



CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

OTIMIZANDO DESEMPENHO DE FRONT-END EM WEBSITES PARA HTTP2

PEDRO COLEN CARDOSO

Orientador: Prof. Flávio Coutinho
Centro Federal de Educação Tecnológica de Minas Gerais – CEFET-MG

BELO HORIZONTE
MAIO DE 2015

PEDRO COLEN CARDOSO

OTIMIZANDO DESEMPENHO DE FRONT-END EM WEBSITES PARA HTTP2

Trabalho de Conclusão de Curso apresentado ao Curso
de Engenharia da Computação do Centro Federal de
Educação Tecnológica de Minas Gerais.

Orientador: Flávio Coutinho
Centro Federal de Educação Tecnológica
de Minas Gerais – CEFET-MG

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO
BELO HORIZONTE
MAIO DE 2015

PEDRO COLEN CARDOSO

OTIMIZANDO DESEMPENHO DE FRONT-END EM WEBSITES PARA HTTP2

Trabalho de Conclusão de Curso apresentado ao Curso
de Engenharia da Computação do Centro Federal de
Educação Tecnológica de Minas Gerais.

Trabalho aprovado. Belo Horizonte, 24 de novembro de 2014

Flávio Coutinho
Orientador

Co-Orientador

Professor
Convidado 1

Professor
Convidado 2

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO
BELO HORIZONTE
MAIO DE 2015

Espaço reservado para dedicatória. Inserir
seu texto aqui...

Agradecimentos

Inserir seu texto aqui... (esta página é opcional)

"You can't connect the dots looking forward you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future. You have to trust in something: your gut, destiny, life, karma, whatever. Because believing that the dots will connect down the road will give you the confidence to follow your heart, even when it leads you off the well worn path."(Steve Jobs)

Resumo

Abstract

Lista de Figuras

Figura 1 – Média de Bytes por Página por Tipo de Conteúdo em 2011	1
Figura 2 – Média de Bytes por Página por Tipo de Conteúdo em 2015	2
Figura 3 – Exemplo de requisição HTTP	6
Figura 4 – Exemplo de resposta HTTP	6
Figura 5 – Visão Geral do Protocolo HTTP	7
Figura 6 – HTTP com (a) múltiplas conexões e requisições sequenciais. (b) Uma conexão persistente e requisições sequenciais. (c) Uma conexão persistente e requisições em pipeline	11
Figura 7 – Multiplexação de fluxos no HTTP/2 (a) dois fluxos separadas (b) fluxos multiplexadas	17
Figura 8 – Comparação entre modelo de <i>web</i> original e modelo utilizando AJAX	20

Lista de Tabelas

Tabela 1 – Impacto do desempenho de <i>website</i> na receita.	2
--	---

Lista de Quadros

Quadro 1 – Etiquetas para cabeçalhos HTTP.	8
Quadro 2 – Métodos HTTP.	9
Quadro 3 – Códigos de estado HTTP.	9
Quadro 4 – Mudanças introduzidas no HTTP/1.1.	12
Quadro 5 – Mudanças introduzidas no HTTP/2.	19

Lista de Algoritmos

Lista de Abreviaturas e Siglas

AJAX	Asynchronous JavaScript and XML
ASCII	American Standard Code for Information Interchange
CDN	Content Delivery Network
CSS	Cascade Style Sheet
DNS	Domain Name System
DOM	Document Object Model
IETF	Internet Engineering Task Force
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPbis	Hypertext Transfer Protocol bis
HTTPS	Hypertext Transfer Protocol Secure
HTTP-WG	Hypertext Transfer Protocol Working Group
JS	JavaScript
KB	Kilobytes
MIME	Multi-Purpose Internet Mail Extensions
ms	Milissegundo
PNG	Portable Network Graphics
RFC	Request for Comments
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XHR	XMLHttpRequest
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

Sumário

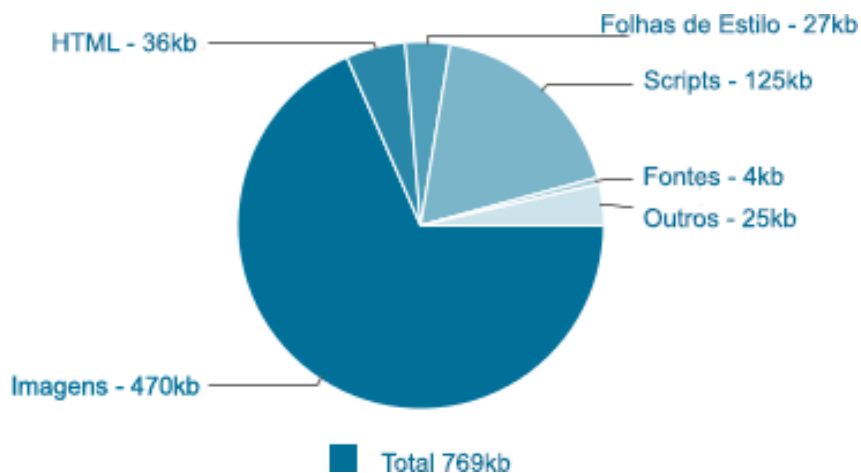
1 – Introdução	1
1.1 Motivação	3
1.2 Objetivos	3
2 – Fundamentação Teórica	5
2.1 O Protocolo HTTP	5
2.2 Visão Geral	7
2.3 HTTP/1.0 VS HTTP/1.1	9
2.4 HTTP/1.1 VS HTTP/2	15
2.5 AJAX	19
2.6 Web 2.0	21
3 – Trabalhos Relacionados	22
3.1 <i>High Performance Web Sites</i>	22
3.1.1 Regra 1: Faça menos requisições HTTP	22
3.1.2 Regra 2: Use Redes de Entrega de Conteúdo (CDN)	23
3.1.3 Regra 3: Adicione cabeçalhos de expiração	23
3.1.4 Regra 4: Utilize <i>gzip</i> no componentes	23
3.1.5 Regra 5: Coloque folhas de estilo no topo da página	24
3.1.6 Regra 6: Coloque <i>scripts</i> no fim da página	24
3.1.7 Regra 7: Evite expressões CSS	24
3.1.8 Regra 8: Faça arquivo <i>JavaScripts</i> e folhas de estilo externos	24
3.1.9 Regra 9: Reduza o número de pesquisas de DNS	25
3.1.10 Regra 10: Mimifique arquivo <i>JavaScript</i>	25
3.1.11 Regra 11: Evite redirecionamentos	25
3.1.12 Regra 12: Remova <i>scripts</i> duplicados	26
3.1.13 Regra 13: Configure <i>ETags</i>	26
3.1.14 Regra 14: Habilite <i>cache</i> para AJAX	26
3.2 <i>Even Faster Web Sites</i>	26
3.2.1 Entendendo performance em AJAX	27
3.2.2 Criando aplicações <i>web</i> responsivas	27
3.2.3 Dividindo carga inicial	28
3.2.4 Carregando <i>scripts</i> sem bloqueios	28
3.2.5 Lidando com <i>scripts</i> assíncronos	29
3.2.6 Posicionando blocos de <i>scripts</i> em linha	29
3.2.7 Escrevendo <i>JavaScripts</i> eficientes	29

3.2.8	Escalando usando <i>Comet</i>	30
3.2.9	Indo além do <i>gzip</i>	30
3.2.10	Otimizando imagens	30
3.2.11	Quebrando domínios dominantes	31
3.2.12	Entregando o documento cedo	31
3.2.13	Usando <i>Iframes</i> com moderação	31
3.2.14	Simplificando seletor CSS	32
4	– Metodologia	33
4.1	Delineamento da pesquisa	33
4.2	Coleta de dados	33
5	– Análise de Resultados	34
5.1	Situação atual	34
5.2	Análise dos dados coletados	34
6	– Conclusão	35
	Referências	36
	 Apêndices	 38
	APÊNDICE A–Nome do Apêndice	39
	APÊNDICE B–Nome do Apêndice	40
	 Anexos	 41
	ANEXO A–Nome do Anexo	42
	ANEXO B–Nome do Anexo	43

1 Introdução

Desde que a *World Wide Web* foi proposta pelo cientista e pesquisador britânico Tim Bernes-Lee em 1989 ([CONNOLLY, 2000](#)), as páginas *web* vêm mudando de maneira acelerada. Nos últimos 4 anos, o tamanho médio de uma página *web* passou de 769kB para 2061kB, um expressivo aumento de 168%. Esse fato pode ser percebido observando os gráficos nas [Figura 1](#) e [Figura 2](#), gerados com a ajuda do *website* HTTP Archive¹. O primeiro foi gerado com dados de 15 de Abril de 2011 e o segundo com dados de 15 de Abril de 2015 e os dois mostram a média de *bytes* por página por tipo de conteúdo nas páginas *web*. Mas a informação mais relevante está localizado na parte debaixo do gráfico e mostra o tamanho médio de uma página *web* nas respectivas dadas.

Figura 1 – Média de Bytes por Página por Tipo de Conteúdo em 2011

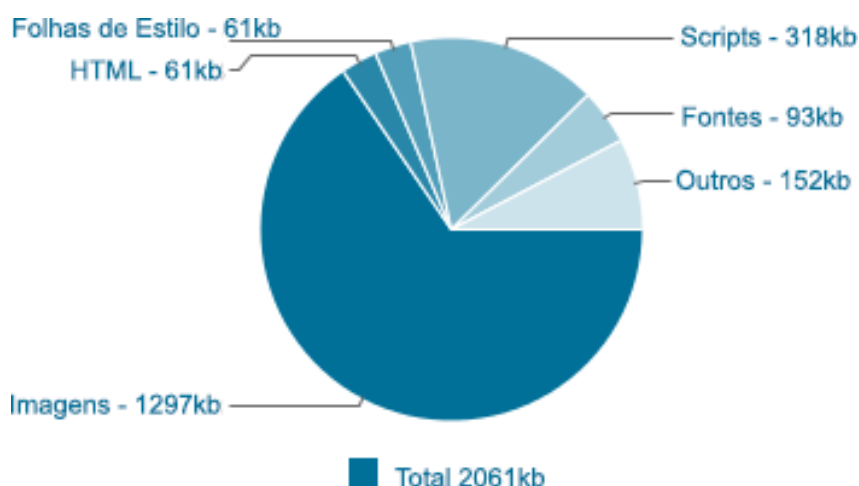


Fonte: Adaptado de [Archive \(2015a\)](#)

Apesar dessa grande mudança no tamanho das páginas (e consequentemente dos *websites*), a maneira como *websites* são entregues dos servidores para os clientes não sofreu nenhuma alteração desde 1999, ano de lançamento da RFC 2616 que especificou o HTTP/1.1 ([GROUP, 1999](#)). Como explicado por ([TANENBAUM, 2011](#)), o HTTP é um protocolo simples na camada de aplicação de requisições e respostas que é executado em cima da camada de transporte do protocolo TCP. O HTTP ficou famoso por ser fácil de entender e implementar e ao mesmo tempo cumprir sua função de transferência de recursos em rede com um bom desempenho. Contudo, o aumento no tamanho dos *websites* começou a fazer com que o tempo de resposta das páginas *web* ficasse muito grande e, como mudanças no HTTP seriam muito difíceis, pois teriam de envolver esforços de muitas partes interessadas na *World Wide Web* (como fabricantes de navegadores e

¹<http://httparchive.org/>

Figura 2 – Média de Bytes por Página por Tipo de Conteúdo em 2015



Fonte: Adaptado de [Archive \(2015b\)](#)

mantenedores de servidores), os desenvolvedores passaram a ter de criar outras formas de resolver esse problema.

