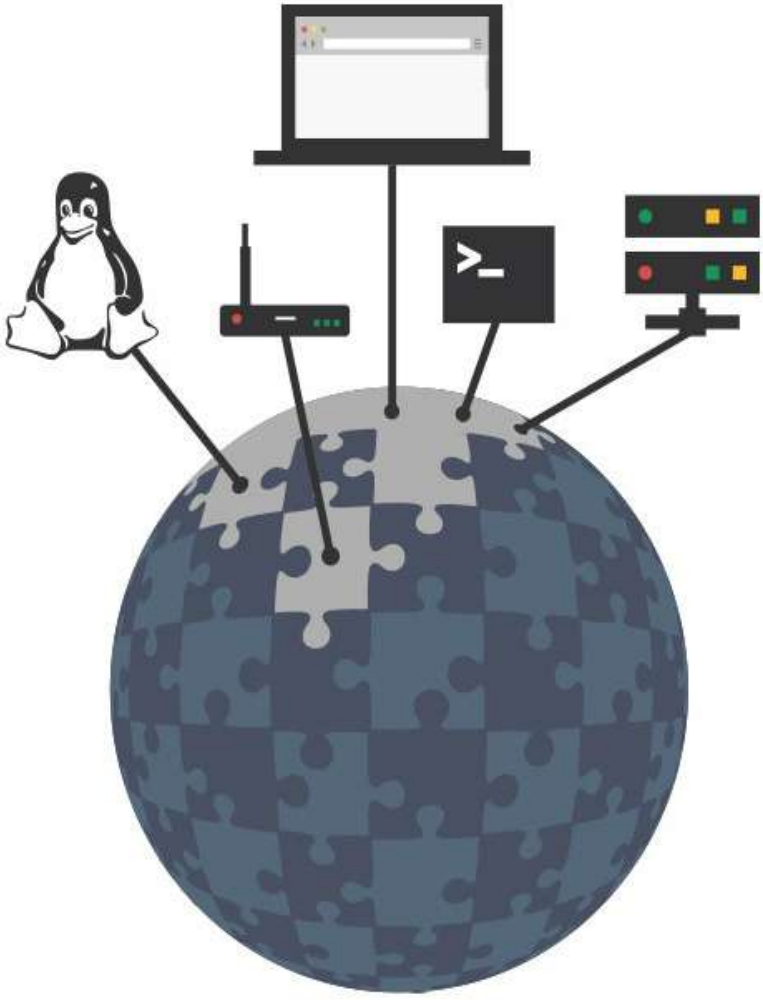
Desconstruindo a Web

As tecnologias por trás de uma requisição



Casa do Código

WILLIAN MOLINARI

Adriano Almeida

Livros para o programador

Rua Vergueiro, 3185

www.casadocodigo.com.br

ISBN

Impresso e PDF: 978-85-5519-210-4

EPUB: 978-85-5519-211-1

MOBI: 978-85-5519-212-8

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

PREFÁCIO

Tudo começou com um estudo despretensioso sobre como uma requisição para um site funciona. A ideia era mostrar o máximo possível do que acontecia em uma requisição web, durante uma palestra de 50 minutos. Depois de 40 horas de estudo e dedicação, os slides da apresentação estavam prontos, mas havia muito mais detalhes nesse conteúdo do que era possível demonstrar naquela mídia e com aquele tempo.

Durante a construção do material, foi possível perceber que há vários estudos isolados e documentos de arquitetura que mostram separadamente cada uma das peças de uma requisição web. É possível encontrar material sobre como as redes funcionam para entregar o conteúdo de um ponto ao outro, como os protocolos trabalham para encapsular os dados, como é a arquitetura de um *framework* de desenvolvimento web, como o servidor de aplicação se integra com o framework etc. Apesar disso, não há nada que mostre, do começo ao fim, o que acontece desde o momento em que o usuário aperta a tecla *Enter* do teclado até a página estar completamente carregada na tela.

Este livro vem cobrir esse espaço, trazendo um estudo unificado e focado no entendimento de como a internet funciona, utilizando uma requisição de uma página HTML como exemplo. Durante este estudo, vamos passar por diversas áreas de conhecimento distintas. Passaremos pela criação visual da interface no navegador, pela montagem dos pacotes dentro do kernel do sistema operacional, pelos dispositivos de rede da internet, e até mesmo pela integração do código da aplicação com o servidor web.

Para quem é

Desenvolvedores de aplicações para a internet, programadores curiosos com a web, administradores de sistema, engenheiros de rede, ou qualquer outra pessoa que crie conteúdo para a internet e esteja disposta a ir fundo na tecnologia para entender o exato papel do seu trabalho no todo.

O estudo assume um conhecimento básico de como funciona o ambiente de desenvolvimento para a internet. Esse conhecimento básico significa saber que um desenvolvedor programou o site, possivelmente usou algum *framework*, colocou em um servidor de alguma empresa de hospedagem que possui Apache ou NGINX, e registrou o domínio para que seja fácil de acessar via internet.

Apesar de o livro explicar muitos dos termos, ele assume que quem está lendo já sabe o que significa sistema operacional, processo e *thread*, por exemplo. Esses e mais alguns outros termos básicos não serão detalhados para deixar a explicação mais direta, mas uma busca rápida na internet resolve o problema para continuar o entendimento. Familiaridade com alguma linguagem de programação também será útil em alguns pontos do estudo, em que vamos mais a fundo no desenvolvimento do software estudado.

Aqui há informações úteis tanto para quem é menos experiente na área de internet como para quem já trabalha com isso há algum tempo. Um especialista em desenvolvimento de aplicações, por exemplo, encontrará algumas informações redundantes para a sua área de conhecimento. Apesar disso, ele conhecerá mais sobre todas as tecnologias que envolvem a entrega de conteúdo para o usuário.

O conteúdo deste livro foi pensado para que os especialistas possam passar rapidamente pelo conteúdo de sua especialidade, focando em todos os outros componentes da infraestrutura da requisição.

Estrutura

O livro está dividido em 10 capítulos, sendo que cada um deles se dedica a uma parte específica de uma requisição web. O conteúdo foi organizado dessa forma para facilitar a leitura por tópicos e áreas de interesse, além de ajudar os mais experientes a filtrar tópicos.

Para contribuir com o fluxo de leitura, todos os capítulos são divididos em três partes: conteúdo principal, resumo do conteúdo e referências do capítulo.

O conteúdo principal pode focar em diversas áreas, dependendo da parte estudada. Há capítulos mais focados na parte de desenvolvimento, outros na parte de redes de computadores, e outros em softwares que geralmente são administrados por um administrador de sistemas.

Caso não haja interesse em se aprofundar em determinada área de conhecimento da requisição web, ainda será possível acompanhar o resumo no fim do capítulo. Ele não entrará em detalhes, mas vai proporcionar uma forma de continuar o estudo sem perder o fluxo das informações.

O estudo

O estudo começa com o usuário digitando a URL de um site na barra de endereços do navegador e apertando a tecla *Enter*. Nesse momento, vamos começar juntos uma jornada que passa por dezenas de tecnologias até que a requisição esteja completa e a página totalmente carregada na tela do usuário.

Apesar de este estudo não ser focado em uma tecnologia específica, vamos definir os softwares que serão estudados. Faremos isso para poder ir mais fundo no estudo e entender qual trabalho precisa ser executado em uma parte específica da requisição. Podemos utilizar como exemplo o navegador, que para esse estudo será o Chromium. Por ter definido esse navegador como nossa

escolha de estudo, podemos olhar o código-fonte e os documentos de design que explicam por que determinadas implementações foram feitas.

É comum a sua tecnologia preferida não estar listada como parte do estudo, pois há uma vasta lista de possibilidades para cada parte de uma requisição web. Apesar disso, a tecnologia que vamos estudar desempenhará um papel parecido para cumprir o que é necessário para entregar o conteúdo, proporcionando o conhecimento que estamos buscando.

A aplicação que o usuário vai acessar é o site deste livro. Essa aplicação tem código aberto, assim como grande parte dos softwares utilizados nesse estudo. Essa aplicação foi criada utilizando o framework *Ruby on Rails* e possui pouco código, para facilitar o estudo. Ela está sendo executada em um servidor externo, e é disponibilizada via *NGINX* e *Phusion Passenger* para a internet.

Vamos assumir que o usuário está utilizando um sistema *GNU/Linux*, que tem código aberto e podemos ver tudo o que precisarmos. Por ser um sistema operacional livre e utilizar muitos softwares de código aberto, poderemos ver onde cada um deles interage com o sistema operacional para conseguir o que precisam.

Não se preocupe se você conhece pouca coisa dessa sopa de letrinhas em forma de nomes de tecnologias, vamos entrar em mais detalhes durante o estudo. Lembre-se de que **o propósito é entender como a internet funciona** independente da tecnologia usada, explorando os conceitos que guiam cada uma delas para prover o que temos hoje. Sempre que possível haverá exemplos de uso de tecnologias diferentes das que foram descritas, a fim de fazer comparações e ajudar no entendimento, ou até mesmo para satisfazer a curiosidade.

Profundidade

Como dito no início deste prefácio, há muitos estudos sobre cada uma das várias áreas que uma requisição web passa. Alguns deles são livros com mais que o dobro de páginas deste, abordando cada detalhe dos protocolos de uma rede, por exemplo. O foco deste livro não é detalhar ao máximo cada uma das etapas, mas mostrar tudo o que é necessário para entender mais a fundo como uma requisição web funciona.

Fica fora do escopo estudar como é a sequência de bits utilizados para criar o cabeçalho de um protocolo, como o teclado trata uma tecla sendo apertada, como o kernel recebe a interrupção de I/O do teclado, como a GPU calcula os frames de vídeo e coisas do gênero. Todos esses assuntos têm alguma ligação com uma requisição web e são utilizados para alguma coisa útil desse contexto, mas nesse estudo eles não têm um papel determinante para o entendimento do todo.

Este livro segue uma linha pragmática de estudo, sem chegar aos níveis mais básicos de hardware e sistema operacional, mas chegando o mais próximo possível para mostrar por onde os dados passam. A ideia é prover informação suficiente para que seja possível entender o todo. Com isso, prover uma forma de avaliar onde uma melhoria de performance poderia ser encaixada, ou onde um determinado problema pode estar.

Referências externas

Por ser um estudo geral que envolve muitas áreas de conhecimento, todos os capítulos deste livro possuem referências externas que dão apoio ao conteúdo estudado. As referências apresentadas não são de leitura obrigatória, sendo que sua explicação básica será feita junto com o conteúdo do livro, deixando-as apenas como uma extensão. Em alguns casos, elas são deixadas apenas para os mais curiosos e corajosos que podem estar

interessados em saber sobre os mínimos detalhes da tecnologia.

É importante ressaltar que quase todas as referências citadas neste livro estarão em inglês, por ser o idioma com mais informações sobre os assuntos aqui estudados. Esse fato não deve atrapalhar a leitura, pois como já foi dito, as referências apenas estendem o livro, e não são totalmente necessárias para a sua compreensão.

AGRADECIMENTOS

Primeiramente, agradeço à minha esposa **Jacqueline Molinari** por tudo, principalmente pelo apoio durante as centenas de horas e dezenas de fins de semana em que eu estava fechado no escritório escrevendo este livro. Sem o apoio dela, ele não existiria.

Aos meus amigos, **Daniel Sousa**, **Renan Ranelli**, **Mateus Linhares** e **Vinicius Baggio**, por conseguir tempo para revisar e contribuir sugerindo alterações e inclusões de conteúdo para o livro. Muito obrigado!

Agradeço também à minha família, **Wilson**, **Evanilda** e **Lilian Molinari**, por me darem o alicerce para formar meu caráter e alcançar minhas conquistas. Se cheguei aonde estou, devo isso a eles.

E a todos os gigantes que estão nas referências, cujos ombros me serviram de apoio para ver mais longe.

SOBRE O AUTOR

Willian Molinari, mais conhecido como **PotHix**, é formado em Sistemas de informação pela Fundação Santo André e trabalha com desenvolvimento de software há mais de 10 anos. Durante sua carreira, passou por diversas linguagens de programação, desde ASP no começo da carreira até Python, Ruby e Golang atualmente. Além de trabalhos freelancer para fora do país, ele passou desde pequenas empresas e startups com cerca de 5 funcionários, até grandes empresas com mais de 1.200.

Por gostar de jogos, ele resolveu investir parte do seu tempo livre para desenvolvê-los, só pela diversão. Ficou em segundo lugar em um concurso de desenvolvimento de jogos do Itaú Cultural em 2009 e participou de alguns outros GameJams. Desde que começou a investir o tempo livre nesse hobby, foram feitos 4 pequenos jogos, sendo que 3 deles utilizam JavaScript e recursos do HTML5. O jogo que exigiu mais dedicação e trouxe mais desafios foi o **Skeleton Jigsaw** (http://plaev.github.io/skeleton-jigsaw).

Foi revisor técnico de dois livros. O primeiro é o *Ruby on Rails: Coloque sua aplicação web nos trilhos*, escrito por *Vinícius Baggio Fuentes* e publicado pela *Casa do Código*. O segundo é o *HTML5 Game Development Hotshot*, escrito por *Makzan* e publicado pela *Packt Publishing*.

Costuma participar de vários eventos de tecnologia e é um dos fundadores do grupo de usuários Ruby de São Paulo, mais conhecido como **Guru-SP** (https://gurusp.org/). Ele estava no primeiro encontro do grupo feito em 2008, e vem ajudando na organização desde então.

É palestrante e já passou por mais de 7 estados brasileiros

durante as dezenas de palestras feitas. Só em 2015, foram 14 palestras sobre desenvolvimento de software e arquitetura.

Sempre que possível disponibiliza códigos open source na sua conta do GitHub (https://github.com/pothix) e nas organizações a ela relacionadas. Não é muito fã de redes sociais, mas de vez em quando posta sobre tecnologia na sua conta do Twitter (https://twitter.com/pothix), além de manter sua conta do LinkedIn (https://www.linkedin.com/in/willianmolinari) atualizada com as últimas empresas que passou e os desafios que enfrentou.

Sumário

Versão: 19.9.25

Capítulo 1

E NO COMEÇO, HAVIA O NAVEGADOR

Aqui começamos nossa jornada! Como bom aventureiro em terras desconhecidas, é importante ter um mapa. Como não conseguiria representar todos os leitores com um único desenho, vamos utilizar a aparência do programador que seja mais fácil de encontrar. Além disso, esse é um desenho que o autor está acostumado a fazer de uma forma não tão ruim.

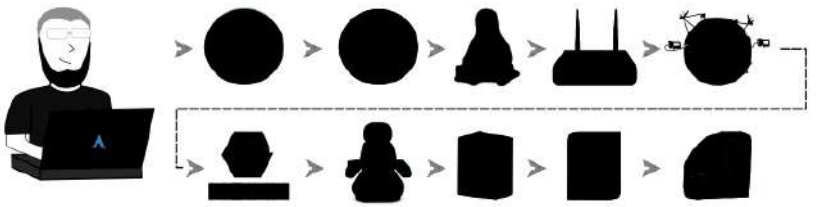


Figura 1.1: O início da jornada

Por enquanto, não temos informações sobre nenhum passo, mas vamos conhecendo cada um deles ao longo da jornada.

Neste capítulo passaremos por todo o trabalho que o navegador precisa fazer para entender o que o usuário realmente quer, antes de passar para o sistema operacional. Algumas coisas sobre a comunicação de rede serão mencionadas de forma superficial para que possamos entender o trabalho do navegador, mas focaremos nelas mais à frente.

Para iniciar, vamos imaginar um cenário propício: o usuário está em frente ao seu computador com o navegador aberto. Então, ele lembra de uma *URI* que um amigo indicou, sobre um livro que explica como uma requisição web funciona. Com isso, ele decide acessar o site e digita a URI na barra de endereços no navegador: `desconstruindoaweb.com.br`.

Relembrando do que foi descrito há pouco no *Prefácio* (ah... é claro que você leu o prefácio, né?), agora seria possível descer bem fundo. Poderíamos ir até o hardware explicando a interrupção que cada tecla do teclado gerou e como isso foi interpretado. Poderíamos passar pelo sistema operacional, seguindo pelo gerenciador de janelas — caso o sistema operacional tenha essa separação —, e mais algumas camadas até chegar ao navegador para que o caractere referente a essa tecla seja interpretado.

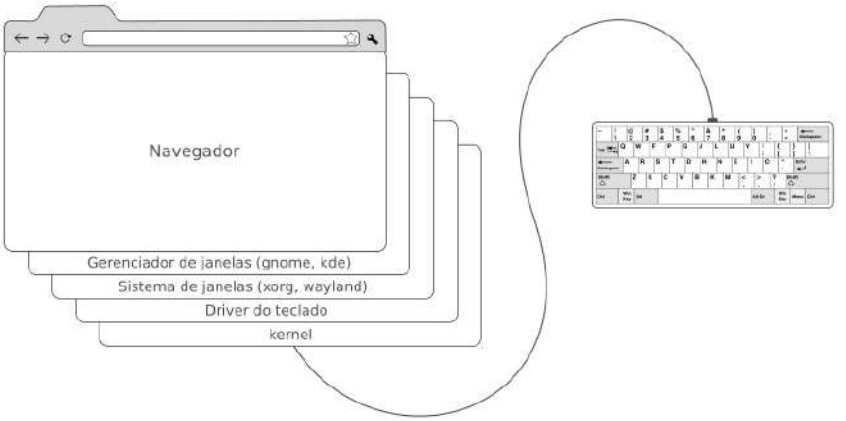


Figura 1.2: Exemplo das várias camadas entre o teclado e o navegador em um sistema GNU/Linux

Em vez disso, vamos seguir uma abordagem prática e focar nas partes que podem ter alguma relação com a vida de quem trabalha com internet. Portanto, assumiremos que a URI foi digitada na barra de endereços e a tecla *Enter* foi pressionada, assim dando início ao fluxo da requisição no navegador.

A seção de performance de navegação do W3C[1] nos dá uma boa ideia desse fluxo dentro do navegador.

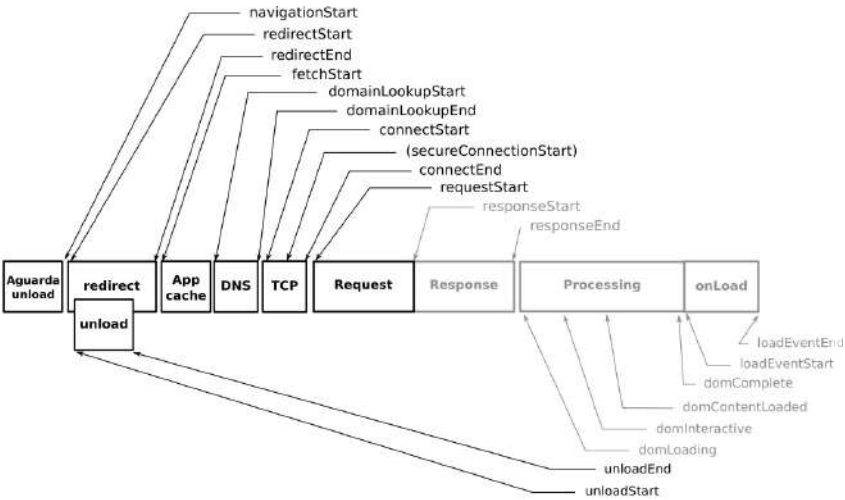


Figura 1.3: Fluxos de uma requisição no navegador. Baseado na imagem disponível em https://www.w3.org/TR/navigation-timing

Ele passa pelos seguintes estados: espera pela URL, possível redirecionamento, possível *cache*, DNS, TCP, requisição, resposta, processamento e página carregada. No primeiro e segundo capítulo, vamos seguir até o momento da requisição e voltaremos a falar sobre a resposta próximo ao fim da nossa jornada.

Neste primeiro capítulo, estudaremos como o navegador faz para chegar até a resolução de DNS, passando por todas as etapas que mencionamos no parágrafo anterior.

1.1 O NAVEGADOR DE ESTUDO

Em nossa jornada, vamos escolher a tecnologia que será usada para cada um dos passos. Com isso, será possível entrar nos detalhes de implementação e tirar a magia do processo.

Na data de escrita deste livro, o navegador mais utilizado é Google Chrome[2], representando mais de 57% do uso de navegadores no mundo. Esse valor é mais do que o dobro da utilização do Firefox, que está em segundo lugar.

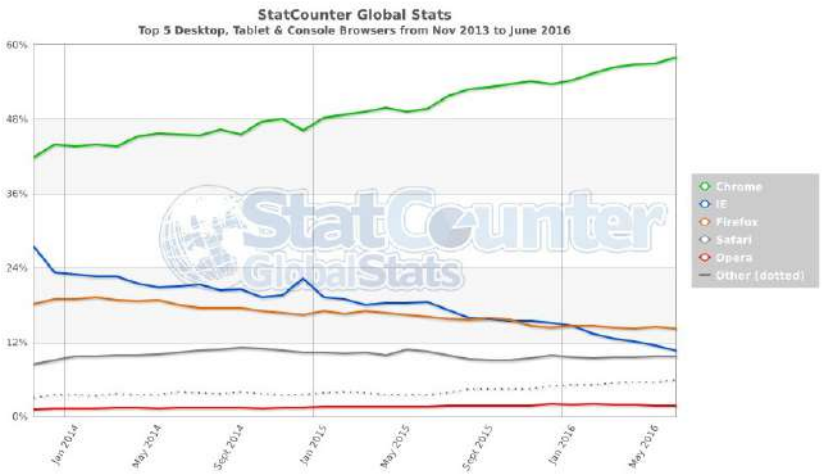


Figura 1.4: Gráfico de utilização dos navegadores no StatCounter, disponível em http://gs.statcounter.com/#browser-ww-monthly-201311-201606

Por esse motivo, usaremos o *Chromium*[3] como nosso navegador de estudo.

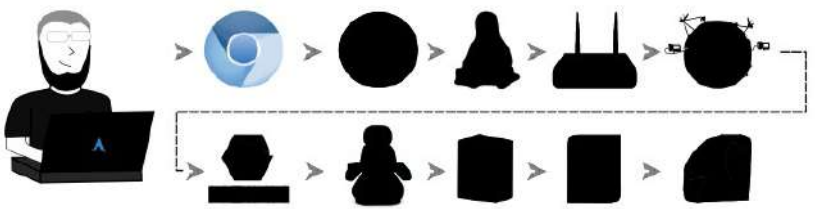


Figura 1.5: Chromium e o início da jornada

A principal diferença entre o Chromium e o Google Chrome é que o primeiro tem código aberto — conhecido como *open source*, em inglês — e não adiciona plugins pagos por padrão. O Google

Chrome usa o Chromium como base e adiciona alguns softwares proprietários, como codecs de MP3, por exemplo. Além disso, as novas versões são testadas pelos desenvolvedores do Google antes de enviar aos usuários[4].

1.2 ENTENDENDO O CONTEÚDO DA BARRA DE ENDEREÇOS

O Chromium pode receber requisições para carregamento de páginas de várias formas, e cada uma delas possui um fluxo diferente para ser iniciado. Ele poderia receber essa requisição vinda de um clique em um *link* da página, via JavaScript, ou pelo caminho que vamos seguir, que é a inserção da URI diretamente na barra de endereços.

Ao receber a URI, o navegador começa sua peregrinação para entregar ao usuário o que ele pediu. Para fazer isso, ele precisa primeiro entender o que o usuário digitou na barra de endereços, para então buscar o que ele quer. Apesar de parecer uma pergunta boba, o navegador precisa respondê-la antes de continuar: **o que o usuário digitou na barra de endereços é uma URI?**.

Essa pergunta não é tão estranha, afinal, já passamos do longínquo ano de 2015 e o *dr. Brown* já veio para o futuro ver as maravilhas modernas. Nesse futuro, os navegadores já fazem buscas diretamente pela barra de endereços.

O navegador possui configurações que definem qual buscador ele usará quando alguma coisa que não for uma URI for encontrada. Por esse motivo, é necessário que o navegador faça uma análise do que foi digitado para entender o que significa. Se o texto digitado não for uma URI, ele faz a busca utilizando seu buscador configurado; caso contrário, ele segue para o próximo passo. O nosso caso cai na segunda opção devido ao texto digitado ser a URI

`desconstruindoaweb.com.br` , fazendo o navegador seguir em frente.

Algumas pessoas podem não estar familiarizadas com o nome *URI*, que significa *Uniform Resource Identifier*, mas talvez tenham ouvido falar do nome mais comum, que é *URL*, ou *Uniform Resource Locator*. Para falar mais sobre esses termos, vamos ao momento tão esperado por muitos, a menção da nossa primeira *RFC*! Afinal, quem não gosta de ler dezenas de páginas técnicas em texto puro, para entender como funciona uma URI?

As RFCs, ou *Request For Comments*, são documentos que pesquisadores, engenheiros, cientistas de computação e outras pessoas de áreas correlatas disponibilizam, para revisão e possível criação de um padrão. Apesar do nome estranho, as RFCs não são apenas documentos para levantar discussões. Eles definem os padrões de muitas coisas fundamentais para a internet que conhecemos hoje.

O primeiro passo para que seja criado um padrão é fazer a submissão de um rascunho, conhecido como *draft*. Esse rascunho vai seguir o processo definido pelo *IETF* até que seja aprovado. O *IETF*, sigla para *Internet Engineering Task Force*, é responsável por cuidar de todos estes padrões e criar grupos de trabalho para que novos apareçam. Muitas RFCs serão mencionadas no decorrer deste livro, mas nenhuma será de leitura obrigatória, vamos mantê-las apenas como referência.

A diferença entre URI e URL, em uma tradução livre, é:

```
Uma URI pode ser definida como um localizador, enquanto uma URL
provê também informações sobre como este recurso pode ser
acessado.
```

Um exemplo para transformar uma URI em uma URL é adicionar o protocolo. Para ler a versão original, se estiver curioso,

você pode olhar no capítulo 1.1.3 da *RFC3986*[5], que está disponível nas referências.

No nosso caso, o navegador percebeu que o texto que ele recebeu, `descontruindoaweb.com.br` , é uma URI. Como ele precisa da URL para chegar até o destino, esse problema terá de ser resolvido antes de continuar.

1.3 ESCOLHENDO O PROTOCOLO

Para que o navegador possa fazer a conexão, será necessário ter um endereço completo que siga um padrão parecido com esse:

```
[protocolo://desconstruindoaweb.com.br]
```

No endereço digitado pelo usuário, não foi especificado nenhum protocolo antes do endereço do site, portanto, o navegador utilizará o famoso protocolo *HTTP (Hypertext Transfer Protocol)*. Ele será estudado com mais detalhes no *capítulo 4*. Antes de inserir diretamente o protocolo padrão para formar a URL, o navegador verifica a existência de um metadado chamado *HTTP Strict Transport Security*, mais conhecido como **HSTS**.

O *HSTS* é definido pela *RFC6797*[6]. Ele é uma especificação que dá aos sites a chance de dizer que preferem ser acessados somente via conexão segura. Por conexão segura, estamos nos referindo ao *HTTPS*, que será descrito com mais detalhes no *capítulo 5*.

O navegador recebe a informação do HSTS via cabeçalhos do *HTTP*[7], que funcionam como metadados para quem está recebendo o pacote. Os cabeçalhos possuem muitos tipos de informação que guiam diversas coisas na conexão e na forma de armazenar os dados. O cabeçalho responsável pelo HSTS é chamado de *Strict-Transport-Security* e é enviado na resposta usando a sintaxe padrão de cabeçalhos:

```
Strict-Transport-Security: max-age=31536000
```

O atributo `max-age` está em segundos e, nesse caso, equivale a 1 ano. Nesse exemplo, o navegador vai dar preferência ao HTTPS sobre HTTP durante 1 ano, até que o cabeçalho expire. Não se preocupe, pois seu *framework* de desenvolvimento provavelmente já faz isso para você, bastando configurá-lo para isso.

Figura 1.6: Exemplo de como seria o HSTS visto no Developer Tools do Chromium

No nosso caso, a aplicação do `desconstruindoaweb.com.br` não está devolvendo esse cabeçalho. Ela foi feita dessa maneira para que possamos testar tanto HTTP quanto HTTPS durante o estudo. Caso a configuração estivesse forçando o uso de HTTPS, o navegador receberia o cabeçalho do HSTS e redirecionaria a requisição prefixada com `https://` em vez de `http://`. Com isso, a nossa URI `desconstruindoaweb.com.br` se tornaria a URL `https://desconstruindoaweb.com.br`.

