

# Benchmark HTTP1.1 vs HTTP2

**Disciplina:** Programação Distribuída e Paralela

**Professor:** Claudio Fernando Resin Geyer

**Alunos:** Eduardo Spitzer Fischer - 00290399

Rodrigo Paranhos Bastos - 00261162

Universidade Federal do Rio Grande do Sul  
Instituto de Informática

## INTRODUÇÃO

Este trabalho apresenta uma série de experimentos que comparam a performance dos protocolos de rede HTTP/1.1 e HTTP/2, duas revisões do protocolo que é a base da comunicação de dados na *World Wide Web*, sendo utilizados para servir a maioria das páginas que visitamos hoje em dia.

O protocolo HTTP foi pensado originalmente em uma época em que as páginas acessadas na *Internet* eram consideravelmente diferentes do que são hoje, em particular, eram compostas principalmente por texto e conteúdo estático, enquanto que no presente é comum que contenham conteúdo dinâmico, muitas imagens, vídeos, sons e interatividade. A vontade dos entes envolvidos de manter seus produtos o mais compatíveis e acessíveis quanto possível, bem como o rápido desenvolvimento do campo, fez com que a nova *Web* fosse construída em cima dos protocolos e ideias originais, tais como o HTTP/1.1, mesmo que este não fosse pensado para essa realidade. O protocolo HTTP/2 foi pensado como uma grande revisão do HTTP/1.1 para adequá-lo a essa nova realidade, prometendo melhorias significativas de performance se comparado com o HTTP/1.1 para as aplicações modernas.

Dentre as diferenças e novos recursos do HTTP/2 em comparação ao HTTP/1 pode-se citar:

1. Multiplexação completa de solicitação e resposta em uma mesma conexão.

O HTTP/1 é um protocolo sequencial, em que para cada conexão aberta o navegador solicita apenas um arquivo por vez, tendo de esperar a sua entrega para solicitar o próximo. Para contornar isso em face de páginas que referenciam múltiplos arquivos, comuns hoje em dia, os navegadores modernos abrem diversas conexões em paralelo, o que incorre em *overhead* de gerenciamento bem como maior uso de recursos. O HTTP/2 consegue, para cada origem, multiplexar na mesma conexão várias requisições e respostas.

2. *Server Push*

O HTTP/2 permite ao servidor enviar múltiplas respostas a um cliente, a partir de uma única requisição, sem que o cliente as tenha solicitado explicitamente. Isso permite que o servidor se antecipe a requisições que sabe que o cliente faria de qualquer forma. Por exemplo: ao receber a requisição que pede pelo arquivo *html* da página, o servidor pode já enviar junto como resposta não só o arquivo *html* mas também o *CSS*, *javascript*, ícones e outros componentes.

### 3. Priorização de requisições

Através do HTTP/2 o navegador pode informar ao servidor como deseja que as requisições sejam priorizadas, por meio de pesos e árvores de dependência. Um exemplo é mais uma vez a requisição de arquivos *CSS* e *javascript*, em que, no HTTP/2 os arquivos de *CSS* serão priorizados em relação aos de *js*, mesmo que sua requisição apareça no *DOM* posteriormente.

### 4. Camada de enquadramento binário

Ao contrário do HTTP/1.1 que é um protocolo baseado em texto, o HTTP/2 é baseado em quadros binários, cada um associado a um dos fluxos de dados da conexão. Essa característica é o que permite que várias “pequenas partes” de requisições e respostas se misturem na mesma conexão e sejam remontadas na ponta, bem como outras das melhorias do protocolo. Mesmo com essa mudança, a semântica dos métodos, verbos e cabeçalhos HTTP foi mantida, de modo que não é necessário uma grande mudança nos *softwares* para suportar o HTTP/2.

### 5. Compressão automática

O HTTP/2 realiza compressão automática dos dados usando o chamado algoritmo HPACK. Além da compressão usual *gzip*, que é padrão no HTTP/2 e tem de ser habilitada no HTTP/1, o HPACK utiliza uma *lookup-table* para comprimir os dados tirando proveito de informações do contexto da conexão: os *headers* comuns, como *GET* e *POST*, possuem como entradas na tabela números correspondentes, podendo ser substituídos por estes valores nos pacotes que de fato são enviados, economizando-se assim espaço. Mas não só isso, quando novos *headers* são utilizados na conexão, caso não existam entradas para os mesmos ou seus valores na tabela, entradas são criadas e, a partir desse momento, esses novos headers também passam a ser substituídos nas mensagens por valores numéricos mais enxutos. O mapeamento desses *headers* e valores para valores na *lookup-table* se dá utilizando o conhecido algoritmo de [codificação de Huffman](#).

## OBJETIVO

Pelo fato do HTTP/2 significar uma mudança de paradigma, algumas das técnicas desenvolvidas ao longo do tempo para lidar com as limitações do HTTP/1.1 possivelmente perdem o sentido e, em teoria, poderiam inclusive se tornar prejudiciais. Nosso objetivo com este trabalho foi confeccionar uma *página web* com características relevantes no que diz respeito às diferenças entre os protocolos e medir seu tempo de carregamento com cada um dos protocolos e usando diferentes técnicas em sua confecção e disponibilização, para averiguar em termos quantitativos quais os ganhos de performance de fato auferidos e o efeito de cada uma dessas técnicas no contexto de cada um dos protocolos. Também foram criados *endpoints* de *API* que servem diferentes conteúdos, com o mesmo objetivo.

Os experimentos fizeram uso das técnicas/recursos de *inlining* e *server push*, a página utilizada consiste em um mural de imagens e as *APIs* servem ou um texto longo ou um texto curto, como detalhado em mais detalhes nas seções a seguir de *Metodologia*.

## METODOLOGIA

O experimento é composto de três componentes principais: objetos de teste, servidores HTTP e ferramentas de teste e avaliação dos resultados.