Técnicas de otimização de desempenho passaram a ser estudadas e implementadas por muitas empresas que queriam ter seus *websites* entregues mais rapidamente a seus clientes. Por muitos anos, a grande maioria dessas empresas focou seus esforços em otimizações para o *back-end*, principalmente para os seus servidores, o que hoje em dia pode ser considerado um erro. Como explicado por [Souders \(2007, p. 5\)](#) no que ele chamou de "Regra de Ouro do Desempenho": "Apenas 10-20% do tempo de resposta do usuário final são gastos baixando o documento HTML. Os outros 80-90% são gastos baixando todos os componentes da página."

Dessa forma ficou claro que técnicas de otimização de desempenho para o *front-end* dos *websites* deveriam se tornar prioridade quando procura-se melhorar o tempo de resposta para o usuário final. Para entender o quão importante esse tempo de resposta se tornou para empresas que baseiam seus negócios em vendas de serviços ou produtos na Internet, basta observar os dados da tabela [Tabela 1](#) expostos por Steve Souders na conferência Google I/O de 2009:

Tabela 1 – Impacto do desempenho de *website* na receita.

Empresa	Piora no tempo de resposta	Consequencia
Google Inc.	+500ms	-20% de tráfego
Yahoo Inc.	+400ms	-5% à -9% de tráfego
Amazon.com Inc.	a cada +100ms	-1% de vendas

Fonte: ([SOUDERS, 2009b](#))

Steve Souders tornou-se um grande evangelizador da área de otimização de

desempenho de *front-end* de *websites*. Em seus livros, *High Performance Websites* (SOU-[DERS, 2007](#)) e *Even Faster Websites* (SOU-[DERS, 2009a](#)) ele ensina técnicas de como tornar *websites* mais rápidos, focando nos componentes das páginas. E em 2012, ele lançou seu terceiro livro, *Web Performance Daybook* (SOU-[DERS, 2012](#)), como um guia para desenvolvedores que trabalham com otimização de desempenho de *websites*.

1.1 Motivação

Após mais de 15 anos sem mudanças, o protocolo HTTP (finalmente) receberá uma atualização. A nova versão do protocolo, chamada de HTTP2, teve sua especificação aprovada no dia 11 de Fevereiro de 2015, (GROU-[P, 2015b](#)), e deverá começar a ser implantada, a partir de 2016. Muitas mudanças foram feitas com o objetivo de melhorar o desempenho e a segurança da *web*. Além disso, o HTTP2 foi desenvolvido para ser compatível com suas versões anteriores, não sendo necessárias mudanças em servidores e aplicações antigos para funcionar baseados no novo protocolo.

Com as novas funcionalidades do HTTP2 a caminho algumas coisas devem mudar na área de otimização de desempenho de *websites*. Como pode ser percebido em (STEN-[BERG, 2014](#)), o HTTP2 foi desenvolvido para melhorar o desempenho de todos os *websites* e aplicações *web*, e não apenas dos poucos que podem aplicar técnicas de otimização. Isso torna difícil uma previsão o resultado da aplicação de técnicas desenvolvidas para os protocolos HTTP/1.0 e HTTP/1.1. Acredita-se que algumas das técnicas antigas podem não apenas não melhorar o desempenho dos *websites* como podem acabar piorando o tempo de resposta para o usuário final.

No decorrer dos próximos anos o HTTP2 deve seguir o mesmo caminho do HTTP/1.1 e se tornar o protocolo mais utilizado da *web*. Apesar de todo o esforço do HTTPbis (grupo responsável por desenvolver a especificação do HTTP2) em desenvolver um protocolo que garanta o melhor desempenho de *websites* e aplicações, sempre é possível ser mais rápido se as medidas certas forem tomadas.

1.2 Objetivos

Este trabalho tem como objetivo analisar o comportamento de técnicas de otimização de desempenho de *websites* desenvolvidas para os protocolos HTTP/1.0 e HTTP/1.1 quando aplicadas a *websites* usando o protocolo HTTP2 e, se necessário, propor técnicas específicas para o novo protocolo.

Para realização do objetivo principal, os seguintes objetivos específicos foram determinados:

1. Fazer uma análise comparativa das versões do protocolo HTTP
2. Avaliar os ganhos de desempenho das técnicas propostas por Steve Souders ao aplicá-las ao HTTP2
3. Se necessário, propor novas técnicas de otimização de desempenho de *websites* específicas para o HTTP2

2 Fundamentação Teórica

Este capítulo apresenta os principais conceitos relacionados ao funcionamento e à utilização do protocolo HTTP, bem como traça comparações entre suas versões. O bom entendimento do funcionamento do protocolo, bem como do uso que é feito dele na *web*, é essencial para a compreensão das técnicas de otimização de desempenho que serão apresentadas e avaliadas neste trabalho.

2.1 O Protocolo HTTP

Nas década de 1970 e 1980 a comunidade científica estava trabalhando arduamente para fazer importantes descobertas. O problema é que por causa das distâncias geográficas era muito difícil de compartilhar informações e isso atrasava os tão esperados avanços da ciência. Apesar dos avanços na área de rede de computadores, essas não conseguiam resolver o problema dos cientistas. Existiam várias redes espalhadas pelos Estados Unidos e Europa, mas como cada uma possuía topologia e sistemas operacionais diferentes elas não se comunicavam entre si, limitando a troca de informação aos computadores conectados na mesma rede. Visando resolver esse problema em 1989, o cientista do CERN¹, Tim Berners-Lee, propôs uma rede de computadores global para ajudar a comunidade científica a compartilhar o conhecimento gerado em diferentes partes do mundo, acelerando assim o desenvolvimento tecnológico. Para isso Tim precisaria de um formato padrão de arquivos que pudesse ser executado em qualquer computador, assim desenvolveu uma linguagem de marcação de hipertextos, que ficou conhecida como HTML, e um programa para executar esses arquivos, um navegador *web*. Esses arquivos precisariam ser servidos sempre que fossem requisitados, então Berners-Lee criou um servidor *web*. E por último era necessário um protocolo para conectar o navegador ao servidor, então foi desenvolvido o protocolo para transferência de hipertextos, chamado de HTTP. Ao final de 1990, Tim Berners-Lee já havia terminado de desenvolver todos os requisitos necessários para sua rede, e decidiu chamá-la de *World Wide Web*, hoje conhecida apenas como *web*.

O protocolo HTTP baseava-se na troca de informações por meio de mensagens formadas por caracteres ASCII, ou seja, mensagens de texto. O navegador *web* enviava uma requisição, como a exemplificada na [Figura 3](#), e o servidor retornava uma resposta, como a da [Figura 4](#). O protocolo era simples.

¹Organização Europeia para Pesquisa Nuclear, conhecida como CERN na sigla em inglês, é uma organização de pesquisas em física e engenharia que realiza experimentos para tentar compreender as estruturas fundamentais do universo.

Figura 3 – Exemplo de requisição HTTP

```
GET /index.html HTTP/1.1
Host: www.example.com
```