1.4 O CAMINHO ATÉ A REDE

Com o protocolo definido, o navegador vai continuar o processo de busca dos dados no computador de destino. Para requisitar esses dados, ele precisa criar um pacote de requisição e enviar pela rede até o destino, que no caso é o `desconstruindoaweb.com.br`.

O fluxo de uma requisição no Chromium não é um processo simples. Por ser um projeto feito para funcionar em vários sistemas operacionais, ele possui bastante código e abstrações para lidar com todas as condições necessárias para atender uma requisição web, em cada um desses ambientes.

Para facilitar essa separação e melhorar a performance, o

Chromium é dividido em vários processos que executam funções específicas. Esses processos se comunicam entre si para enviar e receber novas tarefas. Para ter uma ideia, vamos executar o Chromium com apenas uma aba que acessa o `desconstruindoaweb.com.br`:

```
$ chromium --incognito https://desconstruindoaweb.com.br
```

Em outro terminal, vamos ver quantos processos foram criados:

```
$ ps xufa | grep [c]hromium | wc -l
19
```

> **Curiosidade sobre esse grep**
>
> O comando `grep`, usado para conseguir o número de processo, está utilizando colchetes na letra **c** do chromium para evitar que o comando `grep chromium` também apareça na lista. Isso funciona graças às habilidades de expressões regulares do `grep`.

Executando em um computador comum, apenas um site criou **19** processos do Chromium. Isso acontece porque ele provisiona toda a infraestrutura para atender uma requisição, independente de ser um site ou vários. Esse valor pode ser alterado com configuração ou parâmetros quando o navegador é iniciado, além da possibilidade de variação entre computadores e sistemas operacionais.

O que vemos do navegador é apenas um desses processos, que é o principal. Ele é responsável por gerenciar os outros processos, que são especializados em outras tarefas, como a renderização das páginas, por exemplo.

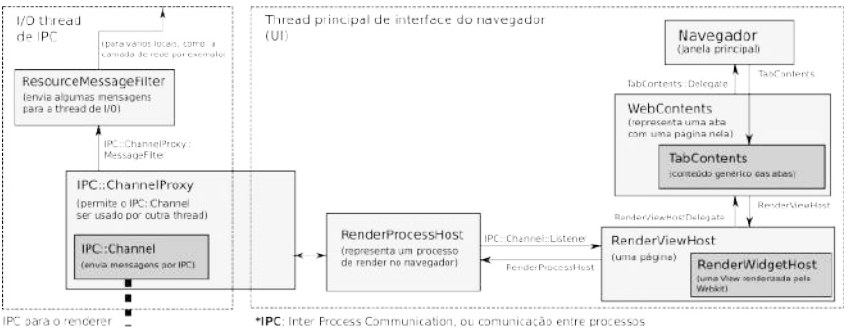


Figura 1.7: O processo do navegador. Baseado na arquitetura disponível em: http://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome

A parte que nos interessa nesse momento é relacionada ao *I/O*, ou entrada e saída de dados. Ela é executada em *threads* separadas para evitar que uma chamada que dependa da rede bloqueie o navegador. Por usar threads, o navegador cria um fluxo adicional dentro do processo, mantendo o fluxo principal funcionando independentemente. Com isso, conseguimos utilizar o navegador normalmente enquanto estamos aguardando uma página carregar.

Esse código de I/O utiliza o código de rede do Chromium[8] para fazer a interpretação dos protocolos e lidar com outras tarefas relacionadas. A documentação lista um fluxo de mais de 10 classes para que ele consiga chegar até os mecanismos de rede do sistema operacional.

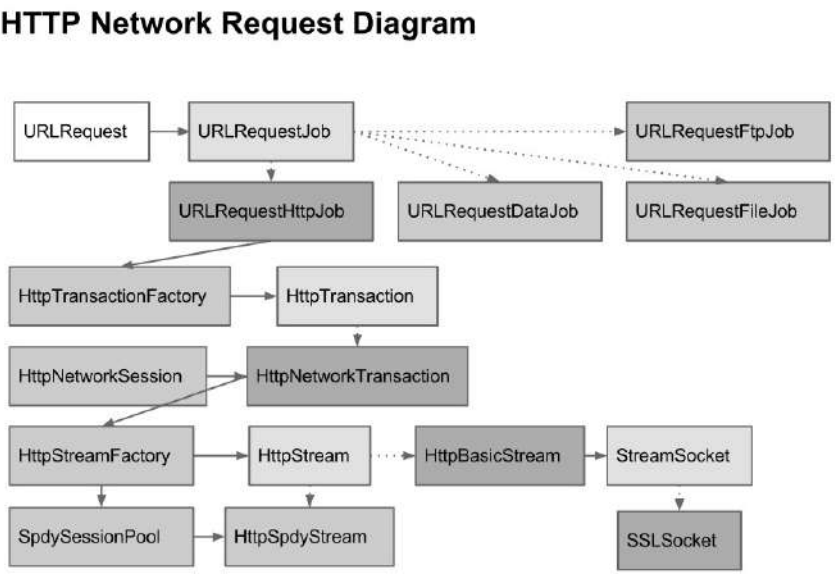


Figura 1.8: Camadas para uma chamada de rede no Chromium. Fonte: http://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome

Não é prático estudar cada uma dessas classes, mas vamos utilizar esse fluxo durante as próximas seções.

1.5 O CACHE DA URL

Agora que o navegador sabe qual é a URL completa, ele vai procurar se há algum cache relacionado a ela. O cache é uma cópia local dos arquivos que o servidor envia. Ou seja, se houve alguma requisição para essa URL no passado o navegador pode tê-los guardado.

É possível que o conteúdo do site seja somente estático, como um site institucional que não é atualizado com frequência. Com isso, o cache do navegador se faz muito útil ao desenvolvedor, pois vai reduzir a quantidade de requisições no servidor, trazendo mais velocidade no carregamento da página por ter o conteúdo completo localmente.

Há várias formas de manter o site no cache do navegador e os desenvolvedores se favorecem de cabeçalhos como *Cache-Control*, *Expires* e *Etag*, por exemplo. Imagino que você adivinharia onde está descrito o processo de cache com todos os detalhes. Sim, claro que é em uma RFC! A *RFC7234*[9] tem essa definição, mostrando todo o processo de cache e cabeçalhos utilizados. Apesar de o conteúdo ser bem acadêmico, é o melhor lugar para quem tem interesse em se aprofundar no tema.

Vamos explorar aqui o processo mais comum, que será o suficiente para entender uma requisição básica, que é o nosso caso.

Para vias de estudo, vamos assumir que o Chromium encontra uma versão que foi acessada há algum tempo. Para fazer isso, ele utiliza uma classe chamada `HttpCache` e as classes relacionadas a ela, como a `HttpCache::Transaction` [10], que é responsável por gerenciar tudo que é ligado a cache. Essa classe procura os dados no disco local, que por padrão fica em `~/.cache/chromium`, e verifica se pode usá-los. Caso não seja possível, o fluxo segue para a rede.

Há algumas formas de verificar se um cache é válido. Uma delas é verificar se a quantidade de segundos descrita no atributo `max-age`, parte do cabeçalho `Cache-Control`, é maior que a diferença entre a data de acesso e o horário atual. Caso o tempo de cache já tenha se extinguido, será necessário buscar por conteúdo atualizado no servidor.

Caso o cabeçalho `Cache-Control` não esteja presente, o navegador pode verificar a data no cabeçalho `Expires`. Se, ao comparar com a data atual, ele verificar que a data descrita já passou, o cache será invalidado.

Sabendo que o conteúdo do cache está expirado, ou seja, está há mais tempo gravado localmente do que o servidor acha que é o certo para se deixar em cache, o navegador vai buscar por conteúdo

novo. É possível que, apesar de o cache estar mais antigo que o esperado, o conteúdo ainda esteja atualizado quando comparado com o servidor. Nesse caso, o navegador faria todo o download do conteúdo novamente sem necessidade. Por esse motivo, as Etags podem ser enviadas em um cabeçalho chamado `If-None-Match` junto com o conteúdo. O servidor decide se deve enviar todo o conteúdo novamente, ou apenas uma resposta com um *status 304 Not Modified*, dizendo ao navegador que ele pode entregar o mesmo conteúdo e renovar a data do cache que ele já tem.

Figura 1.9: Cabeçalhos de cache no developer tools do Chromium em uma requisição real

Os status do HTTP são formas mais simplificadas de dar dicas sobre o que aconteceu na requisição, para quem está recebendo a resposta. No caso mencionado no parágrafo anterior, o servidor poderia enviar uma resposta com o *status 304*, que significa *não modificado*. Ao receber 304 como status, o navegador vai saber exatamente como agir.

Figura 1.10: Status 304, Not Modified para um arquivo CSS acessado pela segunda vez seguida

Obviamente nossa jornada não vai terminar aqui e não serviremos um conteúdo estático baseado no cache do navegador,

afinal, onde ficaria toda a diversão!? :)

Vamos assumir que o navegador não conseguiu validar o cache de forma alguma e vai precisar buscar conteúdo novo.

1.6 O NAVEGADOR E A RESOLUÇÃO DE NOMES

Nesse momento, o navegador possui a URL completa e sabe que não é possível usar nenhum cache que ele possui, portanto, é chegada a hora de desbravar a internet. Para sair para a internet, é necessário usarmos os endereços certos, e nosso endereço `https://desconstruindoaweb.com.br` não faz muito sucesso entre os equipamentos de rede.

Esses equipamentos lidam diretamente com endereços *IP* e outros protocolos que não são tão bonitos para humanos, como é a nossa URL. Com isso, o navegador precisa encontrar o endereço IP referente à URL que ele possui.

Um endereço IP é representado por um conjunto de números, sendo 32bits em um endereço *IPv4* e 128bits para um endereço *IPv6*. Um endereço IPv4, que é o que vamos usar no exemplo, tem o seguinte formato: **177.153.1.102**. Imagino que você já tenha visto por aí. Entraremos em mais detalhes sobre isso no *capítulo 2*.

Para que o navegador vá buscar mais informações sobre o domínio, ele mantém a URL dividida em 3 partes:

- **https://** — o protocolo;
- **desconstruindoaweb.com.br** — o domínio;
- **/** — o caminho a ser acessado no site. Caso não exista, o navegador vai adicioná-la.

Com essa divisão, o navegador sabe o protocolo de **como** acessar

o site, o domínio para **onde** acessar, e o caminho para **qual parte** do site será acessada.

Como dito anteriormente, é necessário transformar o domínio em um endereço IP. Para isso, alguns navegadores possuem um cache interno com um mapa entre domínios e endereços, como é o caso do Chromium. Por usar esse mecanismo, ele evita fazer novamente o processo completo de resolução de nomes ou repassar essa tarefa para agentes externos ao processo.

Como descrito no capítulo *High Performance Networking in Chrome*[11] do livro *The Performance of Open Source Applications*, que está disponível gratuitamente online[12], o Chromium possui duas formas de resolução de nomes. A primeira delas é um sistema desenvolvido por sua própria equipe de desenvolvimento, que é executado junto com o navegador. Na segunda forma, ele usa bibliotecas do sistema operacional para gerenciar a resolução de nomes. Será esta que vamos utilizar em nosso estudo para que possamos adentrar nos mecanismos do sistema operacional.

Para lidar com a resolução de nomes, o código do Chromium usa uma classe chamada `HostResolverImpl` [13]. Ela faz um cache da resolução de nomes para não precisar ir até o sistema operacional em todas as requisições, além de outros controles para evitar chamadas duplas. O seu principal trabalho é fazer a interface com as função de resolução de nomes do sistema operacional em execução.

Vamos estudar a interação desse código com as bibliotecas do sistema operacional no próximo capítulo.

1.7 RESUMO

O navegador precisa primeiro entender o que foi digitado na barra de endereços, que pode ser uma *URL* válida ou algum texto a

ser buscado em algum motor de busca. O usuário digitou apenas a *URI* `desconstruindoaweb.com.br` , deixando a cargo do navegador entender o que queremos. O navegador, por sua vez, percebe que o que foi digitado é uma *URI* e a transforma em uma *URL*. Para fazer isso, ele adiciona o protocolo e outras informações, se necessário. Como não estamos utilizando o cabeçalho *HSTS*, a aplicação não usou *HTTPS*, adicionando o protocolo HTTP na URL.

Com a *URL* completa, o navegador verifica se já possui cache para aquele dado. Caso ele possua e este ainda esteja válido, o navegador o serve diretamente. Se não houver cache, ou ele estiver inválido, o navegador começa o processo de buscar o conteúdo necessário na internet.

Para buscar o conteúdo na internet, é necessário saber o endereço *IP* do site em questão. O navegador possui um sistema interno de resolução de nomes que faz cache e busca as informações na internet, se necessário. Para nossos estudos, vamos utilizar a implementação do Chromium que depende de bibliotecas do sistema operacional para fazer a resolução de nomes.

1.8 REFERÊNCIAS

1. Seção sobre performance de navegação no W3C — https://www.w3.org/TR/navigation-timing/
2. Site do Google Chrome — https://www.google.com/chrome/browser/desktop/
3. Site do projeto Chromium — https://www.chromium.org/
4. Diferença entre o Chromium e o Google Chrome — https://chromium.googlesource.com/chromium/src/+/master/docs/chromium_browser_vs_google_chrome.md

5. *RFC3986*, definição de *URL* e *URI* — http://www.ietf.org/rfc/rfc3986.txt

6. *RFC6797*, definição do *HSTS* — https://tools.ietf.org/html/rfc6797

7. *RFC2616*, na página 31, possui a definição dos cabeçalhos do *HTTP* — https://tools.ietf.org/html/rfc2616#page-31

8. Repositório da parte de *networking* do Chromium — https://chromium.googlesource.com/chromium/src/net/

9. *RFC7234* e o cache no *HTTP* — https://tools.ietf.org/html/rfc7234

10. Classe responsável pelo cache no Chromium —https://chromium.googlesource.com/chromium/src/+/master/net/http/http_cache_transaction.cc

11. Capítulo sobre *High Performance Networking in Chrome* — http://aosabook.org/en/posa/high-performance-networking-in-chrome.html

12. Livro *The Performance of Open Source Applications* — http://aosabook.org/en/index.html

13. Classe responsável por resolver nomes no Chromium — https://chromium.googlesource.com/chromium/src/+/master/net/dns/host_resolver_impl.cc

Capítulo 2

O SISTEMA OPERACIONAL E A RESOLUÇÃO DE NOMES

O navegador fez o que tinha de ser feito e entendeu a URI que o usuário digitou, tirando uma URL disso. Mas tentar chegar do outro lado da internet só com a URL não é possível. Fazer isso seria como viajar para um país que usa um idioma e um alfabeto diferente do qual você conhece, e tentar encontrar um lugar. Fora da cidade natal da requisição, que é o computador local, o idioma falado tem palavras como *endereços MAC*, *endereços IP* e outras coisas mais.

Neste capítulo, vamos estudar como o navegador faz para conseguir essa tradução. Veremos como funciona a função `getaddrinfo` e como ela ajuda a traduzir um domínio para um endereço IP junto ao sistema operacional.

2.1 DEFININDO O SISTEMA OPERACIONAL

Antes de estudar a comunicação do navegador com as bibliotecas externas, vamos primeiro conhecer o sistema operacional que eles estão instalados. Cada sistema operacional pode possuir seu próprio conjunto de bibliotecas para ajudar no trabalho de traduzir um endereço IP para um domínio.

No nosso estudo, usaremos o sistema *GNU/Linux*.

POR QUE GNU/LINUX E NÃO SOMENTE LINUX?

Alguns podem achar estranho chamar o Linux de GNU/Linux, mas esse nome existe pelo fato de as distribuições utilizarem uma boa parte dos softwares básicos da GNU para funcionar. O Linux que temos hoje é uma junção do kernel, criado pelo Linus Torvalds, com uma boa parte do sistema GNU, criado por Richard Stallman, como uma forma de trazer liberdade para o uso de software.

Devido à popularidade do kernel Linux e à melhor forma de pronúncia, o nome GNU não é usado. Esse fato acaba sendo desleal com todo o trabalho feito pelo Richard Stallman, portanto, usaremos o termo GNU/Linux.

Poderíamos fixar uma distribuição como o Ubuntu, Debian, CentOS, ArchLinux, ou qualquer outra entre as dezenas que estão espalhadas por aí. Apesar disso, nesse estudo vamos utilizar apenas o termo GNU/Linux, pois o conteúdo estudado terá o mesmo conceito em qualquer uma dessas distribuições. A ideia não é mostrar funcionalidades específicas de um determinado tipo de distribuição, mas sim o básico de como o sistema lidará com a requisição web.

É interessante lembrar de que, mesmo que o sistema GNU/Linux não seja usado no seu computador pessoal, a probabilidade do servidor de destino da requisição utilizá-lo é grande. Esse é o caso do `desconstruindoaweb.com.br` , logo, todos os conceitos que veremos vão se aplicar quando a requisição voltar.

2.2 A GLIBC E AS CHAMADAS DE SISTEMA

Agora que fomos apresentados ao sistema operacional que está executando o navegador do usuário, podemos estudar as bibliotecas que estão entre os dois. A forma que o nosso navegador de estudo vai usar para fazer a interface com o sistema operacional é por meio da função `getaddrinfo`, que está definida na *glibc* do sistema.

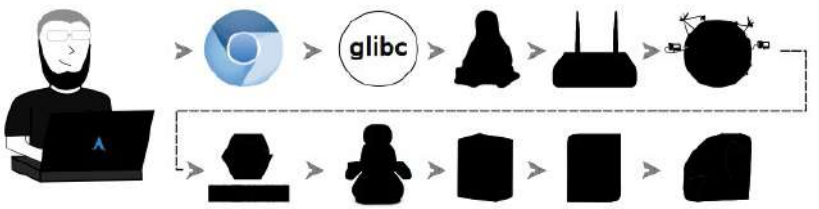


Figura 2.1: A glibc

A `glibc` [1], é a `libc` do projeto GNU[2]. Todo sistema da família Unix, ou *Unix-like*, precisa de uma `libc`. A `glibc` cumpre esse papel com os sistemas que usam os softwares da GNU.

A finalidade da `libc` é definir as funções mais básicas, como as interfaces para algumas chamadas de sistema, conhecidas como *syscalls*. Ela também fornece alguns comandos mais primitivos, como o `printf`, usado para imprimir informações na tela, e o `malloc`, usado para alocação de memória.

Curiosidades sobre os detalhes

A `libc` é uma das principais bibliotecas do sistema operacional GNU/Linux e segue um padrão importante chamado *POSIX (Portable Operating System Interface)*[3].

Para entender como os softwares do seu computador estão alocando memória (caso eles estejam utilizando a função `malloc` padrão para isso, claro) basta olhar na página da `glibc` e procurar por *malloc.c*[4], que é o arquivo que define a função que aloca memória, conhecida como `malloc`.

O mesmo vale para outras funções conhecidas, como o `printf`, `open`, `exit` etc. Basta olhar na raiz do projeto[5] e navegar por seus vários arquivos.

Mas não só de funções mais básicas é feita a `glibc`, há também funções que fazem muito mais do que expor syscalls de uma maneira mais amigável. A `getaddrinfo` é uma dessas funções que fazem bastante coisa não somente ligado ao sistema operacional.

2.3 A FUNÇÃO QUE RESOLVE NOMES

Antes da função `getaddrinfo`, os programadores tinham de usar uma função chamada `gethostbyname` para retornar o IP de um determinado domínio. Essa função dava mais trabalho para o programador, pois, ao utilizá-la, era necessário fazer códigos diferentes para cada uma das versões do protocolo IP. Com a função `getaddrinfo` o desenvolvedor recebe ambos os IPs e pode escolher qual caminho seguir.

A finalidade da função `getaddrinfo` é traduzir um nome para

endereços IP, além de trazer informações úteis para a criação de uma conexão. O sistema operacional provê uma interface para a criação de conexões, e a função `getaddrinfo` transforma um domínio na estrutura de dados que ele precisa para criar essa conexão. Para quem gosta de conhecer as definições formais, a função `getaddrinfo` faz parte do padrão *POSIX*, sendo possível ver a sua definição formal no site da especificação[6].

Um dos passos para traduzir um domínio em um endereço IP é verificar o sistema de *hosts* que os sistemas operacionais mantêm. No GNU/Linux, esse sistema utiliza um arquivo de texto, localizado em `/etc/hosts` .

O sistema de *hosts* provê uma forma de o usuário informar que um determinado IP possui um determinado domínio. Um exemplo de uso seria a migração do servidor do `desconstruindoaweb.com.br` para um novo servidor, mais moderno e atualizado. Por um tempo, teríamos duas aplicações idênticas sendo executadas, uma em cada servidor. Entretanto, o domínio `desconstruindoaweb.com.br` ainda apontaria para o servidor original.

Nós poderíamos usar o nosso arquivo `/etc/hosts` local para alterar o IP que o domínio `desconstruindoaweb.com.br` está apontando. Com isso, seria possível acessar a nova aplicação utilizando o mesmo domínio, bastando adicionar uma nova linha com o IP do novo servidor e o domínio que queremos acessar.

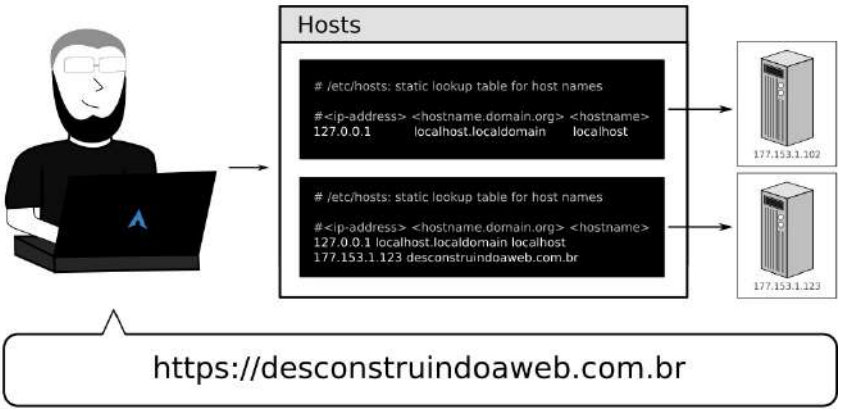


Figura 2.2: Utilizando o sistema de hosts para alterar o domínio na máquina local

Vamos ver um exemplo prático da função `getaddrinfo` e de como o uso do arquivo `/etc/hosts` influencia nos dados que ela retorna. Essa função não possui um comando para ser executado diretamente do terminal, mas podemos executá-la com uma linha de alguma linguagem de script. Como exemplo, vamos utilizar Python e Ruby:

```
$ python -c 'import socket;print socket.getaddrinfo("desconstruind
oaweb.com.br","http")'

[(2, 1, 6, '', ('177.153.1.102', 80)), (2, 2, 17, '', ('177.153.1.
102', 80)), (2, 1, 132, '', ('177.153.1.102', 80)), (2, 5, 132, ''
, ('177.153.1.102', 80))]

$ ruby -e "require 'socket'; p Socket.getaddrinfo('desconstruindoa
web.com.br', 'http')"

[["AF_INET", 80, "177.153.1.102", "177.153.1.102", 2, 1, 6], ["AF_
INET", 80, "177.153.1.102", "177.153.1.102", 2, 2, 17], ["AF_INET"
, 80, "177.153.1.102", "177.153.1.102", 2, 1, 132], ["AF_INET", 80
, "177.153.1.102", "177.153.1.102", 2, 5, 132]]
```

Podemos ver que, apesar da diferença de sintaxe, os valores são os mesmos em ambos os casos. Esses são os valores necessários para o sistema operacional criar um *socket*, que vamos estudar mais a fundo no próximo capítulo, quando chegarmos no sistema operacional.

Vamos adicionar uma nova linha no arquivo `/etc/hosts`, para ter certeza de que as modificações feitas lá são refletidas no resultado da função `getaddrinfo`:

```
$ su -c 'echo "177.153.1.123   desconstruindoaweb.com.br" >> /etc/
hosts'
```

Com o arquivo modificado, executaremos os mesmos comandos para ver o resultado:

```
$ python -c 'import socket;print socket.getaddrinfo("desconstruind
oaweb.com.br","http")'

[(2, 1, 6, '', ('177.153.1.123', 80)), (2, 2, 17, '', ('177.153.1.
123', 80)), (2, 1, 132, '', ('177.153.1.123', 80)), (2, 5, 132, ''
, ('177.153.1.123', 80))]

$ ruby -e "require 'socket'; p Socket.getaddrinfo('desconstruindoa
web.com.br', 'http')"

[["AF_INET", 80, "177.153.1.123", "177.153.1.123", 2, 1, 6], ["AF_
INET", 80, "177.153.1.123", "177.153.1.123", 2, 2, 17], ["AF_INET"
, 80, "177.153.1.123", "177.153.1.123", 2, 1, 132], ["AF_INET", 80
, "177.153.1.123", "177.153.1.123", 2, 5, 132]]
```

Podemos ver que, em ambos os casos, o valor retornado foi o que adicionamos no arquivo `/etc/hosts`. Lembre-se de remover essa linha do seu arquivo de *hosts* para evitar problemas. ;)

Agora que a função `getaddrinfo` retornou os endereços IP referentes ao domínio que informamos, podemos escolher qual dos resultados vamos utilizar. Entre a lista de endereços IP retornados, podemos ter tanto *IPv4* como *IPv6* e a aplicação que fará a escolha de qual usará.

2.4 O PROTOCOLO IP E SUAS VERSÕES

O protocolo IP é responsável por gerenciar o endereço dos pacotes. Esse endereço vai ajudar os dispositivos de rede, como roteadores, a guiar um pacote pela rota certa até o destino. Atualmente, ele possui duas versões, uma é conhecida por IPv4 e a

outra por IPv6, devido ao número de suas versões que vai no cabeçalho do pacote.

Cadê o IPv5?

Podemos deixar passar que as várias versões anteriores a 4 praticamente não existem, afinal, os protocolos eram apenas experimentos na época. Mas a pergunta que fica na cabeça é: *o que aconteceu com a versão 5?*.

Apesar de nunca ter sido conhecido popularmente como IPv5, o protocolo que tem o número 5 no campo de versão do IP é um protocolo chamado *Internet Stream Protocol*, também conhecido como ST-II+[7].

Como o número 5 já estava em uso por esse protocolo, foi mais simples utilizar o número 6 para fazer a nova versão do protocolo IP, sem gerar incompatibilidades.

Considerando o ano de 2016, temos mais de 3 bilhões e 400 milhões de usuários conectados à internet[8]. Esse número só cresce nos últimos anos, trazendo a necessidade de se criar um protocolo que disponha de mais endereços que o IPv4.

O IPv4, definido pela *RFC791*[9], possui um espaço de 32 bits no cabeçalho, podendo compor até 2^{32} ou 4.294.967.296 endereços distintos. Na época da criação do protocolo, esse número parecia mais do que suficiente para suportar todos os dispositivos conectados à internet. Como podemos ver hoje em dia, temos muito mais dispositivos do que se podia imaginar algumas décadas atrás, e essa quantidade de endereços se tornou pequena para a demanda.

O IPv6, definido pela *RFC2460*[10], possui espaço de 128 bits no cabeçalho, podendo compor até 2^{128} endereços distintos. Esse número equivale à quantia de `340.282.366.920.938.463.463.374.607.431.768.211.456` endereços distintos, o que daria uma quantidade suficiente de endereços para cada um dos 3 bilhões e 400 milhões de usuários que mencionamos. Uma conta simples com Ruby para exemplificar:

```
340_282_366_920_938_463_463_374_607_431_768_211_456 / 3_400_000_00(

# => 100083049094393665724521943362
```

Considerando a quantidade atual de usuários de internet, teríamos **apenas** `100.083.049.094.393.665.724.521.943.362` endereços para cada um. O problema da distribuição de endereços IP estará resolvido por um bom tempo, ficando apenas o problema da velocidade de conexão Wi-Fi em eventos de tecnologia.

Como o IPv6 e o IPv4 são protocolos diferentes, não é possível utilizá-los juntos de uma forma transparente. Com isso, temos sites que possuem uma infraestrutura para IPv4 e outra para IPv6, deixando a cargo do cliente escolher qual delas usar.

