



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 z krótkiego wstępu jasno i wyczerpująco opisującego oraz uzasadniającego cel pracy, trzech rozdziałów (2-4) zawierających opis istniejących podobnych rozwiązań, komponentów rozpatrywanych jako kandydaci do tworzonego systemu i wreszcie zagadnień wydajności wirtualnych rozwiązań. Piąty rozdział to opis środowiska obejmujący opis konfiguracji środowiska oraz przykładowe ćwiczenia laboratoryjne. Ostatni rozdział pracy to opis możliwości dalszego rozwoju projektu.

Słowa kluczowe: praca dyplomowa, LaTeX, jakość

THESIS TITLE

Abstract

This thesis presents a novel way of using a novel algorithm to solve complex problems of filter design. In the first chapter the fundamentals of filter design are presented. The second chapter describes an original algorithm invented by the authors. It is based on evolution strategy, but uses an original method of filter description similar to artificial neural network. In the third chapter the implementation of the algorithm in C programming language is presented. The fifth chapter contains results of tests which prove high efficiency and enormous accuracy of the program. Finally some possibilities of further development of the invented algorithms are proposed.

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

1	Wstęp	1
2	HTTP/2	3
2.1	Historia	3
2.2	Protokół binarny	3
2.3	Multiplexing	4
2.4	Prioretyzacja	5
2.5	Server push	5
2.6	Kompresja nagłówków	6
3	Budowa aplikacji	8
3.1	node-spdy – konfiguracja serwera	8
3.2	Server Push	9
4	Testy	13
4.1	Chrome DevTools	13
4.2	Przygotowanie testów	15
4.3	Porównanie prędkości protokołu HTTP/2 i HTTP/1.1	17
4.4	Porównanie prędkości przy bezpiecznym połączeniu HTTP/1.1	17
4.5	Porównanie prędkości dla HTTP/1.1 z włączonym CACHE	20
4.6	Porównanie prędkości przy wykorzystaniu Server Push	21
5	Wnioski	22
	Bibliografia	23

Rozdział 1

Wstęp

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

1. pamięć cache przeglądarki, dzięki której nie musimy przesyłać 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. 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 plików – 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 styli 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 ma wprowadzić protokół HTTP/2 faktycznie pozwalały zapomnieć o wspomnianych środkach, to życie wielu developerów stałoby się dużo łatwiejsze. Dzięki temu mogliby się oni ten czas poświęcić na rozwój aplikacji.

Za pomocą własnoręcznie stworzonej aplikacji chcę przekonać się, czy wprowadzone funkcje faktycznie mają tak ogromny wpływ na szybkość komunikacji pomiędzy klientem i serwerem.

Swoją aplikację stworzyłem wykorzystując zestaw oprogramowania MEAN – 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 uruchumieniowe 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,
- dobra znajomość języka JavaScript oraz jednoczesna chęć rozwoju umiejętności tworzenia aplikacji w tym języku,
- chęć poszerzenia wiedzy dotyczącej budowania aplikacji internetowych za pomocą technologii javascriptowych,
- nie ukrywam, że znaczący wpływ na moją decyzję miała również popularność języka JavaScript 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. 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