Figura 4 – Exemplo de resposta HTTP

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Content-Type: text/html

<html>
<head>
  <title>An Example Page</title>
</head>
<body>
  Hello World, this is a very simple HTML document.
</body>
</html>
```

Apesar de ter sido utilizado desde 1990, o HTTP recebeu sua primeira documentação oficial em 1991, quando recebeu um número de versão e passou a ser chamado de HTTP/0.9. A ideia é explicada por [Grigorik \(2013\)](#) da seguinte maneira:

- Uma conexão TCP era aberta entre o navegador (chamado também de cliente) e o servidor
- A requisição do cliente era uma cadeia simples de caracteres ASCII
- A resposta do servidor era uma torrente de caracteres ASCII que representava um arquivo HTML
- A conexão era fechada após a transferência do documento

Com o passar dos anos, a *World Wide Web* de Tim Berners-Lee cresceu rapidamente e isso fez com que o uso do HTTP aumentasse muito em pouco tempo. Viu-se a necessidade de um protocolo mais robusto e estruturado, mas que mantivesse a simplicidade do HTTP/0.9, pois esta era vista como o motivo por trás do sucesso do protocolo. Então o IETF ² passou a coordenar a criação de especificações para o HTTP e criou o HTTP-WG ³ que tinha a função definir as especificações das versões seguintes do protocolo. Em 1996 foi definido o HTTP/1.0 ([GROUP, 1996](#)), em 1999 o HTTP/1.1 ([GROUP, 1999](#)) e em 2015 foi aprovada a especificação para o HTTP2 ([GROUP, 2015b](#)).

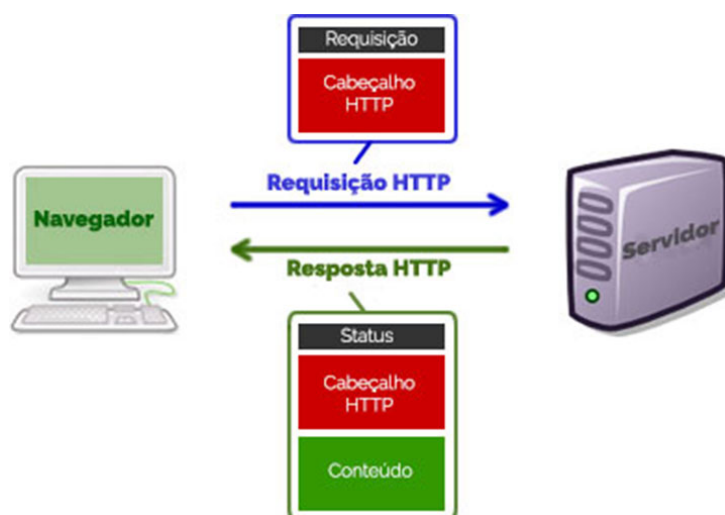
²Força Tarefa de Engenharia da Internet, conhecida como IETF na sigla em inglês, é uma comunidade aberta formada por profissionais que se preocupam com a evolução da Internet e por isso procuram formalizar as técnicas utilizadas na rede global de computadores.

³Grupo de Trabalho do HTTP, conhecido como HTTP-WG na sigla em inglês, é um grupo formado por profissionais escolhidos pelo IETF

2.2 Visão Geral

Como descrito por [Tanenbaum \(2011, p. 683\)](#), o HTTP é um simples protocolo de requisições e respostas. As requisições e respostas são compostas por um cabeçalho e um conteúdo, e são enviadas do cliente para o servidor. O HTTP é um protocolo independente de estado, ou seja, cada tupla requisição-resposta pode ser tratada de maneira independente, sem que as anteriores e futuras interfiram nela. O modelo, ilustrado na [Figura 5](#), é bem simples e direto, fácil de ser replicado.

Figura 5 – Visão Geral do Protocolo HTTP



Fonte: Adaptado de [Saenz \(2014\)](#)

O HTTP foi desenvolvido para funcionar na camada de Aplicação do protocolo TCP, conectando as ações do usuário à camada de Apresentação. Contudo, de acordo com [Tanenbaum \(2011, p. 684\)](#), ele se transformou em um protocolo da camada de Transporte, criando uma maneira de processos se comunicarem através de diferentes redes. Hoje em dia não são apenas os navegadores *web* que utilizam o protocolo HTTP para se comunicar com servidores, aplicações como tocadores de mídias, anti-vírus, programas de fotos, dentre outras utilizam o HTTP para trocar informações de maneira simples, rápida e eficiente.

Os cabeçalhos HTTP definem características desejadas ou esperadas pelas aplicações e servidores, como tipo de codificação de caracteres ou tipo de compressão dos dados. Existem várias etiquetas padrões que podem ser utilizadas nos cabeçalhos e os desenvolvedores podem ainda criar etiquetas próprias para serem utilizadas dentro das aplicações - por definição, caso um cliente ou servidor receba uma etiqueta que não reconhece ele simplesmente a ignora. No [Quadro 1](#) são apresentadas algumas das etiquetas mais utilizadas, mas existem muitas outras que não foram citadas e que podem variar com a versão do protocolo.

Quadro 1 – Etiquetas para cabeçalhos HTTP.

Etiqueta	Tipo	Conteúdo
<i>Accept</i>	Requisição	Tipo de páginas que o cliente suporta
<i>Accept-Encoding</i>	Requisição	Tipo de codificação que o cliente suporta
<i>If-Modified-Since</i>	Requisição	Data e hora para checar atualidade do conteúdo
<i>Authorization</i>	Requisição	Uma lista de credenciais do cliente
<i>Cookie</i>	Requisição	Cookie definido previamente enviado para o servidor
<i>Content-Encoding</i>	Resposta	Como o conteúdo foi codificado (e.g. <i>gzip</i>)
<i>Content-Length</i>	Resposta	Tamanho da página em <i>bytes</i>
<i>Content-Type</i>	Resposta	Tipo de <i>MIME</i> da página
<i>Last-Modified</i>	Resposta	Data e hora que a página foi modificada pela última vez
<i>Expires</i>	Resposta	Data e hora quando a página deixa de ser válida
<i>Cache-Control</i>	Ambas	Diretiva de como tratar <i>cache</i>
<i>ETag</i>	Ambas	Etiqueta para o conteúdo da página
<i>Upgrade</i>	Ambas	O protocolo para o qual o cliente deseja alterar

Fonte: Adaptado de [Tanenbaum \(2011\)](#)

O conteúdo de uma resposta HTTP pode assumir diferentes formatos (como, HTML, CSS e JavaScript) e a definição desse formato é feita com a etiqueta *Content-Type* enviada no cabeçalho de resposta. O conteúdo é a maior parte de uma resposta HTTP e os desenvolvedores devem se esforçar para reduzi-lo ao máximo, garantindo que a comunicação de dados seja rápida e eficiente.

Por executar em cima do protocolo TCP, o HTTP precisa que uma conexão TCP seja aberta para poder realizar a troca de dados entre o cliente e o servidor. Como essa conexão é gerenciada depende da versão do protocolo. Após a abertura da conexão, a requisição pode ser enviada. Na primeira linha da requisição, são definidas a versão do protocolo e a operação que será realizada. Apesar de ter sido criado apenas para recuperar páginas *web* de um servidor, o HTTP foi intencionalmente desenvolvido de forma genérica, possibilitando a extensibilidade do seu uso. Sendo assim, o protocolo suporta diferentes operações, chamadas de métodos, além da tradicional requisição de páginas *web*. A lista completa de métodos com suas descrições pode ser vista no [Quadro 2](#). Vale ressaltar que esses métodos são *case sensitive*, ou seja, o método *get*, por exemplo, não existe.

Dos métodos citados no [Quadro 2](#), GET e POST são os mais utilizados pelos navegadores *web* e serão os mais utilizados neste trabalho. O método GET é utilizado para recuperar informações do servidor e o método POST para enviar informações para o servidor.

Sempre que uma requisição é enviada por um cliente, este recebe uma resposta, mesmo que a requisição não possa ser cumprida pelo servidor, que então enviará uma resposta comunicando o cliente do ocorrido. Na primeira linha do cabeçalho de resposta

Quadro 2 – Métodos HTTP.

Método	Descrição
GET	Ler página <i>web</i>
HEAD	Ler cabeçalho de página <i>web</i>
POST	Anexar à página <i>web</i>
PUT	Armazenar página <i>web</i>
DELETE	Remover página <i>web</i>
TRACE	Imprimir requisição de entrada
CONNECT	Conectar através de um <i>proxy</i>
OPTIONS	Listar opções para uma página <i>web</i>

Fonte: Adaptado de [Tanenbaum \(2011\)](#)

se encontra o código do estado da resposta em formato numérico de três dígitos. O primeiro dígito deste código define a qual sub-grupo ele pertence. O [Quadro 3](#) mostra os sub-grupos existentes e o significado de cada um deles. Cada um destes sub-grupos possui vários códigos com diferentes significados e, com a evolução do protocolo, mais códigos foram sendo inseridos para lidar com necessidades específicas.

Quadro 3 – Códigos de estado HTTP.

Código	Significado	Exemplo
1xx	Informação	100 = servidor concorda em lidar com requisição do cliente
2xx	Sucesso	200 = sucesso na requisição; 204 = nenhum conteúdo presente
3xx	Redirecionamento	301 = página foi movida; 304 = <i>cache</i> ainda é válida
4xx	Erro do cliente	403 = página proibida; 404 = página não encontrada
5xx	Erro do servidor	500 = erro interno de servidor; 503 = tente novamente mais tarde

Fonte: Adaptado de [Tanenbaum \(2011\)](#)

A *cache* é uma característica importante do HTTP. O protocolo foi construído com suporte integrado para lidar com este requisito de desempenho. Os clientes e os servidores conseguem gerenciar *caches* com a ajuda dos cabeçalhos de requisição e resposta, mas o tamanho da *cache* é definido pelo navegador. O problema destas memórias locais é saber o momento de utilizar os dados armazenados nelas ou de pedir novos dados ao servidor.

As versões do protocolo HTTP foram corrigindo falhas identificadas quando ele passou a ser utilizado em larga escala. Mas, apesar das mudanças, a essência continua a mesma: um simples protocolo de requisição e resposta.

consulta para desenvolvedores que queriam utilizar o protocolo em suas aplicações. Sendo assim, apesar de todo o debate por trás da especificação do HTTP/1.0 (GROUP, 1996), aprovada em 1996, o documento apenas explicou os usos comuns do protocolo, mas não chegou a definir padrões de como ele deveria ser aplicado, como explica (KRISHNAMURTHY et al., 1999). Por isso, logo após a sua aprovação, o HTTP-WG já começou a trabalhar na (GROUP, 1999), para poder corrigir os erros existentes no HTTP/1.0 com a criação de uma nova versão, o HTTP/1.1.

Várias funcionalidades importantes foram adicionadas no HTTP/1.1, e o HTTP-WG teve o cuidado de manter a compatibilidade entre as versões do protocolo, levando em consideração que o HTTP/1.0 já era amplamente utilizado e não podia se esperar que todos os *websites* e aplicações se adaptassem de uma hora para outra. Esse fato também levou o HTTP-WG a criar um protocolo que fosse flexível a mudanças futuras (lembrando que no HTTP todas as etiquetas que um cliente ou servidor não reconhecem são simplesmente ignoradas). Considerando-se a intensão de se criar um protocolo que possa se estender de acordo com as necessidades do ambiente, as primeiras mudanças no HTTP/1.1 que valem ser citadas foram a criação de duas novas etiquetas para cabeçalhos, *Upgrade* - uma maneira do cliente informar qual versão do protocolo ele suporta - e *Via* - que define uma lista dos protocolos suportados pelos clientes ao longo do caminho de uma transmissão.