Para ajudar com esse problema, foi desenvolvido um algoritmo chamado *Happy eyeballs*, que provê uma implementação recomendada para lidar com os dois protocolos de uma forma inteligente.

2.5 HAPPY EYEBALLS

O *Happy eyeballs*, definido pela *RFC6555*[11], é um algoritmo que favorece o uso de IPv6 sobre o IPv4, além de ajudar a tornar a resolução de nomes mais rápida quando há registro para ambos.

Vamos a uma explicação passo a passo do que acontece no

algoritmo:

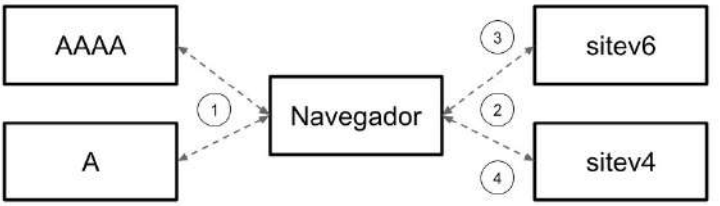


Figura 2.3: Interações do navegador durante o Happy Eyeballs

1. O algoritmo envia simultaneamente duas requisições para o servidor de domínios, uma pedindo o registro IPv6 (registro *AAAA*) e outra pedindo o registro IPv4 (registro *A*). Caso não receba resposta de um dos dois, o outro é utilizado.
2. Ao receber resposta dos dois, o cliente, que no nosso caso é o navegador, tenta se comunicar com ambos e aguarda as respostas.
3. Caso o IPv6 responda, ele vai ser favorecido sobre o IPv4. Nesse caso, o navegador envia um comando *reset* para o IPv4, desistindo da conexão.
4. Caso o IPv6 não responda a nenhuma das tentativas, o cliente continua a conexão com o IPv4 que já estava em andamento.

Atualmente, vários sistemas implementam esse algoritmo, incluindo navegadores como o Opera, Chrome e Firefox, o sistema operacional OS X e até mesmo o comando `curl` .

O Happy eyeballs no getaddrinfo e na implementação de DNS do Chromium

Como mencionado no capítulo anterior, apesar de estarmos estudando apenas a versão via `getaddrinfo`, a equipe do Chromium implementou seu próprio código para converter domínios em IPs. Eles decidiram seguir por esse caminho para ter mais controle sobre todas as requisições de tradução de domínios, algo que não era possível com a implementação via `getaddrinfo`.

Para entender a otimização que a equipe do Chromium conseguiu por usar seu próprio sistema de resolução de domínios, vamos utilizar um exemplo. O navegador dispara as chamadas para o servidor de DNS pedindo os registros de domínio do IPv6 e IPv4 paralelamente. Ambos retornam, e o navegador usa o IPv6 caso esteja totalmente funcional, senão ele aguarda a versão IPv4 para utilizá-la.

Na versão com `getaddrinfo`, isso não é possível. Como vimos nos exemplos com Python e Ruby, a função `getaddrinfo` retorna os dois endereços juntos, deixando o processo um pouco mais lento.

Com o IP escolhido, o navegador vai enviar uma chamada para conseguir a informação que o usuário quer daquele domínio.

Neste capítulo, ainda assumimos que o DNS funciona magicamente, trazendo os IPs que precisamos. No próximo, vamos descobrir o que ele faz para conseguir esses IPs e como o sistema operacional faz para se comunicar com ele.

2.6 RESUMO

Após separar o domínio a ser acessado da URI, o navegador sabe como procurar pelo endereço que precisa para seguir seu caminho pela internet. No nosso caso, ele vai evitar sua implementação interna de resolução de nomes e delegar para o sistema operacional essa tarefa. O navegador faz isso através de uma função da glibc chamada `getaddrinfo` . Essa função faz o trabalho de fazer as requisições para os registros de IPv4 e IPv6 no servidor de DNS. Com essas informações, o sistema operacional pode criar uma conexão com o servidor de destino, possibilitando o envio de informações para esse endereço.

Com os endereços IPv6 e IPv4 (caso tenha registro para os dois), o navegador vai escolher qual deles utilizar. Para fazer essa escolha, ele usa um algoritmo chamado Happy Eyeballs. Este tenta ao máximo prevalecer o uso do IPv6, sem prejudicar a performance quando ele não está disponível.

2.7 REFERÊNCIAS

1. glibc — https://www.gnu.org/software/libc/
2. O projeto GNU — http://www.gnu.org/gnu/thegnuproject.pt-br.html
3. Especificação *POSIX* — http://pubs.opengroup.org/onlinepubs/9699919799/
4. Código da função *malloc* — https://sourceware.org/git/?p=glibc.git;a=tree;f=malloc
5. Raiz do projeto da *glibc* — https://sourceware.org/git/?p=glibc.git;a=tree

6. `getaddrinfo` no site da especificação *POSIX* — http://pubs.opengroup.org/onlinepubs/9699919799/functions/freeaddrinfo.html

7. *RFC1819* que define o protocolo *Internet Stream Protocol*, suposto IPv5 — https://tools.ietf.org/html/rfc1819

8. Dados da *Internet live stats* sobre os usuários de internet — http://www.internetlivestats.com/internet-users/

9. *RFC791* que define o IPv4 — https://tools.ietf.org/html/rfc791

10. *RFC2460* que define o IPv6 — https://www.ietf.org/rfc/rfc2460.txt

11. *RFC6555* que define o Happy Eyeballs — https://tools.ietf.org/html/rfc6555

Capítulo 3

RESOLUÇÃO DE NOMES NA REDE

Agora que vimos o fluxo de como o navegador interage com a *glibc* e consegue o IP referente a um domínio, estudaremos **como** o processo completo funciona. Vamos partir da execução da função `getaddrinfo`, passando pelas camadas para sair do sistema operacional, pelos servidores responsáveis por cuidar da tradução do domínio, até a entrega do endereço IP para o navegador.

Este capítulo vai andar bem próximo ao capítulo anterior, e tentará ser uma versão "não chata" das explicações básicas das aulas de redes que vemos por aí. Além disso, ele será uma base para o que vamos utilizar nos próximos.

3.1 O MODELO OZZY

```
O que? Modelo Ozzy? O Osbourne? Nunca ouvi falar nesse modelo!
```

Na realidade esse é o **modelo OSI**, mas como esse é um nome mais acadêmico, alguns passariam correndo só de ver o título. O **modelo OSI** é um padrão ISO[1], que define um modelo conceitual para as comunicações. Esse modelo é dividido em sete camadas, sendo elas:

1. Física
2. Enlace

3. Rede
4. Transporte
5. Sessão
6. Apresentação
7. Aplicação

Em vez de explicar cada uma delas, vamos seguir por um caminho menos conceitual, utilizando um modelo simplificado. Não é do escopo deste livro explicar todo o conceito da camada OSI, mas caso esteja curioso, o livro *Computer Networks*[2], do conhecido professor Tanenbaum, aborda o modelo completo. No nosso estudo, usaremos um dos modelos que ele apresenta, que é uma simplificação do modelo OSI com coisas mais práticas. A partir de agora, chamaremos esse nosso modelo simplificado de "Modelo Ozzy" para distingui-lo do modelo conceitual.

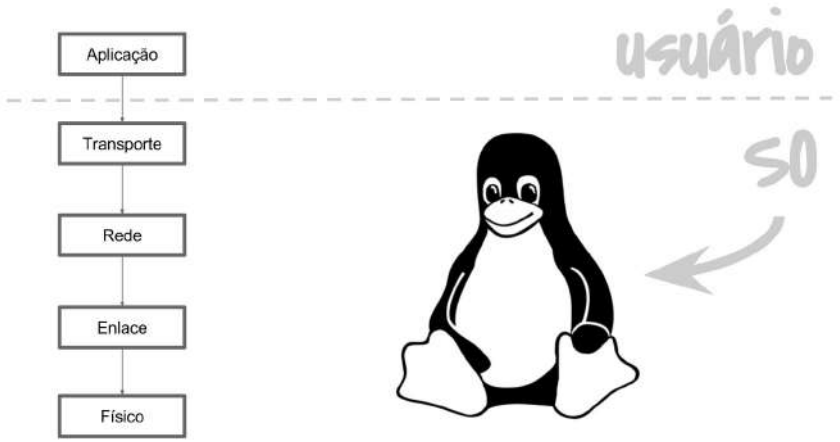


Figura 3.1: Modelo Ozzy

Esse modelo tem apenas cinco camadas, e não sete como o anterior. São elas:

- **Camada de aplicação** — Aqui ficam os protocolos de

nível mais alto, como é o caso do HTTP (para transferência de *hipertexto*), SMTP (para envio de e-mail), FTP (para transferência de arquivos) e DNS (para resolução de nomes), que são os que vamos estudar agora, e mais alguns.

- **Camada de transporte** — Nessa camada ficam protocolos como TCP e o UDP. Esses protocolos são responsáveis por empacotar os dados e manter controle do conteúdo para que sejam enviados.

- **Camada de rede** — Lida com a entrega dos pacotes no destino e seu caminho para chegar até lá. Ela é responsável pelo roteamento dos pacotes pela rede, direcionando-os para o melhor caminho. O IP é um dos protocolos dessa camada.

- **Camada de enlace** — Cuida de enviar as mensagens entre dois dispositivos conectados, evitando erros e cuidando do envio e recebimento. *Ethernet* e *Wi-Fi* ficam nessa camada, vamos estudá-los mais a fundo no *capítulo 6*.

- **Camada física** — Envia os bits para o outro lado por um meio físico. Alguns exemplos de meio físico são os cabos e as ondas de rádio.

Numeração das camadas

Vamos mencionar a camada de aplicação como a primeira no nosso modelo "Ozzy", por estarmos olhando do ponto de vista da requisição web. É importante salientar que, no modelo OSI, a camada de aplicação é última, com o número 7.

No nosso modelo, as primeiras camadas são as mais próximas das aplicações com que estamos acostumados a lidar, como o navegador, por exemplo. As últimas são as camadas mais distantes, que ficam mais próximas do hardware.

A camada de aplicação não é gerenciada pelo kernel do sistema, portanto, seus protocolos são implementados por aplicações do usuário. Um exemplo de aplicação de usuário é o navegador, que implementa o protocolo HTTP para se comunicar com o servidor.

Vamos começar nossa análise de como conseguir o domínio. Para isso, usaremos um protocolo que fica na camada de aplicação, o *DNS*.

3.2 O PROTOCOLO DNS

O DNS, sigla para *Domain Name System*, é um protocolo para resolução de nomes que fica na camada de aplicação. Por estar nessa camada, cabe aos softwares implementarem uma forma de utilizá-lo, ou seja, o *kernel* (o núcleo do sistema operacional) não possui implementação como acontece com os protocolos das outras camadas.

Como vimos no capítulo anterior, o Chromium possui sua própria implementação de DNS, que pode ser uma ótima fonte de pesquisa para os mais curiosos. O código-fonte está disponível no diretório `chromium/src/net/dns` do Chromium[3].

Para esse estudo, vamos seguir o mesmo caminho que já estávamos traçando e utilizar a função `getaddrinfo` da glibc. Ela será responsável por fazer o trabalho de interagir com o protocolo DNS[4].

Existem vários tipos de registros no protocolo DNS que são responsáveis por guardar informações sobre o domínio em questão.

Os mais importantes para o nosso estudo são:

- *A* — O endereço IPv4 do domínio;
- *AAAA* — O endereço IPv6 do domínio;
- *CNAME* — Um apelido para o domínio em questão.

Os registros A e AAAA são equivalentes, sendo que cada um serve para uma versão do protocolo IP. Quando buscamos o endereço IP de um determinado domínio, os registros do tipo A e AAAA são os lugares para procurar.

Vamos usar um comando para consulta de DNS para ver o registro A para o nosso domínio. Como estamos utilizando o GNU/Linux para estudo, usaremos o comando `dig`, mas poderia ser outro, como o `nslookup`, por exemplo.

```
$ dig desconstruindoaweb.com.br

; <<>> DiG 9.10.4 <<>> desconstruindoaweb.com.br
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 10234
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITION
AL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;desconstruindoaweb.com.br.    IN    A

;; ANSWER SECTION:
desconstruindoaweb.com.br. 21599 IN    A    177.153.1.102

;; Query time: 255 msec
;; SERVER: 192.168.1.1#53(192.168.1.1)
;; WHEN: Sat May 28 12:42:54 BRT 2016
;; MSG SIZE  rcvd: 70
```

A seção `ANSWER SECTION` mostra que ele trouxe o endereço `177.153.1.102` que estava na seção `A`. O IP da aplicação pode mudar com o tempo, portanto, é possível que o retorno não seja o

mesmo se esse comando for executado no futuro. Mas independente do IP retornado, o conceito é o mesmo.

Já o CNAME é utilizado para colocar apelidos e criar subdomínios. Um exemplo de uso pode ser visto no GitHub Pages[5], que cria um subdomínio caso você crie uma página estática para o seu usuário. Fazendo uma requisição na página de algum usuário, podemos ver o CNAME em ação:

```
$ dig pothix.github.io
; <<>> DiG 9.10.4 <<>> pothix.github.io
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 57614
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL:
 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;pothix.github.io.        IN    A

;; ANSWER SECTION:
pothix.github.io.    3599    IN    CNAME    github.map.fastly.net.
github.map.fastly.net.    29    IN    A    199.27.79.133

;; Query time: 356 msec
;; SERVER: 192.168.1.1#53(192.168.1.1)
;; WHEN: Sat May 28 12:45:20 BRT 2016
;; MSG SIZE  rcvd: 96
```

Novamente na seção `ANSWER SECTION`, está o que estamos procurando. O domínio `pothix.github.io` tem uma entrada CNAME que aponta para `github.map.fastly.net`. Não se preocupe muito com os nomes que o GitHub retorna, mas sim com o mapeamento de subdomínio criado utilizando CNAME. Isso será o suficiente para continuarmos nossos estudos, portanto, podemos avançar para as próximas camadas.

Antes de continuar, há algo faltando nessa seção, e é claro que temos centenas de páginas de RFC do protocolo DNS para os mais

curiosos. O protocolo DNS é definido pela *RFC1035*[6]. Ao lê-la e ler seus anexos, é possível encontrar todos os detalhes mais obscuros do protocolo.

3.3 DNS E O UDP/IP

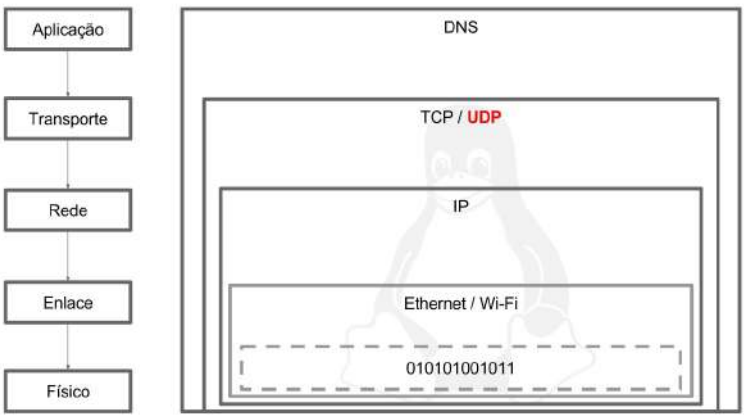


Figura 3.2: Camadas que o protocolo DNS vai passar

No modelo "Ozzy", a primeira camada é a de aplicação. Por estar em cima de todas as outras, ela as utiliza para se completar e atingir seus objetivos, que é enviar e receber conteúdo do servidor. As próximas camadas são a de transporte, que contém os protocolos TCP e UDP, e a de rede, que contém o protocolo IP. As duas andam juntas e é mais interessante olharmos os protocolos juntos também. Por padrão, o DNS usa o protocolo UDP juntamente como o protocolo IP para envio dos pacotes, portanto, daremos uma olhada nesse protocolo "*UDP/IP*".

O protocolo UDP (*User Datagram Protocol*) é um protocolo definido pela *RFC768*[7] de 1980, que incrivelmente tem apenas 3 páginas. No protocolo UDP, é quem está usando que deve definir as

portas de origem e destino, o tamanho do pacote, o *checksum*, e os dados para envio. O *checksum* é um código gerado baseando-se no conteúdo do pacote para fazer a verificação de integridade dos dados quando chegar ao destino.

Já o protocolo IP, como vimos no capítulo anterior, é responsável por guiar um pacote de um ponto a outro. Esse protocolo ajuda dispositivos de rede, como roteadores, a enviar um pacote pela rota certa até que ele chegue ao seu destino. Para esse estudo, utilizaremos a versão 4 do IP, também conhecido como IPv4. Apesar do crescimento no uso da versão 6 do IP, a versão 4 ainda domina a internet[8], sem previsão para mudança imediata.

O *UDP/IP* é a junção dos dois protocolos, criando uma forma de empacotar e enviar os dados até o destino. Essa combinação tem uma particularidade, pois o UDP não controla a forma como os pacotes vão chegar, e não se preocupa se algum dos dados vai se perder no meio do caminho.

Uma analogia sobre como o UDP/IP funciona seria pensar em uma carta comum sendo colocada no correio. Quando vamos usar o serviço de correio, nós preparamos um envelope (UDP), colocamos o endereço (IP), colocamos o conteúdo dentro do envelope (dados) e mandamos para a agência do correio (internet). Após ser entregue na agência, nós não sabemos o que aconteceu com o envelope até que ele chegue ao destino e a pessoa que o recebeu nos avise de alguma forma, caso ela faça. Devido à natureza do UDP/IP, o cliente precisa controlar se a resposta foi recebida e, caso não seja, ele mesmo precisa reenviar a requisição.

Curiosidade sobre o meio de transporte do UDP

A analogia com o correio está na natureza do protocolo UDP. O argumento *SOCK\DGRAM*, que a glibc passa para o socket do sistema operacional, significa *datagram*, ou **datagrama** em português. O nome *datagram* vem da mistura da palavra *data*, ou *dados*, e *telegram* de *telegrama*.

Os conjuntos de bits gerados pelos protocolos vão ser enviados para a internet por meio de um *socket*, criado via *syscall* no sistema operacional.

3.4 OS SOCKETS

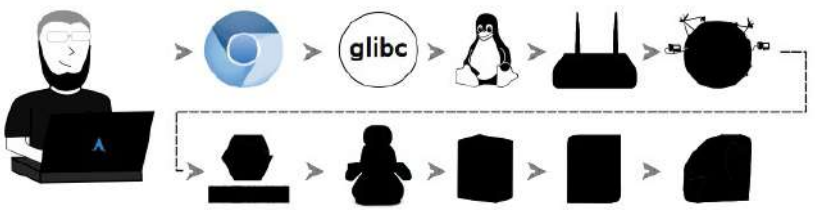


Figura 3.3: Adentrando no sistema operacional

Um socket é uma abstração criada pelo sistema operacional para fazer a transferência de dados de um ponto a outro. Ele vai ser usado em vários pontos da nossa jornada. Primeiramente, será utilizado para conectar o computador local ao servidor de domínios, e depois para fazer a conexão com servidor do site do `desconstruindoaweb.com.br`.

Existem alguns tipos diferentes de socket. O *Unix Socket*, por exemplo, é usado para fazer comunicação entre processos, ou seja,

um processo coloca dados lá e outro pode consumir. O tipo de socket que vamos utilizar em nosso estudo, e que a glibc usa na função `getaddrinfo`, é o *socket de rede*. A partir de agora, vamos nos referir ao *socket de rede* apenas como *socket* para facilitar, ok? :)

No nível de sistema operacional, o socket é um *file descriptor*. E nessa hora alguns podem se perguntar:

```
Peraí, um file o quê?
```

Como analogia, imagine que, para se comunicar pela internet, você está criando um arquivo, só que ao ler ou escrever nesse arquivo, o sistema operacional envia para a rede os dados da transação. Vale lembrar que isso acontece para o caso do *socket de rede*, foco do nosso estudo. Esse é o motivo que faz algumas pessoas afirmarem que *em sistemas Unix tudo é um arquivo*.

Ao pedir um socket para o sistema operacional via `glibc`, nós podemos informar alguns parâmetros:

```
// https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/posix/
getaddrinfo.c;hb=HEAD#l2515
fd = __socket (af, SOCK_DGRAM, IPPROTO_IP);
```

O parâmetro `af` é uma informação interna que não precisamos nos preocupar, mas os outros dois parâmetros são importantes para nós. Como segundo parâmetro, devemos passar qual protocolo da camada de transporte vamos utilizar, sendo `SOCK_STREAM` para TCP e `SOCK_DGRAM` para UDP. Devido ao DNS usar UDP por padrão, a `glibc` usa `SOCK_DGRAM`. O terceiro parâmetro diz qual o protocolo da camada de rede que deve ser utilizado, e no nosso caso usaremos o protocolo IP, ou `IPPROTO_IP`, no código.

Curiosidade da camada de rede

Só falamos do protocolo IP na camada de rede, mas há outros protocolos também. Um outro protocolo que também fica nessa camada é o *ICMP*. Esse protocolo é usado pelo comando `ping` do GNU/Linux, que também está disponível em outros sistemas operacionais. A função do comando `ping` é enviar pequenos pacotes de controle para um destino especificado. Esses pacotes geralmente são utilizados para verificar se o servidor de destino está apto a respondê-los.

Tomando o GNU/Linux como exemplo, caso a glibc quisesse utilizar ICMP, ela usaria `IPPROTO_ICMP` em vez de `IPPROTO_IP`.

Após conseguir o *file descriptor*, que é a variável `fd` no código, a `glibc` usa a syscall chamada `connect` [9]. Como estamos utilizando `SOCK_DGRAM`, e por consequência UDP, essa syscall vai apenas atribuir o endereço do destinatário no socket, e não fará uma conexão. Isso acontece porque o UDP não faz conexões, apenas "envia a carta", como descrevemos na analogia do correio.

Podemos ver o processo acontecendo quando utilizamos o comando `strace`. Ele observa todas as syscalls que estão sendo executadas por um determinado processo. Para usá-lo, vamos fechar o navegador caso ele esteja aberto e abrir uma nova instância com apenas uma aba. Isso vai facilitar na distinção do retorno do comando.

Para ver as syscalls executadas pelo Chromium, basta executar o seguinte comando como usuário *root*:

```
# strace -f -e socket,connect -p PIDS,DO,CHROMIUM,AQUI
```

Cada computador vai ter os processos do Chromium com seus próprios ***P**rocess **ID**entifiers*, portanto, substitua `PIDS,DO,CHROMIUM,AQUI` por seus próprios PIDs. Eles podem ser conseguidos utilizando o comando `ps` com as flags `aux` (eu uso `xufa` por achar melhor a combinação de palavras, e para mim chega a ser engraçado). Ao executar esse comando, você deve ver, entre várias outras coisas, algo como isto:

```
[pid  8739] socket(PF_INET6, SOCK_DGRAM, IPPROTO_IP) = 134
[pid  8739] connect(134, {sa_family=AF_INET6, sin6_port=htons(53),
 inet_pton(AF_INET6, "2001:4860:4860::8888", &sin6_addr), sin6_flo
winfo=0, sin6_scope_id=0}, 28) = -1 ENETUNREACH (Network is unreac
hable)
[pid  9010] socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 134
[pid  9010] connect(134, {sa_family=AF_INET, sin_port=htons(0), si
n_addr=inet_addr("177.153.1.102")}, 16) = 0
[pid  8739] socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 134
[pid  8739] connect(134, {sa_family=AF_INET, sin_port=htons(80),
sin_addr=inet_addr("177.153.1.104")}, 16) = -1 EINPROGRESS (Operat
ion now in progress)
```

Ali está mostrando boa parte do que vimos nos últimos capítulos. Vamos seguir com a explicação passo a passo:

- `socket(PF_INET6, SOCK_DGRAM, IPPROTO_IP)` — A syscall `socket` cria novos sockets seguindo o conceito que estudamos há pouco. O argumento `PF_INET6` diz que o protocolo usado é IPv6, o `SOCK_DGRAM` informa que estamos usando UDP e o `IPPROTO_IP` diz que estamos usando o protocolo IP. Portanto, temos um socket UDP/IP com IPv6.
- `connect(134, {sa_family=AF_INET6, ...})` — Aqui ele está tentando se conectar via IPv6 utilizando a syscall `connect`, fazendo algo parecido com o *Happy Eyeballs* que vimos no capítulo anterior. O `Network is unreachable` nos mostra que não há IPv6 configurado para este servidor.
- `socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) =`

`134` — Essa chamada para a syscall `socket` é parecida com a da primeira linha, mas nesse caso ele está pedindo um socket UDP/IP utilizando IPv4 em vez de IPv6.

- `connect(134, {sa_family=AF_INET ...})` — Aqui acontece a conexão para buscar o domínio.

A partir daí começam as conexões para buscar o conteúdo do site, fazendo a conexão TCP com `SOCK_STREAM` e o `connect` para ela.

3.5 O QUE O DNS FAZ PARA OBTER O QUE PRECISA

Agora que estudamos como as coisas são feitas no computador local, vamos seguir para a próxima fase e ver o que acontece para que o endereço IP desse domínio seja retornado. Todo o processo que passamos até agora foi para entrar em contato com o `nameserver`, também conhecido como *servidor de DNS*, configurado no computador que estamos usando.

A configuração de onde está o `nameserver` depende do sistema operacional. No GNU/Linux, ela é feita através do arquivo `/etc/resolv.conf`, por exemplo. O servidor de DNS configurado nesse arquivo será responsável por encontrar os endereços IP do servidor que contém o site do `desconstruindoaweb.com.br`. Roteadores residenciais geralmente possuem a funcionalidade de agir como um servidor de DNS local. Se esse for o caso, será possível ver algo como `192.168.1.1` na sua configuração de DNS.

Após fazer a requisição para o nosso servidor de DNS configurado, utilizando os métodos que estudamos até agora, o processo começa.

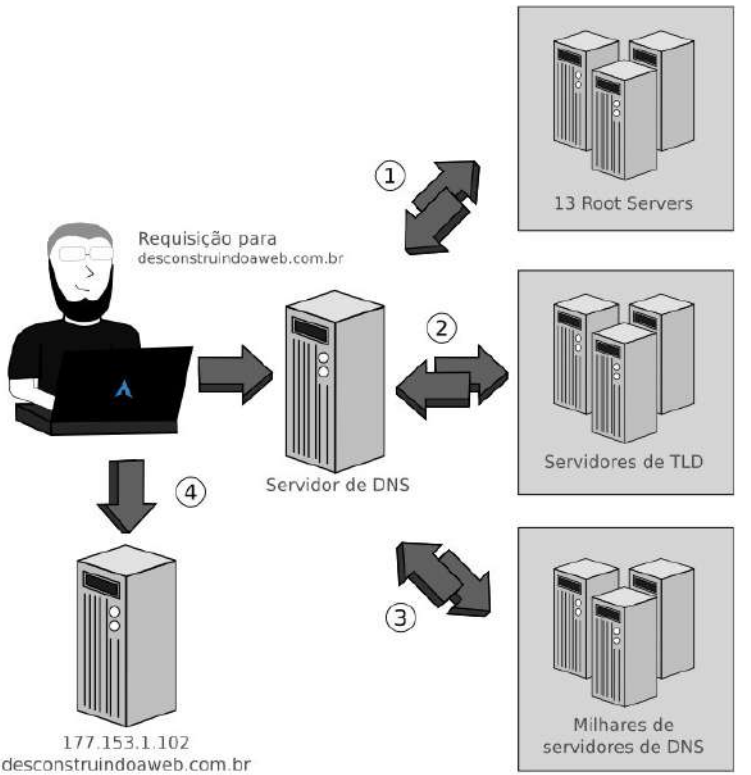


Figura 3.4: Resolução de domínios. Baseado na imagem disponibilizada em: Verisign Domain Name Industry Brief, June 2007

Para fins de estudo, vamos assumir que o servidor de DNS não possui nenhuma informação guardada e precisará fazer o processo completo. Esses servidores quase sempre têm essas informações guardadas em *cache*, portanto, não precisam executar todos os passos que vamos passar.

O nosso servidor de DNS não sabe nada sobre o nosso domínio, `desconstruindoaweb.com.br`, logo ele precisa ir por partes. A primeira coisa a se fazer vai ser adicionar um `.` (ponto) no final do domínio, para torná-lo um *Fully Qualified Domain Name*, também conhecido como *FQDN*.

