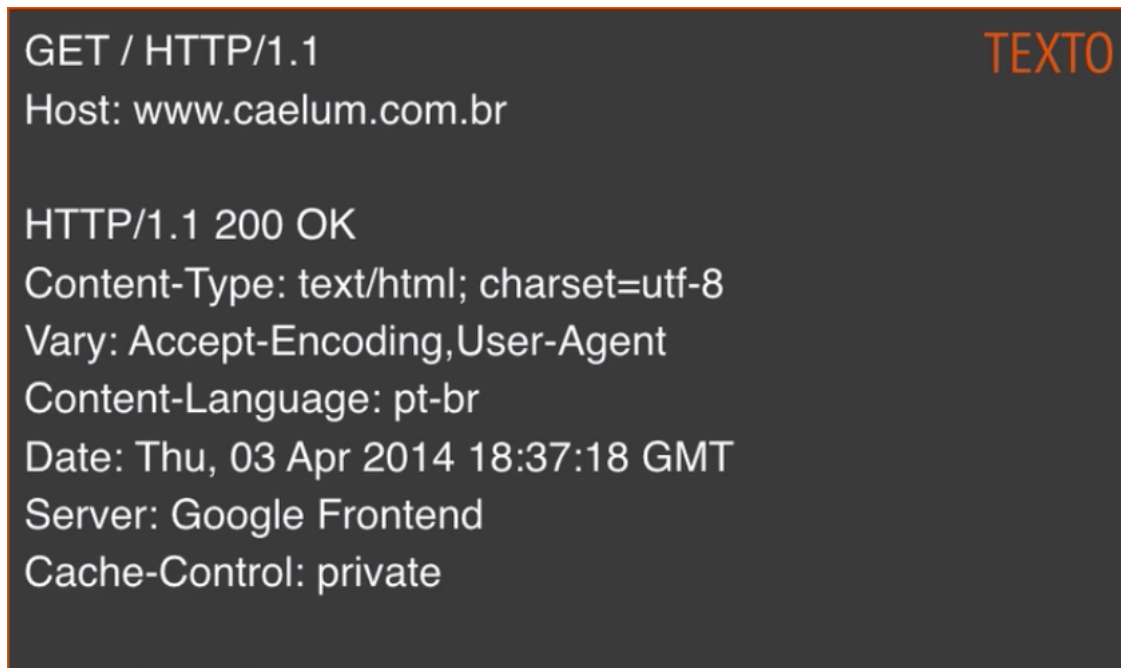


Uma coisa bem importante é falarmos sobre o novo protocolo HTTP, o HTTP 2.0 que está por trás de todas as páginas da web. Vamos entender um pouco o que é HTTP e o que esse novo modelo contribui e influência em nosso dia a dia.