### Objetos de teste:

Os objetos de teste são aqueles que foram servidos ao cliente através dos protocolos HTTP 1.1 e HTTP 2, sendo eles:

- **Página web contendo um grid de 15 imagens:**

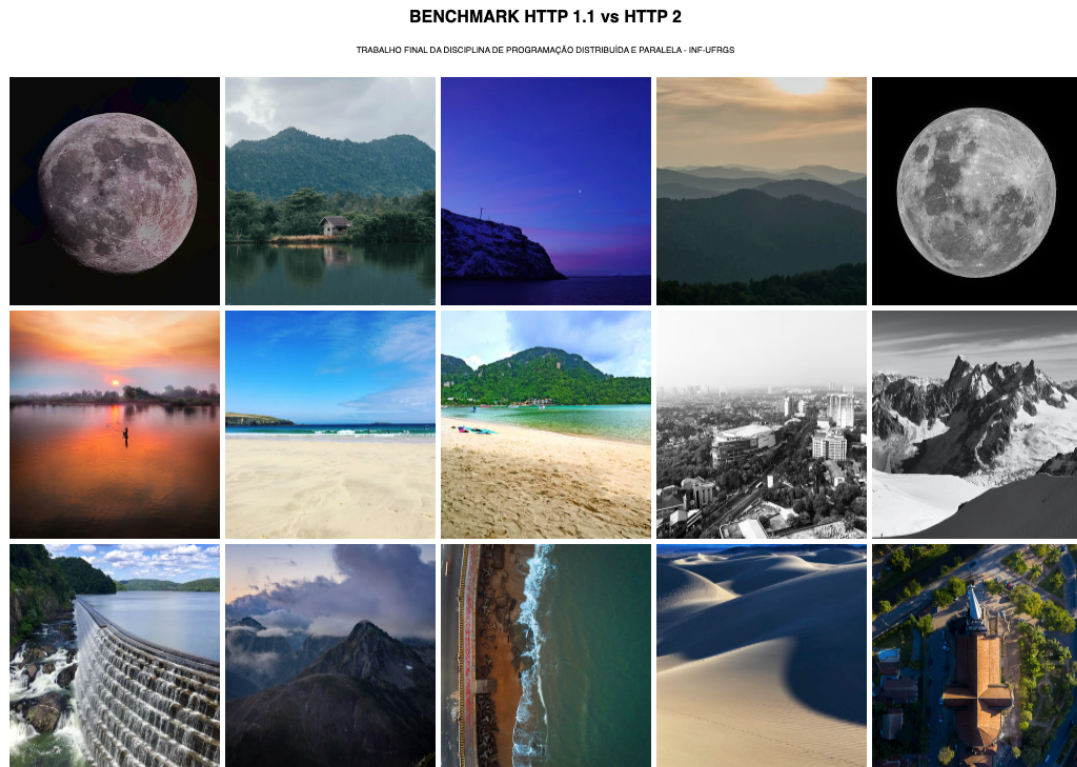


Imagem da página web desenvolvida para o experimento

A página web final final é composta por 17 arquivos distintos, sendo eles:

- 1 arquivo fonte (*index.html*)
- 1 arquivo de estilo (*style.css*)
- 15 imagens (*img1.jpeg*, ..., *img15.jpeg*)

Para fins de comparação, a mesma página web foi também desenvolvida utilizando a técnica de *inlining*, que consiste em inserir o estilo css bem com o conteúdo binário de cada imagem dentro do próprio arquivo *index.html*, resultando na mesma página contida em um único arquivo.

Os códigos fonte das páginas web desenvolvidas podem ser encontrados nos arquivos *index.html* e *index\_inline.html* da pasta de código anexada.

## - API servindo texto estático

O segundo caso de uso dos protocolos HTTP que testamos diz respeito ao envio de texto estático através do protocolo. Nesse caso, lidamos com um arquivo único, sem requisições secundárias de dependências.

Foram testados dois casos:

- Texto estático curto (44KB)
- Texto estático longo (2MB)

Os arquivos de texto estáticos servidos podem ser encontrados em `public/small-text.txt` e `public/large-text.txt` da pasta de código anexada.

## Servidores HTTP:

A ideia inicial era servir os objetos de teste descritos acima através de 2 servidores HTTP distintos, sendo eles:

- 1 servidor HTTP 1.1
- 1 servidor HTTP 2

Depois dos testes iniciais, percebemos que a maioria dos navegadores atuais não permite o uso do protocolo HTTP 2 em conexões não seguras, isto é, os navegadores forçam a utilização dos protocolos seguros SSL e TLS. Como trocamos o HTTP 2 pelo HTTPS 2 e considerando o fato de que os protocolos de segurança adicionam um *overhead* sobre o desempenho da conexão, decidimos incluir mais um servidor a fim de tornar mais justa a comparação, o HTTPS 1.1.

A lista final dos servidores implementados é a seguinte:

- 1 servidor HTTP 1.1 (sem SSL/TLS)
- 1 servidor HTTPS 1.1 (com SSL/TLS)
- 1 servidor HTTPS 2 (com SSL/TLS)

Todos os servidores foram implementados na linguagem Javascript, utilizando o runtime Node.js 15.14.0 e o framework Express.js, que facilita a criação de *endpoints* http e a disponibilização de arquivos estáticos. Para o servidor http2, foi utilizada a biblioteca *spdy*, desenvolvida pela Google e amplamente usada para este fim.

O código fonte dos servidores pode ser encontrado no arquivo `servers.js` da pasta de código fonte anexada.

Para a utilização da tecnologia segura HTTPS nos nossos servidores, foi necessário obter um certificado SSL. Como os testes ocorreram em um ambiente local, foram geradas chaves criptográficas localmente e a partir delas, um certificado SSL auto assinado. Para isso, utilizamos a ferramenta *openssl*, executando os seguintes comandos em um terminal Linux:

```
$ openssl genrsa -aes256 -out server.pass.key 4096
$ openssl rsa -in server.pass.key -out server.key
$ openssl req -nodes -new -key server.key -out server.csr
```

Feito isso, obtemos 2 arquivos: a chave privada `server.key` e o certificado `server.crt`. Ambos os arquivos foram utilizados na criação dos servidores HTTPS, como pode ser visto no trecho de código abaixo:

```
// Certificados SSL utilizados para a criptografia HTTPS
const credentials = {
  key: fs.readFileSync(__dirname + '/server.key'),
  cert: fs.readFileSync(__dirname + '/server.crt')
}

// Aqui criamos os 3 servidores distintos (HTTP1.1, HTTPS1.1 e HTTPS2)
const http1Server = http.createServer(app)
const https1Server = https.createServer(credentials, app)
const https2Server = spdy.createServer(credentials, app)
```

Todos os arquivos necessários para a execução do código e reprodução do experimento estão presentes na pasta enviada junto deste relatório. Para reprodução exata basta executar a seguinte sequência de comandos: (Obs.: É necessário ter o Node.js instalado na máquina)

```
$ npm install
$ npm start
```