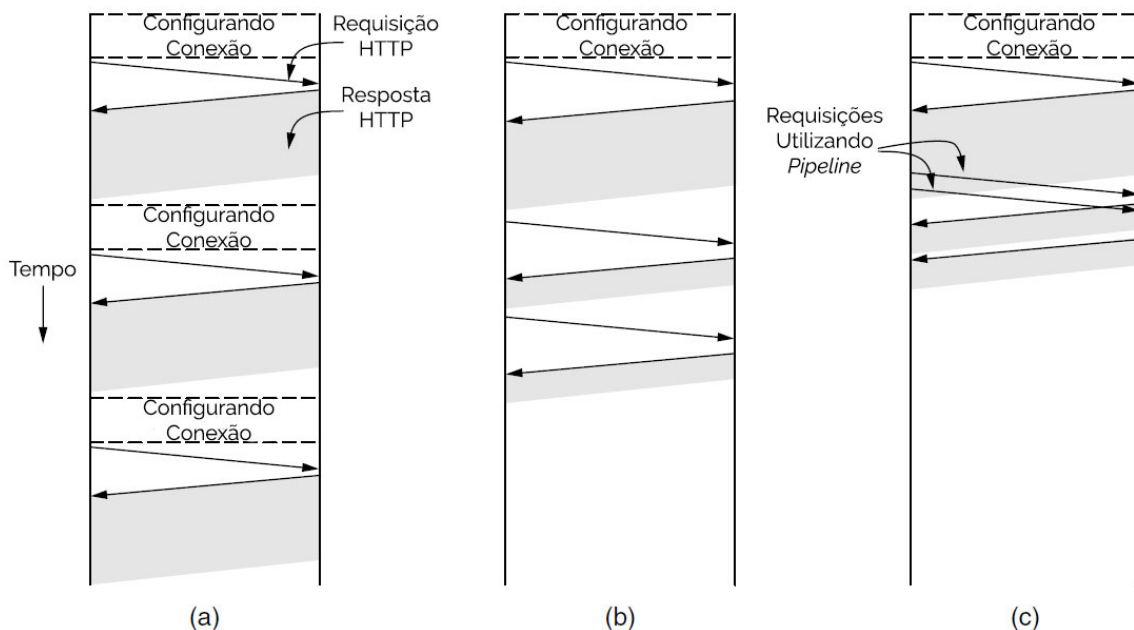
Como dito anteriormente, o protocolo HTTP foi construído com suporte integrado para *cache*. Mas o mecanismo de *cache* do HTTP/1.0 era muito simples e não permitia que o cliente ou o servidor definisse instruções diretas de como a memória deveria ser utilizada. O HTTP/1.1 tentou corrigir esse problema com a criação de novas etiquetas para cabeçalhos. A primeira delas é a *ETag*, que define uma cadeia de caracteres única para um arquivo. Além do próprio conteúdo, essa cadeia utiliza a data e a hora da última modificação no arquivo, logo pode ser utilizada para verificar se dois arquivos são idênticos. O HTTP/1.1 também definiu novas etiquetas condicionais para complementar a já existente *If-Modified-Since*. As etiquetas *If-None-Match*, *If-Match* e *If-Unmodified-Since*, passaram a poder ser usadas para verificar se arquivos em *cache* estão atualizados ou não. Uma das mudanças mais significativas no mecanismo de *cache*, foi a etiqueta *Cache-Control* que possibilita definir novas diretrizes para o uso da *cache*, como tempo de expiração relativos e arquivos que não devem ser armazenados.

O HTTP/1.0 tinha muitos problemas em gerenciar a largura de banda. Não era possível enviar partes de arquivos, sendo assim mesmo se o cliente não precisasse de um arquivo inteiro, ele teria de recebe-lo. No HTTP/1.1, foram então criados a etiqueta *Range*, o tipo *MIME multipart/byteranges* e o tipo de compressão *Chunked* para que cliente e servidores pudessem trocar mensagens com partes de arquivos. Para complementar esse novo mecanismo foi incluído um novo código de resposta, 100, que informava um

cliente que o corpo de sua requisição deve ser enviado. Além de poder enviar partes de arquivos, o HTTP/1.1 garante a compressão dos dados durante todo o caminho da transmissão. Nesse sentido, foi incluída a etiqueta *Transfer-Encoding*, que complementa a *Content-Encoding* indicando qual codificação foi utilizada na transmissão ponto a ponto.

O modelo original do protocolo HTTP utilizava uma conexão TCP para cada transmissão. Esse processo era extremamente danoso para o desempenho de *websites* e aplicações, pois se gastava muito tempo na criação e configuração de novas conexões e nos momentos iniciais da conexão (quando, por definição, ela é mais lenta). Para corrigir esse problema, o HTTP/1.1 define conexões persistentes como seu padrão. Conexões persistentes permitem que clientes e servidores assumam que uma conexão TCP continuará aberta após a transmissão de dados, e que esta poderá ser utilizada para uma nova transmissão. Além disso, foi definido que o HTTP/1.1 utilizaria *pipeline*, isto quer dizer que clientes não precisam aguardar a resposta de uma requisição para enviarem uma nova requisição, como era o padrão do HTTP/1.0. Os ganhos com essas novas técnicas podem ser notados na [Figura 6](#).

Figura 6 – HTTP com (a) múltiplas conexões e requisições sequenciais. (b) Uma conexão persistente e requisições sequenciais. (c) Uma conexão persistente e requisições em pipeline



Fonte: Adaptado de [Tanenbaum \(2011\)](#)

Uma funcionalidade desejada no HTTP/1.1 era a de poder fazer requisições para outros servidores além daquele da página principal sendo acessada. Dessa forma desenvolvedores podem hospedar arquivos CSS e JavaScript em um servidor e imagens em outro por exemplo. Isto se tornou possível com a adição da etiqueta *Host*, com a qual

o cliente pode definir qual é o caminho do servidor que será utilizado na requisição. Caso a etiqueta *Host* não esteja definida no cabeçalho, é assumido que o caminho do servidor é o caminho da página principal.

Como concluem [Krishnamurthy et al. \(1999\)](#): os protocolos HTTP/1.0 e HTTP/1.1 diferem em diversas maneiras. Enquanto muitas dessas mudanças têm o objetivo de melhorar o HTTP, a descrição do protocolo mais do que triplicou de tamanho, e muitas dessas funcionalidades foram introduzidas sem testes em ambientes reais. Esse aumento de complexidade causou muito trabalho para os desenvolvedores de clientes e servidores *web*.

No [Quadro 4](#) encontra-se um resumo das mudanças introduzidos no protocolo HTTP/1.1.

Quadro 4 – Mudanças introduzidas no HTTP/1.1.

Cabeçalhos	
Etiqueta	Descrição
<i>Accept-Encoding</i> ⁴	Lista de codificações aceitas.
<i>Age</i>	O tempo que o conteúdo está salvo no <i>proxy</i> em segundos.
<i>Cache-Control</i>	Notifica todos os mecanismos de cache do cliente ao servidor se o conteúdo deve ser salvo.
<i>Connection</i>	Controla a conexão atual.
<i>Content-MD5</i>	Codificação binária de base-64 para verificar conteúdo da resposta.
<i>ETag</i>	Identificador único de um conteúdo.
<i>Host</i>	Indica o endereço e a porta do servidor que deve ser utilizado pela mensagem.
<i>If-Match</i>	Realize a ação requisitada se, e somente se, o conteúdo do cliente é igual ao conteúdo do servidor.
<i>If-None-Match</i>	Retorna código 304 se o conteúdo não foi modificado.
<i>If-Range</i>	Se o conteúdo não foi modificado, envie a parte solicitada, se não, envie o conteúdo novo.
<i>If-Unmodified-Since</i>	Envie o conteúdo se, e somente se, ele não foi modificado na data esperada.

<i>Proxy-Authentication</i>	Pede uma requisição de autenticação de um <i>proxy</i> .
<i>Proxy-Authorization</i>	Credenciais de autorização para se conectar a um <i>proxy</i> .
<i>Range</i>	Faz a requisição de apenas uma parte de um conteúdo.
<i>Trailer</i>	Indica que o grupo de cabeçalhos está presente em uma mensagem.
<i>Transfer-Encoding</i>	A forma de codificação usada para se transferir o conteúdo para o usuário.
<i>Upgrade</i>	Pede para o servidor atualizar para outro protocolo.
<i>Vary</i>	Informa quais partes do cabeçalho de requisição devem ser levadas em conta para descobrir se um recurso em <i>cache</i> deve ser utilizado ou se este recurso deve ser solicitado no servidor.
<i>Via</i>	Informa o servidor de proxies pelos quais a requisição passou.
<i>Warning</i>	Mensagem genérica de cuidado para possíveis problemas no corpo da mensagem.
<i>WWW-Authenticate</i>	Indica tipo de autenticação que deve ser utilizada para acessar entidade requerida.
Métodos	
Método	Descrição
<i>OPTIONS</i>	Requer informações sobre os recursos que o servidor suporta.
Estados ⁵	
Código	Descrição
100	Confirma que o servidor recebeu o cabeçalho de requisição e que o cliente deve continuar a enviar a mensagem desejada.
206	O servidor está entregando apenas uma parte de um conteúdo por causa da etiqueta de <i>Range</i> na requisição do cliente.
300	Indica as múltiplas opções disponíveis para o cliente.

409	Indica que a requisição não pôde prosseguir por causa de um conflito.
410	Indica que o conteúdo requisitado não está mais disponível e não estará disponível no futuro.
Diretivas	
Diretiva	Descrição
<i>chunked</i>	Utilizado para envio de conteúdo em partes.
<i>max-age</i>	Determina qual é o tempo máximo que um conteúdo deve ficar salvo em <i>cache</i> .
<i>no-store</i>	Indica que o conteúdo não deve ser salvo em <i>cache</i> .
<i>no-transform</i>	Indica, que o conteúdo não deve ser modificado por <i>proxies</i> .
<i>private</i>	Indica que o conteúdo não deve ser acessado sem autenticação.
Tipos de MIME	
Tipo	Descrição
<i>multipart/byteranges</i>	Indica que o conteúdo que está sendo enviado é apenas uma parte de um todo.
Funcionalidades	
Nome	Descrição
<i>Content negotiation</i>	Escolhe a melhor representação disponível para um conteúdo.
<i>Persistent connection</i>	Após o termino de uma requisição HTTP a conexão continua aberta e pode ser utilizada por outras requisições.
<i>Pipeline</i>	O cliente não precisa esperar que a resposta de uma requisição retorne antes de enviar outra requisição.

⁴A etiqueta *Accept-Encoding* já existia no HTTP/1.0, mas era pouco utilizada por causa da sua especificação consufa, por isso por redefinida na versão 1.1.

⁵Além dos citados ainda foram adicionados outros estados no HTTP/1.1, mas a lista ficaria muito extensa. Logo foram descritos os estados que podem influenciar no desempenho do *front-end*

2.4 HTTP/1.1 VS HTTP/2

O HTTP/1.1 é robusto e flexível, e isso permitiu que passasse a ser utilizado em aplicações diversas de maneira eficiente. Ao longo dos anos o IETF acrescentou algumas extensões ao protocolo para corrigir erros pontuais, mas o HTTP atendia as necessidades da rede mundial de computadores.

No início do século XXI, os *websites* começaram a mudar. Eles se tornaram mais complexos e consequentemente maiores, muitas fontes e folhas de estilo eram utilizadas e cada página passou a possuir vários arquivos de JavaScript. Além disso, eles deixaram de ser estáticos e passaram a responder dinamicamente às ações dos usuários. Hoje em dia, muitas requisições HTTP são necessárias para se montar uma página *web* e essas requisições podem ser longas e demoradas. Esse aumento da complexidade das páginas *web* fez com que o HTTP/1.1 se tornasse um gargalo de desempenho para os *websites*, então em 2007 o IETF formou o grupo HTTPbis (onde o "bis" quer dizer "de novo" em latim). Mas o grupo só começou as discussões sobre a nova versão do protocolo em 2012, terminando de redigir as especificações em 2014. Após revisões, a especificação oficial da nova versão do HTTP foi aprovada no início do ano de 2015 e deve começar a ser utilizada em 2016. A nova versão do HTTP passou a se chamar HTTP/2. As casas decimais, que eram comuns na nomenclatura das outras versões, deixaram de existir, agora, caso mudanças sejam necessárias, serão lançadas novas versões do protocolo ao invés de sub-versões.