Todos os domínios absolutos e não ambíguos devem terminar com um ponto no final. Essa informação parece estranha, afinal não adicionamos um `.` no final das URLs, e mesmo assim acessamos os sites sem problema. Essa é mais uma das mágicas que o navegador faz atualmente, adicionar o `.` se necessário.

Com isso, temos o domínio `desconstruindoaweb.com.br.` que será quebrado da seguinte maneira:

1. `.`
2. `br.`
3. `desconstruindoaweb.com.br.`

A resolução de domínio, como todo protocolo, é um processo longo e com vários detalhes, mas vamos focar no que é necessário para entender o fluxo básico. Vamos analisar cada passo do fluxo de resolução do domínio, baseando-se na *Figura 3.4* e na pequena lista que vimos há pouco:

1. O `.` (ponto) que foi adicionado na URL vai representar a query do servidor para algum root server. Nosso servidor de DNS não tem nenhuma informação adicional, apenas o endereço de um root server. Portanto, o servidor de DNS vai perguntar a ele sobre a entrada A do domínio `desconstruindoaweb.com.br.` para tentar descobrir seu IP diretamente.

   Os root servers não possuem essa informação, mas eles sabem quem pode tê-la, pois é sua função saber quais são os servidores de *TLD*, ou *Top-Level Domain*. Os servidores de *TLD* são responsáveis por domínios do tipo `br`, `com.br`, `com`, `io` e outros. O root server vai quebrar a URL, assim como fizemos há pouco, para descobrir qual servidor de TLD é o responsável por esse tipo de domínio. Para o `desconstruindoaweb.com.br`, ele vai enviar para o servidor

que responde aos domínios `br`.

2. Com a informação dos servidores de *TLD*, o servidor de DNS pode ir até eles e fazer a mesma pergunta. Ao fazê-la, ele vai receber a mesma resposta: *"Não tenho essa informação, mas sei quem pode ter"*. Os servidores de *TLD* possuem a lista de outros servidores responsáveis pelos domínios. Baseado na consulta, ele vai encaminhar uma lista de servidores que melhor se enquadram para respondê-la.

3. O nosso servidor vai então para essa lista procurar o servidor autoritativo para o domínio `desconstruindoaweb.com.br`. O servidor autoritativo é o servidor de DNS que possui autoridade para responder qual é o IP registrado para aquele domínio. O servidor autoritativo para o `desconstruindoaweb.com.br` está nessa lista e responde com o IP para o servidor. Na data de escrita deste livro, o IP retornado é o `177.153.1.102`.

4. Com essa informação, o nosso servidor de DNS responde para o navegador, que agora já pode continuar a requisição para o servidor certo.

Em cada um dos passos, o servidor de DNS vai fazer o cache do valor recebido para que futuras consultas aconteçam mais rapidamente.

É possível ver esse estudo na prática utilizando um software chamado `dnstracer`. Ele não costuma vir instalado por padrão nos sistemas operacionais, então, consulte a documentação do seu sistema para saber como instalar. Caso esteja utilizando uma distribuição GNU/Linux como o Ubuntu, um `apt-get install dnstracer` deve resolver.

Podemos ver um passo a passo parecido com o que fizemos,

bastando executar o `dnstracer` com o argumento `-4` para usar IPv4, e `-s .` para usar os *root servers* como requisição inicial.

```
$ dnstracer -s . -4 -o desconstruindoaweb.com.br

Tracing to desconstruindoaweb.com.br[a] via A.ROOT-SERVERS.NET
A.ROOT-SERVERS.NET [.] (198.41.0.4)
 |\_ a.dns.br [br] (2001:12ff::10) Not queried
 |\_ a.dns.br [br] (200.160.0.10)
 |   |\_ c.sec.dns.br (200.189.40.11) Got authoritative answer
 |    \_ a.sec.dns.br (200.160.0.11) Got authoritative answer
      \_ a.sec.dns.br (2001:12ff::11) Not queried
 |\_ b.dns.br [br] (200.189.41.10)
 |   |\_ c.sec.dns.br (200.189.40.11) (cached)
 |    \_ a.sec.dns.br (200.160.0.11) (cached)
 |    \_ a.sec.dns.br (2001:12ff::11) Not queried

(...)

a.sec.dns.br (200.160.0.11)
desconstruindoaweb.com.br -> 177.153.1.102

c.sec.dns.br (200.189.40.11)
desconstruindoaweb.com.br -> 177.153.1.102
```

O resultado do comando foi alterado para melhorar a legibilidade, além da remoção de algumas linhas, que foram substituídas por `(...)`. Fica como exercício identificar cada um dos passos que estudamos.

Com isso, terminamos a jornada de como conseguir o IP para se comunicar com o servidor do `desconstruindoaweb.com.br`. No próximo capítulo, vamos estudar como o protocolo HTTP usa o protocolo TCP para chegar ao servidor de destino.

3.6 RESUMO

Uma requisição web passa pelas várias camadas do "modelo Ozzy", que é um modelo simplificado que nomeamos e usamos para ilustrar as camadas. Nesse modelo, existem 5 camadas: de aplicação, de transporte, de rede, de enlace e física. O protocolo DNS fica na

camada de aplicação e é gerenciado pelas aplicações do usuário, ou seja, o Chromium ou a `glibc`, e não pelo kernel.

Nós vamos procurar por entradas *A* do DNS, pois elas possuem o endereço IPv4 que esperamos para conectar no site referente ao domínio que queremos acessar, no caso `desconstruindoaweb.com.br`. As implementações de DNS usam pacotes UDP por padrão, e no nosso caso estamos usando a implementação da `glibc` como base.

Para requisitar o DNS, a `glibc` vai criar um socket UDP usando a syscall chamada `socket`. Após criar o socket, ela usa a syscall `connect` para atribuir o endereço. Com isso, os pacotes UDP já podem ser enviados. O protocolo UDP **não** é orientado à conexão, portanto, ao enviá-lo, não haverá certeza de entrega nem ordem de chegada dos pacotes. Estamos chamando somente de UDP, mas aqui estamos usando o *UDP/IP*, que é a junção dos dois protocolos. Essa junção proporciona o envio e entrega dos pacotes no destinatário, usando a camada de transporte e rede respectivamente.

Com isso, é feita a comunicação do computador local com o servidor de DNS configurado e o processo de busca do IP do domínio começa. Primeiro, o servidor de DNS pergunta aos root servers qual é o IP do domínio do `desconstruindoaweb.com.br`. Eles vão passar a referência dos servidores de *Top-Level Domain*, que por sua vez enviarão uma lista de servidores que podem saber sobre o domínio.

Um desses servidores será o servidor autoritativo do nosso domínio e vai enviar a informação sobre o IP que foi registrado para uso. Ao receber essa informação, o nosso servidor de DNS vai encaminhá-la para nós como resposta à nossa requisição.

3.7 REFERÊNCIAS

1. Definição do modelo OSI na ISO — http://www.iso.org/iso/catalogue_detail.htm?csnumber=20269

2. TANENBAUM, A. S.; WETHERALL, D. *Computer Networks*. Singapore: Pearson Education, 2011.

3. Código de DNS do Chromium — https://code.google.com/p/chromium/codesearch#chromium/src/net/dns/dns_protocol

4. Código da função `getaddrinfo` — https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/posix/getaddrinfo.c;hb=HEAD#l2321

5. GitHub pages — https://pages.github.com/

6. *RFC1035* e a definição do protocolo DNS — https://tools.ietf.org/html/rfc1035

7. *RFC768* e a definição do protocolo UDP — https://tools.ietf.org/html/rfc768

8. Métricas do uso de IPv4 e IPv6 — http://www.worldipv6launch.org/measurements/

9. Fazendo o *connect* na glibc — https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/posix/getaddrinfo.c;hb=HEAD#l2526

Capítulo 4

TRANSFERINDO HYPERTEXTO

Todo o caminho da resolução de nomes já foi percorrido, conseguimos resolver o domínio do site que queríamos e conseguir seu endereço IP. Agora vamos estudar o que o protocolo HTTP precisa fazer para requisitar a informação do site do `desconstruindoaweb.com.br` e trazê-la até o nosso computador.

4.1 O BÁSICO DO HTTP

O protocolo HTTP também fica na camada de aplicação, como vimos no nosso modelo "Ozzy" no capítulo passado. Sabemos que o HTTP não vai ser administrado pelo kernel, portanto, o Chromium que vai cuidar da implementação do protocolo[1].

As versões do protocolo mais utilizadas atualmente são as da série 1.x, sendo que a mais nova dessa linha é a 1.1. Essa versão do protocolo HTTP é definida pela *RFC2616*[2], e seus vários anexos. O protocolo é composto apenas de texto, e conseguimos exemplificá-lo de uma forma sucinta utilizando o comando `telnet`. O que esse comando faz é executar uma conexão direta com um determinado endereço e porta. Um exemplo seria:

```
$ telnet desconstruindoaweb.com.br 80
Trying 177.153.1.102...
Connected to desconstruindoaweb.com.br.
Escape character is '^]'.
```

Nesse caso, abrimos uma conexão com o `desconstruindoaweb.com.br` na porta 80, que é a padrão do protocolo HTTP. Ele resolveu o domínio, seguindo um processo parecido com o que estudamos nos capítulos anteriores, e conseguiu o endereço IP `177.153.1.102`. Com o IP em mãos, ele iniciou uma conexão e nos deixou com acesso para digitar o que quisermos. O que for digitado agora vai ser enviado para o servidor de destino, é como se o `telnet` entregasse um terminal que, em vez de estar rodando `bash` — como é comum para a maioria dos sistemas *Unix-like* —, está rodando HTTP. Esse exemplo não é real, mas serve como uma metáfora.

Aqui, vamos usar o protocolo HTTP para requisitar a página principal do `desconstruindoaweb.com.br`:

```
GET / HTTP/1.1
Host: desconstruindoaweb.com.br
```

Como o protocolo é baseado em texto, podemos utilizá-lo diretamente via `telnet`. É claro que é uma versão bem simplificada, mas é funcional e vai nos retornar:

```
HTTP/1.1 200 OK
(...)
```

Onde está o `(...)` virão várias informações sobre os cabeçalhos utilizados pelo protocolo e, logo em seguida, o conteúdo do *HTML* da página web que pedimos.

Vamos analisar com calma cada um dos passos que fizemos anteriormente. Na primeira linha, nós usamos o texto `GET / HTTP/1.1`. Esta diz ao servidor web que queremos fazer um `GET` no caminho `/`, ou a raiz da aplicação, utilizando o protocolo `HTTP` com a versão `1.1`, que é a versão mais atual da série 1.x do protocolo.

Para entender melhor o que faz o `GET` do comando, precisamos

conhecer melhor os tipos de requisição do HTTP. Ele possui vários métodos, ou verbos, e cada um tem sua função. Os mais comuns são:

- `GET` — Deve ser utilizado apenas para receber informações, como foi nosso caso.
- `POST` — Deve ser usado quando for necessário enviar informações novas para serem adicionadas ao recurso da URL. **Exemplo:** `POST http://biblioteca.com/livros` criaria um novo livro com os dados enviados.
- `PUT` — Deve ser utilizado para atualizar recursos que estão em uma determinada URL. **Exemplo:** `PUT http://biblioteca.com/livros/2` altera os dados do livro com *id* 2 para o que foi enviado.
- `DELETE` — Deve ser usado para pedir ao servidor que exclua o recurso da URL em questão. **Exemplo:** `DELETE http://biblioteca.com/livros/2` exclui o livro referente ao *id* 2.

Para vermos isso na prática, podemos acessar `http://desconstruindoaweb.com.br` no Chromium e olhar as informações do *Developer Tools*. Lá vai estar disponível o método do HTTP que foi usado para fazer a requisição. Nesse caso, o navegador vai fazer um `GET` para a raiz do site para conseguir seu conteúdo.

Figura 4.1: O método GET no developer tools do Chromium

Conhecer esses métodos já é o suficiente para entender boa parte de como uma requisição funciona, e também será o suficiente para os nossos estudos. Mas saiba que eles não são os únicos! Na seção 9 da *RFC2616*[3] tem uma descrição de todos os métodos que podem ser usados e quais são suas finalidades.

Na segunda linha, utilizamos o parâmetro `Host: desconstruindoaweb.com.br` . Essa informação é um dos cabeçalhos[4] do HTTP, que vimos levemente no primeiro capítulo.

Os cabeçalhos são metadados que podem ser utilizados para enviar informações relevantes para o servidor e para quem está no meio caminho da conexão. As informações presentes nos cabeçalhos podem ser usadas no meio do caminho por servidores de cache, servidores web e outros softwares.

Nessa requisição, estamos utilizando o cabeçalho `Host` com o valor `desconstruindoaweb.com.br` . Ele é necessário pois o servidor de destino pode possuir várias aplicações sendo executadas no mesmo IP. O servidor usa o valor desse cabeçalho para saber para qual dessas aplicações ele deve requisitar o arquivo que foi solicitado, que no nosso caso será o que está disponível na raiz da aplicação, ou `/` .

A próxima linha é a resposta do HTTP, e parte dela ainda é interessante para o nosso estudo. O texto `HTTP/1.1 200 OK` diz

que o servidor retornou o *status* 200, que significa que a requisição foi concluída com sucesso. No primeiro capítulo, comentamos:

```
Os status do HTTP são formas mais simplificadas de dar dicas sobre
o que aconteceu na requisição para quem está recebendo a resposta.
```

Existem várias classes de status, sendo que cada uma possui uma classe de números que têm significados diferentes. Vamos listar essas classes e os status mais comuns em uma requisição web:

- **1xx**: Informativo, a requisição foi recebida e o processo continua.
- **2xx**: Sucesso! Ocorreu tudo certo com a requisição.
  - **200**: **OK**, usado quando a requisição foi processada e entregue com sucesso.
  - **201**: **Created**, geralmente é usado quando a aplicação cria um objeto e quer avisar o cliente que a criação foi executada com sucesso.
  - **202**: **Accepted**, geralmente utilizado quando algo assíncrono vai ser executado, portanto o servidor retorna que a requisição foi aceita e depois avisa quando tudo estiver concluído.
- **3xx**: Redirecionamento, alguma ação precisará ser tomada para completar a requisição.
  - **301**: **Moved Permanently**, usado quando uma aplicação mudou de URL e não pretende mais voltar para onde estava. Uma das utilizações desse status é quando uma aplicação muda de nome e o endereço antigo é mantido para redirecionar para o novo.
  - **304**: **Not Modified**, como vimos no *capítulo 1*, esse status é utilizado para cache. Quando o arquivo não foi modificado, o servidor responde com o status 304.
- **4xx**: Erro no cliente, possivelmente a requisição foi mal

feita.
    - **403**: **Forbidden**, é usado para quando o usuário que está fazendo a requisição não tem permissão para acessar esse recurso.
    - **404**: **Not Found**, é usado quando o recurso que está sendo requisitado não foi encontrado.
- **5xx**: Erro no servidor, a requisição parece válida, mas o servidor não consegue processar.
    - **500**: **Internal Server Error**, é usado quando o servidor não conseguiu processar a requisição por causa de algum erro inesperado.
    - **502**: **Bad Gateway**, é usado pelo servidor web que está sendo utilizado como proxy, ou seja, que está servindo as requisições para um ou mais servidores abaixo dele. Caso ele não receba uma resposta válida do servidor que está abaixo, ele retorna o status 502.

Isso é o básico que deve ser entendido sobre os *status* para o nosso estudo sobre o que acontece uma requisição web. Todos os detalhes sobre os status apresentados e os vários outros disponíveis para utilização estão descritos na *RFC7231*[5], que pode ser usada como referência, caso necessário.

Com isso, conseguimos fazer a nossa primeira requisição HTTP manualmente. Mas vamos entender **como** esse processo todo funciona por baixo dos panos.

4.2 O HTTP E O TCP

O protocolo HTTP fica na camada de aplicação e é suportado por protocolos que ficam abaixo dele, assim como vimos no capítulo anterior com o DNS. Para relembrar, o DNS é suportado

por UDP/IP por padrão, logo, não tem controle de conexão. É como enviar carta comum pelo correio, você nunca sabe quando chega ou se chega. Já o protocolo HTTP é diferente, pois usa o protocolo TCP/IP para o envio das informações. Usando esses protocolos, é possível manter uma conexão e garantir a entrega dos dados.

O TCP é um protocolo bem diferente do UDP. Ele foi pensado para quem precisa ter certeza de que o dado enviado será recebido na mesma ordem pela aplicação de destino. Tanto o protocolo *FTP*, que é usado para envio de arquivos, quanto o protocolo *SSH*, utilizado para conseguir um shell (uma instância de terminal) remoto, usam TCP para ter essa garantia. E é claro que junto com eles está o nosso protocolo de estudo desse capítulo, o HTTP.

O fluxo de criação de uma nova conexão será bem parecido com o que estudamos no capítulo anterior, ou seja, haverá uma chamada para a syscall `socket` com alguns parâmetros. Entre esses parâmetros estará o `SOCK_STREAM` para dizer que será utilizado uma conexão TCP para o transporte dos dados. Podemos ver isso no final da saída do comando `strace` que executamos também no capítulo anterior:

```
[pid  8739] socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 134
[pid  8739] connect(134, {sa_family=AF_INET, sin_port=htons(80),
sin_addr=inet_addr("177.153.1.104")}, 16) = -1 EINPROGRESS (Operat
ion now in progress)
```

Vamos por partes! Começando pela primeira linha:

```
[pid  8739] socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 134
```

Aqui temos a chamada para o socket com `SOCK_STREAM` e `IPPROTO_TCP`, o que retorna o número de *file descriptor* 134 para um *socket TCP/IP* como pedimos.

Já na linha seguinte:

```
[pid  8739] connect(134, {sa_family=AF_INET, sin_port=htons(80),
```

```
sin_addr=inet_addr("177.153.1.104")}, 16) = -1 EINPROGRESS (Operat
ion now in progress)
```

Aqui temos uma chamada para a syscall `connect`. Nela a syscall vai criar uma conexão entre os dois pontos usando o socket 134 criado na linha anterior, e fazer o *three-way handshake*. Nesse caso, fica bem diferente do que vimos com o UDP anteriormente, que apenas atribui o endereço do destinatário por não ter controle de conexão.

4.3 O THREE-WAY HANDSHAKE DO TCP

Ao iniciar a conexão, o TCP vai começar o seu processo de negociação entre o computador local, também conhecido como **cliente**, e o computador remoto, também conhecido como **servidor**. Esse processo é chamado de *three-way handshake*, ou em uma tradução **bem literal**: aperto de mãos de três vias. Esse nome foi dado devido aos 3 passos necessários para que o processo esteja completo. A partir de agora, chamaremos esse processo apenas de *handshake* para facilitar.

Para nos ajudar a entender melhor como o *handshake* funciona, vamos usar uma ferramenta chamada Wireshark[6], que está disponível para os sistemas operacionais mais comuns. Essa ferramenta faz a captura de pacotes que estão trafegando na interface de rede, o que possibilita um estudo mais detalhado dos dados.

Ao executar o Wireshark, vamos configurá-lo para começar a ouvir a interface de rede principal. Nesse exemplo, vamos usar *Wi-Fi* em um sistema GNU/Linux, escutando na interface chamada *wlp3s0*.

Figura 4.2: Interfaces no Wireshark

Ao clicar em *Start*, ele vai começar a captura de pacotes. Com isso, será possível ver na tela principal várias linhas mostrando o endereço de origem, endereço de destino, protocolo, tamanho e outras informações.

Agora vamos abrir o Chromium, que é nosso navegador de estudo, e acessar a URL *http://desconstruindoaweb.com.br* para que o Wireshark possa capturar tudo o que acontecer na interface de rede que configuramos. Quando o site terminar de carregar, podemos parar a captura de pacotes e começar a nossa análise.

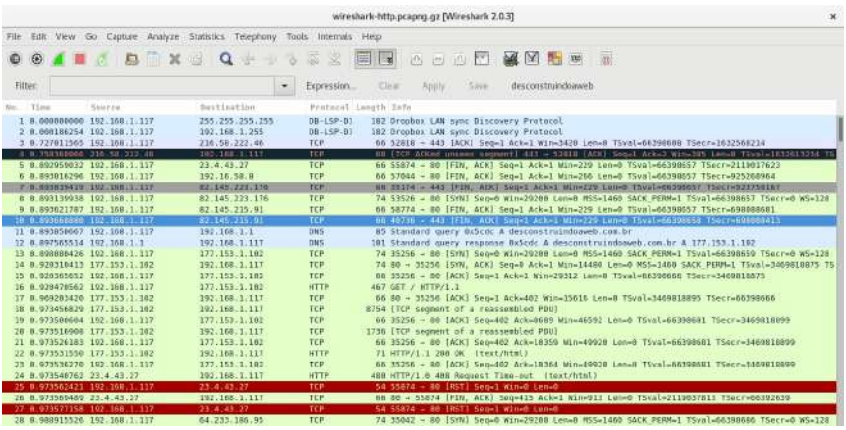


Figura 4.3: Pacotes coletados pelo Wireshark

Uma conexão TCP possui muitos estados que definem cada momento de uma tentativa de conexão. Todos esses estados são

definidos com detalhes na RFC793[7], inclusive com desenho representando cada um dos fluxos.

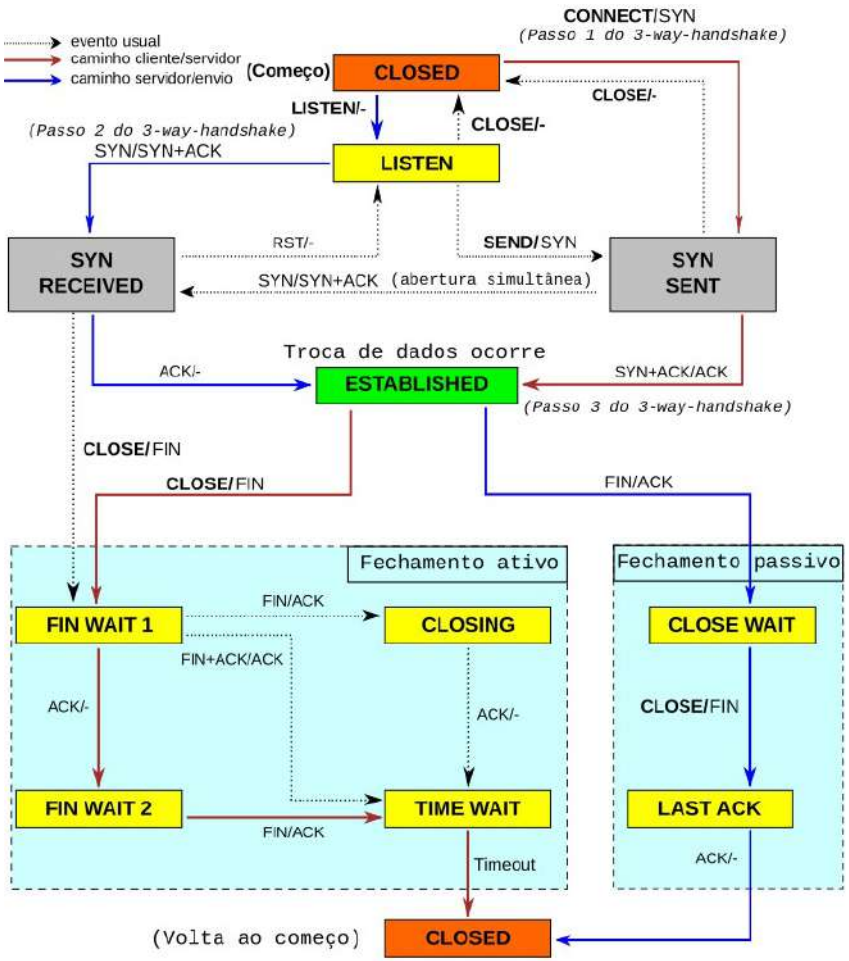


Figura 4.4: Estados do TCP. Traduzido do trabalho do Scil100, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=30810617 | RFC793

Seguindo o conceito que usamos neste livro, vamos seguir da maneira mais simples possível para entender o fluxo. Vamos passar pelos 3 passos fundamentais para que o *handshake* esteja completo, sem precisar adentrar nos detalhes de cada um dos fluxos que

podem aparecer em caso de problemas.

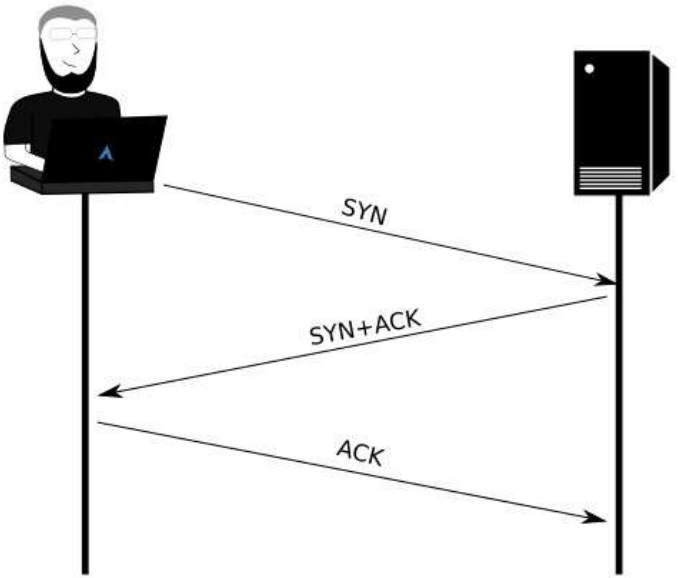


Figura 4.5: O three-way handshake

A princípio a conexão está fechada, ou no estado *CLOSED*, e os dois computadores ainda não se conhecem. Eles precisam de uma maneira para se identificar. Logo, nós, como computador de origem, vamos enviar uma mensagem pedindo para iniciar uma conexão. Nessa mensagem haverá vários atributos como: porta de origem e destino, número de sequência, endereço IP de destino e identificadores do tipo da requisição.

Logo no início da captura já podemos ver dois pacotes referentes à resolução de nomes do DNS. Um deles é responsável por fazer a busca no servidor de DNS e o por entregar a resposta ao cliente.

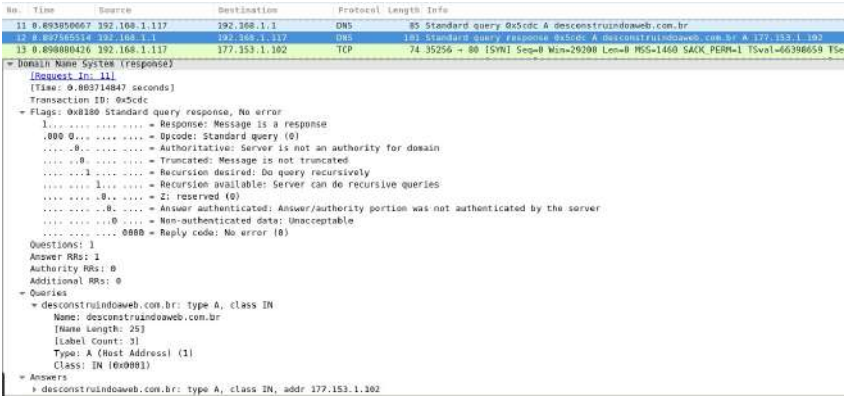


Figura 4.6: Pacotes referentes ao DNS

Em seguida, estão os pacotes referentes ao three-way handshake. Vamos analisá-los focando principalmente no número de sequência e no tipo de requisição.

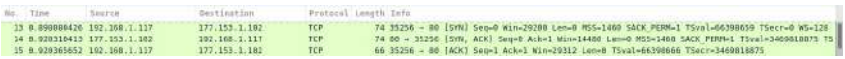


Figura 4.7: Pacotes referentes ao three-way handshake

O número de sequência, que vamos chamar de *SEQ*, é um número gerado que vai sendo incrementado a cada requisição. Já o tipo da requisição mostra para o servidor de destino qual a finalidade dessa requisição, que no nosso caso é abrir uma nova conexão. Assim, vamos utilizar *SYN* como tipo.

Agora que o pacote está pronto, basta enviar para o servidor, terminando o primeiro passo do *handshake*.