Feito isso, os 3 servidores HTTP estarão em execução. Os endpoints disponíveis para cada um dos servidores são os seguintes:

#### Servidor HTTP 1.1:

- Página HTML + CSS + Imagens: <http://localhost:8000/>
- Página HTML + CSS + Imagens (Inlining): <http://localhost:8000/inline>
- Texto 44Kb: <http://localhost:8000/small-text.txt>
- Texto 2Mb: <http://localhost:8000/large-text.txt>

#### Servidor HTTPS 1.1:

- Página HTML + CSS + Imagens: <http://localhost:8001/>
- Página HTML + CSS + Imagens (Inlining): <http://localhost:8001/inline>
- Texto 44KB: <http://localhost:8001/small-text.txt>
- Texto 2MB: <http://localhost:8001/large-text.txt>

#### Servidor HTTPS 2:

- Página HTML + CSS + Imagens: <http://localhost:8002/>
- Página HTML + CSS + Imagens (Server Push): <http://localhost:8002/push>
- Página HTML + CSS + Imagens (Inlining): <http://localhost:8002/inline>
- Texto 44KB: <http://localhost:8002/small-text.txt>
- Texto 2MB: <http://localhost:8002/large-text.txt>

## Ferramenta de teste e avaliação dos resultados:

A ideia original do trabalho consistia em utilizar a ferramenta JMeter (<https://jmeter.apache.org/>) para a realização das avaliações dos tempos de carregamento relativos a cada um dos cenários de teste. O jMeter é uma aplicação desenvolvida em Java e permite que sejam disparados e sumarizados diversos tipos de requisição, incluindo requisições HTTP e HTTPS. Embora o jMeter não apresente por padrão suporte para requisições HTTP 2/HTTPS 2, encontramos um plugin da comunidade que adiciona esse suporte (<https://github.com/Blazemeter/jmeter-http2-plugin>).

Após a realização dos primeiros testes, entretanto, percebemos que embora a aplicação funcione muito bem para requisições HTTP 1.1, o plugin da comunidade que dá suporte a nova versão do HTTP não funciona como esperado, apresentando diversas falhas durante a execução e medições incorretas dos resultados.

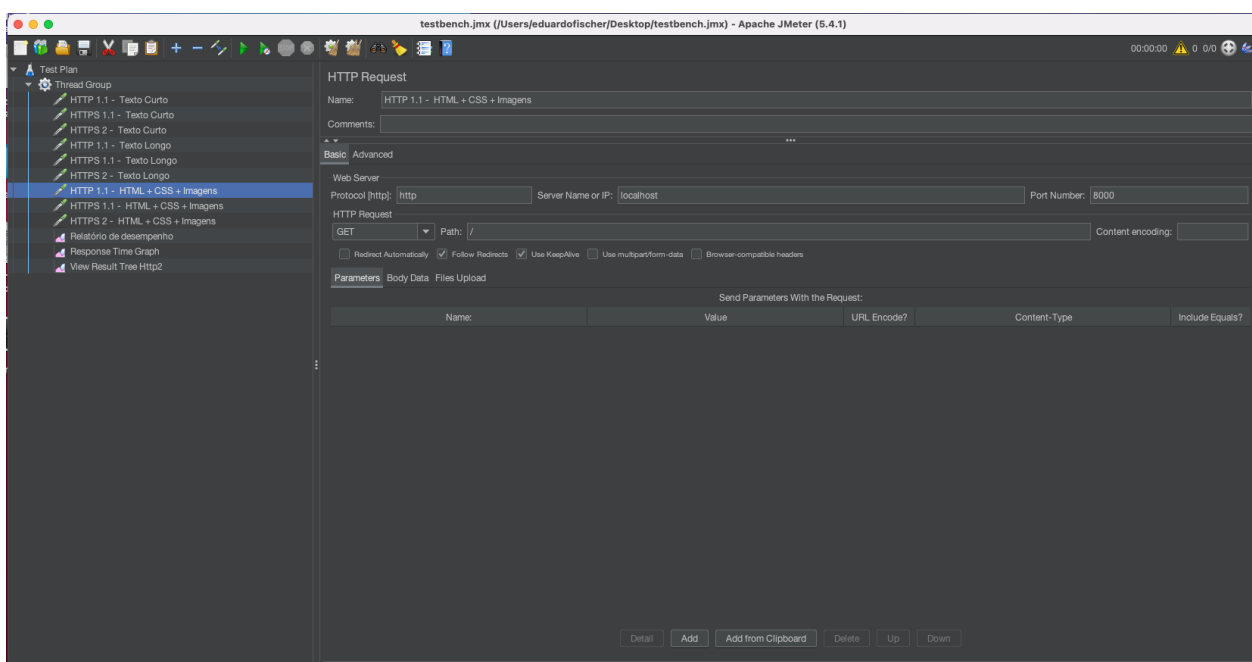


Imagem do software JMeter durante as tentativas de testes

Como alternativa, resolvemos realizar os testes utilizando a ferramenta de Networking do pacote Chrome DevTools (<https://developer.chrome.com/docs/devtools/>). Além de ser de fácil acesso, já que é distribuída juntamente com o navegador Google Chrome, a ferramenta é composta por uma alta gama de recursos que vieram a ser úteis para os testes, tais como possibilidade de simular atrasos na rede, controle do uso de cache, gráficos em cascata dos tempos de carregamento de cada um dos arquivos bem como informações de tempo detalhadas.