HTTP/2

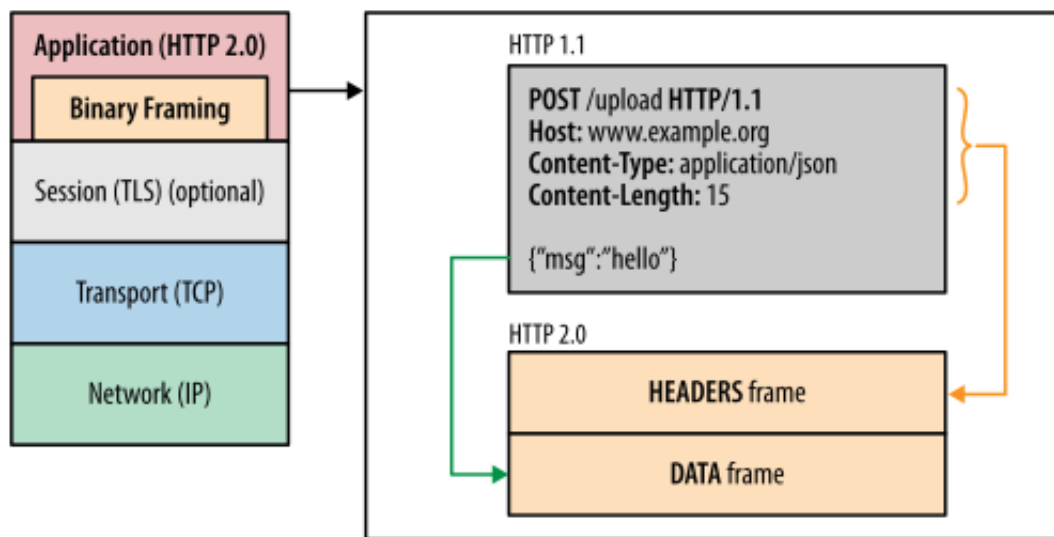
2.1 Historia

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 (**TODO**odnośnik RFC 7540 i może 7541), projekty były rozwijane równolegle. SPDY było wykorzystywane do testów nowych funkcjonalności, 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 może zostać utworzonych wiele dwukierunkowych strumieni danych przesy-



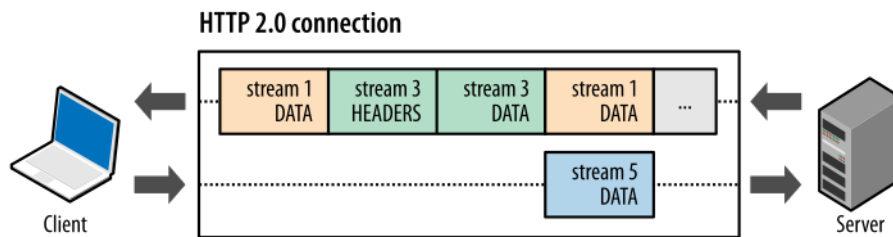
Rysunek 2.1: Schemat ramki protokołu HTTP/2

łających wiadomości HTTP. Taka wiadomość to w rzeczywistości zapytanie od klienta lub odpowiedź serwera składające się z ramek. Każda ramka natomiast musi przynajmniej posiadać 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 HOL (head-of-line blocking ODNOSNIK DO JAKIEGOŚ ŹRÓDŁA?). Aby wykonywać zapytania równolegle należało utworzyć kilka zapytań TCP, co obciąża serwer oraz jest 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 zabiegów jak:

- scalanie plików (WEBPACK),



Rysunek 2.2: Schemat wykorzystania multiplexingu w HTTP/2

- wykorzystywanie spritów,
- domain sharding (DOCZYTAC).

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

2.4 Prioretyzacja

2.5 Server push

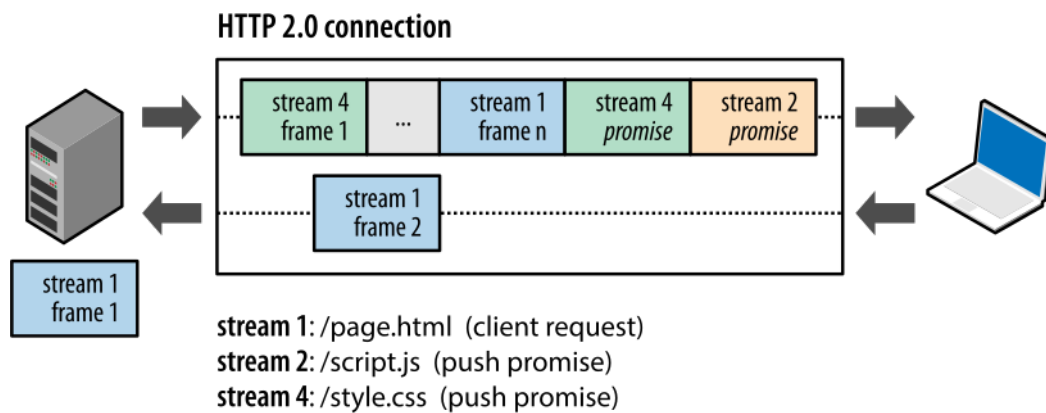
Wykorzystując protokół HTTP/1.1 nie mamy możliwości otrzymania zasobu, o który nie poprosiliśmy wysyłając zapytanie. Powoduje to opóźnienia na przykład podczas ładowania strony internetowej. Zanim otrzymamy skrypty czy arkusze styli, które wykorzystuje nasza strona musi ona o nie poprosić. Zapytanie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

```

1 <!-- CSS -->
2 <link rel="stylesheet"
3       href="libs/bootstrap/dist/css/bootstrap.min.css">
4 <link rel="stylesheet"
5       href="libs/font-awesome/css/font-awesome.min.css">
6
7 <!-- JS -->
8 <script src="libs/angular/angular.min.js"></script>