O HTTP/2 têm o objetivo de corrigir o problema de latência existente na versão anterior. Apesar do sistema de *pipeline*, o HTTP/1.1 é muito sensível à latência, ou seja, apesar de conseguir uma grande quantidade de dados existem problemas quanto ao tempo de viagem das requisições e respostas. Isso acontece porque o *pipeline* do HTTP/1.1 é muito difícil de ser gerenciado e muitas vezes fecha conexões que deveriam ter ficado abertas. O problema é tão grande que [Stenberg \(2014\)](#) afirma que, mesmo nos dias de hoje, muitos navegadores *web* preferem desativar o *pipeline*. Para corrigir este problema, o HTTP/2 propõe mudanças na forma como as informações são trocadas entre clientes e servidores. Assim como ocorreu na mudança da versão 1.0 para a versão 1.1, o novo protocolo não deve alterar nenhum paradigma já existente. As aplicações que utilizam o HTTP/1.1 devem continuar funcionando no HTTP/2, os formatos de arquivos, as URL e as URI devem ser mantidos e o usuário final não deve ter de fazer nada para poder aproveitar das melhorias do novo protocolo. Com isso, para tentar criar um protocolo que funcionasse no mundo real tanto quanto no teórico, o HTTPbis decidiu se inspirar no protocolo SPDY. O SPDY ([INC., 2012](#)) é um protocolo para troca de dados entre clientes e servidores *web*. Ele foi criado pela Google em 2010 como uma alternativa ao HTTP. O SPDY tem como objetivo aumentar a velocidade dos *websites* e aplicações que o utilizam, melhorando o desempenho da *web* como um todo. A escolha

de basear o HTTP/2 neste protocolo, veio do fato dele já vir sendo utilizado por várias aplicações ao longo dos anos e ter se provado um conceito funcional e eficiente.

A primeira mudança no HTTP/2 está na forma como ele escreve suas requisições e respostas. Em suas versões anteriores, o protocolo optou por utilizar o formato ASCII para estruturar suas requisições e respostas, mas era difícil separar as partes dos cabeçalhos e tratar espaços em branco indesejados. Para resolver esse problema o HTTP/2 é um protocolo binário. Assim é mais simples quebrar requisições e respostas em quadros, compará-los e comprimir as informações. Entre as desvantagens dessa representação binária estão o fato de que os cabeçalhos HTTP não serão mais compreensíveis sem a ajuda de ferramentas de visualização de pacotes binários e que a depuração do protocolo dependerá de analisadores de pacotes.

Outro problema muito discutido entre os especialistas em desenvolvimento para a *web* é a segurança da rede. Para garantir a proteção de seus usuário, alguns *websites* e aplicações optam por utilizar serviços de rede seguro via TSL ⁶. O TLS é um protocolo que promove a segurança entre as partes envolvidas em comunicações de dados através de autenticações e criptografias. Quando o HTTP é utilizado em conjunto com o TLS ele recebe o nome de HTTPS. Mas como explica Tanenbaum (2011, p. 853), o HTTPS é simplesmente o protocolo HTTP, as diferenças estão no momento do transporte dos dados, quando o protocolo TLS realiza ações para garantir a segurança. Foi muito discutida a ideia de fazer o uso do TLS obrigatório no HTTP/2, mas isto iria forçar todos os *websites* e aplicações a se adaptarem para poderem se adequar ao protocolo. Então, ficou decidido que o uso do TLS continuaria opcional na nova versão.

Utilizando a representação binária, o HTTP/2 possibilita a multiplexação de fluxos de dados. Como explica Stenberg (2014), uma fluxo é uma associação lógica criação por uma sequencia de quadros. No HTTP/2, uma conexão possui vários fluxos e por isso vários componentes podem ser transferidos ao mesmo tempo. Para esse processo funcionar o protocolo multiplexa esses fluxos no momento do envio e as separa novamente na chegada. A Figura 7 ilustra a multiplexação de fluxos.

Um problema no HTTP/1.1 era a garantia de que um componente A já teria sido baixado quando outro componente B que depende de A fosse executado. Apesar de o HTML garantir isso, essa limitação impedia que a paralelização de *downloads* fosse maior. No HTTP/2 foram adicionados os mecanismos de prioridade e dependência. Eles tornaram possível indicar quais fluxos devem ser baixados primeiro e quais são suas interdependências. Dessa forma, os desenvolvedores de páginas *web* podem paralelizar ao máximo seus componentes e o protocolo cuidará de evitar erros.

A compressão de dados é um fator importante para o aumento de desempenho

⁶http://pt.wikipedia.org/wiki/Transport_Layer_Security

Figura 7 – Multiplexação de fluxos no HTTP/2 (a) dois fluxos separadas (b) fluxos multiplexadas



(a)



(b)

Fonte: Adaptado de [Stenberg \(2014\)](#)

do HTTP. Com o passar dos anos, as requisições e respostas aumentaram de tamanho, e os algoritmos de compressão existentes para o SPDY e o HTTPS não se provaram eficientes contra ataques de terceiros. Logo notou-se que este era um ponto crítico para a nova versão do protocolo, então o HTTPbis decidiu por criar o HPACK, que será o novo formato de compressão para cabeçalhos HTTP/2. Visando a robustez desse formato, foi criada uma nova especificação exclusiva para o HPACK, que detalha como ele funciona e como deve ser usado ([GROUP, 2015a](#)).

Assim como na versão anterior, o HTTP/2 possui mecanismos para lidar com a *cache*. As etiquetas existentes no HTTP/1.1 continuam a existir, mas uma nova funcionalidade foi adicionada na nova versão, o *Server Push*. O *Server Push* tem o objetivo de permitir que o cliente consiga componentes da maneira rápida, mesmo que seja a primeira vez que ele requisite aquele componente. A *Server Push* funciona da seguinte maneira: o cliente requisita um componente X. O servidor então sabe que é provável que este mesmo cliente vá requisitar o componente Y nos próximos instantes. Sendo assim o servidor pode enviar o componente Y antes mesmo de receber o pedido por ele. Essa funcionalidade é algo que o cliente deverá especificar explicitamente, mas existe grande expectativa quanto às melhorias que ela pode trazer. Além dessa melhoria no sistema de *cache*, o HTTP/2 inclui uma nova etiqueta para impedir o desperdício de banda de transmissão. Se o servidor começar a enviar um componente com um tamanho específico e perceber que aquele componente não é mais útil, ele pode cancelar o envio com a etiqueta *RST STREAM*, evitando que o fluxo de envio fique ocupado com dados desnecessários por muito tempo.

Caso um *website* ou uma aplicação deseje transferir o cliente para outro servidor sem ser o requisitado, ou até mesmo para outra porta, ele poderá utilizar a etiqueta Alt-Svc. Com essa etiqueta o servidor informa ao cliente para onde ele deve ir, então o cliente deve tentar se conectar de maneira assíncrona no caminho sugerido pelo Alt-Svc e utilizar aquela conexão apenas se ela se provar confiável. Essa etiqueta foi criada com a intenção de informar clientes que os dados requisitados estão disponíveis também em um servidor seguro.

A escolha pela utilização de envio através de fluxos, tem o objetivo de aumentar a velocidade do protocolo e a quantidade de dados que pode ser enviada de uma só vez. Cada um desses fluxos possui sua própria janela de envio independente, o que garantirá que se um fluxo falhe os outros continuem funcionando. Para impedir o envio de dados e parar todos os fluxos abertos, o protocolo inclui a etiqueta *BLOCKED*, que informa que existem dados para serem enviados, mas algo está impedindo o processo de continuar.

O protocolo HTTP/2 traz grandes mudanças na estrutura dos dados que serão enviados e recebidos. A essência continua a mesma, um protocolo de requisições e respostas, mas, a medida que o protocolo evolui, novas formas de aprimoramento do desempenhos dos *websites* e das aplicações estão sendo acrescentadas ao HTTP

O documento completo com toda a especificação do HTTP/2 pode ser encontrado em ([GROUP, 2015b](#)). O [Quadro 5](#) mostra um resumo das mudanças detalhadas anteriormente.

Quadro 5 – Mudanças introduzidas no HTTP/2.



Funcionalidade	Descrição
HTTP/2 binário	Ao invés de utilizar caracteres ASCII para representar informações, o HTTP/2 é binário, o que facilita a comparação de informações, o envio de dados e outras funcionalidades.
Fluxos multiplexados	Se existem dois componentes para serem enviados, o protocolo pode optar por multiplexá-los em uma única stream e enviar os dois ao mesmo tempo.
Prioridades e dependências	Caso existam, o cliente pode definir quais componentes possuem prioridade para serem baixados primeiro. Além disso pode informar se existem dependências entre os componentes para garantir que quando um arquivo seja baixado todos os outros necessários para o seu funcionamento já estejam no cliente.
<i>HPACK</i>	Novo sistema de compressão de cabeçalhos para o HTTP/2.
<i>RST_STREAM</i>	Uma maneira de cancelar o envio de componentes.
<i>Server Push</i>	Habilidade do servidor de enviar um arquivo X para o cliente caso ele veja como provável que o cliente vai precisar desse arquivo no futuro próximo.
Janelas individuais de fluxo	Cada fluxo de envio possui sua própria janela que pode ser gerenciada individualmente, assim caso um fluxo falhe os outros continuam.
<i>BLOCKED</i>	Forma do cliente ou servidor informar a outra parte que existe algo impedindo que o envio de dados continue.
<i>Alt-Svc</i>	O servidor pode informar ao cliente de caminhos alternativos para acessar os dados requisitados.

2.5 AJAX

AJAX é a sigla em inglês para "~~Assíncrono JavaScript + XML~~", e foi inventada por [Garrett \(2005\)](#). Mas isso não quer dizer que Garrett tenha inventado o modelo AJAX utilizado pelos *websites* e aplicações.

Como dito pelo próprio [Garrett \(2005\)](#), "AJAX não é uma tecnologia". Na reali-

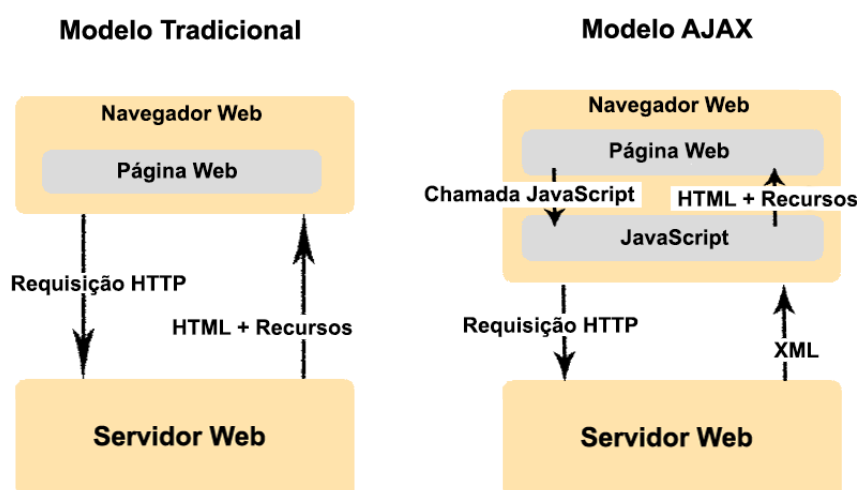