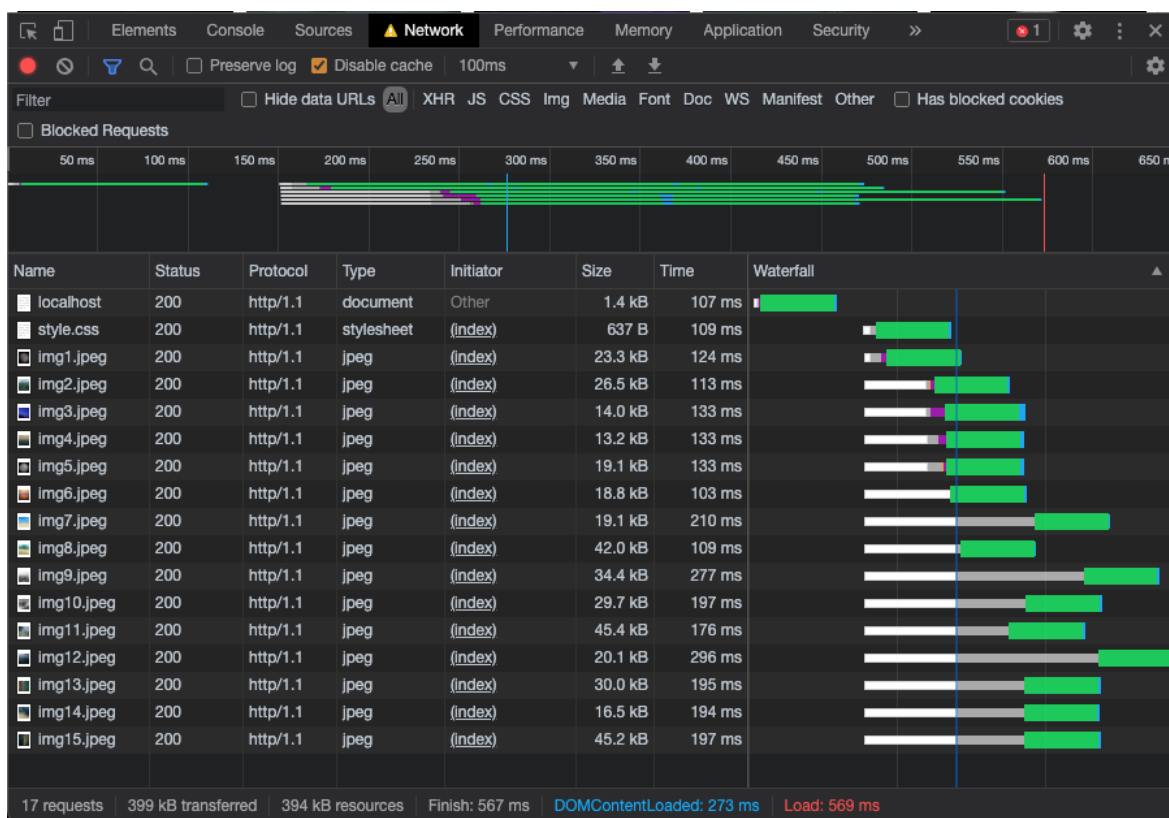


Imagem da ferramenta Chrome DevTools

Os testes foram realizados com os servidores e o testador hospedados em uma mesma máquina. Com a finalidade de simular um ambiente mais próximo ao da internet real, foram definidos os seguintes parâmetros na ferramenta:

- Atraso na rede: 100ms
- Uso de cache do cliente: desabilitado

Para cada combinação de objeto de teste vs servidor HTTP, foram realizadas 10 medições de tempo total de carregamento. A partir disso, foram extraídas as médias para cada caso de teste e gerados gráficos comparativos entre as diversas versões do protocolo HTTP para cada objeto de teste.

Para visualizar os mesmos gráficos de cascata mencionados acima, basta depois de seguir os [passos para subir os servidores](#) abrir os endereços impressos no console no navegador [Google Chrome](#), apertar F12, navegar para a aba **Network** do painel aberto e atualizar a página.

Os resultados dessa análise podem ser vistos a seguir:

## RESULTADOS

### COMPARAÇÃO 1 - Página HTML + Estilo CSS + 15 Imagens

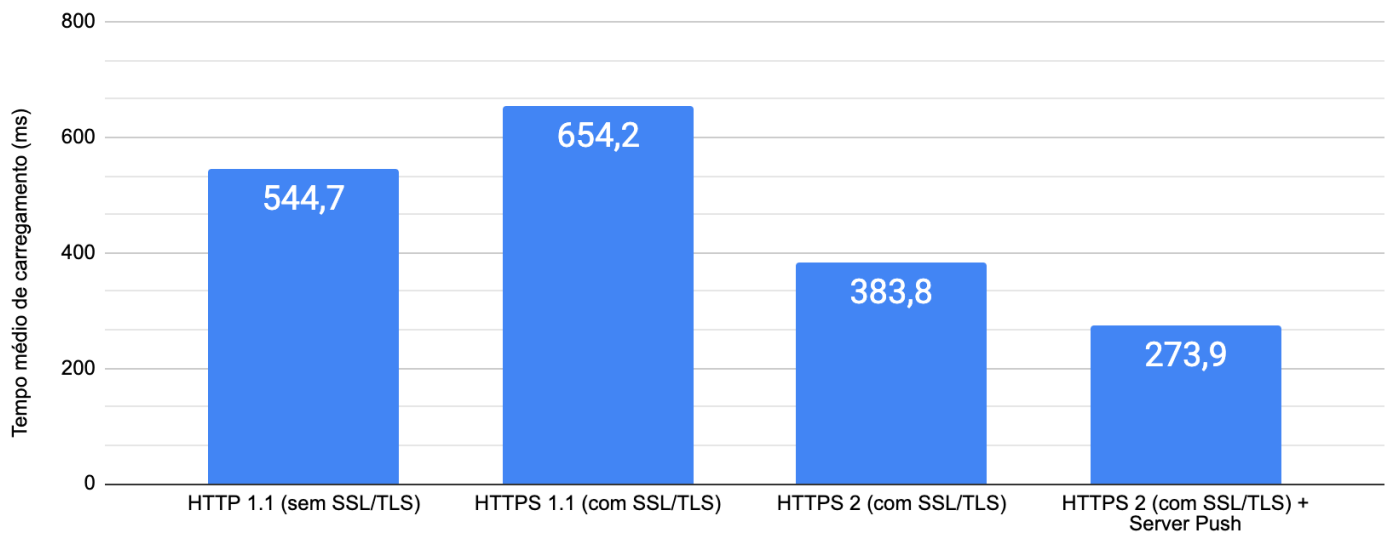
Nessa primeira avaliação, medimos o tempo de carregamento de uma página web complexa, composta por 1 arquivo index.html seguindo de 16 arquivos com dependências.

Vale lembrar que além dos testes com as diferentes versões do protocolo, realizamos também um teste implementando o recurso de server push no servidor HTTPS 2, isto é, no momento que o cliente faz a requisição para o arquivo index.html, o servidor automaticamente envia todos os arquivos das dependências.

O gráfico abaixo compara o tempo médio de carregamento desta página web com cada uma das diferentes versões do protocolo, bem como com a função server push do HTTPS 2.