Figura 4.8: Pacote referente ao primeiro passo do handshake

O servidor web está em pleno funcionamento e no estado *LISTEN*, que diz que está aguardando conexões. Nesse momento, ele recebe nossa requisição identificada como *SYN*, ou seja, um pedido de requisição, que era exatamente o que ele estava aguardando! O **servidor** então cria um número de sequência para ele, incrementa o número *SEQ* que recebeu do **cliente**, e adiciona um novo tipo para a requisição, o *ACK*.

Com isso, o servidor vai enviar de volta uma requisição contendo os tipos *SYN* e *ACK* para quem requisitou a conexão. O *ACK* é um *acknowledge*, ou uma confirmação de que a requisição foi recebida e o *SYN* diz que o servidor está disposto a abrir a conexão. Assim que o **servidor** envia um pacote com essas informações para o **cliente**, o segundo passo do *handshake* está completo.

Figura 4.9: Pacote referente ao segundo passo do handshake

Após executar o primeiro passo do handshake, enviando o *SYN* para o servidor de destino, nós ficamos aguardando o retorno sobre o nosso pedido de conexão. Ao receber a requisição com um *ACK* referenciando o número identificador da nossa conexão, já incrementado em 1, sabemos que os dados foram recebidos. Essa requisição, além de confirmar o recebimento da nossa requisição, também possui um *SYN* dizendo que aceita nosso pedido de conexão! :)

Com essa informação, podemos considerar a conexão como estabelecida, pois todos as partes estão de acordo. Mas ainda há um detalhe: o servidor de destino ainda não sabe disso! Ele nos enviou a resposta dizendo que aceita a conexão, mas ainda não sabe se nós a recebemos ou se ainda pretendemos continuar o processo.

O terceiro passo do handshake serve para confirmar que recebemos as informações como o esperado, e já estamos prontos para iniciar a conexão. A nova requisição segue quase o mesmo processo da primeira, mas com algumas alterações. O *SEQ* é enviado de volta acrescido em 1, para identificar que os dados foram recebidos e são referentes àquela conexão. O tipo da requisição também é alterado, e dessa vez vai como apenas *ACK*, confirmando o recebimento dos dados do servidor de destino e, com isso,

completando o *handshake*.

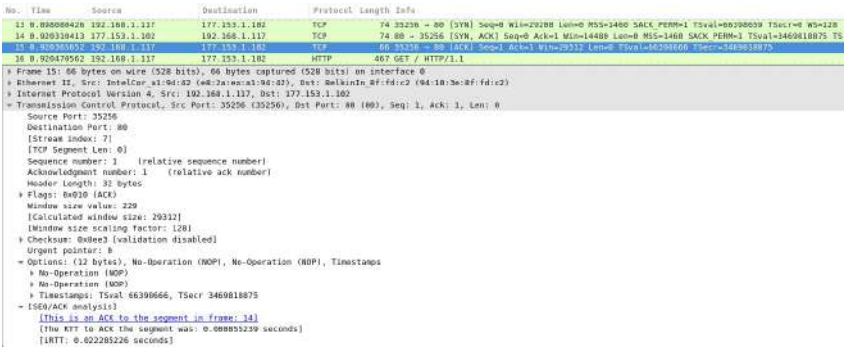


Figura 4.10: Pacote referente ao terceiro passo do handshake

Agora que o handshake está completo, ambos os servidores ficam no estado *ESTABLISHED*, ou conexão estabelecida. Nesse estado, os dados podem trafegar livremente em ambos os lados, seguindo o mesmo padrão de *SEQ* e *ACK*, enquanto a conexão estiver válida. Ela estará válida até que um dos lados decida fechá-la, quebrando assim o acordo feito no handshake.

4.4 A REQUISIÇÃO HTTP DO NAVEGADOR

Até agora nós fizemos uma requisição HTTP manualmente e inspecionamos a resposta para ver o que o servidor responderia para essa requisição. Agora estudaremos como é uma requisição feita por um navegador moderno, usando a captura de pacotes do Wireshark para facilitar a tarefa.

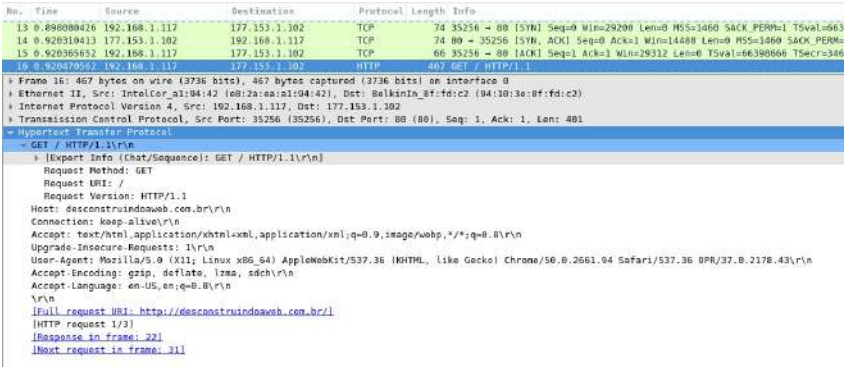


Figura 4.11: Detalhes do pacote HTTP

O pacote que está logo após os três pacotes do three-way handshake é o da nossa requisição HTTP feita via navegador. Ao olhá-lo mais de perto, é possível perceber que não há muita novidade além do que já vimos.

Vamos detalhar cada linha do pacote:

- `GET / HTTP1.1` — É a requisição do navegador, que por sinal é idêntica à que fizemos via `telnet` no início deste capítulo. :)
- `Host` — É o hostname que discutimos há pouco. Ele ajuda a identificar quando há mais de uma aplicação no mesmo IP.
- `Connection` — Esse cabeçalho do HTTP é usado para que o servidor mantenha a conexão aberta por mais tempo. Em vez de abrir e fechar a cada requisição, o servidor mantém a conexão aberta até que a opção `close` seja usada. Esse cabeçalho está em uso por compatibilidade, pois esse é o comportamento padrão do HTTP/1.1[8].
- `Accept` — São os tipos de conteúdo que o navegador aceita processar.

- `Upgrade-Insecure-Requests` — É um cabeçalho usado para dizer ao servidor que o navegador prefere conteúdo por uma conexão segura.
- `User-Agent` — É o que identifica o navegador.
- `Accept-Encoding` — Diz os tipos de codificação que o navegador pode receber. Esse cabeçalho é de suma importância para quem desenvolve para a internet, pois ele diz que o navegador consegue processar requisições com conteúdo comprimido com `gzip`, `deflate` ou `lzma`. É uma boa prática habilitar a compressão no servidor web para reduzir a quantidade de dados trafegados e a velocidade da entrega para o cliente.
- `Accept-Language` — Os idiomas que o navegador está usando e/ou aceita receber.

Apesar de estarmos olhando em uma camada mais baixa, no nível de pacotes e protocolos, conseguimos ver essas informações de cabeçalhos diretamente pelo navegador usando a ferramenta de *Developer Tools* do Chromium. Quando olhamos a requisição por lá, podemos ver que os cabeçalhos são os mesmos que acabamos de estudar via Wireshark.

Figura 4.12: Informações sobre a requisição no developer tools do Chromium

Apesar de ter alguns cabeçalhos que adicionam funcionalidades

à conexão, é possível ver que nossa requisição feita por `telnet` não deixou a desejar.

4.5 O HTTP/2

Tanto o HTTP 1.1 quanto o *HTTP/2* utilizam o protocolo TCP na camada de transporte. Portanto, o que vimos na seção anterior sobre o three-way handshake se aplica a ambos.

O **HTTP/2**, que também já foi conhecido como *HTTP/2.0*, é baseado no protocolo *SPDY* (para parecer com *speedy* quando falado). O SPDY foi criado pelo Google para melhorar a velocidade e compressão de dados em páginas de internet. Eles estavam em busca de criar um novo padrão e chegaram a submeter um *draft*[9] para possivelmente virar uma das nossas queridas RFCs.

O HTTP/2 começou a ser criado pelo grupo de trabalho *httpbis*[10], e depois de muitos meses teve a *RFC7540*[11] criada e aprovada.

As aplicações web, como a do `desconstruindoaweb.com.br` que estamos estudando, não precisam ser modificadas para utilizar a versão 2 do protocolo HTTP. As alterações no protocolo afetam as camadas mais baixas, que lidam com o envio e compressão de dados, portanto a troca de protocolos poderia ser feita apenas alterando o servidor web.

As principais diferenças entre a versão 1.1 e a versão 2 são:

- **Protocolo binário em vez de texto puro:** melhora a comunicação mas impossibilita o exemplo que fizemos com `telnet` na versão 1.1 do HTTP no começo deste capítulo, pois teríamos de digitar o protocolo binário no terminal.
- **Multiplexação:** remove o problema de ter apenas uma

resposta por conexão, que faz com que os clientes criem várias conexões para conseguir tudo o que precisam mais rápido. Com a multiplexação, o servidor pode enviar várias respostas em vários pedaços para o cliente, fazendo com que ele construa os pacotes novamente ao recebê-los por inteiro.

- **Compressão de cabeçalhos:** todos os cabeçalhos do HTTP que antes eram texto puro agora são comprimidos e reduzem a quantidade de transferência gasta. De acordo com o FAQ[12] da página oficial do HTTP/2, utilizar cabeçalhos comprimidos pode reduzir até 8 vezes o tempo de transferência do cabeçalho. Isso ocorre principalmente por causa do algoritmo *Slow Start*[13] do TCP, que faz um controle mais conservador de envio de pacotes no começo da conexão para evitar congestionamento na rede.
- **Permite ao servidor enviar dados antes que sejam pedidos:** se o servidor souber que uma página precisa de 10 arquivos para que esteja completamente carregada, ele pode prepará-los e enviá-los proativamente. Ao entender o HTML e descobrir que vai precisar de mais arquivos, o navegador vai primeiro olhar seu cache local e vai descobrir que os arquivos já estão lá, pois o navegador os enviou junto com o primeiro pedido.

De acordo com os dados disponíveis na pesquisa da w3techs[14] em abril de 2016, que utiliza como base os primeiros 10 milhões de sites do ranking de popularidade do site Alexa.com, o HTTP2 é usado em apenas **7.3%** deles. Baseando-se nessa informação, vamos seguir nossa jornada pelo utilizando o HTTP 1.1, que é o caminho mais conhecido e que vai agregar mais conhecimento no contexto atual. Não só vamos ver o HTTP/1.1, como vamos vê-lo de forma

segura, com *HTTPS*!

4.6 RESUMO

O protocolo HTTP tem duas versões que valem menção. A primeira é o HTTP/2 que promete ser o futuro das requisições web, trazendo várias inovações na forma de lidar com o envio dos dados. A segunda, e mais utilizada, é o HTTP/1.1, que é um protocolo totalmente texto e que pode ser reproduzido utilizando comandos de conexão, como o `telnet`.

O HTTP tem alguns tipos de métodos, e o que usamos para pedir uma página de internet é o `GET`. O básico de uma conexão HTTP/1.1 para requisitar uma página via `telnet` é usar `GET / HTTP/1.1`, que vai fazer uma chamada `GET` na raiz da aplicação, ou `/`, utilizando o protocolo `HTTP/1.1`.

O HTTP funciona em cima de TCP, portanto, precisa fazer o three-way handshake antes de qualquer conexão. O three-way handshake é a parte inicial de uma conexão no TCP, na qual ambos os lados se conhecem e estabelecem uma confiança temporária para a conexão.

O processo começa com o computador de origem, ou cliente, enviando um pedido de conexão (*SYN*) para o computador de destino, ou servidor. O computador de destino recebe e envia de volta uma confirmação de que recebeu (*ACK*), juntamente com um pedido de conexão (*SYN*). No terceiro e último passo, o computador de origem estabelece a conexão e envia uma confirmação (*ACK*) para o computador de destino, para que ele faça o mesmo.

Com a conexão fechada, os dois começam a trocar dados utilizando o mesmo processo de envio e confirmação.

4.7 REFERÊNCIAS

1. Implementação do protocolo HTTP no Chromium — https://code.google.com/p/chromium/codesearch#chromium/src/net/http/
2. *RFC2616* e a definição completa do HTTP — https://tools.ietf.org/html/rfc2616
3. *RFC2616* seção 9, métodos do HTTP — https://tools.ietf.org/html/rfc2616#section-9
4. *RFC2616*, na página 31, possui a definição dos cabeçalhos do HTTP — https://tools.ietf.org/html/rfc2616#page-31
5. *RFC7231* e os status codes do HTTP — https://tools.ietf.org/html/rfc7231#page-47
6. Wireshark — https://www.wireshark.org
7. *RFC793* e os estados do TCP — https://tools.ietf.org/html/rfc793#page-23
8. Conexões persistentes na RFC do HTTP/1.1 — https://tools.ietf.org/html/rfc7230#section-6.3
9. *Draft* do protocolo SPDY — https://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00
10. Grupo de trabalho *httpbis* na IETF — http://tools.ietf.org/wg/httpbis
11. *RFC7540*, a definição do HTTP/2 — https://tools.ietf.org/html/rfc7540
12. Página oficial do HTTP/2 e explicação sobre cabeçalhos comprimidos — https://http2.github.io/faq/#why-do-we-need-header-compression

13. *RFC5681* e o *Slow Start* do TCP — https://tools.ietf.org/html/rfc5681#page-4

14. Pesquisa sobre a utilização do HTTP/2 — http://w3techs.com/technologies/details/ce-http2/all/all

Capítulo 5

HTTPS E SUA SEGURANÇA

Agora que passamos pelo processo de como o HTTP/1.1 faz para enviar os dados de um lado para o outro, estudaremos o que acontece quando o `s` é introduzido no nome do protocolo. O HTTPS é de suma importância para a internet atual e muita coisa que acontece no meio do caminho é mágica para nós, ainda mais por ser um assunto denso e possuir algoritmos complicados no processo.

Vamos adentrar esse mundo, simplificando o máximo possível o processo para entendê-lo.

5.1 O HTTPS

HTTPS significa *HTTP over TLS*, que é literalmente a utilização do protocolo HTTP dentro de uma conexão TLS. O HTTPS é especificado pela *RFC2818*[1], que apesar de bem curta, mostra como uma conexão segura deve ser utilizada para a transferência dos dados de forma criptografada. Para que isso seja feito, *todos* os pacotes HTTP devem ser enviados usando essa conexão.

O navegador distingue uma conexão HTTPS quando vê o início da URL com `https://`, como em *https://desconstruindoaweb.com.br*. Após detectar o protocolo, ele vai seguir o processo de resolução de nomes que vimos nos primeiros capítulos do livro, e fazer o handshake do TCP que vimos

no capítulo anterior.

Vamos colocar a mão na massa para ver o que está realmente acontecendo quando fazemos uma chamada HTTPS. Para isso, usaremos novamente o Wireshark para fazer a análise dos pacotes de rede. Com ele, vamos poder rever os fluxos que vimos em capítulos passados e entender o HTTPS na prática.

A primeira coisa a fazer é selecionar a interface de rede principal, como fizemos no capítulo anterior — que para esse caso será a `wlp3s0` —, e clicar em `Start` para os pacotes começarem a ser coletados.

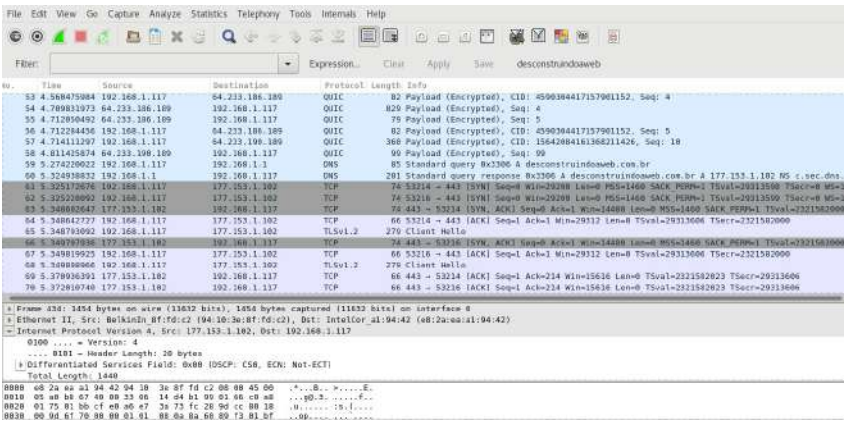


Figura 5.1: Pacotes chegando no Wireshark

Dessa vez, vamos acessar a URL do `desconstruindoaweb.com.br` utilizando **https**! No Chromium, acessaremos `https://desconstruindoaweb.com.br` , e assim que a página terminar de carregar, paramos a captura de pacotes no Wireshark.

Nesse pouco tempo que deixamos capturando os pacotes, o Wireshark capturou centenas deles (634 neste exemplo). Para facilitar os estudos, vamos fazer um pequeno filtro nos dados para

isolar apenas os pacotes da comunicação do servidor com o cliente, e vice-versa. Basta saber o IP do seu computador local e do servidor de destino, e fazer um filtro no campo *Filter* do Wireshark.

Utilizando `192.168.1.117` como IP de origem e `177.153.1.102` como destino, temos:

```
(ip.src == 192.168.1.117 and ip.dst == 177.153.1.102) or (ip.dst =
= 192.168.1.117 and ip.src == 177.153.1.102)
```

Com isso, removemos todo o tráfego que não vai ser útil para esse caso, e reduzimos os pacotes quase que pela metade. Olhando a sequência de pacotes, é possível identificar a chamada para a resolução de DNS, pedindo por um registro do tipo *A* para `desconstruindoaweb.com.br` como vimos nos primeiros capítulos do livro. Logo em seguida, estará o *three-way handshake* do TCP, com seu *SYN*, *SYN ACK*, *ACK* que estudamos no capítulo passado.

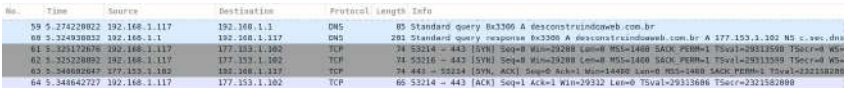


Figura 5.2: Resolução de DNS e handshake do TCP

A diferença que vemos nesse caso é a porta de destino. Por padrão, as conexões HTTPS seguem pela porta `443` em vez da porta `80`. No nosso exemplo, as conexões estão saindo de portas altas como a `53214` no nosso computador, e indo para a porta `443` no servidor do `desconstruindoaweb.com.br`.

Agora que os dois servidores já estabeleceram uma conexão utilizando o handshake do TCP, será estabelecida uma conexão segura com o servidor de destino, para só depois começar a enviar pacotes HTTP. Para fazer essa transferência segura, o HTTPS usa o protocolo TLS.

5.2 O QUE É O TLS

De acordo com a *RFC5246*[2] que define a versão 1.2 do protocolo **TLS**, ele foi criado como uma evolução do SSLv3. Os documentos de especificação do HTTPS recomendam o uso de HTTP sobre TLS em vez de SSL, que é uma versão mais antiga do protocolo.

> **MAIS SOBRE O SSL**
>
> Ainda hoje, não é comum ouvir as pessoas falando sobre o TLS, mas ouvimos com frequência que precisamos utilizar SSL para deixar a nossa conexão segura. Então, qual é a diferença?
>
> O **SSL** foi criado pela Netscape na década de 90, e até hoje ainda é utilizado para algumas poucas conexões. O **TLS** é a nova versão do SSL, desvinculando totalmente o nome da antiga Netscape do protocolo e trazendo melhorias.

A ideia principal do TLS é adicionar segurança de uma forma transparente a uma comunicação. Com isso, o protocolo que usar o TLS pode continuar o mesmo, apenas utilizando-o como um encapsulamento que leva a informação de um lado para o outro de forma segura. No caso do HTTPS, temos o HTTP sendo utilizado seguindo essa premissa — o mesmo protocolo que já estamos acostumados, mas com uma camada adicional de segurança.

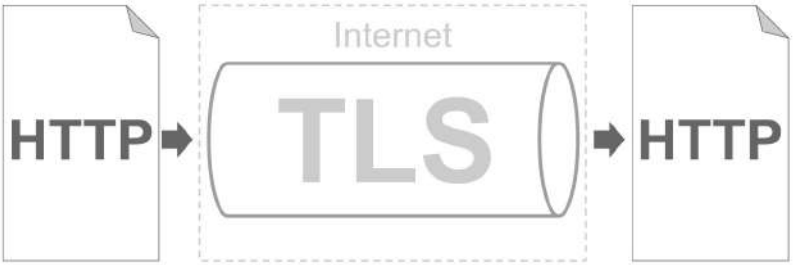


Figura 5.3: O TLS como túnel seguro de transporte

Internamente, o TLS possui duas camadas:

- *TLS Record Protocol*: é a camada mais baixa do protocolo, e é responsável por fazer o encapsulamento das camadas mais altas como foi comentado há pouco.
- *TLS Handshake Protocol*: é responsável por fazer com que o cliente e o servidor se autentiquem. Por meio dessa camada, cliente e servidor negociam um algoritmo de encriptação e chaves de criptografia antes que as aplicações transmitam ou recebam os primeiros dados.

O TLS, como outros protocolos que trabalham com segurança da informação, é bem complicado. Esses protocolos dependem de vários cálculos matemáticos para criação de chaves aleatórias e outras coisas que demandam um livro só de explicações.

Vamos continuar seguindo o que foi definido no começo do livro e estudar a forma mais simples de entender como o processo completo funciona. Assim, passaremos superficialmente pelos conceitos matemáticos dos algoritmos de criptografia envolvidos nas chaves do TLS e focaremos no processo como um todo.

Nosso foco será no TLS Handshake Protocol por ser parte

integrante do que é necessário conhecer quando se está lidando com aplicações para a internet.

5.3 O HANDSHAKE DO TLS

O TLS depende de alguma forma de transporte confiável, e no nosso caso, o HTTPS utiliza TCP da mesma forma como é feito com HTTP. Como vimos no Wireshark, o *handshake do TCP* já foi feito, e agora o cliente e o servidor começam a trocar informações para fechar uma conexão TLS.

Muita coisa precisa ser feita antes que o primeiro pacote de *Application data*, ou dados de aplicação, seja enviado. Dezenas de pacotes são trocados para que o handshake aconteça, então vamos olhá-los mais de perto.

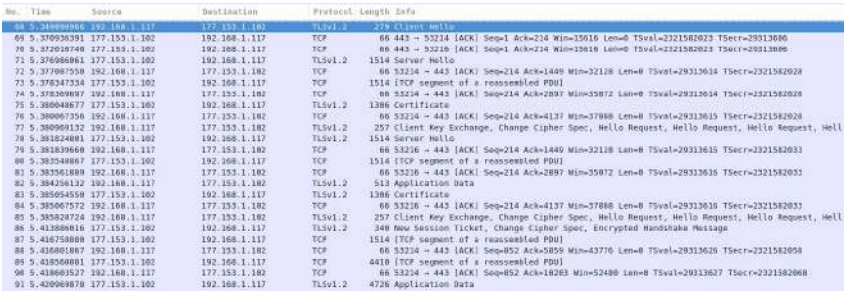


Figura 5.4: Troca de pacotes para o handshake do TLS

Client Hello

A primeira mensagem enviada pelo cliente se chama **Client Hello**. Nessa primeira requisição, o cliente envia várias informações interessantes, sendo algumas delas:

- `Protocol version` — É a versão do TLS que estamos utilizando. Para essa conexão, está sendo usada a versão 1.2 do TLS.

- `Cipher suites` — Uma lista com todos os *Ciphers* que nosso navegador local aceita utilizar na criptografia dos dados. Esse envio é importante para que o servidor saiba a capacidade de criptografia que o cliente tem para que ele possa escolher uma delas.
- `Random` — Nesse campo, há um conjunto de bytes aleatórios e um *unix timestamp*, que é um número inteiro que representa uma data em segundos, contando desde 1 de janeiro de 1970.
- `Extension` — Existem também várias extensões que são definidas pela *RFC3546*[3]. Um exemplo é a extensão chamada `server\_name` que define o *domínio* da aplicação que estamos acessando. Como o TLS acontece bem antes da conexão HTTP começar, essa extensão faz o trabalho que o cabeçalho `Host` desempenha em uma conexão HTTP comum. Portanto, o servidor de destino vai utilizar essa informação para encontrar a aplicação certa para direcionar a conexão caso haja mais de uma no mesmo IP.

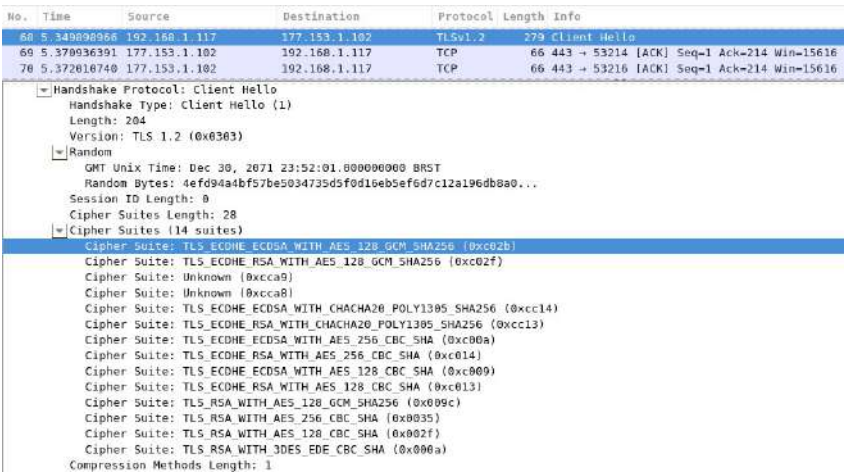


Figura 5.5: Lista de ciphers no Client Hello

Após o envio de cada um dos pacotes do *TLSv1.2*, é possível ver no Wireshark o recebimento dos pacotes de ACK do TCP, que está fazendo a entrega deles, garantindo uma conexão confiável.

Server Hello

Quando o servidor recebe o `Client Hello`, ele precisa devolver um **Server Hello**, para confirmar as informações recebidas e enviar o que for necessário para que a conexão continue a ser estabelecida. Vamos olhar mais de perto algumas informações dessa mensagem:

- `Protocol version` — Diz o protocolo aceito para continuar a conexão, nesse caso o servidor aceitou em usar o `TLSv1.2`.
- `Cipher Suite` — Entre a lista de ciphers que o navegador enviou, o servidor decidiu que vai utilizar `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)` a criptografia dos dados. Esse monte de letrinhas tem muito a nos dizer e voltaremos para ele em breve.
- `Random` — Assim como no `Client Hello`, há um conjunto de bytes aleatórios e um `unix timestamp`.

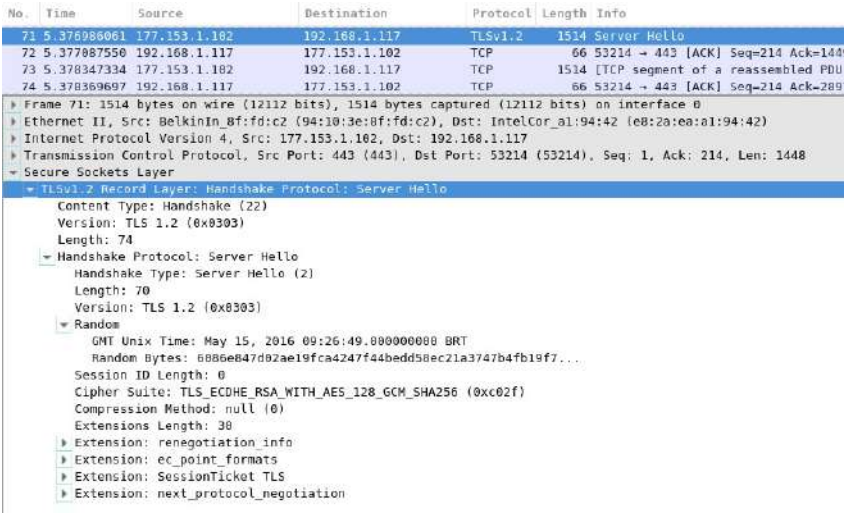


Figura 5.6: Desmembrando as informações do Server Hello

Ainda há alguns passos para terminar o `Server Hello`. O primeiro deles é enviar um certificado para ser avaliado pelo cliente. Essa funcionalidade do TLS deve ser usada sempre que o método de troca de chaves utilizar um certificado para autenticação, e o nosso caso se enquadra nisso.