Review HTTP

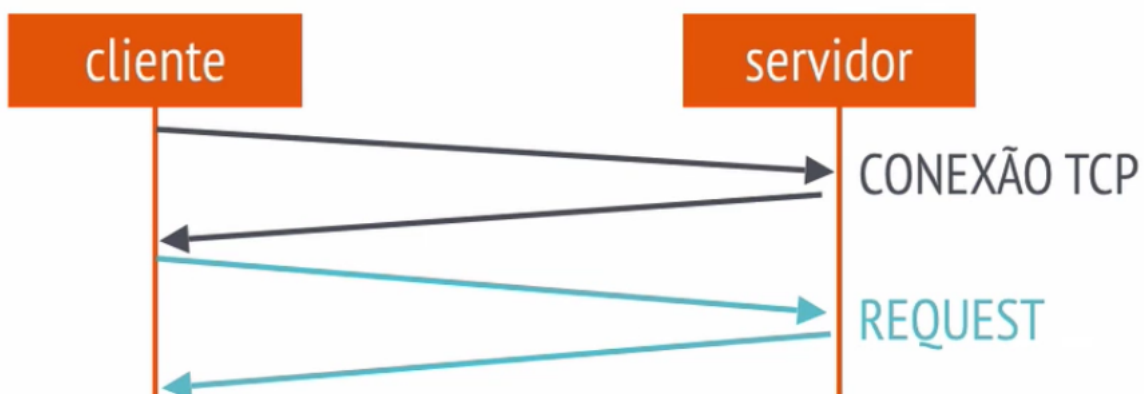
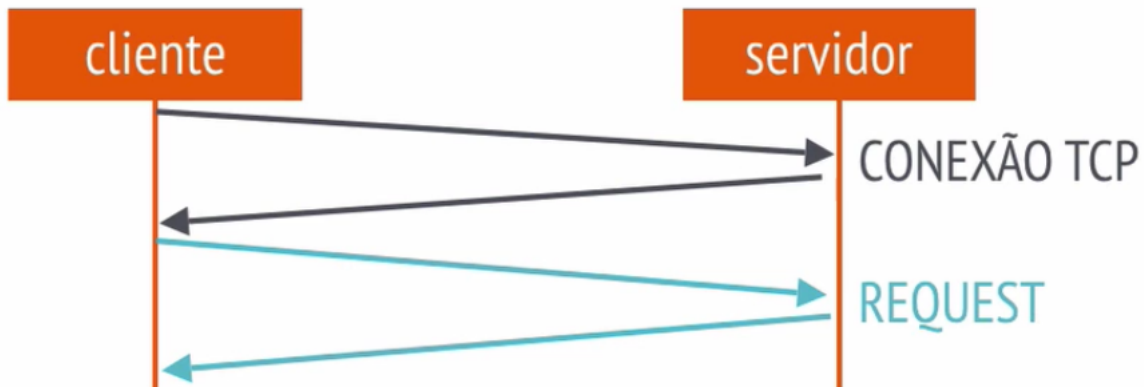


```
GET / HTTP/1.1
Host: www.caelum.com.br

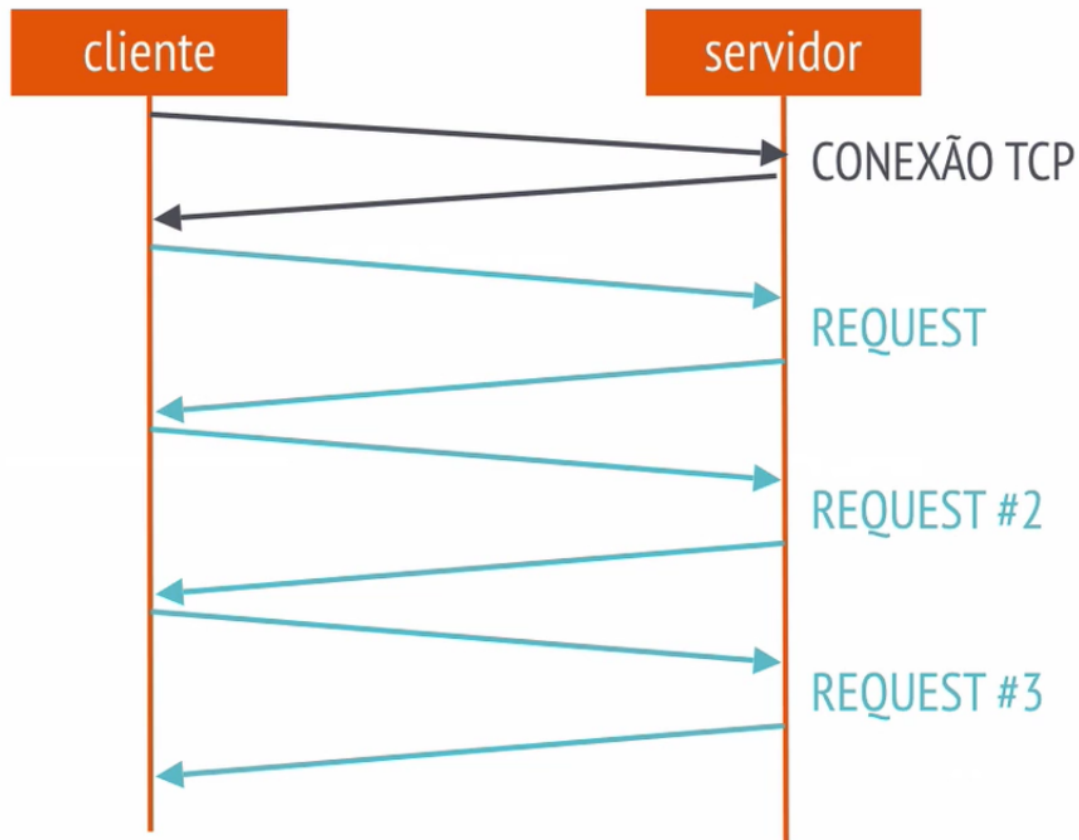
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Vary: Accept-Encoding,User-Agent
Content-Language: pt-br
Date: Thu, 03 Apr 2014 18:37:18 GMT
Server: Google Frontend
Cache-Control: private
```

O protocolo HTTP é baseado em texto, ou seja, quando nos conectados temos que digitar um texto para enviar um GET, um post e etc. Segundo ponto, ele é baseado em *request* e *response* e entre isso existe uma espera. O ponto de espera é importante, porque é nesse momento que o cliente não enxerga nada. Outros pontos que são interessantes, é que ele é baseado na ideia de métodos, "URLs" e *status quo* e temos também as respostas. Tanto no *request* quanto no *response* nós temos cabeçalhos, *headers*. Eles fazem parte do protocolo e mandam algumas coisas, por exemplo, o *content tip*, o tipo de *cache* que estamos utilizando e etc. Esse é o HTTP Clássico.

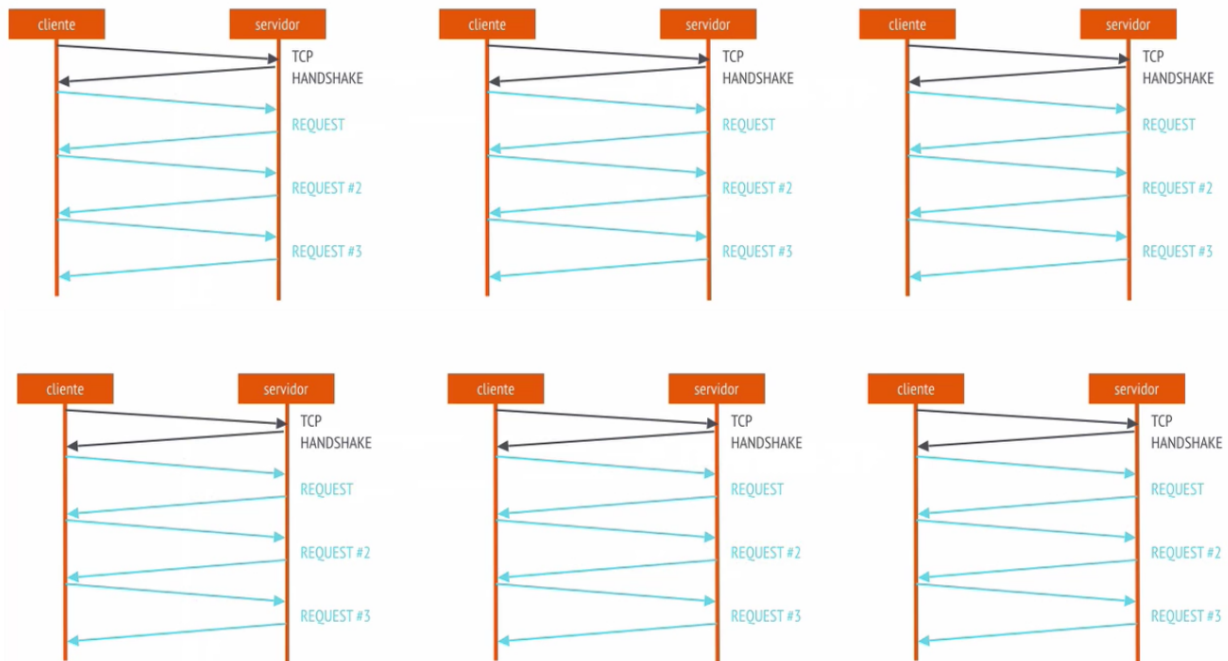
Esse é um desenho para tentarmos entender como funciona o envio das requisições e respostas do servidor:



O objetivo é tentar mostrar o que acontece se quisermos fazer duas requisições. No HTTP clássico a ideia é que cada requisição precisava de uma conexão TCP diferente. Mas isso é péssimo! Pois, quanto mais conexões temos, mais pagamos o preço por abrir elas. Uma mudança, que na verdade vem antes do HTTP 2.0 é a possibilidade de reaproveitar conexões, colocando diversos *requests* em uma conexão, é o *keep alive* do HTTP 1.0.



Aqui, temos um problema, pois a próxima requisição só pode ser enviada quando a resposta da anterior chegar. Esse desenho mostra como o processo é sequencial e por isso pode acabar se tornando uma cascata gigantesca. Para evitar esse tipo de cenário os navegadores fazem mutiplas conexões, conforme a representação da imagem abaixo:



Individualmente cada requisição é sequencial, mas como temos seis conexões paralelas, na verdade possuímos seis requests paralelos por *hostname*. Tudo isso ocorre com HTTP 1.

Baseado nesse cenário existem algumas boas práticas:

- 1) diminuir requests: juntar CSS e javascript, diminuir número de imagens realizando *sprites* e embutir recursos *inline* no HTML;
- 2) diminuir tráfego: habilitar gzip no servidor, minificação de "CSS", *javascript* e etc... Uma prática que não mencionamos, utilizar domínios sem *cookies*, um problema dos *cookies* é que uma vez que eles são configurados eles são reenviados para o servidor a cada nova requisição;
- 3) paralelizar requests: fazer múltiplos *hostnames*, ainda que o navegador tenha um limite e, por fim, trabalhar com CDNs para distribuir em outros servidores.

Existem diversas práticas que podem ser realizadas em torno das limitações e características que discutimos do HTTP 1.0.

Novo HTTP/2

Todos esses princípios que discutimos estão relacionados a estrutura básica do HTTP 1. A partir de 2015 temos a introdução do HTTP 2 que altera um pouco o funcionamento das coisas. A semântica permanece a mesma, entretanto, do ponto de vista do uso, a implementação sofre algumas melhorias.

No HTTP/1.1, quando fazemos o *request* e pegamos o *response* da página poderíamos habilitar o *gzip* como um meio de otimização. Mas, no HTTP clássico, não existe *gzip* de *header*. O que é um grande problema, pois temos diversos *headers* grandes.

Uma primeira mudança que o HTTP/2 traz é a compressão de *headers*. Ela não é feita usando GZIP, e sim HPACK que é mais eficiente e seguro para comprimir *headers*. Os cabeçalhos no HTTP/2 são naturalmente mais leves pois eles já são compactados. Mas não só isso, no HTTP/2 teremos também o uso do TLS que é trabalhar com segurança, tudo que está compactado ainda será transferido de maneira binária e segura. No HTTP/2 é e não é obrigatório usar site seguro, explico, no protocolo do HTTP/2 nada obriga o uso de sites seguros, mas na prática é obrigatório, pois nenhum navegador moderno suporta HTTP/2 sem criptografia. As mudanças do HTTP/1.1 para o 2 devem e estão sendo feitas de maneira gradual, mas como funciona hoje a web, não podemos atualizar o servidor e o navegador e achar que isso está "ok", pois na *web* temos diversos intermediários e o problema consiste em que se o miolo não for atualizado, coisas podem quebrar.

No HTTP/2 temos:

- Header Binário;
- GZIP obrigatório (antes era necessário habilitá-lo);
- HPACK nos *headers*;
- TLS, com criptografia necessária para não quebrar a internet;

Vamos observar algo, temos na imagem abaixo dois *requests* diferentes de HTTP/1. Quebramos o GET em dois, uma linha mostra o método e qual o *path*. Estamos mostrando dois *requests*, um para "home" e outro para o "style.css".

HTTP/1.1

```
:method: GET
:path: /
Host: www.caelum.com.br
Accept: text/html,application/xhtml+xml;q=0.9,image/jpeg
Accept-Encoding: gzip
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_2)
Cookie: SID=786dads78asdbad876asdhjvb28
```

```
:method: GET
:path: /style.css
Host: www.caelum.com.br
Accept: text/html,application/xhtml+xml;q=0.9,image/jpeg
Accept-Encoding: gzip
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_2)
Cookie: SID=786dads78asdbad876asdhjvb28
```

Repare que entre um *request* e outro o que muda, na verdade é o `path`. O resto, os *headers*, são iguais. No HTTP 1, mesmo os *headers* sendo iguais temos que ficar trafegando o tempo todo e isso acarreta em um desperdício. No HTTP/2, se você já enviou os *headers* antes você não precisa enviar de novo, é mandado só aquilo que muda. A conexão mantém o estado dos *headers* que tínhamos antes.