## Página HTML + CSS + 15 Imagens



É possível perceber que entre os protocolos da versão 1.1, a versão segura com SSL/TLS teve um desempenho ligeiramente pior do que a versão tradicional. Isso se deve ao overhead da criptografia adicionada sobre a comunicação. Comparando com o protocolo na versão 2 entretanto, podemos observar que a nova versão do protocolo tem um desempenho bastante superior nesse cenário, chegando a ter um tempo de carregamento 1,7X menor que o HTTPS 1.1 e 2,4X menor tendo a função server push habilitada.

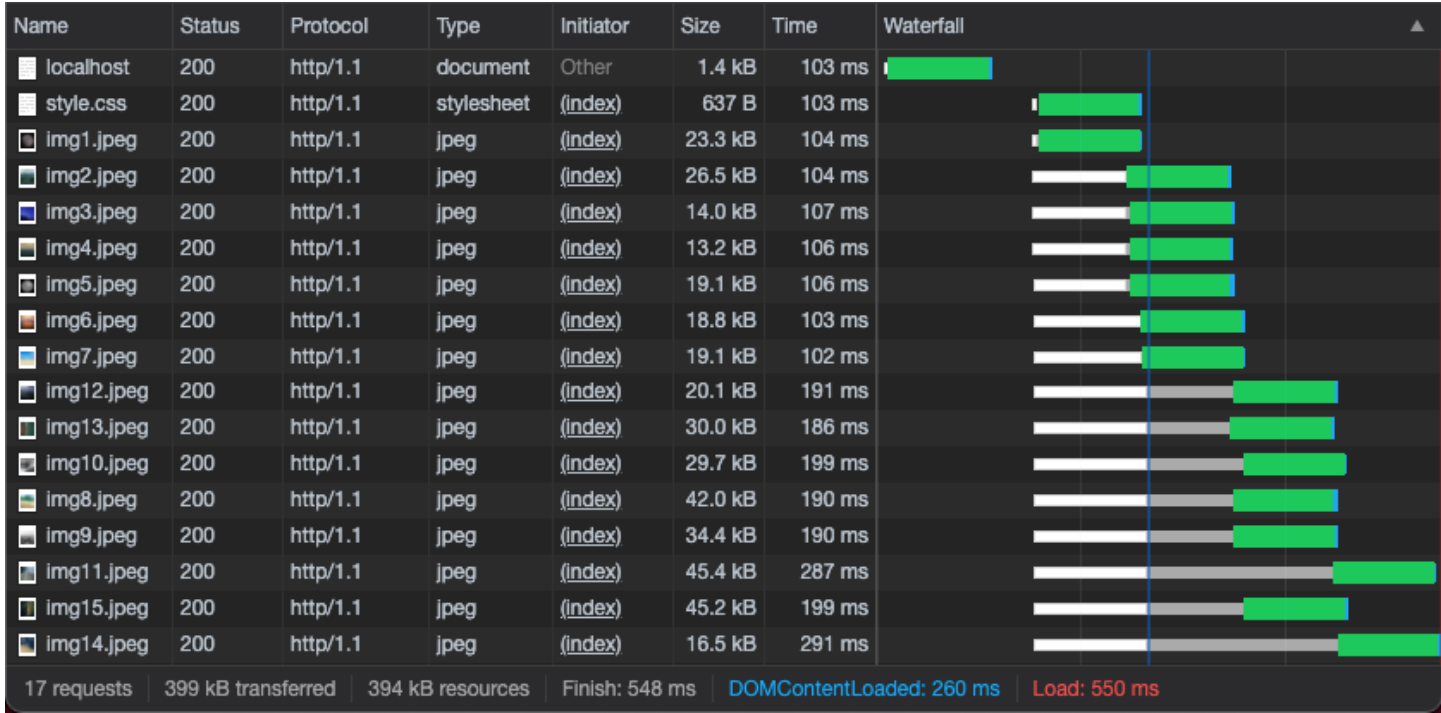
Na tabela abaixo é possível visualizar os dados coletados no experimento:

Página HTML + CSS + 15 Imagens				
Teste	Tempo total de carregamento (ms)			
	HTTP 1.1 (sem SSL/TLS)	HTTPS 1.1 (com SSL/TLS)	HTTPS 2 (com SSL/TLS)	HTTPS 2 (com SSL/TLS) + Server Push
1	533	541	432	292
2	539	686	410	274
3	536	683	384	269
4	554	664	381	276
5	552	694	349	270
6	538	677	358	264
7	541	543	356	269
8	542	670	377	294
9	555	689	384	264
10	557	695	407	267
Média	544,7	654,2	383,8	273,9