```

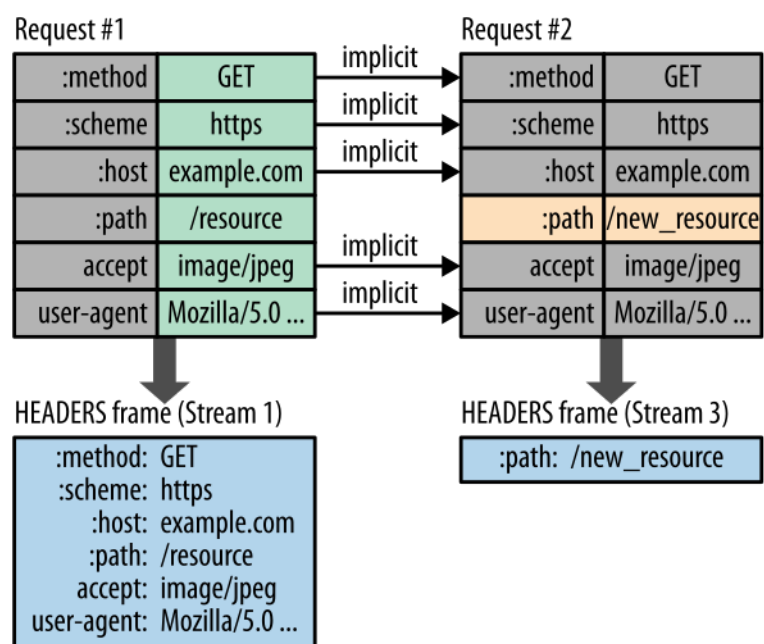
Takie rozwiązanie, chociaż w wielu przypadkach jest pożądane, tutaj jedynie spowalnia działanie aplikacji. Jeżeli mamy pewność, że użytkownik będzie potrzebował danych zasobów 2.3 możemy mu je od razu udostępnić, co zdecydowanie skraca czas ładowania aplikacji i dzięki temu unikam niechcianego efektu,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



Rysunek 2.3: Schemat Server push HTTP/2

2.6 Kompresja nagłówków

Kolejną ważną, chociaż pozornie niezauważalną zmianą jest kompresja nagłówków. Pomimo, że problem może wydawać się marginalny, to te kilka bajtów w każdym zapytaniu potrafi dość mocno wydłużyć czas komunikacji z serwerem. HTTP/2 wykorzystuje format kompresji zwany HPACK, 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).



Rysunek 2.4: Kompresja nagłówków w HTTP/2

Rozdział 3

Budowa aplikacji

3.1 node-spdy – konfiguracja serwera

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

1. plain – jeśli opcja jest włączona, to serwer wykorzystuje tekstową wersję protokołu HTTP,
2. ssl – ustawienia zabezpieczeń połączenia,
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 funkcjonalności.

Na koniec musimy sprawić, 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.

```
1 var spdy          = require ( 'spdy ' );
2 var port          = 8080;
3 var fs             = require ( 'fs ' );
```

```

4 var app                                = express();
5
6     ...
7     ...
8
9 const options = {
10   spdy: {
11     plain: true,
12     ssl: false,
13     protocols: ['h2', 'http/1.1'],
14   },
15   key: fs.readFileSync(__dirname + '/server.key'),
16   cert: fs.readFileSync(__dirname + '/server.crt')
17 };
18
19 spdy
20   .createServer(options, app)
21   .listen(port, (error) => {
22     if (error) {
23       console.error(error);
24       return process.exit(1);
25     } else {
26       console.log('Listening on port ' + port + '.');
27     }
28   });

```

Jak widać podstawowa konfiguracja serwera HTTP/2 z wykorzystaniem biblioteki node-spdy(ODNOSNIK) jest dość prostym zadaniem. Należy jedynie pamiętać o wszystkich ustawieniach oraz wygenerowaniu kluczy dla połączenia SSL, bez których serwer niestety nie ruszy.

3.2 Server Push

Najbardziej wymagającą częścią projektu było zaimplementowanie funkcjonalności 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.

