>
- [9] I. Grigorik, „High Performance Browser Networking“, 2013  
rozdział: <https://hpbrowser.com/http1x/#domain-sharding>
- [10] R. Peon, H. Ruellan, „HPACK: Header Compression for HTTP/2“, 2015.  
<https://tools.ietf.org/html/rfc7541>
- [11] <https://github.com/indutny/node-spdy>

[12] <https://developer.chrome.com/devtools>

[13] <https://github.com/nodejs/http2>