Passamos por toda essa explicação para chegar nas *Head Tables* que são comuns ao protocolo e evita ficar renovando a mesma coisa o tempo inteiro. *Headers* que foram enviados anteriormente não precisam ser reenviados, ele salva isso. O protocolo HTTP semanticamente continua a ser *statles*, mas na implementação da conexão ele guarda coisas a mais, os *Head Tables*. Isso é uma otimização de performance.

Agora, após termos conhecimento sobre o HTTP/ 2, podemos riscar algumas coisas de nossa lista. O *gzip* ele continua a ser essencial, mas a compressão das coisas deveria ser habilitada, agora é automática. Não precisamos nos preocupar com os domínios de *cookies*, eles são enviados uma única vez e depois são reaproveitados e, inclusive compactados. Algumas otimizações que fazíamos antes podemos cortar pelo simples fato de atualizarmos para o HTTP/2. Observe como está nossa lista:

OTIMIZAÇÕES WEB

diminuir
requests

JUNTAR CSS E JS
SPRITES
INLINE DE RECURSOS

diminuir
tráfego

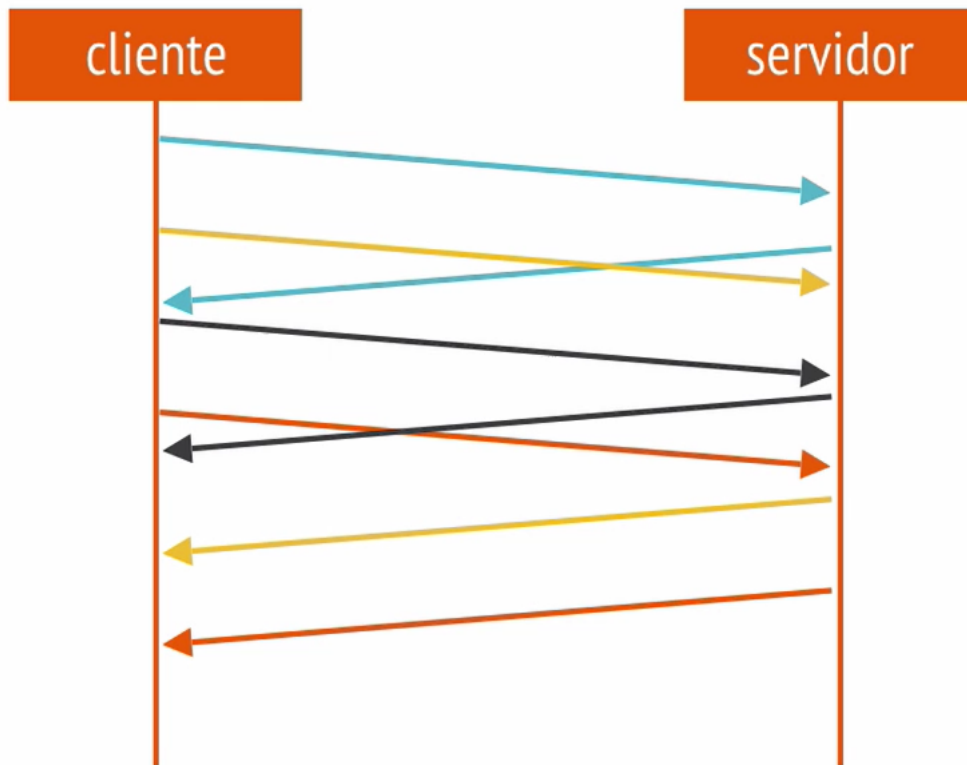
GZIP ★
MINIFICAÇÃO
DOMÍNIOS SEM COOKIES

paralelizar
requests

MÚLTIPLOS HOSTNAMES
CDNs

No HTTP 1.1 uma ideia de otimização era usar o *keep-alive*. Entretanto, isso tem o problema de que enquanto a primeira requisição não chegar a segunda não é disparada. Ainda nesse HTTP um recurso para tentar resolver esses problemas é o *pipelining* que significa mandar mais de uma requisição, simultaneamente. Isso possui um detalhe, que a ordem das respostas seja a mesma ordem das requisições que enviamos, o que cria um problema também, se tivermos uma resposta demorando ela trava as demais respostas. Isso cria um problema que chamamos de *Head of Line Blocking*. O *Pipeline* foi algo que na prática não acabou sendo implementado.