Patrząc na poniższy przykład postaram się przybliżyć sposób, w jaki

udało mi się zaimplementować tę funkcjonalność. 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 zasymulowanej 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 jednym z kolejnych rozdziałów, obarcza dewelopera pewną odpowiedzialnością. W związku z tym, że klient nie prosił o dane zasoby musimy koniecznie wypełnić nagłówki przesyłanego zasobu, aby przeglądarka mogła odpowiednio zareagować na odebrany zasób. Na przykład po prostu go odrzucić, gdy na przykład 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`.

```
1 app.get('/push', function(req, res) {
2   fs.readFile('/push.html', function read(err, data) {
3     if(err) {
4       throw err;
5     }
6     content = data;
7     res.write(content)
8   })
9
10  fs.readFile('/jquery.js', function read(err, data) {
11    if(err) {
12      throw err;
13    }
14    content = data;
15    var stream = res.push('/libs/jquery/dist/jquery.js', {
16      status: 200, // optional
17      method: 'GET', // optional
18      request: { accept: '*/*' },
```

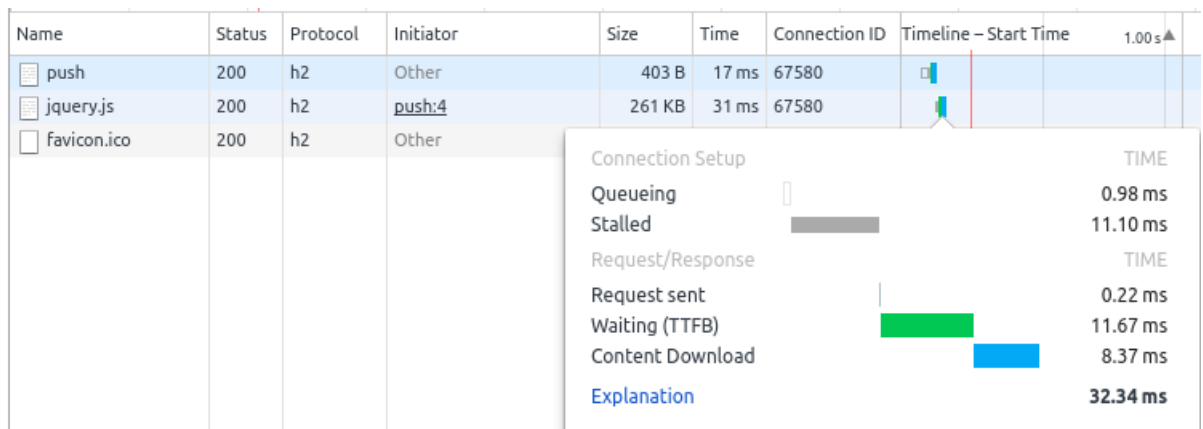


```






19     response: { 'content-type': 'application/javascript' }
20   })
21   stream.on('error', function(err) {
22     console.log(err);
23   })
24   stream.end(content)
25   res.end();
26 })
27 })

```

Ciekawy jest też wygląd okienka szczegółów zapytania w Google DevTools. Na rysunku 3.1 widzimy przykład zapytania bez wykorzystania server push. Znajdują się tutaj typowe informacje dla zwykłego zapytania HTTP. Dokładnie mam w tym wypadku na myśli czas wysłania zapytania oraz oczekiwania na początek odpowiedzi od serwera. Natomiast na rysunku 3.2 nie ma tych informacji. Świadczy to o tym, 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. Widać też, że czas zapytania to prawie 3 razy mniej 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

Name	Status	Protocol	Initiator	Size	Time	Connection ID	Timeline – Start Time	1.00 s ▲
 push	200	h2	Other	394 B	30 ms	32566		
 jquery.js	200	h2	Push / push:4	261 KB	5 ms	32566		
 favicon.ico	200	h2	Other					