dade AJAX é a definição de como várias tecnologias podem ser utilizadas em conjunto para ler componentes *web* de maneira assíncrona. Estas tecnologias incluem:

- HTML ou XHTML⁷
- CSS
- JavaScript
- DOM⁸
- XML e XSLT⁹
- Requisições XMLHttpRequest (XMLHttpRequest)¹⁰

A Figura 8 mostra uma comparação entre o modelo clássico de chamadas *web* e o modelo utilizando AJAX. O modelo AJAX adiciona uma camada intermediária entre o usuário e o servidor.

Quando o usuário interage com a página *web* ele manda uma mensagem para essa nova camada, desenvolvida em JavaScript, que é responsável por requisitar do servidor somente os dados necessários para a interação do usuário e atualizar apenas a parte da página necessária para terminar essa interação. Dessa forma o AJAX impede que as páginas tenham de ser inteiramente recarregadas a cada interação e consegue melhorar a experiência do usuário.

Figura 8 – Comparação entre modelo de *web* original e modelo utilizando AJAX



Fonte: Adaptado de Scott (2007)

⁷<http://en.wikipedia.org/wiki/XHTML>

⁸<http://www.w3.org/DOM/>

⁹<http://www.w3schools.com/xml/>, <http://www.w3schools.com/xsl/>

¹⁰<https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>

2.6 Web 2.0

A *Web 2.0* não é uma nova versão da *World Wide Web* criada por Tim Berners-Lee. Como explica O'Reilly (2005), o termo *Web 2.0* surgiu de uma discussão entre a companhia de mídia O'Reilly e a empresa de produção de MediaLive, quando estavam preparando uma conferencia sobre a *web*. O que eles chamaram de *Web 2.0* foi a nova forma como a criação de Berners-Lee estava sendo utilizada pelas pessoas.

O desenvolvimento de técnicas AJAX e o conceito de conhecimento coletivo fez com que a *web* deixasse de ser utilizada apenas para mostrar páginas estáticas e passasse a explorar a interação com o usuário. Sendo assim *websites* se tornaram aplicações *web*, que melhoravam a medida que os usuários iam alimentando-as com novas informações. Como exemplo de aplicações que exploram os conceitos definidos na *Web 2.0* podemos citar a Wikipedia ¹¹, o Facebook ¹² e o Google Maps ¹³. Todas essas aplicações utilizam de técnicas AJAX para garantir uma boa experiência e são alimentadas com informações entradas pelos usuários, o que as torna mais dinâmicas e flexíveis do que os *websites* da *Web 1.0* (como ficou conhecida a primeira fase da *web*).

¹¹<https://www.wikipedia.org/>

¹²<https://www.facebook.com/>

¹³<https://www.maps.google.com/>

3 Trabalhos Relacionados

Desenvolvedores estão sempre a procura de maneiras para tornar suas aplicações mais rápidas aos olhos dos usuários finais, isso não é diferente com os *websites*. A área de otimização de desempenho para páginas *web* é tão antiga quanto a *World Wide Web*, mas por muito tempo os esforços foram dedicados à otimização de *back-end*, o que (SOUDERS, 2007) provou não ser o melhor caminho para se atingir o objetivo desejado. De acordo com Souders (2007, p. 5): "Apenas 10-20% do tempo de resposta do usuário final são gastos baixando o documento HTML. Os outros 80-90% são gastos baixando todos os componentes da página.". Então os desenvolvedores podem obter melhores resultados de desempenho caso foquem seus esforços na otimização do *front-end* de seus *websites* ao invés do *back-end*.

Neste capítulo serão apresentadas as ideias principais dos dois livros de Steve Souders. Estes livros tornaram o conhecimento sobre a otimização de desempenho de *front-end* de *websites* acessíveis a todos os desenvolvedores e ajudaram a melhorar a *web* nos últimos anos.

3.1 *High Performance Web Sites*

No livro *High Performance Web Sites*, (SOUDERS, 2007), Steve Souders descreve 14 regras para a otimização de desempenho de *front-end* de *websites*. De acordo com Souders (2007, p. 5): "Se você seguir todas as regras aplicáveis a seu *website*, você vai fazer sua página 25-50% rápida e melhorar a experiência do usuário.". As regras são muito variadas e cobrem de configurações básicas de servidores, a redução do tamanho de arquivos e melhores práticas de programação.

A seguir serão explicadas as 14 regras encontradas no livro, ordenadas, de acordo com Steve Souders, das que causam maior impacto no desempenho às que causam menor impacto.

3.1.1 Regra 1: Faça menos requisições HTTP

Cada componente de uma página *web* gera pelo menos uma requisição e uma resposta HTTP para chegar do servidor ao cliente. Essas chamadas HTTP costumam ser o um grande gargalo de desempenho para *websites* e aplicações *web*, pois como explicou Souders (2007), 80-90% do tempo de montagem dessas páginas é gasto baixando outros componentes além do HTML. Assim uma solução para diminuir o tempo de resposta para o usuário final é diminuir o número de componentes que precisam ser baixados.

Com esse objetivo, as seguintes técnicas devem ser utilizadas:

- CSS Sprites ¹
- Images em linha
- Combinar *scripts* e folhas de estilo ²

3.1.2 Regra 2: Use Redes de Entrega de Conteúdo (CDN)

No HTTP/1.1 o número de conexões TCP que os navegadores *web* abrem em paralelo no mesmo domínio é limitado, com o objetivo de garantir a qualidade de cada conexão. Apesar desse número ser maior no HTTP/1.0, continua existindo um limite para essas conexões paralelas. Outra característica dessas conexões é que elas são afetadas pela distância física entre clientes e servidores, logo um usuário que esteja no Brasil vai demorar mais para acessar componentes que estejam hospedados na China do que componentes que estejam hospedados nos Estados Unidos. Uma maneira de resolver esses dois problemas simultaneamente são as CDNs ³. Ao invés de hospedar todos os componentes do *website* no mesmo servidor, os desenvolvedores deveriam utilizar das CDNs para garantir maior paralelismo e diminuir as distâncias geográficas entre seus sistemas e seus usuários.

3.1.3 Regra 3: Adicione cabeçalhos de expiração

O HTTP foi desenvolvido com suporte nativo para gerenciamento de *cache*. Mas para essa *cache* funcionar de maneira eficiente é necessário que os desenvolvedores configurem seus servidores para informar aos navegadores *web* que certos arquivos devem ficar salvos localmente. A maneira de garantir que os arquivos vão ser salvos em *cache*, é o envio da etiqueta *Expires* no cabeçalho HTTP da resposta que possui o arquivo. Essa configuração não é definida por padrão nos servidores, então os desenvolvedores devem analisar quais arquivos devem utilizar desse recurso e por quanto tempo eles serão válidos. A *cache* é uma ferramenta muito importante na otimização de desempenho, e deixar de usa-la causa grandes danos no tempo médio de resposta de *websites* e aplicações *web*.

3.1.4 Regra 4: Utilize *gzip* no componentes

De acordo com Souders (2007, p. 29) essa é a técnica mais simples e que causa maior efeito no tamanho em *bytes* das páginas *web*. O HTTP e os navegadores *web*

¹http://www.w3schools.com/css/css_image_sprites.asp

²A técnica não sugere a união de *scripts* com folhas de estilo, e sim *scripts* com *scripts* e folhas de estilo com folhas de estilo, diminuindo o número de arquivos existentes.

³http://en.wikipedia.org/wiki/Content_delivery_network

possuem suporte para compressão de todos os tipos de arquivos, os desenvolvedores só precisam garantir que os clientes e os servidores informem uns aos outros que a esse mecanismo está habilitado. Para fazer isso, a etiqueta de cabeçalho *Accept-Encoding* deve ser definida com os formatos de compressão suportados. O *gzip* é um formato de compressão criado especialmente para a *web* e é o mais eficiente para ser utilizado no HTTP.


3.1.5 Regra 5: Coloque folhas de estilo no topo da página

Como disse Souders (2007, p. 38), "essa regra tem menos *haver* com melhorar o tempo de carregamento da página e mais *haver* com como o navegador *web* reage à ordem dos componentes.". Por definição, os navegadores não mostram os elementos da tela até que todas as folhas de estilo tenham sido carregadas, para evitar erros de desenho. Sendo assim, os desenvolvedores devem colocar as folhas de estilo no topo da página (de preferência dentro da etiqueta HTML *HEAD*⁴), pois essas devem ser os primeiros componentes a serem baixados.

3.1.6 Regra 6: Coloque *scripts* no fim da página

Ao contrário das folhas de estilo, os arquivos de *script* devem ser colocados no final do arquivo HTML. Componentes localizados abaixo de arquivos de *script* são bloqueados de serem baixados e desenhados até que o *script* tenha terminado de ser carregado. Isso porque o navegador quer garantir que o *script* esteja pronto caso ele vá executar alguma alteração no restante da página.

3.1.7 Regra 7: Evite expressões CSS

Apesar de já não serem suportadas na maioria dos navegadores *web*, os desenvolvedores não devem utilizar expressões CSS em nenhum caso, mesmo nos casos onde o navegador dá essa alternativa. Este tipo de expressão é calculada toda vez que há uma movimentação na página (desenhos de novos componentes, uso da barra de rolagem, dentre outros) e isso pode afetar, em muito, a performance de um *website* ou aplicação *web*. 

3.1.8 Regra 8: Faça arquivo *JavaScripts* e folhas de estilo externos

Existem duas maneiras de incluir estilos e *scripts* em um *website* ou em uma aplicação *web*. A primeira delas é inserir blocos de código em linha, ou seja, colocar os código diretamente no arquivo HTML, de preferência seguindo as Regras 5 e 6. Apesar de funcionar, essa maneira impede um bom uso da *cache*, porque arquivos

⁴http://www.w3schools.com/tags/tag_head.asp