O servidor então envia o certificado para avaliação do cliente. Apesar de conseguirmos ver pelo Wireshark, podemos ter mais detalhes do certificado pelo próprio Chromium clicando no cadeado verde ao lado da barra de endereços. Ambos mostram a precedência do certificado, até quando vale, quando foi criado e mais algumas informações.

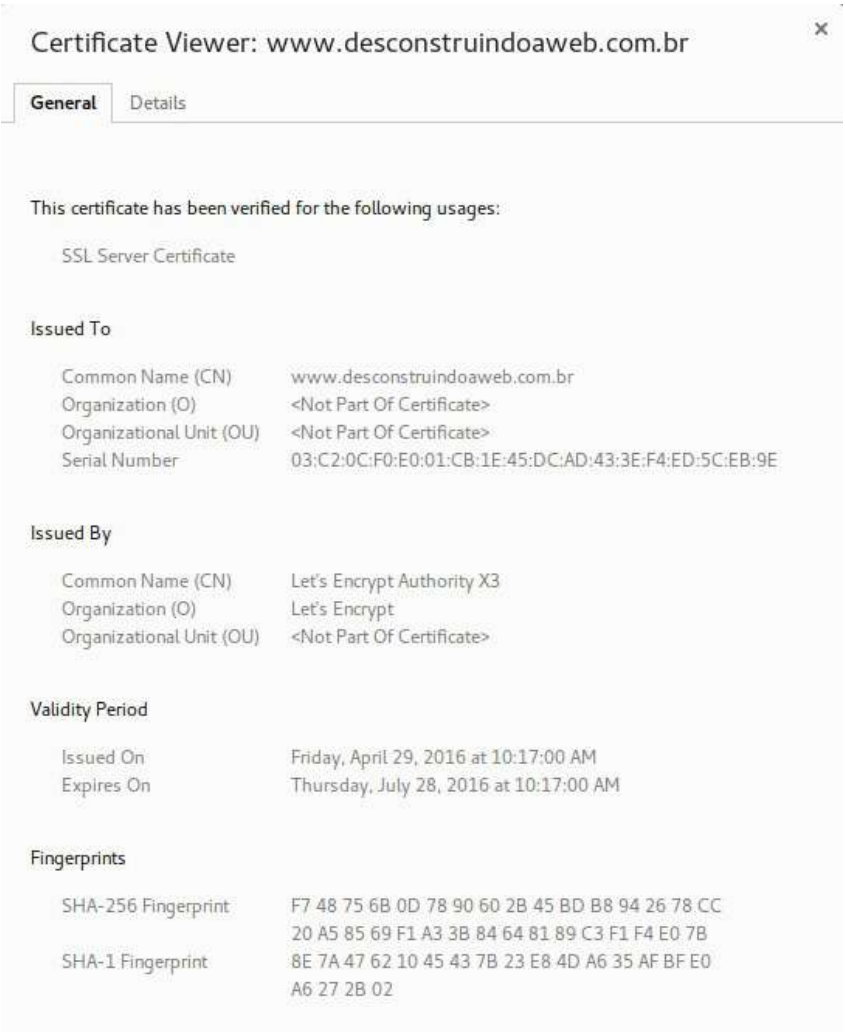


Figura 5.7: Detalhes do certificado no Chromium

O segundo passo é enviar uma mensagem de *Server Key Exchange* com informações sobre o nosso algoritmo de criptografia. Essa mensagem possui informações como: chave pública, tipo de curva e assinatura. Estas são referentes ao algoritmo de *Diffie-Hellman* que vamos conhecer melhor quando entendermos o cipher, portanto, para ficar mais didático, veremos essa parte com

mais detalhes junto com o *Client Key Exchange*.

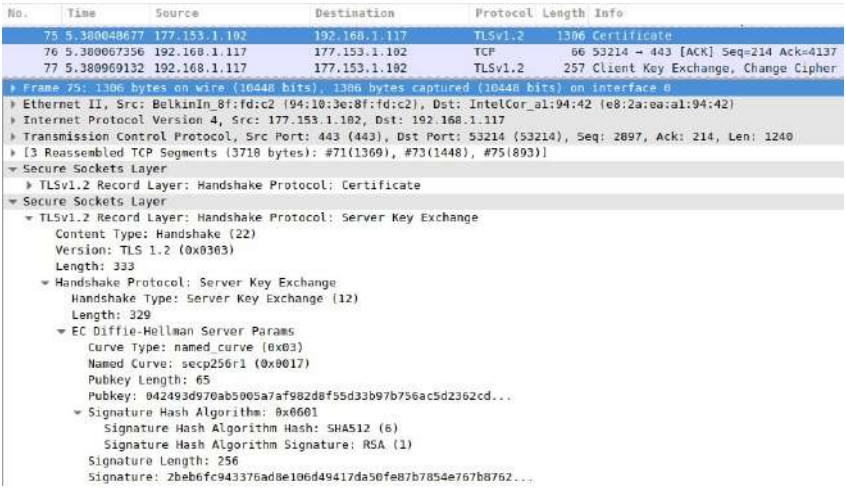


Figura 5.8: Server Key Exchange enviado pelo servidor

Logo em seguida, o servidor envia um `Server Hello Done`, para dizer que essa fase está encerrada e que está aguardando uma resposta do cliente.

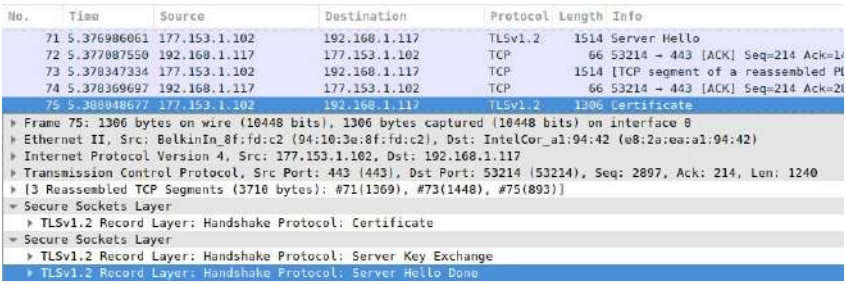


Figura 5.9: Server Hello Done depois do certificado

O certificado e a chave pública

Antes de passar para a próxima seção, precisamos de um pouco de teoria para entender **como funciona** esse negócio de certificado. Vamos primeiro estudar por que ele é tão importante, e depois

seguiremos o fluxo do handshake para ver o que o navegador faz para verificá-lo.

Um certificado digital é um *container* que contém informações que ajudam a garantir a fonte de uma chave pública e sua versão 3 é especificada pela *RFC5280*[4].

Uma pergunta comum que aparece ao ler isso tudo é:

```
"Mas peraí... que negócio é esse de chave pública no certificado?"
```

A chave pública está presente porque o processo do TLS utiliza, além de outras coisas, uma troca de chaves para a criptografia assimétrica das informações. A criptografia assimétrica nos entrega uma forma de encriptação onde há um par de chaves, sendo que uma delas é a *chave privada* e a outra é a *chave pública*.

A **chave privada** deve ser guardada da melhor forma possível e nunca ser revelada para ninguém. É ela que vai garantir que somente você poderá revelar o conteúdo de um texto que for criptografado com a sua chave pública.

Já a **chave pública** pode ser distribuída para todos que você queira que se comunique de forma segura com você. Com isso, sempre que alguém precisar falar diretamente com você, ele poderá criptografar os dados usando a sua chave pública e enviar livremente por um canal não seguro, pois terá certeza de que somente você vai ver o conteúdo, pois só você tem a chave privada para descriptografá-lo.

Diferença entre criptografia simétrica e assimétrica

Os nomes são parecidos, mas a ideia entre as duas é bem diferente. Na criptografia simétrica, há uma chave compartilhada entre os dois lados, e ela é usada para criptografar e descriptografar os dados. Um bom exemplo de criptografia simétrica são as redes Wi-Fi. Para que alguém acesse uma rede Wi-Fi segura, é necessário saber uma senha que é compartilhada entre o roteador Wi-Fi e todos que estão utilizando a rede.

Já na criptografia assimétrica, não existe uma chave que ambos sabem antes da conexão. Ela usa um par de chaves, sendo que o conteúdo criptografado por uma só pode ser revelado pela outra. Um exemplo de criptografia assimétrica que está presente no nosso dia a dia é o próprio HTTPS que estamos estudando neste capítulo. Para acessar um site, nós não temos uma senha compartilhada e, mesmo assim, só o destinatário e o remetente conseguem decifrar o dado em tempo hábil.

O que há por trás desse processo é um daqueles algoritmos matemáticos muito interessantes e complicados que comentamos no início do capítulo. Os detalhes de implementação dele vão bem além do nosso estudo, portanto, não vamos a fundo neles. Para quem quiser entender como é a *matemática* por trás de tudo isso, o vídeo explicativo *Public key cryptography — Diffie-Hellman Key Exchange*[5] é muito recomendado.

Garantindo a confiança do certificado

O principal motivo do certificado é transportar a chave pública e

prover meios para que o outro lado possa utilizá-la de uma segura. Para que isso aconteça, quem receber a chave precisa ter certeza de que ela é realmente de quem ela diz ser. Afinal, qualquer um pode mandar uma chave dizendo que é o *google.com*, não é?

O certificado traz informações para ajudar o cliente a garantir que a chave é de quem ele realmente espera. Para isso, eles possuem uma assinatura de alguém que garante a autenticidade de quem o está enviando. Vamos usar um exemplo fictício baseado em alguns personagens conhecidos, só para ilustrar esse processo:

Frodo tem consigo um objeto secreto e poderoso, e sabe que alguém virá ajudá-lo a levar esse objeto para algum lugar em algum momento. Um certo dia ele recebe uma carta com os seguintes dizeres:

"Olá Frodo, meu nome é Gandalf e fiquei sabendo que você tem um objeto de grande poder por aí. Preciso saber se é verdade, pois vou te ajudar a levá-lo. Você o possui?"

Junto com o documento, ele enviou um selo para que Frodo possa fechar a carta magicamente e enviar de volta. Mas como ele vai ter certeza de que essa chave é do Gandalf mesmo e não de outra pessoa que está tentando saber se ele está com o objeto para atacá-lo? Então, ele vê que na parte de trás desse selo tem uma assinatura que só Elrond, o elfo de Rivendell, poderia fazer.

Frodo pega um livro que contém as assinaturas das pessoas que ele confia, e a assinatura do Elrond está lá. Ele compara as assinaturas e verifica que aquela é realmente a assinatura de Elrond, logo, ele pode confiar que a carta que ele recebeu é do Gandalf mesmo.

Como Frodo confia em Elrond e sabe que ele não assinaria um selo de alguém de Mordor, ele vai confiar que aquela chave é do Gandalf mesmo e responder usando o selo mágico de Gandalf para que ninguém possa ver o conteúdo da carta. Apenas o Gandalf tem o poder de retirar o selo e ver o conteúdo original que está dentro da carta.

Agora trazendo isso para o mundo real, temos:

- **Frodo** como o cliente;
- **Gandalf** como o servidor;
- **Alguém de Mordor** como o possível atacante;
- **A carta** como os dados;
- **Elrond** como **C**ertificate **A**uthority;
- **O livro de assinaturas**, como o pacote de chaves públicas que o cliente tem disponível em seu sistema operacional;
- **O selo** como a chave pública do certificado;
- **O poder de Gandalf** como sua chave privada;
- **A habilidade de escrita de Elrond** como chave privada do Certificate Authority.

O seu computador com certeza tem um pacote com as chaves públicas de muitos *CAs*, também conhecidos como *Certificate Authority*, que são empresas conhecidas e confiáveis. Se um certificado HTTPS foi considerado válido, quer dizer que esse certificado foi assinado com a chave privada de algum dos CAs cuja chave pública está disponível no seu sistema operacional.

Validando o certificado

Agora que entendemos por que o certificado é importante e por que é usado para resolver essa questão de confiança, vamos seguir avaliando como isso se aplica na nossa requisição para o `desconstruindoaweb.com.br`.

O Chromium, que é o cliente no nosso caso, agora tem todas as informações do servidor e vai analisá-las para decidir se vai realmente estabelecer a conexão com o servidor. A única coisa realmente nova no nosso caso é o certificado, pois as outras informações foram apenas confirmações do servidor sobre informações que havíamos enviado.

Figura 5.10: Hierarquia de confiança do certificado do desconstruindoaweb

Vamos olhar mais de perto algumas informações do certificado do `desconstruindoaweb.com.br`, começando com a hierarquia de confiança. Lá temos 3 linhas, sendo a última referente ao certificado do `desconstruindoaweb.com.br` que vamos olhar já já. Mas antes, vamos ver se confiamos nas entidades que garantem as informações desse certificado.

Acima do certificado do `desconstruindoaweb.com.br`, está o certificado chamado *Let's Encrypt Authority X3*, que é o do Let's Encrypt[6]. O Let's Encrypt é um novo serviço que pretende se tornar um CA gratuito que pode ser facilmente automatizado por meio de APIs. Atualmente essas APIs já estão prontas e podem ser usadas por qualquer um que queira gerar um certificado sem custos. Como podemos ver, o `desconstruindoaweb.com.br` usa esse CA.

Na linha de cima do certificado do Let's Encrypt, ainda há mais um chamado *DST Root CA X3*, que é o certificado da IdenTrust[7]. Como a Let's Encrypt ainda não está na lista de CAs confiáveis de todos os navegadores e sistemas operacionais, eles precisam que alguém que o cliente já confie assine os seus certificados. Com a assinatura de um CA como a IdenTrust, eles garantem que alguém confiável confia neles, tornando-os confiáveis também.

Os certificados da IdenTrust são autoassinados, dizendo que eles

mesmos garantem a veracidade dos seus próprios certificados. Isso é bem comum com os *Root certificates* das grandes CAs, pois eles são os certificados que ficam no topo da hierarquia de confiança.

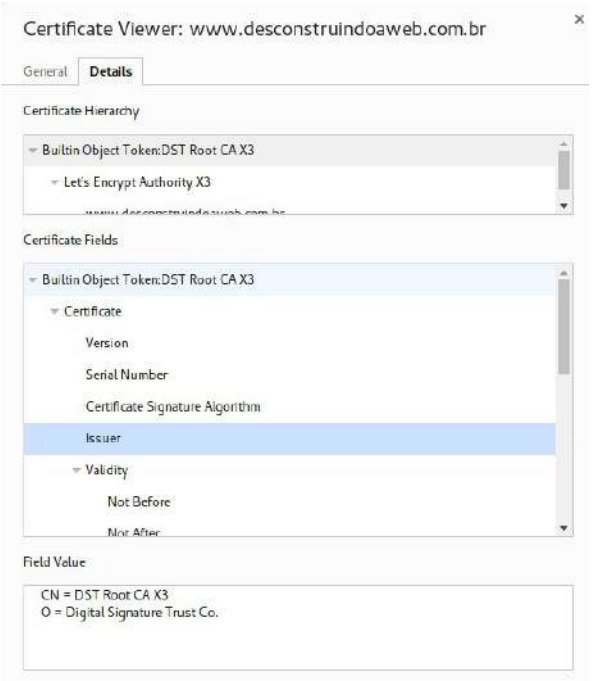


Figura 5.11: Certificado autoassinado da IdenTrust

O Chromium usa uma biblioteca chamada NSS[8] para ter acesso aos certificados instalados. Olhando de uma forma bem básica para as chaves instaladas no nosso sistema de estudo, temos:

```
$ ls /etc/ssl/certs/ | grep DST | grep Root
DST_Root_CA_X3.pem
```

Esse é o mesmo certificado que vimos anteriormente! Você deve possuí-lo também possivelmente em outro local, dependendo do seu sistema operacional. Esse será o certificado usado para verificar a assinatura e confiar no certificado da Let's Encrypt, que por sua vez, nos fará acreditar no certificado do

`desconstruindoaweb.com.br` .

Além de garantir que todas as assinaturas estão corretas e são confiáveis, o navegador vai olhar também a **data de validade**. O certificado contém sua data de criação e expiração, sendo que ele não será válido antes de ser criado e nem depois de expirar, obviamente.

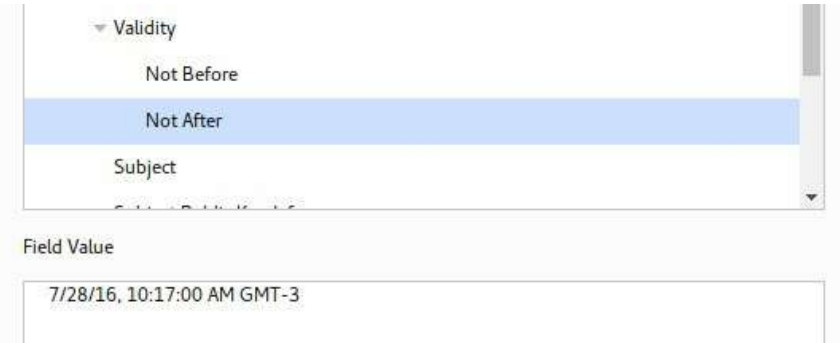


Figura 5.12: Validade do certificado do desconstruindoaweb.com.br

O vencimento do certificado do `desconstruindoaweb.com.br` para essa requisição é `7/28/16, 10:17:00 AM GMT-3` , ou 28 de julho de 2016, que na data de escrita deste livro ainda está válido.

Como os certificados do Let's Encrypt possuem uma data de validade curta, o certificado não será o mesmo na data de publicação. Apesar desse típico problema de viagem no tempo, vamos continuar nosso estudo assumindo que estamos em algum momento de junho do ano de 2016.

Entendendo o cipher

Lembram-se de que o `Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)` ia ser importante em breve? Chegou o momento de estudá-lo com mais detalhes.

Para entender melhor o cipher, vamos utilizar o comando `openssl`:

```
$ openssl ciphers -V ECDHE-RSA-AES128-GCM-SHA256
0xC0,0x2F - ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH Au=RSA Enc
=AESGCM(128) Mac=AEAD
```

Baseando-se na saída do comando `openssl`, podemos estudar cada uma das partes do cipher separadamente:

- `kx=ECDH` — é o algoritmo *Elliptic curve Diffie–Hellman* e o `E` que só aparece no nome do cipher é referente ao uso de *Ephemeral keys*[9]. Esse é mais um daqueles algoritmos matemáticos complicados de criptografia assimétrica que não vamos entrar em tantos detalhes. O que ele faz é garantir a primeira parte do handshake, na qual as duas partes vão concordar com um segredo em comum, para então usar criptografia simétrica, que é bem mais barata em termos de processamento.
- `au=RSA` — é o algoritmo de assinatura usado para autenticar a troca de chaves.
- `enc=AESGCM(128)` [10] — Quebrando um pouco mais, o `AES` [11] é o algoritmo de criptografia simétrica, o `GCM` [12] [13] é um modo de operação (do inglês, *mode of operation*) que provê confidencialidade e autenticação da origem, e por fim o `128` é o tamanho da chave que será gerada para a criptografia simétrica.
- `Mac=AEAD` — O *Authenticated Encryption with Associated Data* (ou `AEAD`) é uma interface para algoritmos de criptografia[14], e o GCM faz parte desses algoritmos.

A troca de chaves (isso, isso, isso)

Agora que sabemos o que significa aquele monte de siglas separadas por *underline* no nome do cipher, vamos usar toda essa informação para entender como a criptografia funciona. Para começar o estudo, veremos como funciona a troca de chaves que começa com o *Server Key Exchange*.

A primeira fase usa o algoritmo de `kx`, ou *Key Exchange*, chamado `ECDHE_RSA`, visto na seção anterior quebrado em `ECDHE` e `RSA`. Vamos evitar toda a matemática por trás das curvas elípticas que deram o nome ao *Elliptic curve Diffie-Hellman* (ou `ECDH`), e vamos apenas entender como ele ajuda no processo de lidar com a chave que será transferida para o servidor. Se você gostar bastante de matemática e quiser se aprofundar, dê uma olhada no paper disponível no site do *Standards for Efficient Cryptography Group*[15].

Na etapa de *Server Key Exchange*, visto na seção *Server Hello* de uma forma superficial, o servidor envia a chave pública gerada pelo algoritmo de Diffie-Hellman juntamente com as informações usadas no processo.

O algoritmo de Diffie-Hellman garante a troca de chaves via criptografia assimétrica como falamos no tópico *O certificado e a chave pública* quando estávamos estudando como o certificado funciona. Esse algoritmo faz isso usando as tais curvas elípticas que comentamos há pouco, e são os valores referentes a essa curva que vemos no Wireshark.

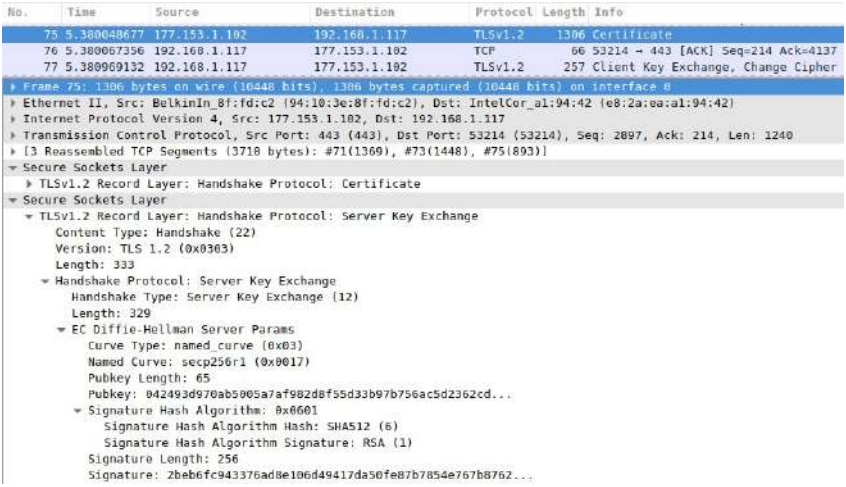


Figura 5.13: Detalhes do Server Key Exchange

É nesse momento que o `RSA` faz a sua parte no processo. Utilizando o Wireshark, podemos ver no campo *Signature Hash Algorithm* que ele está usando o algoritmo `RSA` para o processo de assinatura da chave pública. O servidor utiliza esse algoritmo para provar que a chave enviada é realmente dele e não de outra pessoa que interceptou a conexão e está enviando uma chave diferente. Ele faz isso criando uma assinatura com a chave privada do servidor, utilizando como base os dados da chave pública gerada. Essa assinatura vai ser enviada no campo *Signature* da mensagem, possibilitando ao cliente verificá-la usando a chave pública do certificado, que é par da chave privada usada para gerar a assinatura.

É um processo que possui muitos componentes. Logo, para deixar mais claro, vamos numerar os passos desde o começo da participação das chaves pública e privada:

1. Em algum momento, o dono do servidor gerou a chave pública e privada para si.
2. Durante o `Server Hello`, a chave pública foi enviada junto com o certificado.

3. O servidor envia uma nova chave que foi gerada para essa mensagem e, junto a ela, no campo `Signature`, segue um *hash* gerado como assinatura. Esse hash foi criado utilizando a chave privada gerada pelo dono do servidor, no passo 1, que pode ser refeito pela chave pública que já está com o cliente.

Agora que estudamos melhor o Server Key Exchange que tinha ficado para trás, vamos seguir para o **Client key exchange**.

Nesse passo, o cliente vai receber a mensagem enviada no passo 3, que descrevemos a pouco, e utilizar a chave pública que está no certificado para validar a assinatura. Isso só é possível graças ao algoritmo `RSA` que garante que a chave pública não consiga a mesma assinatura sem um esforço absurdo. Além disso, ele também garante que seja possível validar se ela foi gerada pela chave privada.

Se a assinatura estiver correta, fica comprovado que a chave é do `desconstruindoaweb.com.br`, e então o cliente pode usar os dados da curva enviados pelo servidor para gerar seu par de chaves também. Caso a validação da assinatura não retorne o valor esperado, o cliente vai assumir que quem está enviando a chave não é o servidor que ele está aguardando a conexão, e falhará.

Considerando que tudo correu bem com a validação de assinaturas, o cliente gera seu par de chaves para a conexão e envia a mensagem de *Client Key Exchange* para o servidor. Essa mensagem conterá apenas a sua chave pública, que podemos ver no campo `Pubkey` no Wireshark.

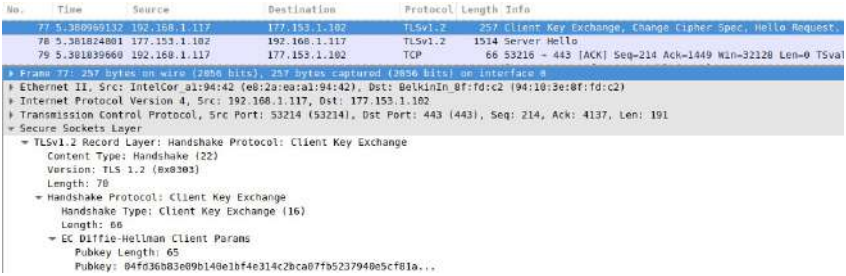


Figura 5.14: Detalhes do Client Key Exchange

Os valores gerados que não foram mostrados a ninguém são a `pre\_master secret` e a `master secret`. A `master secret` é baseada na `pre\_master secret` que é usado na criptografia simétrica.

Fazendo a criptografia acontecer

Até o momento conseguimos ver vários detalhes utilizando o Wireshark, mas a partir de agora já não vai mais ser possível descobrir muita coisa.

O cliente envia uma mensagem de *Change Cipher Spec* dizendo que, a partir de agora, ele vai começar a enviar mensagens criptografadas com a chave que eles decidiram, e pouco depois o servidor faz o mesmo.

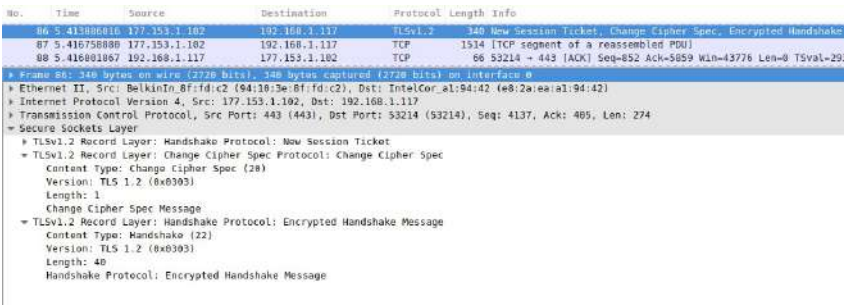


Figura 5.15: Mensagem de change spec

A partir desse momento, não conseguimos ter mais nenhuma informação interna utilizando o Wireshark. Se olharmos os pacotes que aparecem depois dessa mensagem, veremos apenas o campo `Encrypted Application Data: ...` . Este contém os dados encriptados por criptografia simétrica, que usa a chave possuída por ambos.

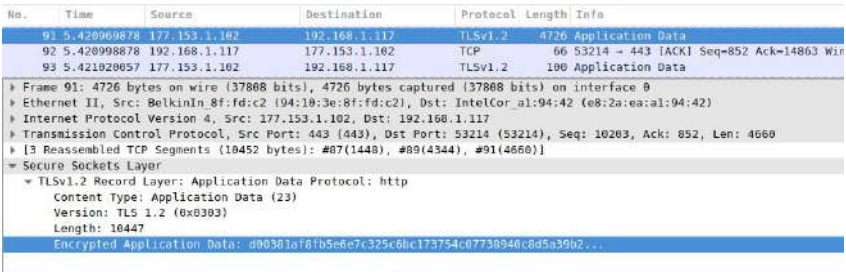


Figura 5.16: Um pacote com dados criptografados no Wireshark

Cliente e servidor começam a usar criptografia simétrica `AES` com `GCM` para a criptografia dos dados. O `AES` (*Advanced Encryption Standard*) foi criado em 2001 para substituir o `DES` (*Data Encryption Standard*) de 1977, que é suscetível a falhas. O `GCM` é utilizado para autenticar os valores que são criptografados com `AES` e enviados. Isso é feito para que o servidor tenha certeza de que nenhum dos dados foi modificado no meio do caminho.

Entrar em detalhes sobre como o `AES` ou o `GCM` trabalham não é muito prático e foge do escopo deste livro. Aqui vão algumas dicas para os mais ~~corajosos~~ curiosos em entender esses protocolos:

- O post do Jeff Moser[16] é um ótimo material para entender como o AES funciona, com desenhos e um link para uma implementação bem comentada.
- O vídeo do David Wong[17] no YouTube mostra de uma forma detalhada qual é o papel do GCM junto com o AES.