A ideia de misturar requisições e respostas é a base do que chamamos de *multiplexing* é a cereja do novo protocolo do HTTP/ 2:



HTTP 2.0 multiplexing

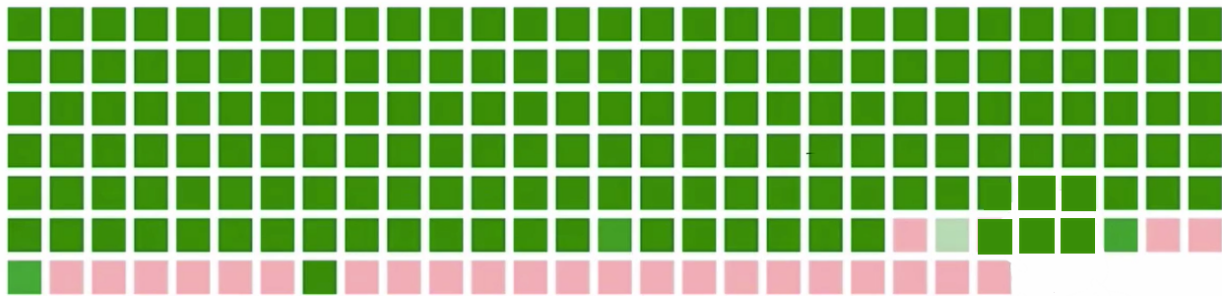
Isso evita, por exemplo, que uma mensagem que demora para voltar acabe acavando as demais respostas. A implementação do protocolo HTTP/2 nas duas pontas não interfere na escrita da página da web, pois, a semântica da HTTP não muda.

Com isso, chegamos a uma conclusão, não precisamos mais de seis conexões, com uma única conexão de TCP conseguimos, de maneira multiplicada, enviar vários *requests* e *responses* e usar essa conexão ao máximo. Isso é ótimo, pois, conseguimos diminuir o uso de recurso, inclusive, diminuir o consumo de bateria de um celular e etc. Basta habilitar o HTTP/2 para ganhar tudo isso de graça.

No HTTP 1 uma boa prática é realizar a concatenação de diversos tipos de arquivos, pois, melhora o tempo de carregamento. Mas, se prestarmos atenção, o tempo do primeiro *script* quando fazemos isso acaba demorando mais, pois agora, o tempo de baixar esse primeiro *script* é o tempo do todo, não temos mais como priorizar as coisas, teríamos que quebrar em arquivos distintos. Essa prática nos concede vantagens, mas ela faz com que tenhamos diversas desvantagens também.

Vejamos uma simulação de como fica se carregarmos os mesmos 200 arquivos em HTTP 2:

Loading 200 JS modules asynchronously in separated files (13 kb each file)



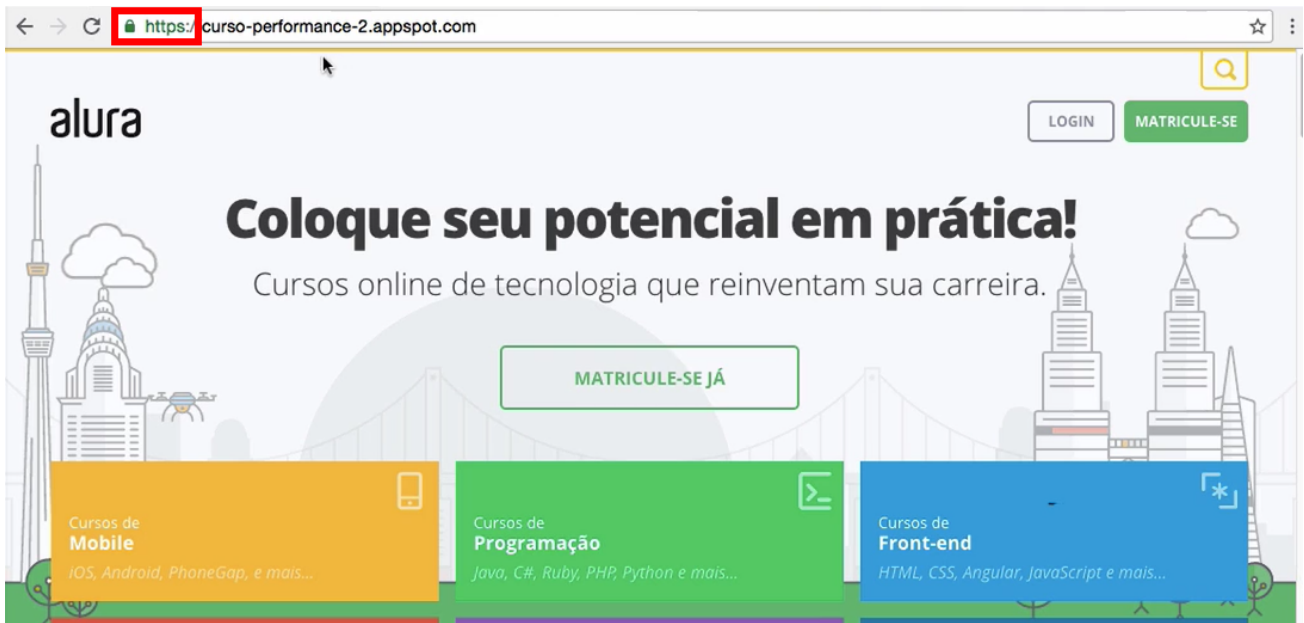
Temos um primeiro *script* priorizado e o todo carrega mais rápido, pois temos uma conexão multiplexada. É diferente do que tínhamos antes, que era de seis em seis, uma verdadeira cascata. Fica mais fácil de desenvolvermos. Uma observação: não existe um limite de *requests*, 1 é muito pouco, mas talvez 200 sejam muitos *requests*. A ideia é que agora temos diversas opções de como conduziremos o processo.

Quando o próprio protocolo faz a multiplexação de forma transparente ele nós dá novas aberturas para não precisarmos daquelas diversas "gambiarras" que fazíamos no HTTP 1.