HTML geralmente não são salvos em *cache*, logo os estilos e *scripts* têm de ser baixados todas as vezes que a página é carregada. Levando em consideração que a Regra 3 está sendo usada, uma maneira mais eficiente de inserir estilos e *scripts* em uma página é colocando-os em arquivos externos e adicionando os arquivos no HTML. Dessa forma os arquivos externos podem ser salvos em *cache* e não precisam ser baixados todas as vezes que o HTML é requisitado.

3.1.9 Regra 9: Reduza o número de pesquisas de DNS

Usar diferentes domínios para hospedar componentes é uma maneira de possibilitar que esses componentes sejam baixados em paralelo. Mas a pesquisa por domínios é cara e pode acabar influenciando no tempo de carregamento de uma página. Por isso os navegadores salvam os domínios que já sabem em *cache*. Levando em consideração esse custo de pesquisa, existe um limite para até quando hospedar componentes em domínios diferentes é benéfico. Os desenvolvedores têm de se analisar a quantidade de pesquisas de DNS para domínios únicos (que não estão em *cache*) estão sendo feitas no carregamento da página e decidir se ainda vale a pena aumentar o paralelismo. Como regra geral os desenvolvedores deveriam distribuir seus componentes em pelo menos dois e no máximo quatro domínios.

3.1.10 Regra 10: Mimifique arquivo *JavaScript*

Mimificar é a técnica de reduzir caracteres desnecessários de códigos. Essa técnica pode ser utilizada para qualquer tipo de arquivo CSS, JS, PHP, HTML, dentre outros. A mimificação reduz o tempo de carregamento de páginas *web* porque reduz o tamanho do arquivo diminuindo o tempo gasto para baixá-lo.

3.1.11 Regra 11: Evite redirecionamentos

Redirecionamentos são situações onde o usuário tenta acessar uma página *web* e essa página o informa que ele deve se encaminhar para outra página. Os redirecionamentos são feitos através dos códigos da família 3xx do protocolo HTTP, onde os diferentes códigos dessa família informam a razão para o redirecionamento.

Redirecionamentos são extremamente danosos para a experiência de uso de *websites* e aplicações *web*, pois nada é mostrado para o usuário até que o redirecionamento seja concluído e o conteúdo HTML termine de ser carregado. Logo desenvolvedores não devem utilizar redirecionamento a não ser em casos onde são indispensáveis.

3.1.12 Regra 12: Remova *scripts* duplicados

Em projetos grandes é comum que vários desenvolvedores mexam no código e isso pode fazer com que o mesmo código seja inserido mais de uma vez. *Scripts* duplicados aumentam o número de chamadas HTTP e isso é muito prejudicial ao desempenho de *websites* e aplicações *web*. Sendo assim os desenvolvedores devem ser muito cuidadosos quanto aos *scripts* que estão sendo carregados em suas páginas e ter certeza de que todos são únicos e necessários.

3.1.13 Regra 13: Configure *ETags*

Muitas vezes os desenvolvedores não configuram como funcionará o mecanismo de *ETags* de seus servidores e isso pode fazer com que o mecanismo de *cache* seja prejudicado. Como o valor do *ETag* de um componente é único para cada servidor, se o cliente requisitar um componente no servidor X e mais tarde requisitar o mesmo componente no servidor Y, mesmo que este tenha sido salvo na *cache* do navegador na primeira vez, ele será baixado novamente. Na maioria das vezes é melhor desabilitar a opção de *ETags* do que correr o risco de influenciar negativamente no sistema de *cache*.

3.1.14 Regra 14: Habilite *cache* para AJAX

As chamadas AJAX melhoram muito a experiência do usuário, pois tornam as páginas *web* mais dinâmicas e responsivas. O problema é que muitas vezes elas não são salvas em *cache* e isso pode afetar o desempenho dos *websites* e aplicações *web*. Para garantir que as chamadas AJAX serão salvas em *cache* os desenvolvedores devem seguir as regras 1 à 13 também para esse tipo de componente, tomando cuidado especial com a Regra 3.

3.2 *Even Faster Web Sites*

Com o objetivo de difundir ainda mais as técnicas de otimização para *front-end* de *websites*, Steve Souders lançou, em 2009, seu segundo livro, intitulado *Even Faster Web Sites*. Nesta obra Souders teve a contribuição de outros 8 profissionais da área de otimização que escreveram 6 dos 14 capítulos do livro.

As técnicas descritas em *Even Faster Web Sites* são mais avançadas, e, consequentemente, mais difíceis de serem aplicadas do que as da obra anterior ~~de Souders~~. Além disso, elas não vêm em formato de regras a serem seguidas, o que facilitava a aplicação das regras descritas em *High Performance Web Site*. Alguns capítulos do livro explicam melhores técnicas de programação, descrevem o funcionamento de mecanismos da *web* e outros ainda reforçam ideias anteriores.

Nesta seção serão resumidas as ideias principais de cada capítulo.

3.2.1 Entendendo performance em AJAX

Para saber se vale a pena otimizar o desempenho de um *website* ou de uma aplicação *web*, os desenvolvedores devem levar em conta o tempo e o esforço que deverão ser empregados no processo de otimização, e conhecer quanto essa otimização pode melhorar a experiência do usuário. Desta forma, pode-se decidir se compensa fazer um investimento nesse processo. O desenvolvedor deve se lembrar que otimizar componentes que não *tem* contribuição expressiva no tempo de carregamento de uma página *web* não é muito relevante para a experiência do usuário.

Os componentes AJAX representam parte importante das aplicações na *Web*. Esses componentes são carregados enquanto o usuário está navegando na página, então deve-se evitar qualquer cenário *onde* a aplicação congele à espera da resposta de uma requisição AJAX. Os desenvolvedores devem identificar quais componentes compensam ser otimizados utilizando os "Eixos do Erro", descrito por Souders (2009a, p. 2) e focar seus esforços nesse componentes.

Durante o processo de otimização de componentes AJAX, os desenvolvedores devem se lembrar:

- A maior parte do tempo do navegador é gasta na construção do DOM e não no *JavaScript*.
- Códigos organizados são mais fáceis de serem otimizados.
- Iterações afetam muito o desempenho de uma aplicação *web*.
- **Truques** não compensam, a não ser que eles se provem realmente eficientes.

3.2.2 Criando aplicações *web* responsivas

Aplicações responsivas são aquelas que respondem ao usuário de maneira rápida e eficiente, fazendo com que ele não sinta que está esperando. Atrasos maiores do que 0,2 segundos causam no usuário a sensação de que o navegador está tendo problemas para conseguir sua resposta, e isso afeta sua experiência de uso. O mecanismo responsável pela responsividade das aplicações é o AJAX.

Em aplicações grandes, muitas chamadas AJAX pode ser executadas ao mesmo tempo e isso é um problema para o *JavaScript*. De acordo com Brendan Eich, criador do *JavaScript*, sua linguagem não possui e nunca vai possuir *threads*. Assim, fica por responsabilidade dos desenvolvedores encontrar técnicas para melhorar o paralelismo de suas aplicações. Enquanto tentando paralelizar seus códigos, um cuidado que os

desenvolvedores devem ter é com o uso de memória. Diferente de Java, o *JavaScript* não possui um coletor de lixo, e o uso excessivo de memória pode afetar o desempenho das páginas *web*.

3.2.3 Dividindo carga inicial

O tempo inicial de carregamento de uma página é um fator muito importante na otimização de desempenho. Usuários não gostam de esperar para poder interagir com uma aplicação ou *website* e a grande maioria desses usuários desiste de esperar se o tempo for muito longo. Sendo assim, os desenvolvedores devem definir quais métodos de seus *scripts* são necessários no evento *onload* (evento executado assim que a página é carregada) e separa-los em arquivos diferentes dos outros métodos. Os métodos que devem ser executados logo que a página for carregada devem ser declarados no início do documento HTML. Os outros podem ser declarados ao final da página ou até mesmo de maneira assíncrona.

3.2.4 Carregando *scripts* sem bloqueios

Scripts possuem um efeito muito negativo no carregamento de páginas. Enquanto estão sendo baixados e executados eles bloqueiam o carregamento de componentes localizados abaixo deles. Então, é muito importante definir quais *scripts* podem ser baixados independentes do resto da página e encontrar maneiras de requisitar esses *scripts* sem bloquear o resto da página. No Capítulo 4 deste livro, Souders (2009a) descreve seis técnicas de como carregar arquivos de *script* externos de forma assíncrona:

- *XHR Eval*, (SOUDERS, 2009a, p. 29)
- *XHR Injection*, (SOUDERS, 2009a, p. 31)
- *Script in Iframe*, (SOUDERS, 2009a, p. 31)
- *Script in DOM Element*, (SOUDERS, 2009a, p. 32)
- *Script Defer*, (SOUDERS, 2009a, p. 32)
- *document.write Script Tag*, (SOUDERS, 2009a, p. 33)

Não existe uma solução única para todos os *websites* e aplicações *web*, então os desenvolvedores devem analisar qual é a melhor para sua situação específica. E caso seja definido que a página terá de ser congelada por algum tempo, os desenvolvedores devem se certificar de que algum indicador de navegador ocupado está sendo mostrado ao usuário⁵.

⁵<http://www.stevesouders.com/blog/2013/06/16/browser-busy-indicators/>

3.2.5 Lidando com *scripts* assíncronos

Quando as técnicas descritas no Capítulo 4, (SOUDERS, 2009a, p. 27), são utilizadas para carregar *scripts* sem bloqueios, cria-se um novo problema. Componentes que são carregados de maneira assíncrona estão sujeitos a condições de corrida⁶. Essas condições tornam imprevisível a ordem na qual os *scripts* serão carregados, o que pode fazer com que ocorram falhas nas dependências de funções. Para evitar essas falhas, Steve Souders descreve 5 técnicas de como garantir que os *scripts* serão carregados em uma ordem na qual serão executados sem erros de dependências:

- *Hardcoded Callback*, (SOUDERS, 2009a, p. 46)
- *Windows Onload*, (SOUDERS, 2009a, p. 47)
- *Timer*, (SOUDERS, 2009a, p. 48)