Server Push

TIME

Receiving Push




7.98 ms

Connection Setup

TIME

Queueing




7.32 ms

Request/Response

TIME

Reading Push



1.37 ms

Explanation

12.01 ms

Rysunek 3.2: Szczegóły zapytania wysłanego z wykorzystaniem Server Push

Rozdział 4

Testy

4.1 Chrome DevTools

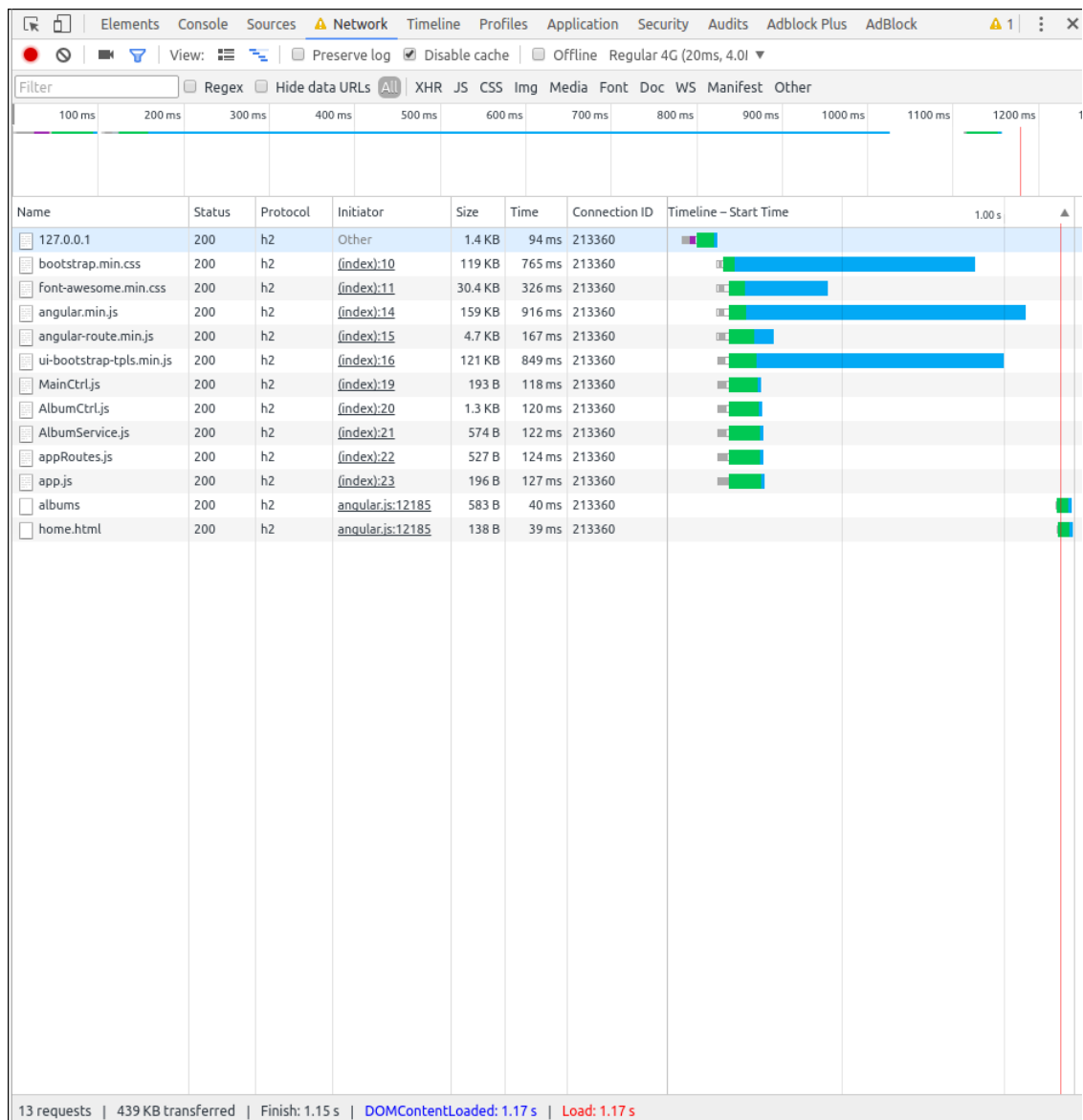
Do pomiaru prędkości oraz uzyskania innych ważnych informacji wykorzystałem narzędzie Chrome DevTools. Opiszę tutaj po krótce co i jak mierzyłem za pomocą tego oprogramowania.