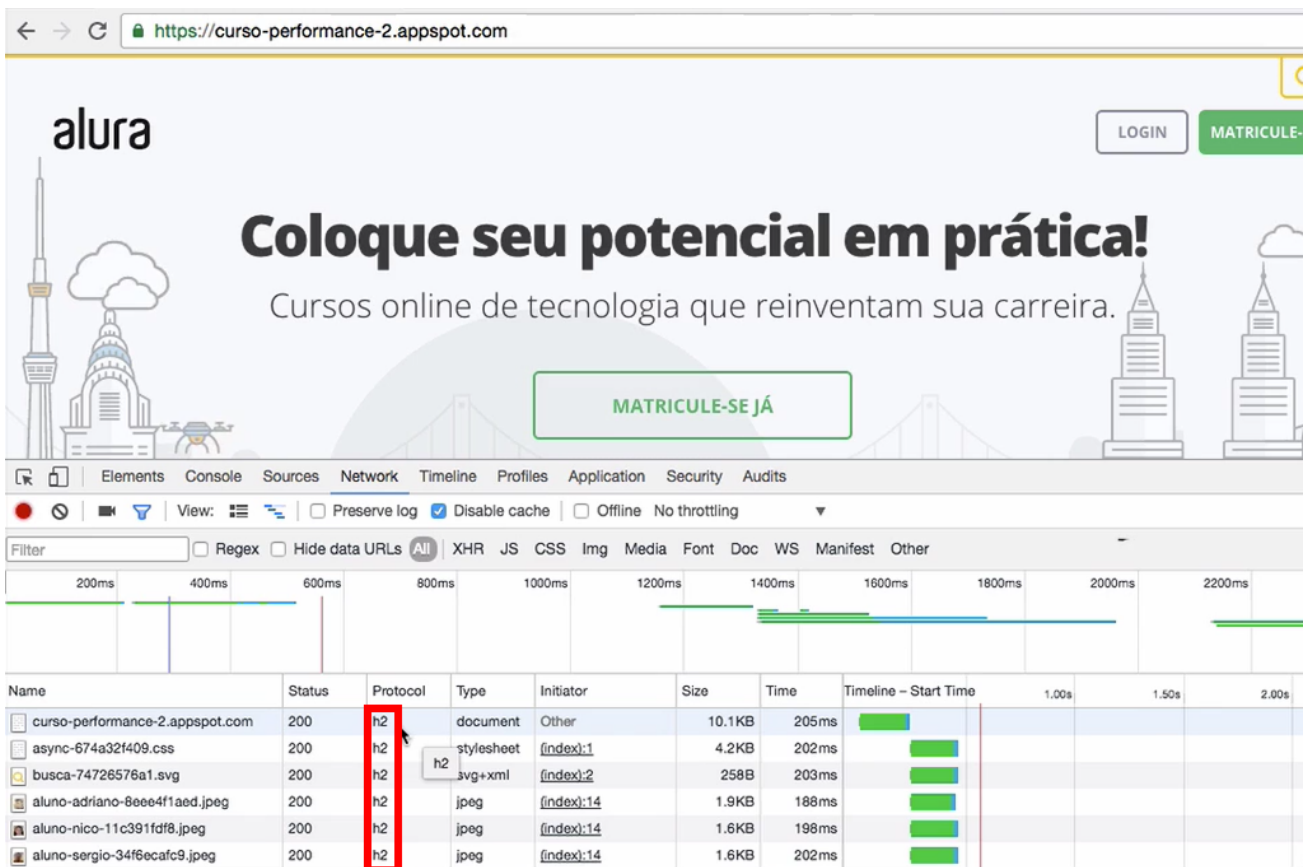
Com o HTTP 2 não precisamos mais da concatenação de arquivos CSS e JS, os *sprites* tão pouco, as requisições são algo que o próprio protocolo faz por nós. Não é mais necessário paralelizar *requests*, pois, não precisamos de múltiplos *hostnames*. Algumas coisas, entretanto, permanecem úteis, por exemplo, o CDN, que diminui latência, mas ele relacionado a paralelizar *request*, já não é tão útil.

O *inline* de recursos, prática que utilizamos anteriormente, possui dois objetivos. O primeiro é diminuir o número de *requests* pois estamos "inlineando" o conteúdo direto no HTML, nesse caso não precisamos fazer isso no HTTP 2 devido a conexão multiplexada. O segundo recurso do "inline" é priorizar conteúdo e isso segue sendo importante e para esse caso ele ainda é bom para ser utilizado.