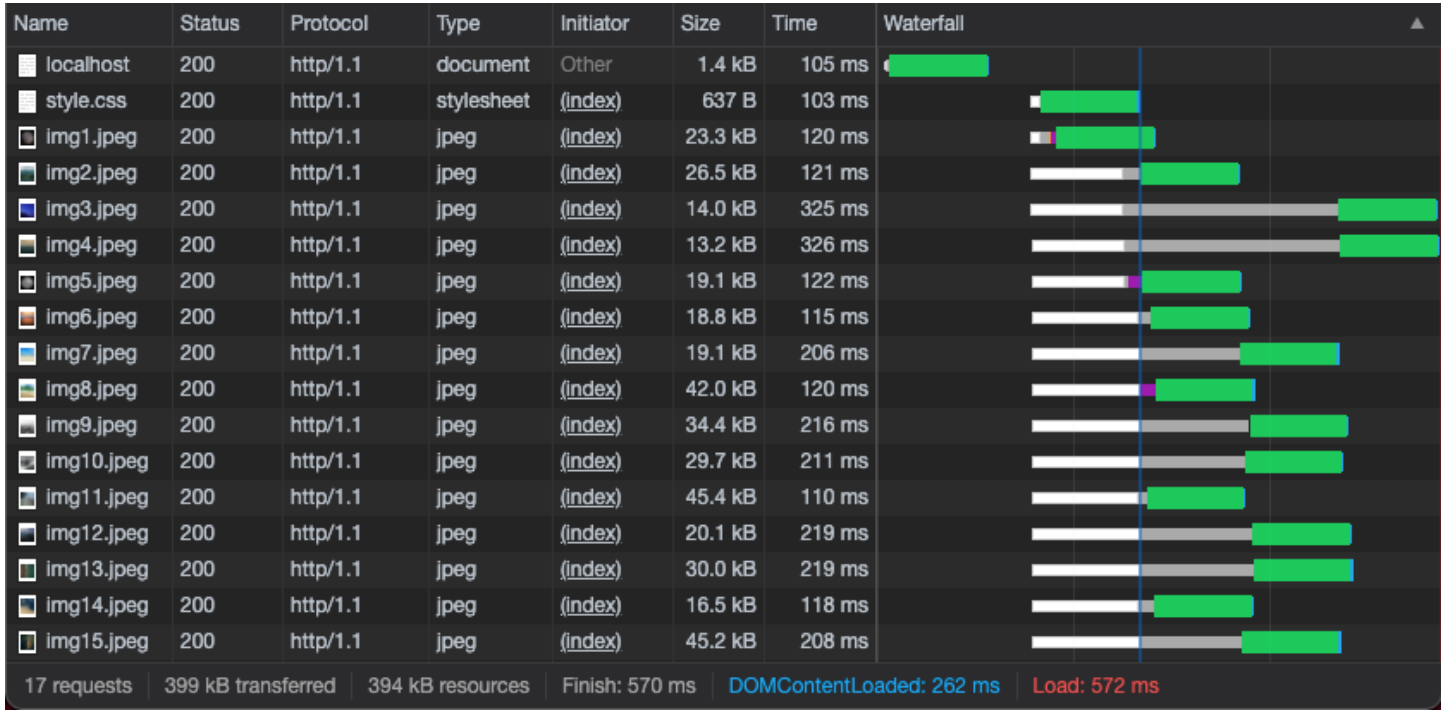
Abaixo é possível visualizar os gráficos do tipo cascata que mostram como as requisições HTTP dos diferentes arquivos foram realizadas no tempo, para cada variação do protocolo.


















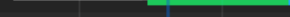

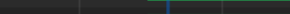
















HTTP 1.1 (sem SSL/TLS)



HTTPS 1.1 (com SSL/TLS)



## HTTPS 2 (com SSL/TLS)

Name	Status	Protocol	Type	Initiator	Size	Time	Waterfall		
 localhost	200	h2	document	Other	1.1 kB	103 ms			
 style.css	200	h2	stylesheet	(index)	417 B	104 ms			
 img1.jpeg	200	h2	jpeg	(index)	23.1 kB	103 ms			
 img2.jpeg	200	h2	jpeg	(index)	26.2 kB	108 ms			
 img3.jpeg	200	h2	jpeg	(index)	13.8 kB	107 ms			
 img4.jpeg	200	h2	jpeg	(index)	13.0 kB	107 ms			
 img5.jpeg	200	h2	jpeg	(index)	18.9 kB	107 ms			
 img6.jpeg	200	h2	jpeg	(index)	18.6 kB	108 ms			
 img7.jpeg	200	h2	jpeg	(index)	18.8 kB	111 ms			
 img8.jpeg	200	h2	jpeg	(index)	41.9 kB	109 ms			
 img9.jpeg	200	h2	jpeg	(index)	34.2 kB	107 ms			
 img10.jpeg	200	h2	jpeg	(index)	29.5 kB	109 ms			
 img11.jpeg	200	h2	jpeg	(index)	45.2 kB	111 ms			
 img12.jpeg	200	h2	jpeg	(index)	19.8 kB	108 ms			
 img13.jpeg	200	h2	jpeg	(index)	29.8 kB	108 ms			
 img14.jpeg	200	h2	jpeg	(index)	16.3 kB	110 ms			
 img15.jpeg	200	h2	jpeg	(index)	45.1 kB	112 ms			
17 requests		396 kB transferred		394 kB resources		Finish: 355 ms	DOMContentLoaded: 259 ms		Load: 392 ms

## HTTPS 2 (com SSL/TLS) + Server Push

Name	Status	Protocol	Type	Initiator	Size	Time	Waterfall
localhost	200	h2	document	Other	1.6 kB	103 ms	
style.css	200	h2	stylesheet	Push / (index)	355 B	2 ms	
img1.jpeg	200	h2	jpeg	Push / (index)	23.0 kB	3 ms	
img2.jpeg	200	h2	jpeg	Push / (index)	26.2 kB	4 ms	
img3.jpeg	200	h2	jpeg	Push / (index)	13.8 kB	5 ms	
img4.jpeg	200	h2	jpeg	Push / (index)	13.0 kB	6 ms	
img5.jpeg	200	h2	jpeg	Push / (index)	18.9 kB	7 ms	
img6.jpeg	200	h2	jpeg	Push / (index)	18.5 kB	7 ms	
img7.jpeg	200	h2	jpeg	Push / (index)	18.8 kB	7 ms	
img8.jpeg	200	h2	jpeg	Push / (index)	41.8 kB	8 ms	
img9.jpeg	200	h2	jpeg	Push / (index)	34.2 kB	9 ms	
img10.jpeg	200	h2	jpeg	Push / (index)	29.5 kB	8 ms	
img11.jpeg	200	h2	jpeg	Push / (index)	45.2 kB	8 ms	
img12.jpeg	200	h2	jpeg	Push / (index)	19.8 kB	9 ms	
img13.jpeg	200	h2	jpeg	Push / (index)	29.8 kB	10 ms	
img14.jpeg	200	h2	jpeg	Push / (index)	16.2 kB	10 ms	
img15.jpeg	200	h2	jpeg	Push / (index)	45.0 kB	10 ms	