- *Script Onload*, (SOUDERS, 2009a, p. 49)
- *Defering Script Tags*, (SOUDERS, 2009a, p. 50)

Ao final do capítulo, Souders ainda ~~explica~~ descreve uma técnica que chamou de "Solução Geral", (SOUDERS, 2009a, p. 59).

3.2.6 Posicionando blocos de *scripts* em linha

Apesar de não gerarem novas requisições HTTP, blocos de *scripts* em linha, ou seja, aqueles posicionados dentro do documento HTML, bloqueiam outros componentes de serem carregados enquanto eles estão sendo executados e isso impede o desenho progressivo da página. Para evitar esse efeito, blocos de *scripts* em linha devem ser movidos para o final da página e, se possível, serem executados de maneira assíncrona. Outro grave problema com esse tipo de *script* é que, quando eles são antecidos por folhas de estilo, eles não começam a executar enquanto os estilos não são carregados. Dessa forma, é uma boa técnica não colocar blocos de *scripts* em linha logo após chamadas de folhas de estilo.

3.2.7 Escrevendo *JavaScripts* eficientes

Neste capítulo, Nicholas Zakas descreve boas técnicas de programação para *JavaScripts* que visam otimizar o desempenho do código. As técnicas descritas são:

- Melhor gerenciamento de escopos, (SOUDERS, 2009a, p.79)

⁶http://en.wikipedia.org/wiki/Race_condition

- Use variáveis locais, (SOUDERS, 2009a, p. 81)
- Otimize acesso à dados frequentes, (SOUDERS, 2009a, p. 85)
- Use condicionais mais eficientes, (SOUDERS, 2009a, p. 89)
- Use iterações mais eficientes, (SOUDERS, 2009a, p. 93)
- Otimize cadeias de caracteres, (SOUDERS, 2009a, p. 99)
- Evite função com tempo de execução longo, (SOUDERS, 2009a, p. 102)
- Habilite contadores de tempo, (SOUDERS, 2009a, p. 103)

3.2.8 Escalando usando *Comet*

Quando dados precisam ser entregues de maneira assíncrona, o mecanismo de AJAX pode não ser o suficiente. *Comet* é um termo que se refere a uma coleção de técnicas, protocolos e implementações que tem como objetivo melhorar o tráfego de dados em baixa latência ⁷.

3.2.9 Indo além do *gzip*

Mesmo com o suporte à compressão de dados com *gzip* habilitado nos *websites* e aplicações, uma média de 15% dos usuários continua recebendo dados sem compressão alguma. A maior causa disso são os *proxies* e os programas de anti-vírus. Estes sistemas modificam os cabeçalhos HTTP para poderem ter acesso aos dados que estão sendo enviados à fim de realizarem suas tarefas.

Se essas entregas sem compressão fossem dissolvidas entre todos os usuários da página *web*, o problema seria menos relevante. Mas o que ocorre é que os mesmos 15% dos usuário estão sempre acessando os *websites* e aplicações de forma mais lenta, o que acaba com a experiência desses usuários e os desencoraja de voltar a acessar a página.

Considerando este fato, Tony Gentilcore, reforça as ideias expostas por Steve Souder no livro *High Performance Web Sites*, (SOUDERS, 2007), de que os desenvolvedores devem fazer seus arquivos os menos possíveis e reduzir o número de chamadas HTTP.

3.2.10 Otimizando imagens

Imagens são consideradas ótimos componentes para se conseguir melhora de desempenho sem ter de abrir mão de funcionalidades. Existem vários padrões de formato para imagens⁸, cada um deles é determinado pelo tipo de compressão que usa

⁷http://ajaxexperience.techtarget.com/images/Presentations/Carter_Michael_ScalableComet.pdf

⁸http://en.wikipedia.org/wiki/Image_file_formats

e cada um deles possui seus prós e contras. Sendo assim, os desenvolvedores devem decidir que tipo de qualidade que eles desejam em suas imagens, então selecionar o formato que se adequa às suas necessidades e otimizar essas imagens o máximo possível.

Como regra geral, os desenvolvedores devem optar por utilizar o formato PNG sempre que possível.

3.2.11 Quebrando domínios dominantes

Mesmo que isso aumente a quantidade de pesquisas de DNS, muitas vezes aumentar o número de domínios nos quais componentes estão hospedados melhora o desempenho de *websites* e aplicações *web*. O motivo disso é que mais componentes podem ser baixados em paralelo, o que diminui o tempo total de carregamento da página. O desafio é encontrar o número de domínios que resulta no menor tempo de carregamento da página. Como regra geral (SOUDERS, 2009a, p. 168) determina que dividir componentes entre dois a quatro domínios gera resultados satisfatórios.

3.2.12 Entregando o documento cedo

Por padrão uma página *web* só começa a baixar os seus componentes depois que o documento HTML é totalmente carregado. Sendo assim, durante algum tempo a única atividade que está sendo realizada pelo navegador é a espera do carregamento do documento HTML. Contudo, existem técnicas para diminuir esse tempo de espera e, conseqüentemente, o tempo de carregamento de uma página.

Uma boa prática é entregar o documento HTML em partes para o navegador, desde que essas partes possam acelerar o processo de carregamento. Para isso Steve Souders sugere que o cabeçalho HTML, onde normalmente são declaradas as folhas de estilo, seja entregue assim que estiver pronto, dessa forma o navegador pode começar a baixar esses componentes antes mesmo que o carregamento do HTML esteja concluído. Essa técnica é possível graças à etiqueta *Chunked Encoding*, adicionada no HTTP/1.1, e às funções de descarregamento de linguagens como PHP e NodeJS.

3.2.13 Usando *Iframes* com moderação

Os *Iframes* são componentes HTML que permitem incorporar documentos dentro de outros documentos. Apesar de terem sido muito utilizados no passado, principalmente para adicionar publicidades às *websites*, os *Iframes* são pouco usados nos dias de hoje. A razão disso é que são componentes muito pesados e que bloqueiam o desenho das páginas *web*.

Como regra geral Steve Souders aconselha não utilizar estes componentes.

3.2.14 Simplificando seletor CSS

Na Web 2.0, folhas de estilo são tão populares quanto *scripts*. *Websites* possuem várias folhas de estilo, mas pouco esforço é empregado em otimizar esses arquivos.

Neste capítulo Souders explica que os navegadores fazem buscas por componentes declarados em folhas de estilo da direita para esquerda, logo os identificadores mais à direita devem ser os mais específicos, para diminuir o tempo de busca. Além disso ele alerta para o mal de códigos duplicados e cita boas práticas de programação para CSS, (SOUDERS, 2009a, p. 195).

4 Metodologia

Inserir seu texto aqui...

4.1 Delineamento da pesquisa

Inserir seu texto aqui...

4.2 Coleta de dados

Inserir seu texto aqui...

5 Análise de Resultados

Inserir seu texto aqui...

5.1 Situação atual

Inserir seu texto aqui...

5.2 Análise dos dados coletados

Inserir seu texto aqui...

6 Conclusão

Referências

- ARCHIVE, H. **Http Archive Abril 2011 Query**. 2015. Disponível em: <<http://httparchive.org/interesting.php?a=All&l=Apr%2015%202011>>. Citado na página 1.
- ARCHIVE, H. **Http Archive Abril 2015 Query**. 2015. Disponível em: <<http://httparchive.org/interesting.php?a=All&l=Apr%2015%202015>>. Citado na página 2.
- CONNOLLY, D. **The birth of the web**. 2000. Disponível em: <<http://www.w3.org/History.html>>. Citado na página 1.
- GARRETT, J. J. **Ajax: A New Approach to Web Applications**. 2005. Disponível em: <<http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/>>. Citado na página 19.
- GRIGORIK, I. **High Performance Browser Networking**. 1st. ed. [S.l.]: O'Reilly, 2013. Citado na página 6.
- GROUP, H. W. **HPACK - Header Compression for HTTP/2**. 2015. Disponível em: <<http://tools.ietf.org/html/draft-ietf-httpbis-header-compression-12>>. Citado na página 17.
- GROUP, H. W. **Hypertext Transfer Protocol version 2**. 2015. Disponível em: <<https://tools.ietf.org/html/draft-ietf-httpbis-http2-17>>. Citado 3 vezes nas páginas 3, 6 e 18.
- GROUP, N. W. **RFC 1945**. 1996. Disponível em: <<http://tools.ietf.org/html/rfc1945>>. Citado 2 vezes nas páginas 6 e 10.
- GROUP, N. W. **RFC 2616**. 1999. Disponível em: <<http://tools.ietf.org/html/rfc2616>>. Citado 3 vezes nas páginas 1, 6 e 10.
- INC., G. **SPDY**. 2012. Disponível em: <<http://dev.chromium.org/spdy>>. Citado na página 15.
- KRISHNAMURTHY, B.; MOGUL, J. C.; KRISTOL, D. M. Key differences between http/1.0 and http/1.1. **Elsevier Science B.V**, 1999. Citado 2 vezes nas páginas 10 e 12.
- O'REILLY, T. **What Is Web 2.0**. 2005. Disponível em: <<http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html?page=1>>. Citado na página 21.
- SAENZ, J. **Building Web Apps with Go**. [s.n.], 2014. Disponível em: <<https://www.gitbook.com/book/codegangsta/building-web-apps-with-go/details>>. Citado na página 7.
- SCOTT, T. 2007. Disponível em: <<http://derivadow.com/2007/01/05/ajax-what-is-it-its-not-dhtml/>>. Citado na página 20.
- SOUDERS, S. **High Performance Web Sites**. 1st. ed. [S.l.: s.n.], 2007. Citado 6 vezes nas páginas 2, 3, 22, 23, 24 e 30.
- SOUDERS, S. **Even Faster Web Sites**. 1st. ed. [S.l.: s.n.], 2009. Citado 7 vezes nas páginas 3, 27, 28, 29, 30, 31 e 32.

SOUDERS, S. Even faster websites. In: . [s.n.], 2009. Disponível em: <https://www.youtube.com/watch?feature=player_embedded&v=aJGC0JSIpPE>. Citado na página 2.

SOUDERS, S. **Web Performance Daybook**. 1st. ed. [S.l.: s.n.], 2012. Two. Citado na página 3.

STENBERG, D. Http2 explained. **ACM SIGCOMM Computer Communication Review**, 2014. Disponível em: <<http://daniel.haxx.se/http2>>. Citado 4 vezes nas páginas 3, 15, 16 e 17.

TANENBAUM, D. J. W. A. S. **Computer Networks**. 5th. ed. [S.l.: s.n.], 2011. Citado 6 vezes nas páginas 1, 7, 8, 9, 11 e 16.

Apêndices

APÊNDICE A – Nome do Apêndice

Inserir seu texto aqui...

APÊNDICE B – Nome do Apêndice

Inserir seu texto aqui...

Anexos

ANEXO A – Nome do Anexo

Inserir seu texto aqui...

ANEXO B – Nome do Anexo

Inserir seu texto aqui...