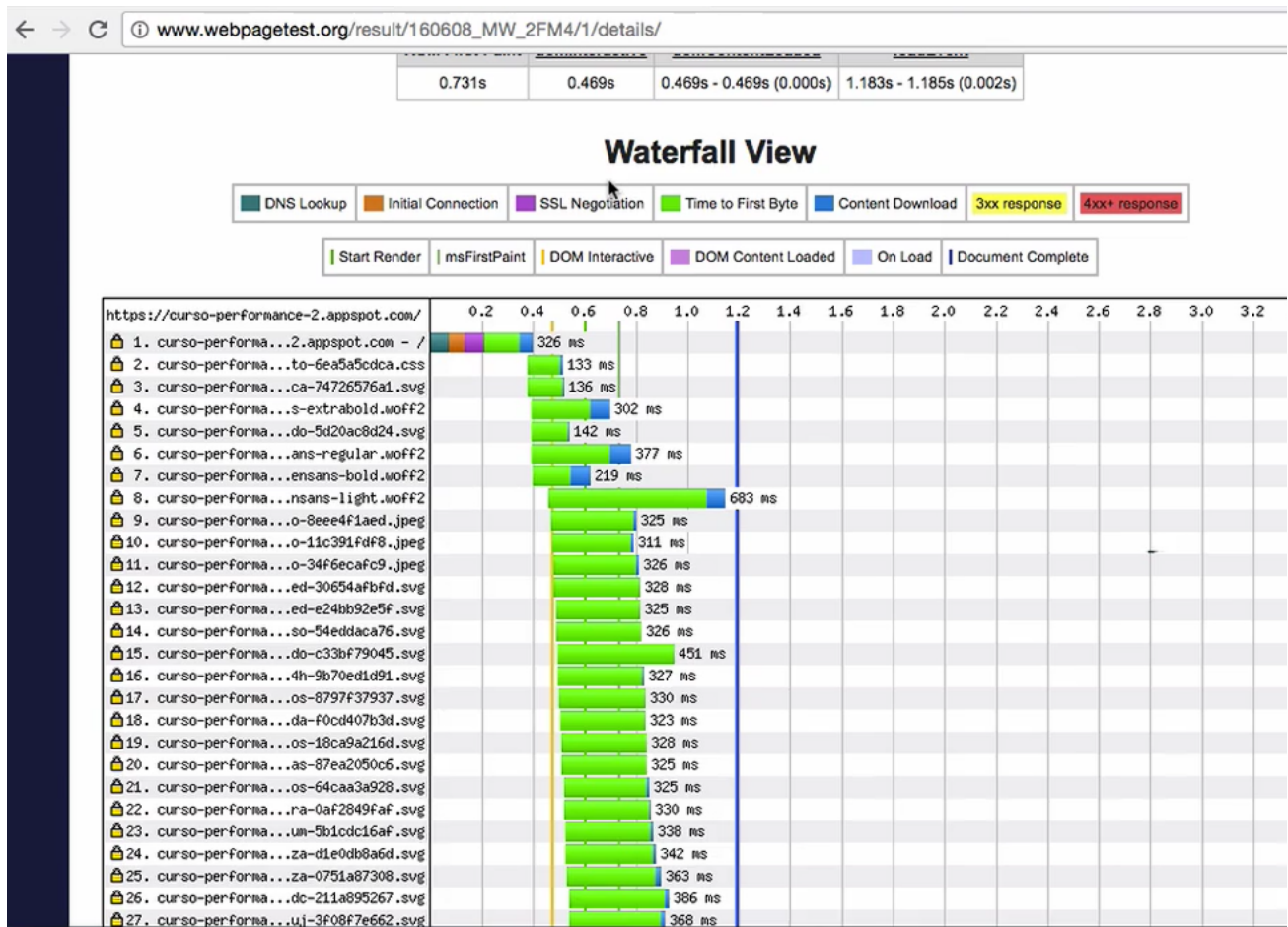
Vamos subir o projeto com HTTP/2 para ver algumas diferenças na prática. Usamos um servidor que suporta o HTTP 2, o *Google App Engine*, aquele mesmo que utilizamos no primeiro curso. Subimos um novo projeto para diferenciar esse daquele do primeiro curso, chamamos de "curso-performance-2". O *Google App Engine* possui uma grande vantagem, que é já vir com suporte nativo, transparente e automático desde que usemos a versão HTTPs do site. Basta escrever no navegador o "https" que ele nos mostra a versão do HTTP/2:



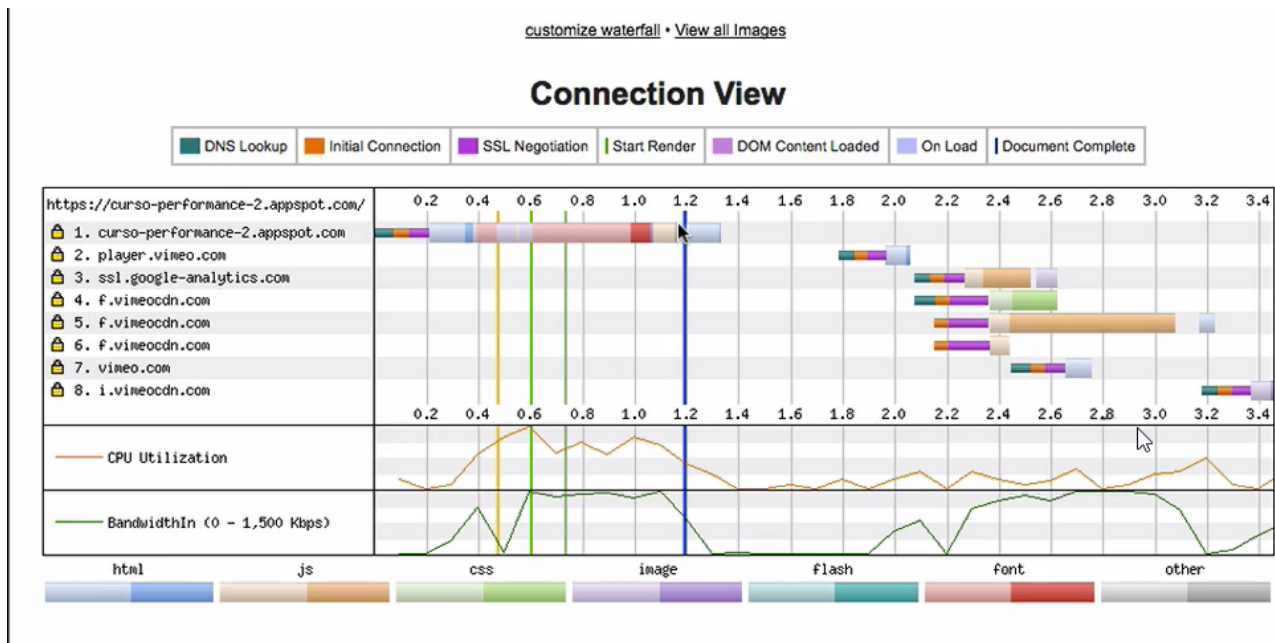
Para identificar que estamos utilizando o HTTP/2 podemos abrir o *console* acessar a aba *Network*. Ative o "Protocol" e poderemos visualizar nessa coluna que todas as requisições que possuem HTTP/2, serão diferenciadas com o "h2":



Vamos rodar no *Web Page Test* a versão do HTTP/2 para vermos algumas coisas interessantes. Vamos analisar, primeiro, o gráfico *Water Fall* que possui diferenças fundamentais em relação ao HTTP 1.