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

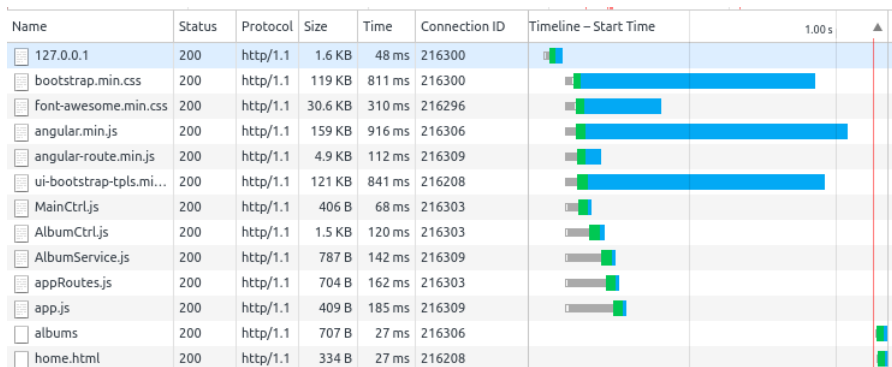
4.2 Przygotowanie testów

Przed przejściem do porównania prędkości działania obu wersji protokołu chciałem zaprezentować pierwsze efekty implementacji protokołu HTTP/2, które przedstawia rysunek 4.3.

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 4.3: Dowód działania protokołu HTTP/2

Widzimy tutaj dwie rzeczy, które powinny nas zainteresować. W sekcji 'Protocol' oznaczonej na rysunku 4.3 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 'Connection ID' (na rysunku 4.3 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 4.4.



Rysunek 4.4: 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 niż te w jego starszym bracie. W swojej pracy stworzyłem możliwość wysłania zapytania, które w odpowiedzi dostaje odpowiedź w postaci samego nagłówka. Dzięki temu można porównać wielkość nagłówków HTTP na konkretnym przykładzie, Wyniki przedstawiają rysunki 4.5 i 4.6.

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

Rysunek 4.5: 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 4.6: Pusta odpowiedź w HTTP/1.1

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

4.3 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.

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 4.7 oraz 4.8 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.

4.4 Porównanie prędkości przy bezpiecznym połączeniu HTTP/1.1

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

Name	Status	Prot...	Initiator	Size	Time	Connection ID	Timeline – Start Time	1.00 s	▲
127.0.0.1	200	h2	Other	1.4 KB	26 ms	18989			
AlbumCtrl.js	200	h2	(index):20	1.3 KB	60 ms	18989			
AlbumService.js	200	h2	(index):21	574 B	63 ms	18989			
appRoutes.js	200	h2	(index):22	491 B	65 ms	18989			
app.js	200	h2	(index):23	196 B	68 ms	18989			
bootstrap.min.css	200	h2	(index):10	119 KB	790 ms	18989			
font-awesome.m...	200	h2	(index):11	30.4 ...	299 ms	18989			
angular.min.js	200	h2	(index):14	159 KB	897 ms	18989			
angular-route.mi...	200	h2	(index):15	4.8 KB	109 ms	18989			
ui-bootstrap-tpls...	200	h2	(index):16	121 KB	823 ms	18989			
MainCtrl.js	200	h2	(index):19	193 B	56 ms	18989			
albums	200	h2	angular.js:12185	583 B	29 ms	18989			
home.html	200	h2	angular.js:12185	138 B	28 ms	18989			
13 requests 439 KB transferred Finish: 1.10 s DOMContentLoaded: 1.13 s Load: 1.13 s									

Rysunek 4.7: Test dla HTTP/2

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 4.8: Test dla HTTP/1.1

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

Wyniki poniższego testu kolejny raz pokazują wyższość nowszej wersji protokołu HTTP. Jednakże różnica 6% nie pozwala powiedzieć, że jest to ogromna różnica. Na podstawie tego testu widzimy też, że w przypadku wykorzystania zabezpieczonego połączenia czasy w przypadku HTTP/1.1 są minimalnie większe, niż bez wykorzystania tego protokołu.

4.5 Porównanie prędkości dla HTTP/1.1 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, w którym nieznacznie wygrywa nowsza wersja protokołu – około 11/

4.6 Porównanie prędkości przy wykorzystaniu Server Push

Próba	HTTP/1.1	HTTP/2
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
ŚREDNIA		

Rozdział 5

Wnioski

Bibliografia

- [1] M. Belshe, BitGo, R. Peon, Google, Inc, M. Thomson, Ed. Mozilla,
„Hypertext Transfer Protocol Version 2 (HTTP/2)“, 2015.
<https://tools.ietf.org/html/rfc7540>
- [2] webpack concept
<https://webpack.js.org/concepts/>