17 requests

396 kB transferred

394 kB resources

Finish: 245 ms

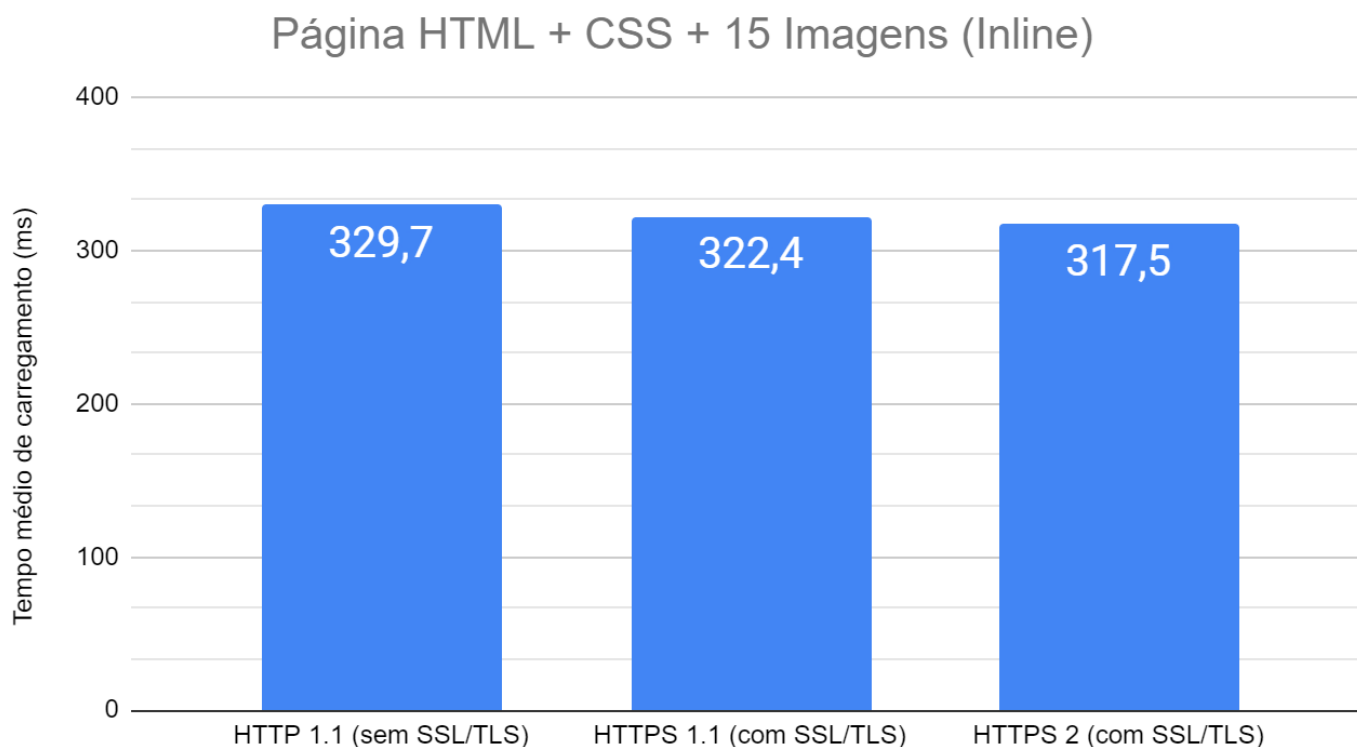
DOMContentLoaded: 249 ms

Load: 296 ms

## COMPARAÇÃO 2 - Página HTML + CSS + 15 Imagens (Inline)

Na segunda avaliação, medimos o tempo de carregamento da mesma página web, mas agora utilizando a técnica de *inlining* que, como foi exposto anteriormente, consiste em embutir no *html* da página o código dos outros arquivos dos quais depende, entregando tudo que é necessário para sua exibição no navegador em um único grande arquivo.

O gráfico abaixo compara o tempo médio de carregamento desta página web com cada uma das diferentes versões do protocolo.



Como era de se esperar e justificando-se o desenvolvimento desta técnica, os tempos obtidos com a utilização de *inlining* mostraram-se em geral menores do que aqueles que obtivemos servindo o conteúdo da maneira convencional, com uma exceção digna de nota: o tempo obtido no teste anterior, com vários arquivos, fazendo-se uso do *Server Push* (273,9 ms) foi *inferior* ao melhor tempo obtido com o *inlining* (317,5 ms) , mostrando que o recurso possibilita ganhos de performance sem que se tenha de pagar o preço da execução de técnicas mais complicadas.

Mais uma vez, o HTTP 2 apresentou os menores tempos no geral, embora as diferenças entre os tempos desse teste estejam dentro da margem de erro esperada, fazendo com que os protocolos tenham virtualmente empatado.

Da mesma forma que anteriormente, abaixo pode se ver o detalhamento dos testes:



### COMPARAÇÃO 3 e 4 - 44 KB de texto e 2 MB de textos puros

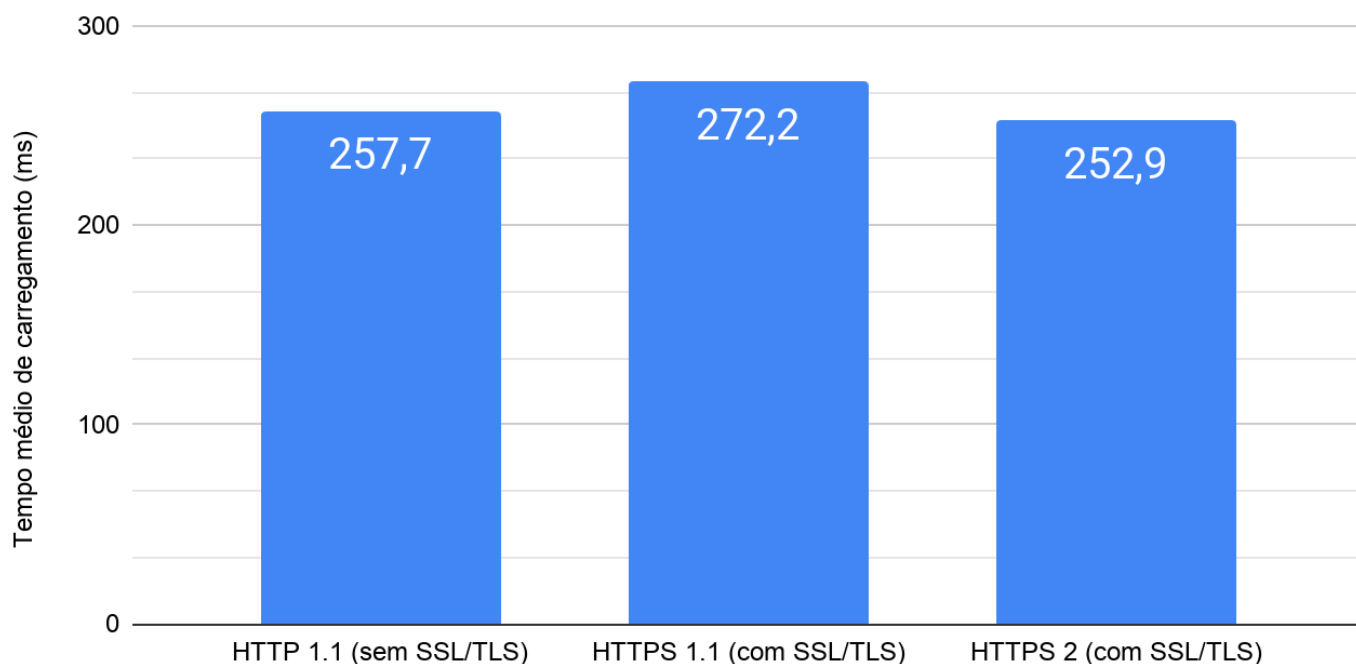
Além de seu uso em páginas web, os protocolos HTTP também são muito usados em APIs, que retornam os dados consultados de forma serializada.