Podemos ver, logo de cara, uma barra roxa na primeira requisição, ela indica o "SSL". Quando colocamos o "https" pagamos o preço de fazer uma negociação de certificados, pois não temos uma conexão sem criptografia. É necessário que se faça uma avaliação desse aspecto, se realmente vale a pena sacrificar o tempo desse carregamento. Em nosso caso são apenas 100 milésimos de segundos e acaba sendo válido. Verifique o gráfico do *Connection View*, ele mostra quais conexões foram abertas usando cada *hostname*. Uma diferença brutal que podemos verificar é que quando usamos o HTTP/2 estamos utilizando apenas uma conexão, multiplexada:



Vamos voltar ao nosso projeto, já realizamos a otimização do *Critical Path*. "Buildamos" todos os "CSS" para que eles funcionassem *inline* e utilizamos o *async* em um único arquivo. Fizemos isso na página da home. Temos uma outra página que é a "cadastrado":



No HTTP/2 podemos quebrar esse único arquivo onde colocamos os "CSS" em dois blocos.

Vamos tirar os dois arquivos, "home-aprenda" e "home" em um bloco distinto. Vamos retirar do bloco principal e vamos colocar ele em um arquivo chamado "home-async.css":

```
22      <!-- build:css assets/css/async.css -->
23      </noscript>
24
25      <link rel="stylesheet" href="assets/css/block-conteudo.css">
26      <link rel="stylesheet" href="assets/css/block-cursoCard.css">
27      <link rel="stylesheet" href="assets/css/block-depoimentos.css">
28      <link rel="stylesheet" href="assets/css/block-elasticMedia.css">
29      <link rel="stylesheet" href="assets/css/block-footer-listaCursos.css">
30      <link rel="stylesheet" href="assets/css/block-footer.css">
31      <link rel="stylesheet" href="assets/css/block-form-erro.css">
32      <link rel="stylesheet" href="assets/css/block-grupoCaelum.css">
33
34      <link rel="stylesheet" href="assets/css/block-highlighted.css">
35      <link rel="stylesheet" href="assets/css/block-painelPlanos.css">
36
37      <link rel="stylesheet" href="assets/css/block-titulos.css">
38
39      <link rel="stylesheet" href="assets/css/home-aprenda.css">
40      <link rel="stylesheet" href="assets/css/home-diferenciais.css">
41
42      <noscript>
43      <!-- endbuild -->
```

Na "home" temos agora, dois arquivos. Esse foi um exemplo prático de como podemos ter arquivos adicionais. Damos um `gulp` e vamos rodar localmente para observar a diferença. Vamos clicar com o botão direito e pedir "View Page Source" e teremos um monte de coisas, poderemos verificar que teremos o "async.css" e o "home.css" que são dois arquivos diferentes. Se quiser, você pode quebrar em mais arquivos.

O ponto onde queríamos chegar é que no HTTP/2 podemos nos dar ao luxo de ter mais arquivos. Podemos fazer isso no final da página. Lembra-se que tínhamos usado a abordagem de usar todos os "scripts" em um único "build"? Essa estratégia, entretanto, acarreta um grande problema, pois o carregamento ocorre de uma vez só. Observando o código fonte veremos que ele gera um arquivo *javascript* com todo o conteúdo dentro. Todo o restante deve ser executado antes que se comece a executar esse arquivo:

```
918 <!-- build:js assets/js/scripts.js async --> |
919 <script src="assets/js/loadcss.js"></script>
920 <script src="assets/js/cssrelpreload.js"></script>
921 <script src="assets/js/menu.js"></script>
922 <script src="assets/js/busca.js"></script>
923 <script src="assets/js/detect.js"></script>
924 <script src="assets/js/footer.js"></script>
925 <script src="assets/js/svg4everybody.js"></script>
926 <script src="assets/js/video.js"></script>
927 <script src="assets/js/lazyload.js"></script>
928 <!-- endbuild -->
```

Vamos tomar uma atitude radical, vamos tirar o "build" e deixaremos esses arquivos serem baixados de maneira independente. Podemos fazer isso no HTTP/2, como queremos que isso seja executado de maneira independente iremos adicionar um `async` e colocar esse atributo em cada um dos *scripts*. Faremos o `gulp` e dando um "open page source" poderemos visualizar todos os *javascripts* carregados de maneira independente.

O foco é priorizar a execução dos "scripts", queremos que aquele que chegar primeiro seja executado primeiro. Fazemos isso carregando arquivos independentes, com caches independentes. Essa tática é uma grande vantagem, ela pode ser feita de maneira assíncrona de forma que o *browser* dispara todos os downloads e conforme eles vão chegando podemos executá-los fora de ordem. Claro que se existem *scripts* que exigem uma ordem, teremos que pensar em outra solução. Com HTTP/2 podemos fazer isso com diversos arquivos. Concluindo, é uma boa prática utilizar o HTTP/2.