Pensando nesse caso de uso, decidimos testar cada um dos protocolos com *payloads* de texto puro.

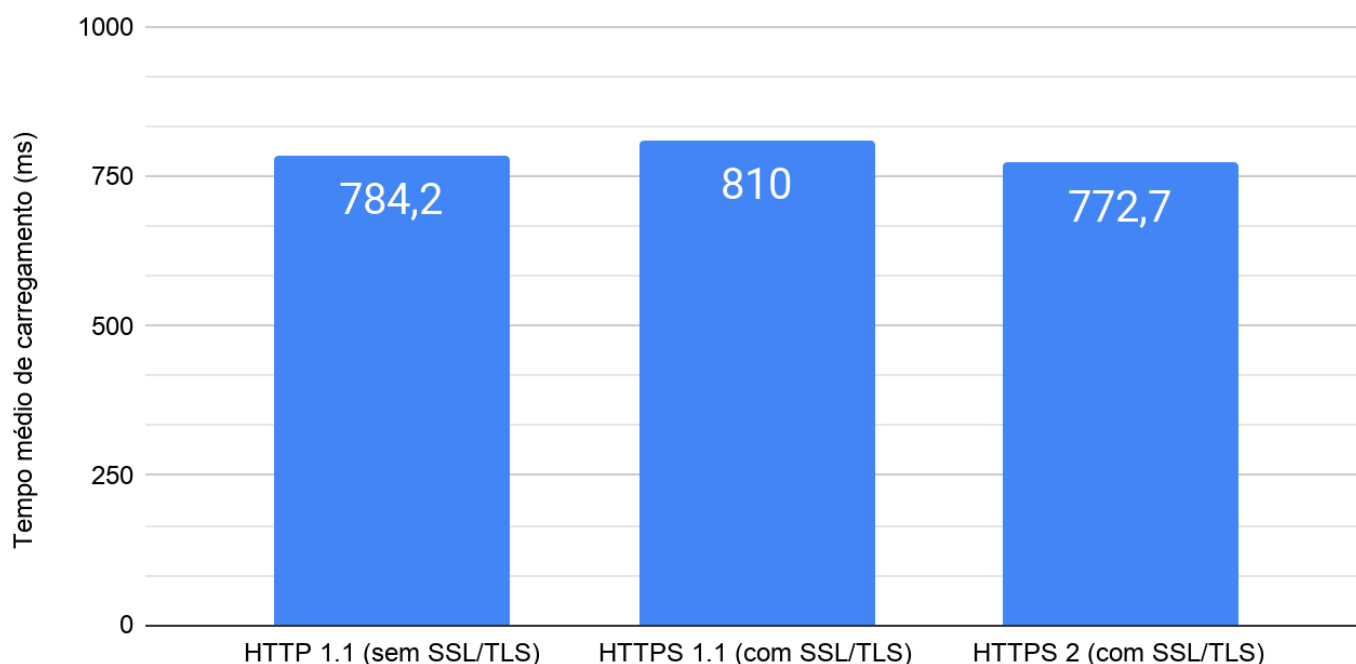
Foram feitos dois testes distintos, um com um texto curto, de 44 Kb de tamanho e outro com um texto longo, com 2 Mb de tamanho.

Nos gráfico abaixo é possível visualizar os resultados de tempo médio de carregamento para em cada um dos protocolos testados:

44 KB de texto simples



2 MB de texto simples



Analisando os resultados, podemos perceber que embora o protocolo HTTPS 2 não tenha vantagens tão significativas nesse caso de uso quanto tem em aplicações web com dependências, a versão mais recente do protocolo continua sendo mais eficiente, superando inclusive a versão não segura do protocolo HTTP 1.1.

No que diz respeito ao tamanho dos *payloads*, não notamos nenhuma influência significativa do tamanho dos textos no desempenho dos protocolos.

Nas tabelas abaixo é possível visualizar os dados coletados no experimento:

44 KB de texto simples			
Teste	Tempo total de carregamento (ms)		
	HTTP 1.1 (sem SSL/TLS)	HTTPS 1.1 (com SSL/TLS)	HTTPS 2 (com SSL/TLS)
1	251	246	239
2	244	246	265
3	253	334	249
4	247	269	248
5	257	272	269
6	253	248	253
7	274	259	246
8	280	332	253
9	269	273	251
10	249	243	256
Média	257,7	272,2	252,9

2 MB de texto simples			
Teste	Tempo total de carregamento (ms)		
	HTTP 1.1 (sem SSL/TLS)	HTTPS 1.1 (com SSL/TLS)	HTTPS 2 (com SSL/TLS)
1	740	772	746
2	778	873	771
3	770	777	775
4	805	826	785
5	818	767	778
6	781	861	803
7	767	765	779
8	806	856	791
9	766	774	760
10	811	829	739
Média	784,2	810	772,7

## CONCLUSÃO

Em adição às suas garantias de segurança (navegadores em sua maioria suportam-no apenas por HTTPS ) e maiores níveis de controle oferecidos (*server push*, *priorização...*) o HTTP/2 realmente confere vantagens significativas na velocidade de carregamento de conteúdo na *web*. Em todos os testes realizados o HTTP/2 teve a melhor performance sendo, ao que nos parece, recomendável se não em todas, na grande maioria das situações.

### FONTES:

Geração de certificados SSL

<https://devcenter.heroku.com/articles/ssl-endpoint#acquire-ssl-certificate>

Instalação do plugin HTTP/2 no JMeter

<https://www.flood.io/blog/jmeter-tutorial-http2-test>

Implementação do recurso de server pushing no servidor HTTP/2

<https://webapplog.com/http2-server-push-node-express/>