Neste momento, tanto cliente como servidor estão com uma chave compartilhada que é bem grande e difícil de ser encontrada por um terceiro. Todos os dados enviados são criptografados com essa chave e os algoritmos ajudam o receptor da mensagem a validar que ela está intacta, tornando um ataque bem improvável com o poder computacional atual.

Improvável com o poder computacional atual?

A última frase do parágrafo anterior é um tanto quanto enigmática. Por que esses algoritmos diminuem a probabilidade de ataque com o poder computacional atual?

A criptografia que utilizamos atualmente se baseia no fato de que não temos processamento o suficiente para encontrar a chave de criptografia por força bruta. Isso quer dizer que, se alguém tentar descobrir a chave por tentativa e erro, ele demoraria uma quantidade enorme de anos, tornando o ataque inviável.

A mesma chave, gerada pelo mesmo algoritmo, poderia ser encontrada por força bruta caso algum computador com processamento absurdamente maior apareça no mercado.

5.4 TESTANDO UMA CONEXÃO HTTPS MANUALMENTE

Depois que todo o túnel estiver criado, o protocolo HTTP é utilizado da mesma forma que vimos no capítulo anterior. Podemos inclusive fazer um teste parecido com o que fizemos usando `telnet`, mas dessa vez utilizando o comando `openssl`.

```
$ openssl s_client -connect desconstruindoaweb.com.br:443

CONNECTED(00000003)
depth=2 O = Digital Signature Trust Co., CN = DST Root CA X3
verify return:1
depth=1 C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
verify return:1
depth=0 CN = www.desconstruindoaweb.com.br
verify return:1
---
Certificate chain
 0 s:/CN=www.desconstruindoaweb.com.br
   i:/C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3
 1 s:/C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3
   i:/O=Digital Signature Trust Co./CN=DST Root CA X3
 2 s:/C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3
   i:/O=Digital Signature Trust Co./CN=DST Root CA X3
---
Server certificate
-----BEGIN CERTIFICATE-----
(...) muitas linhas com o conteúdo do certificado aqui...
-----END CERTIFICATE-----
subject=/CN=www.desconstruindoaweb.com.br
issuer=/C=US/O=Let's Encrypt/CN=Let's Encrypt Authority X3
---
No client certificate CA names sent
Peer signing digest: SHA512
Server Temp Key: ECDH, P-256, 256 bits
---
SSL handshake has read 4370 bytes and written 433 bytes
---
New, TLSv1/SSLv3, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES256-GCM-SHA384
    Session-ID: 3EB4AF39370DCC4ED84B0591B25BE2F8865F89E5D4919E62A6
CD54EA033DA399
    Session-ID-ctx:
    Master-Key: 561398BAF38617CAAEC957AB21EC6C904DECB7E26C00B6AC3A
A0FD24EA926D7844AEC9E4E5914CE230E1D93A41807206
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
```

```
    SRP username: None
    TLS session ticket lifetime hint: 300 (seconds)
    TLS session ticket:
    00a0 - 07 f6 8e 82 3e 37 33 93-fb b6 4a ff 53 11 25 9e
    (...) várias linhas parecidas com a linha anterior para mostra
r o SSL ticket

    Start Time: 1465761436
    Timeout   : 300 (sec)
    Verify return code: 0 (ok)
---
```

Até esse momento, ele fez todos os passos que vimos na seção anterior e mostrou as informações referentes a eles. Agora que a conexão segura está estabelecida, ele deixa o terminal livre para digitarmos comandos, assim como o `telnet` faz. Com isso, podemos usar o protocolo HTTP da mesma forma que vimos no capítulo anterior:

```
GET / HTTP/1.1
Host:desconstruindoaweb.com.br
```

Esse comando vai ser enviado por uma conexão segura até o destino, que vai nos retornar a página referente a raiz do site.

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Status: 200 OK
(...)
```

5.5 O QUE FICA SEGURO?

O HTTPS ganhou ainda mais popularidade quando uma extensão para Firefox chamada firesheep[18] surgiu. Essa extensão mostra como é simples roubar as sessões de outros usuários quando a conexão é feita apenas com HTTP. Isso acontece porque todos os dados ficam disponíveis em texto puro, facilitando para que outros usuários possam capturar os dados na rede.

Utilizando HTTPS, podemos deixar de nos preocupar que alguns dados ficarão expostos como texto puro na internet, caso alguém consiga observar os pacotes no meio do caminho. Como vimos durante todo o capítulo, as únicas informações que ficam disponíveis para alguém observando os dados no meio do caminho são: porta de origem e destino, e o *hostname* de destino, todo o resto é dado criptografado.

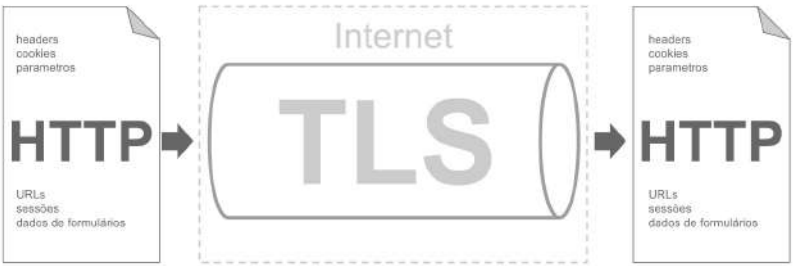


Figura 5.17: Informações protegidas

Em caso de APIs que trafegam dados como *token* em cabeçalho ou na própria URL, o HTTPS se torna primordial. Qualquer um com acesso a uma rede sem fio local poderia capturar os pacotes e roubar essas informações sem muito trabalho. Um exemplo simples disso é uma rede *Wi-Fi* de cafeterias e outros locais públicos, mas poderia ser em qualquer local entre o cliente e o servidor. Vamos olhar mais de perto o que há entre esses dois no próximo capítulo.

5.6 RESUMO

O HTTPS é a forma segura de transportar HTTP do cliente para o servidor. Por baixo dos panos, o HTTPS utiliza o protocolo TLS, que cuida de todo o processo de criptografia dos dados. Usando um sistema como o Wireshark, é possível ver todo o processo de funcionamento do TLS, juntamente com os passos do TCP que

foram vistos no capítulo anterior. Portanto, vamos usá-lo para acompanhar de perto.

O primeiro passo é a resolução de domínio, seguido three-way handshake do TCP como vimos em capítulos anteriores. Logo após essa fase, começa o **handshake do TLS**. A primeira fase do handshake do TLS é o `Client Hello` , no qual o cliente pede uma conexão TLS para o servidor enviando as opções que está disponível a aceitar para que essa conexão seja estabelecida. O servidor recebe e envia um `Server Hello` com as opções que funcionam para ele e, logo em seguida, envia um certificado digital.

O **certificado** possui uma chave pública e alguns dados que ajudam a provar que essa chave pública é realmente do servidor. Um exemplo desses dados é a assinatura de alguma entidade externa que grande parte dos clientes confiam. O servidor envia também uma segunda chave pública caso o cliente aceite o certificado, e essa mensagem é chamada de **Server Key Exchange**. Por último, ele envia um `Server Hello Done` para dizer que terminou sua parte.

O cliente faz a validação do certificado, datas e assinaturas. Caso essas informações estejam de acordo com o esperado, ele valida também as informações enviadas pelo servidor no Server Key Exchange e usa-as para gerar seu par de chaves.

A chave pública é enviada para o servidor na mensagem de **Client Key Exchange** enquanto o cliente mantém sua chave privada, que será usada para gerar a chave de criptografia simétrica que permeará toda a comunicação a partir de agora. Ambos enviam uma mensagem de **Change Cipher Spec** e, a partir daí, todo o conteúdo é criptografado e não conseguimos mais identificar no Wireshark o que está acontecendo dentro da conexão.

5.7 REFERÊNCIAS

1. *RFC2818* e a definição do HTTPS — https://tools.ietf.org/html/rfc2818
2. *RFC5246* e a definição do protocolo TLS. Informações sobre a versão do TLS e SSL no Glossário (apêndice B, página 81) — https://tools.ietf.org/html/rfc5246
3. *RFC3546* e a definição das extensões do TLS — https://www.ietf.org/rfc/rfc3546.txt
4. *RFC5280* e a definição dos certificados — https://tools.ietf.org/html/rfc5280#section-4.1
5. Vídeo de explicação de criptografia assimétrica com matemática simples — https://www.youtube.com/watch?v=YEBfamv-_do
6. Let's Encrypt, CA gratuito e com API — https://letsencrypt.org
7. Certificado da CA chamada Identrust — https://www.identrust.com/certificates/trustid/root-download-x3.html
8. Network Security Services (NSS) da Mozilla — https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS
9. Ciphers de Elliptic Curve Cryptography (ECC) para o TLS — https://tools.ietf.org/html/rfc4492#page-7
10. *RFC5288* AES e GCM — https://tools.ietf.org/html/rfc5288
11. Detalhe e especificação do AES — http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
12. Especificação do GCM —

http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf

13. Explicação reduzida e implementação do GCM — http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf

14. RFC do AEAD, Authenticated Encryption with Associated Data — https://tools.ietf.org/html/rfc5116

15. Especificação dos algoritmos de Elliptic Curve Cryptography — http://www.secg.org/sec1-v2.pdf

16. Explicação do AES com quadrinhos — http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html

17. Vídeo que explica com detalhes o uso do GCM — https://www.youtube.com/watch?v=g_eY7JXOc8U

18. Firesheep: extensão de firefox para roubar sessões de outros usuários — https://codebutler.github.io/firesheep/

Capítulo 6

PARA A INTERNET E ALÉM!

Apesar de nossa jornada já ter chegado ao servidor em muitos momentos, nosso estudo ainda está no protocolo que é executado no cliente para se comunicar com o servidor. Pensando na nossa famigerada camada "Ozzy", nós passamos pela camada de aplicação, onde está DNS e HTTP(s), e pelas camadas de transporte e rede com TCP/IP e UDP/IP.

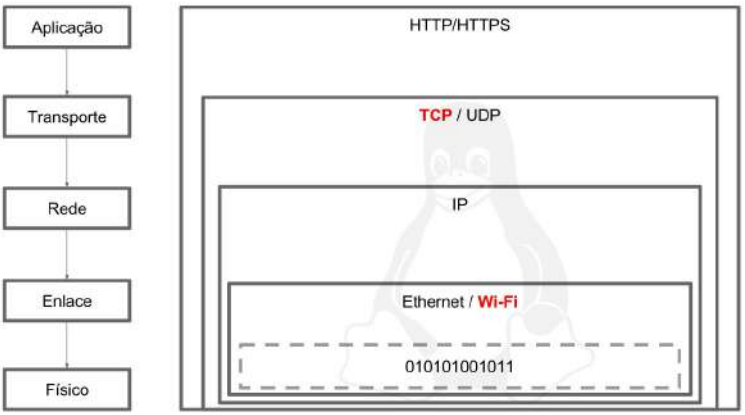


Figura 6.1: Os protocolos do nosso modelo "Ozzy"

Agora vamos um pouco mais a fundo para descobrir como esses dados são enviados para o outro lado. Vamos estudar o que precisa ser feito no cliente para que eles finalmente saiam do computador rumo ao servidor de destino.

6.1 ETHERNET OU WI-FI

Agora começamos nossa jornada para entender como o sistema operacional faz para tirar a informação do cliente e levar ao servidor. Primeiro, vamos nos situar: nós estamos logo depois do pacote TCP ser criado. Apesar de estarmos estudando os protocolos de uma maneira mais completa, testando-os até a chegada ao destino, o nosso estudo chegou apenas até o TCP.

O primeiro passo é descobrir o que o computador cliente precisa fazer para enviar esses dados para a rede local, onde vamos assumir que haverá um roteador para enviá-los para a internet.

Aqui entramos na tal *camada de enlace* do nosso modelo "Ozzy", e já começamos a depender dos equipamentos que estão na *camada física* para enviar a informação para o outro lado. O equipamento necessário nessa fase é a placa de rede, que pode aceitar somente cabo ou pode possuir uma antena *wireless*. Para cada um dos casos, há protocolos diferentes para usarmos. Vamos olhar Ethernet e Wi-Fi, tentando fugir daquelas explicações formais das aulas de redes da faculdade.

Ethernet

Ethernet é um conjunto de tecnologias que provê a rede cabeada que conhecemos atualmente. Essas tecnologias são especificadas por um órgão chamado *Institute of Electrical and Electronics Engineers*, mais conhecido como *IEEE*. Eles não são muito criativos com nomes, e a especificação da Ethernet é chamada de *802.3*. Essa especificação é de 2012 e tem 3.500 páginas[1], divididas em 6 arquivos. Ela mantém os padrões desde a época dos cabos coaxiais até o padrão atual do cabo de par trançado, por esse motivo ela é tão extensa.

Caso você esteja utilizando GNU/Linux, há uma ótima

ferramenta chamada `ethtool` [2], que ajuda na tarefa de obter informação sobre quais partes da Ethernet o nosso computador está utilizando.

A principal interface cabeada do nosso computador de estudo se chama `enp0s25` . Utilizando o comando `ethtool` nessa interface, temos o seguinte resultado:

```
$ sudo ethtool enp0s25
Settings for enp0s25:
    Supported ports: [ TP ]
    Supported link modes:   10baseT/Half 10baseT/Full
                            100baseT/Half 100baseT/Full
                            1000baseT/Full
    Supported pause frame use: No
    Supports auto-negotiation: Yes
    Advertised link modes:  10baseT/Half 10baseT/Full
                            100baseT/Half 100baseT/Full
                            1000baseT/Full
    Advertised pause frame use: No
    Advertised auto-negotiation: Yes
    Speed: 1000Mb/s
    Duplex: Full
    Port: Twisted Pair
    PHYAD: 1
    Transceiver: internal
    Auto-negotiation: on
    MDI-X: on (auto)
    Supports Wake-on: pumbg
    Wake-on: g
    Current message level: 0x00000007 (7)
                   drv probe link
    Link detected: yes
```

Podemos constatar algumas coisas legais sobre o que está sendo usado da Ethernet só com a saída desse comando. A primeira delas é que o computador suporta duas formas de comunicação só de uma via, ou **half** duplex, que é aquele antigo modo de comunicação utilizado com cabo coaxial etc. E além dos half, ele possui três full duplex que fazem troca de dados simultâneo de duas vias, o que consideramos normal hoje em dia.

De acordo com o `Port`, estamos usando cabo de par trançado, aquele azul que todo mundo conhece. Pelo argumento `Speed`, podemos ver que o computador está usando uma rede de 1Gb, portanto, olhando também o argumento `Duplex`, podemos concluir que, de todos os modos que foram anunciados no `Advertised link modes`, o escolhido foi `1000baseT/Full`.

Os tipos `1xbaseT/Full` são os mais comuns atualmente, principalmente em redes domésticas como a que este computador está conectado. Seu nome tem um padrão simples: o primeiro número é referente à quantidade de Mbs, e o **T** vem de *Twisted pair cable*, ou cabo de par trançado.

Mais comum que usar Ethernet atualmente é utilizar uma rede Wi-Fi. Por esse motivo, não vamos focar muito mais nos seus padrões e especificações, e vamos partir para as conexões sem fio.

Wi-Fi

Wi-Fi é uma tecnologia que permite que dispositivos se conectem a uma rede sem fio, também conhecida como *WLAN*, baseada nos padrões *802.11* da IEEE. Assim como o padrão *802.3* que vimos na seção sobre a Ethernet, o padrão 802.11 carece de criatividade de nome e possui uma extensa documentação, com 2.793 páginas[3] na versão de 2012.

O nome Wi-Fi é apenas comercial, que soa **muito melhor** que *Dispositivo compatível com IEEE 802.11*. Por trás do nome Wi-Fi e do logo preto e branco que costumamos ver por aí, existe uma empresa sem fins lucrativos chamada *Wi-Fi Alliance*. Esta é responsável por certificar os dispositivos que seguem os padrões do 802.11.

Figura 6.2: Logo do Wi-Fi para quem é certificado pela Wi-Fi Alliance

As novas versões do Wi-Fi seguem um padrão alfabético, sendo que o mais conhecido no momento é o padrão **N**. Os padrões só possuem essa sigla até se tornarem parte integrante da especificação. Apesar disso, o mercado usa esses nomes para vender seus equipamentos, pois essa letra adicional ajuda a mostrar quais funcionalidades o roteador suporta ou se é compatível.

Apesar de haver uma sopa de letrinhas que diferencia as adições no 802.11, algumas ficaram mais conhecidas:

- **802.11b**, uma das primeiras a fazer sucesso para o usuário final, usa a frequência de 2.4GHz, mas ainda é um pouco lento.
- **802.11g**, possivelmente uma das versões mais conhecidas. Essa versão apareceu na época em que o Wi-Fi começou a ficar mais conhecido entre os usuários comuns de internet. Usa a frequência de 2.4GHz assim como a 802.11b, mas é quase 5x mais rápida no melhor caso.
- **802.11n** é o padrão do momento. Agora em 2016, se você for comprar um roteador simples no Brasil, ele provavelmente será compatível com o padrão **N**. Pode usar tanto a frequência de 2.4GHz como a de 5GHz, além de trazer novidades, como a possibilidade de usar múltiplas antenas para transmissão e recebimento de

dados.

- **802.11ac** é o padrão que tende a assumir em breve. Usa **somente** a frequência de 5GHz, o que o deixa com um alcance um pouco menor, mas a velocidade pode chegar ao dobro do **N**[4], e sofre menos interferência.

Isso é apenas o básico do Wi-Fi. Vamos entender como sair do sistema operacional primeiro e depois podemos olhar para ele de uma maneira mais prática.

6.2 SAINDO DO SISTEMA OPERACIONAL

Como escolhemos Wi-Fi como nosso meio de saída do computador local, vamos ir mais a fundo e ver como é feita a implementação disso. Lembrando de que nós acabamos de sair da criação do pacote TCP na nossa requisição e estamos entrando na camada de enlace. Nessa camada, também conhecida como *link layer*, tudo é gerenciado pelo kernel do sistema operacional.

Para continuar nosso estudo, precisaremos de informações sobre o hardware usado para a conexão Wi-Fi. Para isso, vamos utilizar o comando `lspci`, que lista os dispositivos PCI conectados:

```
$ lspci | grep -i wireless
03:00.0 Network controller: Intel Corporation Wireless 7260 (rev 8
3)
```

Com isso, já sabemos que a placa wireless do nosso computador de estudo é a dual-band `7260` da intel[5]. Por estarmos conectados utilizando essa placa, precisamos ter certeza de que ela está funcionando. Portanto, podemos procurar por um módulo do kernel que implemente as suas funcionalidades.

Para verificar, vamos utilizar o comando `lsmod` que lista os

módulos carregados:

```
$ lsmod | grep wifi
iwlwifi               176128  1 iwlmvm
cfg80211              491520  3 iwlwifi,mac80211,iwlmvm
```

Para ter certeza, vamos pedir mais informações sobre o módulo `iwlwifi` e procurar pelo número do nosso hardware:

```
$ modinfo iwlwifi | grep 7260
firmware:       iwlwifi-7260-13.ucode
```

Bingo! Esse é realmente o driver que estávamos procurando. :)

Nossa ideia nessa parte não é olhar o código do kernel minuciosamente e entender onde cada função está sendo chamada, mas entender como funciona o fluxo das informações para sair do sistema operacional. Vamos olhar na estrutura do kernel para termos uma ideia de como os dados fazem para sair, mas sem se preocupar com o fluxo exato das funções e detalhes de implementação.

Um paper[6], escrito por *Ashwin Kumar Chimada* da *University of Kansas* em 2005, mostra o caminho da saída de um pacote do kernel. Apesar de a data de escrita ser um pouco antiga, muita coisa ainda funciona da mesma maneira. E baseando-nos nele, temos:

- O `socket`, que vimos em capítulos anteriores, na camada de aplicação. Ele está definido em `net/socket.c` [7].
- O `socket` decide o tipo de transporte que será usado no arquivo `net/ipv4/af_inet.c` [8], que no nosso caso vai ser TCP, afinal, é uma chamada HTTP.
- O pacote vai ser construído e, em algum momento, a função `tcp_sendmsg` do arquivo `net/ipv4/tcp.c` [9] vai ser chamada para encaminhar o pacote para a próxima camada.

- A camada de rede vai receber, decidir a rota e construir o pacote IP na função `ip_queue_xmit` do arquivo `net/ipv4/ip_output.c` [10]. E depois enviará para a camada de enlace.
- Na camada de enlace, acontecem algumas coisas mais complicadas antes de chamar o driver diretamente. No meio do caminho, o código passa pelo framework `cfg80211` do kernel e por filas de envio, buffers e outras coisas que não vamos entrar em detalhes.
- Até que esse pacote chega ao nosso driver `iwlwifi`, que está disponível em `net/wireless/intel/iwlwifi` [11]. Esse driver vai lidar com o firmware específico para o hardware instalado, que no nosso caso é o `iwlwifi-7260-13.ucode`, um firmware proprietário da Intel. Ele lida diretamente com o hardware para fazer a placa de rede trabalhar como o esperado.

Graças ao mundo open source, temos a oportunidade de ver o processo quase completo de como as coisas funcionam, apenas assumindo que o software da placa de rede funciona magicamente.

6.3 O CAMINHO PARA O ROTEADOR

Nesse momento, nosso estudo acabou de sair fisicamente do cliente, sendo enviado via ondas de rádio para o roteador Wi-Fi. Não é muito prático estudar como as ondas de rádio e os receptores funcionam, mas há coisas práticas e curiosidades que podemos absorver dessa conexão.

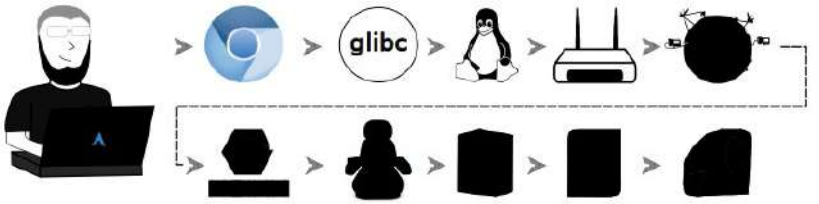


Figura 6.3: A jornada para levar os dados para o roteador

Da placa de rede ao roteador

A *camada de enlace* e a *camada física* são diferentes das outras do nosso modelo "Ozzy". O objetivo final delas é apenas o próximo ponto da rede, enquanto as outras visam o objetivo final. A camada de rede, por exemplo, usa o IP de destino da requisição.

Nessas duas camadas, os endereços de comunicação são diferentes. As placas de rede lidam apenas com endereços MAC e não com endereços IP, como vimos até agora. Todas as placas de rede possuem um endereço MAC, que são representados por 6 números hexadecimais separados por `:` (dois pontos). Um exemplo de um endereço MAC real é `28:d2:44:c1:fa:ae`.

Para descobrir o endereço MAC do outro ponto, é necessário utilizar o protocolo ARP, definido pela *RFC826*[12]. Ele envia uma pergunta para todos os computadores da rede na qual ele está, perguntando quem tem determinado IP. Todos os computadores vão verificar o pedido e responder com seu endereço Mac, caso seja o computador responsável por responder por aquele IP. Com o endereço MAC do próximo ponto, a camada física pode fazer a transferência do sinal.

Curiosidades sobre o envio do sinal

Na seção de introdução ao Wi-Fi deste capítulo, vimos que a versão `g` do protocolo ainda é uma das mais conhecidas, apesar de

a versão `n` já estar quase tomando o seu lugar. Vamos assumir que nossa conexão está utilizando a versão `g` do protocolo, como é o caso para muitas casas aqui no Brasil.

Por usar somente a frequência de 2.4GHz, o wireless `g` compete com vários outros dispositivos. Com isso, podemos esperar que nossa conexão para o `desconstruindoaweb.com.br` brigará com *baby monitors*, micro-ondas, telefones sem fio, alguns dispositivos *bluetooth* e outros dispositivos que operam na mesma frequência[13].

Além de toda a interferência de outros dispositivos, temos também as outras redes Wi-Fi que podem ou não estar compartilhando canais de operação com a sua. Em distribuições GNU/Linux, é possível verificar a quantidade de redes Wi-Fi e seus canais facilmente com o comando `nmcli` do pacote *NetworkManager*:

```
$ nmcli device wifi list | grep Infra | wc -l
11
```

Utilizando nosso computador de estudo, podemos ver que há 11 redes Wi-Fi funcionando simultaneamente e brigando por espaço. Vamos dar uma olhada nos canais:

```
$ nmcli device wifi list | grep Infra | sed 's/.*Infra\s\+\([0-9]\
+\).*/\1/g' | sort -n
1
3
4
6
6
10
11
11
11
136
161
```

Não se preocupe muito com esse comando `sed` maluco, ele

apenas remove outras informações, deixando apenas o canal disponível. Podemos ver que muitas delas operam no mesmo canal, e isso com certeza as deixa mais lentas quando ambas estiverem em uso.

Caso a nossa conexão esteja utilizando o canal 11 dessa lista, possivelmente teremos lentidões para o envio dos pacotes. O Wi-Fi usa o algoritmo `CSMA/CA` (*Carrier Sense Multiple Access with Collision Avoidance*). Por utilizar esse algoritmo, ele garante menos colisão de pacotes, mas a alta interferência gera mais latência[14].

Ao operar em uma rede `g`, é possível que a opção de compatibilidade com redes `b` esteja habilitada. Caso algum dispositivo que suporte apenas a versão `b` do protocolo se conecte a essa rede, ele vai fazer todos os outros dispositivos da rede ficarem mais lentos. Quanto mais pesados forem os pacotes enviados e recebidos pelo dispositivo compatível apenas com wireless **b**, pior será a velocidade para *todos* os outros dispositivos[15].

6.4 A SEGURANÇA DO WI-FI

Vamos voltar ao nosso conhecido Wireshark dos últimos capítulos, mas dessa vez faremos algumas alterações para olhar mais de perto o que está acontecendo durante a conexão Wi-Fi.

A captura padrão do Wireshark considera que todos os pacotes estão chegando via Ethernet[16]. Se olharmos as análises que fizemos anteriormente, veremos que logo após a linha que possui o *Frame* virá uma linha com o nome *Ethernet* e não 802.11 como era o esperado. Para alterar esse padrão, é necessário fazermos algumas configurações, que dependem de hardware e sistema operacional.

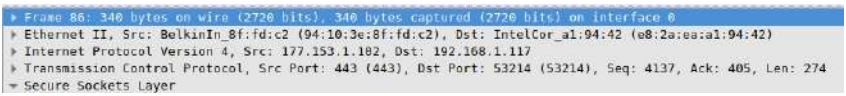


Figura 6.4: Mostrando os pacotes como Ethernet na nossa análise anterior

Por sorte, a placa de rede que vimos na seção anterior é compatível com a funcionalidade de *monitor mode*. Portanto, não vamos ter tanto trabalho para conseguir o que queremos.

A primeira coisa a se fazer é ir ao Wireshark em `Capture > Options`, e clicar duas vezes sobre a interface da placa de rede wireless, que no caso do nosso computador de estudo é a `wlp3s0`.

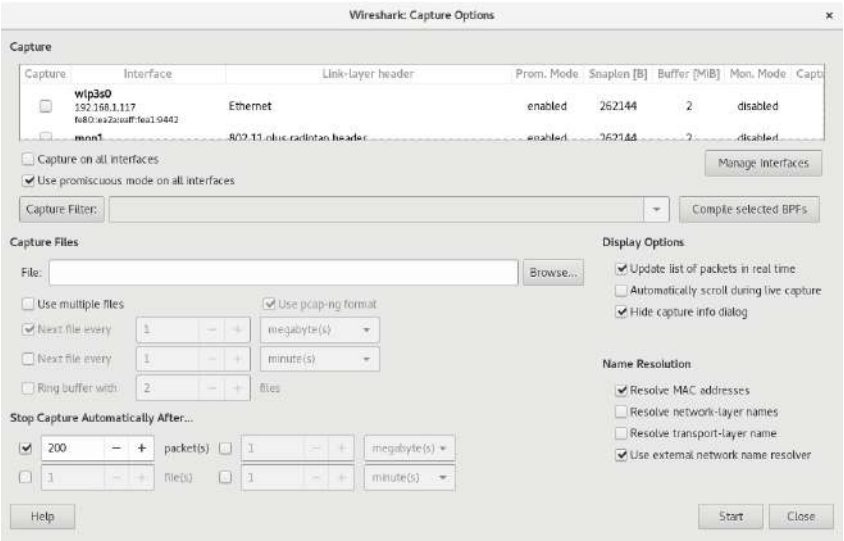


Figura 6.5: Menu de Capture Options do Wireshark — Capture > Options

Ele vai abrir uma nova janela com opções sobre a interface e já terá uma opção chamada `Capture packets in monitor mode`, que é a que estávamos procurando. Vamos deixar essa opção marcada e clicar em `OK`.

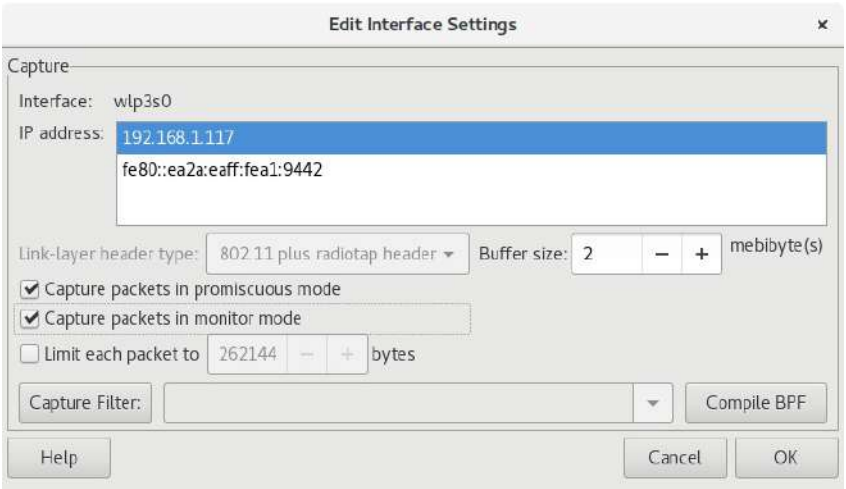


Figura 6.6: Ativando o modo de monitoração da placa wireless

Com isso, podemos clicar em *Start* para começar a captura de dados. Dessa vez, mudaremos as configurações do roteador para que não seja mais necessário o uso de senha para acessar, apenas momentaneamente.

Vamos acessar http://desconstruindoaweb.com.br. Vale salientar que estamos usando **HTTP** em vez de **HTTPS** para esse caso.

A primeira coisa que podemos ver é que os pacotes não estão mais chegando como *Ethernet*, mas sim como pacotes `802.11` , como esperávamos.

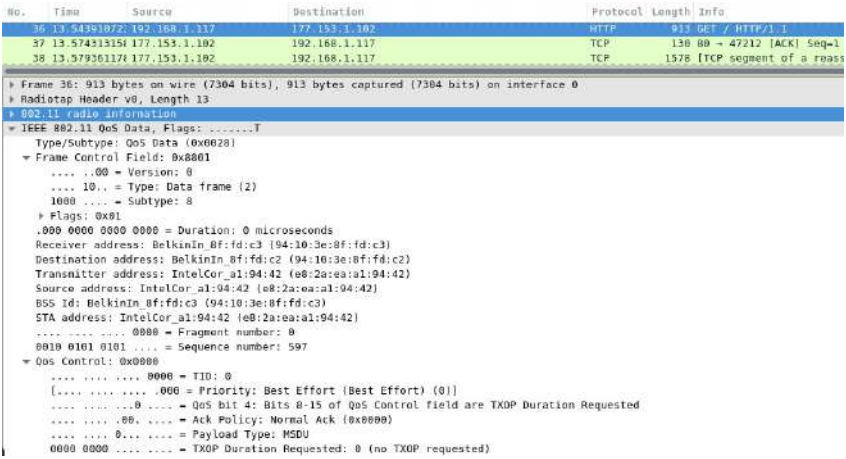


Figura 6.7: Novos campos sobre 802.11 no Wireshark

E a segunda é que os pacotes que apareceram no Wireshark estão **totalmente** em modo texto, o que nos possibilita analisar as informações trafegadas completamente.

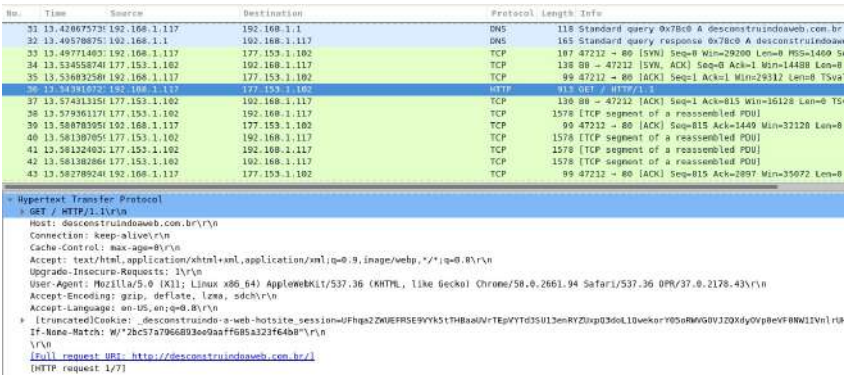


Figura 6.8: Pacotes HTTP em uma rede Wi-Fi pública, claro como o dia

Se alguém conseguir ficar no meio do caminho, ele vai poder ver todas as informações que estamos trafegando. Mas esse não é o pior dos problemas, pois como o Wi-Fi utiliza sinais de rádio, ele distribui todos os dados para todos os dispositivos da rede. Fica a cargo da placa de rede de cada computador decidir se ela é dona do

dado, ou se esse dado é de outra pessoa.

Neste momento é que entra o tal *monitor mode*. Utilizando esse modo, a placa de rede desabilita o filtro que escolhe apenas os pacotes que são direcionados para esse computador, coletando todos os pacotes que recebe.

Como exercício, podemos tentar usar outro dispositivo para acessar alguma outra página, utilizando o mesmo roteador Wi-Fi com conexão pública. Para coletar os dados desse acesso, vamos usar uma ferramenta chamada *aircrack-ng*[17].

O *aircrack-ng* é um pacote que possui um conjunto de ferramentas para fazer análise de segurança em redes Wi-Fi. Os dados capturados por essa ferramenta são compatíveis com o Wireshark, e podemos seguir o mesmo fluxo que estávamos seguindo até agora.

Para fazer o teste, vamos utilizar um celular Android que se conectou a rede com o IP `192.168.1.119` .

Figura 6.9: IP e mac address do telefone usado

Estamos usando um celular Android para esse caso, mas poderia ser qualquer outro dispositivo que possa se conectar a uma rede Wi-Fi.

Vamos acessar qualquer outro conteúdo para ficar fácil de diferenciar quando estivermos olhando os dados capturados. Nesse caso, vamos acessar o blog http://pothix.com escolhido aleatoriamente entre a lista de blogs do autor.

Após parar a captura de dados e analisar os pacotes, podemos ver que o pacote que acessamos com o dispositivo secundário também foi recebido pela nossa placa que está em modo de monitoração.

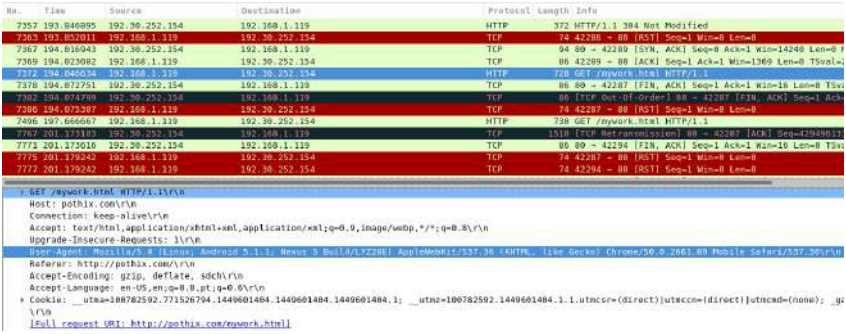


Figura 6.10: Pacotes enviados pelo telefone, acessando o IP 192.30.252.154, podem ser vistos no computador

Por esse motivo, não é recomendado acessar redes Wi-Fi públicas, ainda mais se o site que você está acessando não utilizar HTTPS. Atualmente, é fácil de encontrar aplicações para celular que analisam tráfego em texto puro e fazem coisas como roubar dados de acesso a sites, por exemplo. Todo desenvolvedor web deve conhecer vulnerabilidades como essa para entender um dos principais motivos para se usar HTTPS em sua aplicação.

Como nossos testes acabaram, vamos voltar à configuração do roteador para como ele sempre deveria estar: com senha!

O uso do WPA2 em redes Wi-Fi

O padrão 802.11 especifica alguns padrões de senha, sendo que o mais recomendado atualmente é o *WPA2*. Esse padrão foi especificado na adição **i** do 802.11, conhecida como 802.11i[18], e foi incorporada oficialmente ao protocolo em 2007[19].

Nesta seção, vamos evitar nos aprofundar tanto como fizemos no capítulo sobre o TLS, pois a autenticação não acontece durante a requisição web. Ainda assim, vamos passar pelo básico e deixar as referências para os documentos oficiais, caso haja curiosidade em se aprofundar.

A vantagem do *WPA2* sobre o *WPA* e o *WEP* é o algoritmo de criptografia que ele utiliza. O WPA2 usa o *CCMP*, que significa *CTR with CBC-MAC Protocol*[20] e é parte do AES, que por sua vez, é o mesmo algoritmo de criptografia que mencionamos no capítulo anterior, enquanto estudávamos o TLS.

Esse padrão veio para substituir o antigo padrão WEP, que utiliza *ARC4*[21]. A modificação foi necessária, pois o ARC4 foi quebrado há algum tempo e, atualmente, existem ferramentas que conseguem descriptografar os dados rapidamente. As ferramentas do pacote do `aircrack-ng`, que usamos para capturar pacotes quando estávamos estudando a segurança do Wi-Fi, fazem esse trabalho.

O WPA2 usa um processo de handshake que parece com algumas coisas que vimos no TLS com o 3-way handshake. Ele é conhecido como o *4-way handshake* pelo mesmo motivo que o 3-way handshake é chamado assim, porque possui 4 passos para que os dois concordem que está tudo certo. Uma descrição simplificada do processo do *WPA2 Personal*, uma das opções do WPA2, seria:

1. O roteador (no nosso caso) faz o envio do `ANONCE`, que significa **A**uthenticated **N**umber used **Once**, ou número autenticado para ser usado apenas uma vez. Esse número é gerado aleatoriamente.
2. O cliente envia uma mensagem de `SNOUNCE` para o roteador. Essa mensagem é gerada seguindo o mesmo esquema do `ANOUNCE` que vimos no passo anterior, sendo que a diferença é que o `SNOUNCE` é a versão do cliente. Com o `SNOUNCE` gerado, o cliente gera também uma chave chamada `PTK` (*Pairwise Transient Key*) baseando-se em um conjunto de informações sobre a conexão. Entre eles estão: o `ANOUNCE` enviado pelo roteador, os endereços `MAC` das duas pontas da conexão, uma chave chamada `PMK`, criada baseando-se na

senha do Wi-Fi, e o `SNOUNCE` gerado há pouco. Junto com a mensagem é enviado um código chamado `MIC` (*Message Integrity Code*), usado para garantir a integridade da chave, ou seja, para o cliente verificar que a mensagem não foi modificada no meio do caminho.

3. O roteador recebe o `SNOUNCE` e faz a mesma coisa para gerar a sua versão da chave `PTK`. Ele envia uma chave chamada `GTK` (*Group Transient Key*) que é utilizada para mensagens que são enviadas para várias fontes, também conhecidas como multicast e/ou broadcast, e é compartilhada por todos os clientes da rede. Essa mensagem também vai junto com o código MIC do passo anterior.
4. O cliente confirma o recebimento de todas as chaves e diz que está pronto para trocar dados encriptados.

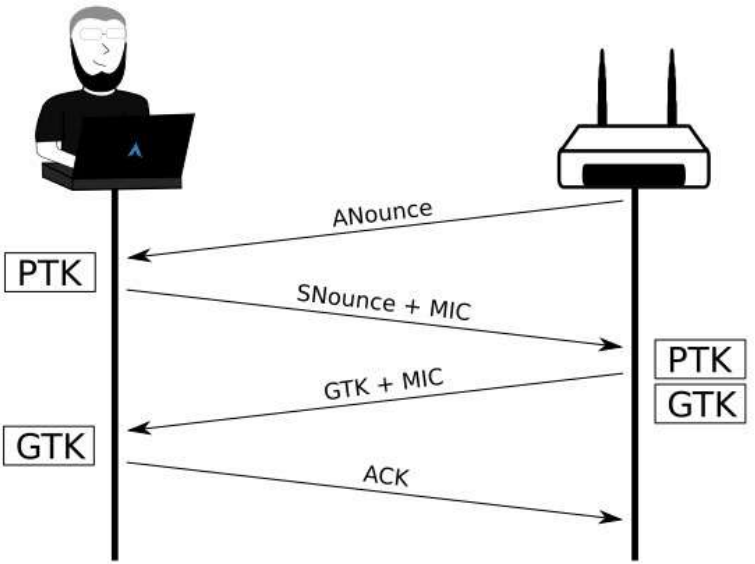


Figura 6.11: Passos de 4-way handshake

A senha que nós temos configurada no nosso roteador Wi-Fi nunca é transitada pela rede. Os dois computadores fazem a

matemática necessária para derivar outras chaves baseadas na senha e verificar a conexão a cada passo.

Se refizermos o mesmo teste da seção anterior, podemos ver que os dados não são mais abertos como antes. Deixando a placa de rede wireless em modo de monitoração, conseguimos capturar pacotes de uma determinada rede, mas estes não estão mais em texto puro. Os pacotes auditados no Wireshark parecem bastante com os pacotes TLS que vimos no capítulo anterior, mas com menos informações.

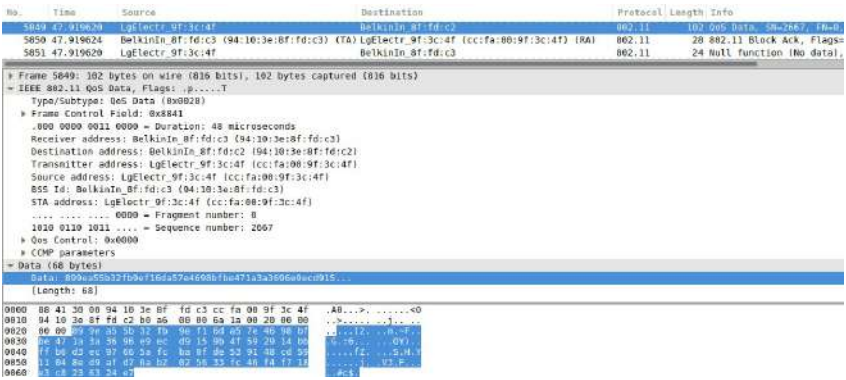


Figura 6.12: Dados de uma conexão feita no celular visto do computador

Muito mais seguro, certo? E isso levaria alguns dos usuários do `desconstruindoaweb.com.br` a perguntar:

```
Quer dizer que, se estou usando WPA2 Personal, não tem problema nã
o utilizar HTTPS?
```

Essa é uma pergunta bem comum. Apesar de o usuário estar bem mais seguro utilizando WPA2 Personal do que usando uma rede Wi-Fi aberta, isso não quer dizer que está totalmente seguro. Como vimos na explicação simples do 4-way handshake, a senha não é transitada em nenhum momento, mas sabemos por experiência de vida que a senha é geralmente disponibilizada para convidados. Se algum desses convidados decidir ficar capturando os

pacotes, ele vai conseguir ver todo o conteúdo em texto puro.

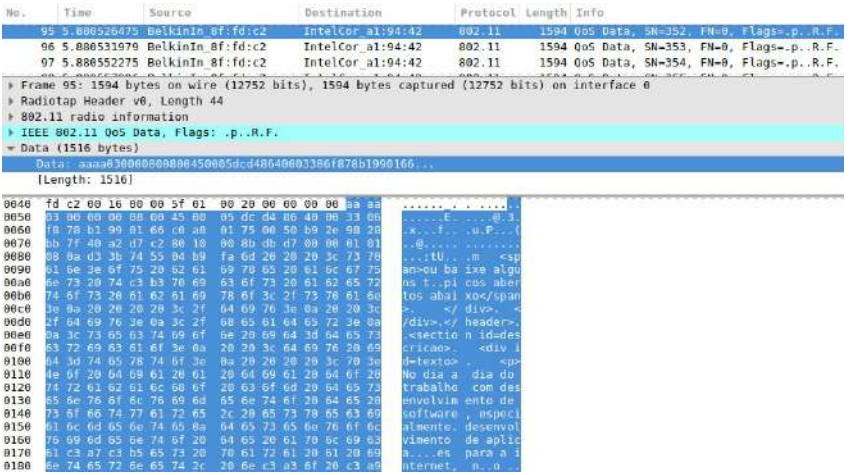


Figura 6.13: Dados dentro de uma conexão WPA2 com HTTP quando temos a senha

Isso reforça ainda mais que não importa qual o Wi-Fi utilizado, utilizar HTTPS para aplicações de internet que trafegam dados sigilosos é muito importante.

6.5 SAINDO PARA A INTERNET

Finalmente nosso estudo saiu da rede local e estamos prontos para desbravar a internet!

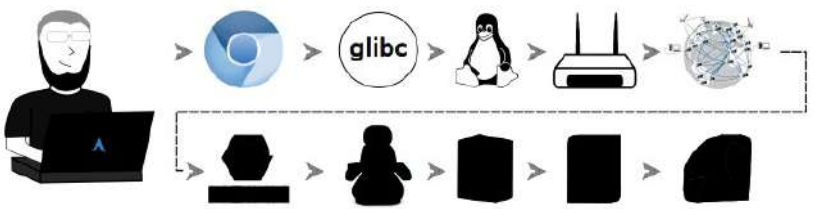


Figura 6.14: A jornada para a internet

A internet é um conjunto de redes conectadas, por isso o nome

inter - **net**. Essas várias redes são conectadas por incontáveis dispositivos de rede que distribuem e encaminham os pacotes para o lugar certo. Os computadores de uma rede estão conectados a um dispositivo de rede, geralmente um switch ou roteador. Esse dispositivo de rede pode se conectar a outro dispositivo, que pode estar conectado a outro, e assim sucessivamente.

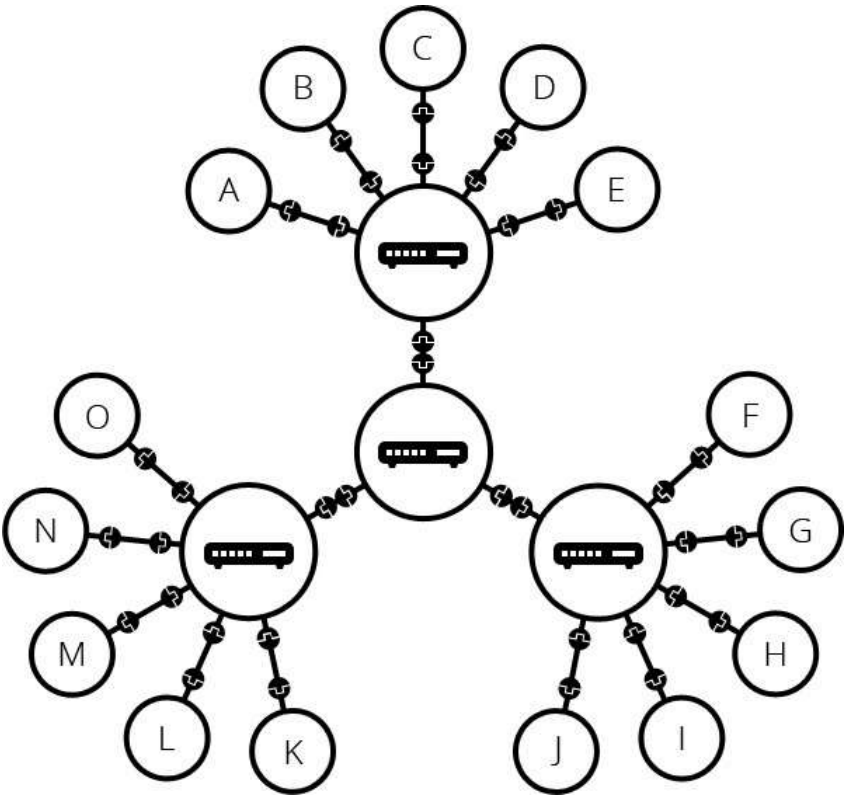


Figura 6.15: Conexão entre roteadores na internet. Fonte: http://cege.la/PE7Skr, feito pelos Contribuidores da Mozilla, licenciado sobre CC-BY-SA 2.5

Mesmo com essa possibilidade de conexão entre os dispositivos de rede, ainda é necessário ter uma forma que leve os dados de um lado para o outro do mundo. Com alguma infraestrutura física dessa, podemos acessar servidores em qualquer lugar do mundo, como o `desconstruindoaweb.com.br` , por exemplo. Atualmente,

já possuímos uma infraestrutura para isso, que foi implementada para o sistema de telefonia.

Para utilizar a infraestrutura do sistema de telefonia, precisamos de um aparelho chamado modem. Este faz a conversão dos dados digitais de internet para um sinal que possa viajar pelos sinais dessa infraestrutura. O modem geralmente é fornecido por uma empresa que chamamos de ISP, que é a sigla para **I**nternet **S**ervice **P**provider, ou provedor de serviço de internet. Essa empresa é a responsável por nos conectar com os vários outros ISPs ao redor do mundo. A NET, Embratel e a GVT são algumas dos ISPs brasileiros, e elas vão fazer parte da nossa jornada na próxima seção.

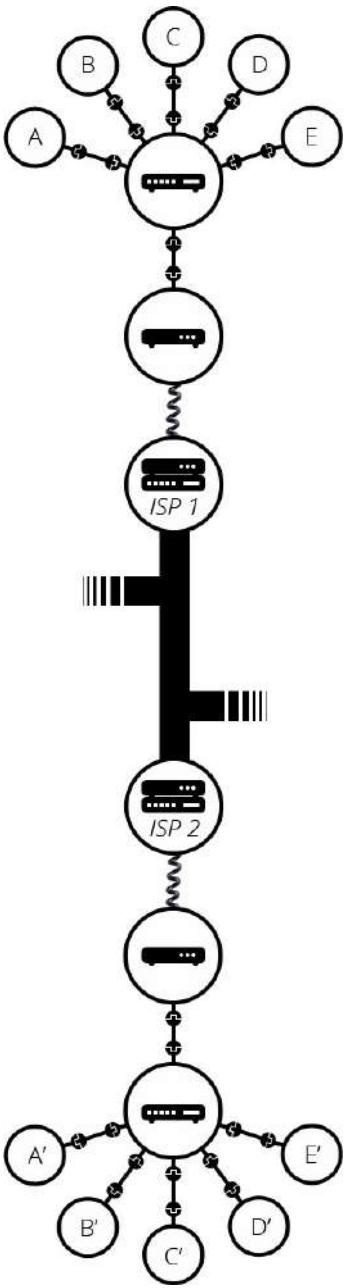


Figura 6.16: Os ISPs e a conexão entre a rede local e o servidor de destino. Fonte: http://cege.la/PE7Skr, feito pelos Contribuidores da Mozilla, licenciado sobre CC-BY-SA 2.5

Traçando a rota

Fora de rede local, existe muita coisa que não conseguimos ter controle algum. Vamos ter de assumir que funcionam e vão nos levar até onde queremos. Para ter uma ideia do que está entre o nosso roteador e o servidor de destino, nós podemos usar o comando `traceroute`. O que esse comando faz é usar o que podemos chamar de *uma marotagem* para descobrir quem está no caminho entre ele e o servidor de destino.

A marotagem está em usar o **T**ime **T**o **L**ive, ou **TTL**, dos pacotes para conseguir informações sobre cada um dos equipamentos no caminho. Um exemplo dessa ideia:

- O cliente envia um pacote com TTL 1 para o endereço do `desconstruindoaweb.com.br`.
- O roteador recebe o pacote, subtrai 1 e percebe que o TTL está 0, portanto avisa quem enviou para enviar novamente, porque o pacote *morreu*.
- Quando o cliente recebe o aviso de que o pacote *morreu*, ele consegue as informações de **onde** ele morreu. Essa informação é extraída baseada no número do TTL utilizado. Se o TTL era 1, ele sabe que esse equipamento é o primeiro com que ele tem contato.
- O cliente então envia um pacote com TTL 2 para o endereço do `desconstruindoaweb.com.br`.
- O roteador recebe o pacote, subtrai 1 e encaminha para o próximo equipamento em rota.
- Esse equipamento recebe o pacote, subtrai 1 e percebe que o TTL está 0..., e assim sucessivamente.

Vamos usar essa ideia esperta para descobrir o que há entre nós e nosso destino! Mas primeiro vamos aplicar algumas opções sobre o `traceroute` para que nosso estudo fique mais fácil. O primeiro

argumento que vamos usar é o `-T` para usar TCP em vez de UDP, que é o padrão do `traceroute`. Isso porque, em algum lugar, os pacotes UDP são bloqueados, portanto nunca chegamos ao final. O segundo argumento será o `-q` e passaremos o valor `1`, que informa ao `traceroute` para fazer apenas uma tentativa para cada um dos *hops*.

Um **hop** é um intermediário em uma comunicação de rede. Ou seja, cada um dos pontos que vamos ver no retorno do comando `traceroute` é um hop. Com isso, temos o comando: `traceroute -q 1 -T desconstruindoaweb.com.br`. Vamos executá-lo e ver a quantidade de hops que há entre nosso computador de estudo e o servidor de destino.

```
$ traceroute -q 1 -T desconstruindoaweb.com.br

traceroute to desconstruindoaweb.com.br (177.153.1.102), 30 hops m
ax, 60 byte packets
 1  palantir (192.168.1.1)  6.239 ms
 2  10.18.128.1 (10.18.128.1)  18.601 ms
 3  c8bd5001.virtua.com.br (200.189.80.1)  19.289 ms
 4  embratel-T0-7-3-0-tacc01.spoph.embratel.net.br (200.178.127.29
)  22.098 ms
 5  ebt-T0-5-0-9-tcore01.spo.embratel.net.br (200.230.158.82)  30.
642 ms
 6  ebt-B10-tcore01.spo.embratel.net.br (200.230.0.2)  37.803 ms
 7  ebt-B21108-tcore01.rjo.embratel.net.br (200.230.251.209)  39.4
99 ms
 8  ebt-B10-tcore01.rjoen.embratel.net.br (200.230.252.157)  35.73
0 ms
 9  ebt-B10-tcore01.rjoen.embratel.net.br (200.230.252.157)  35.72
6 ms
10  peer-T0-4-0-1-puacc01.rjoen.embratel.net.br (200.211.219.74)
34.473 ms
11  gvt-te-0-6-0-7.rc02.rjo.gvt.net.br (177.205.9.31)  35.694 ms
12  gvt-te-0-5-0-12.rc02.spo.gvt.net.br (177.99.249.230)  41.668 m
s
13  locaweb.static.gvt.net.br (177.135.153.234)  35.648 ms
14  186.202.158.5 (186.202.158.5)  39.474 ms
15  186.202.158.6 (186.202.158.6)  40.475 ms
16  dist-aita20-a.noc.locaweb.com.br (186.202.191.88)  39.462 ms
17  177.153.1.102 (177.153.1.102)  35.057 ms
```

Vamos usar a primeira linha e fazer um estudo de cada um dos valores que estão lá para entender o padrão do conteúdo devolvido pelo `traceroute`:

```
1  palantir (192.168.1.1)  6.239 ms
```

- **1** — TTL que foi usado.
- **palantir** — *Hostname* do equipamento em questão.
- **192.168.1.1** — IP que foi utilizado para chegar ao equipamento.
- **6.239 ms** — Tempo em milissegundos da requisição feita para o IP do equipamento. Quando o argumento `-q` não é informado, ele faz três tentativas para cada hop e coloca o tempo das três tentativas um após o outro.

Em alguns casos, cada uma das tentativas utiliza um caminho diferente, e ele coloca cada um dos equipamentos que foram encontrados e o tempo que demorou para chegar até eles, gerando bastante informação.

Antes de começarmos a analisar a trajetória que nossa requisição tomou, vamos assumir que nesse momento estamos em São Paulo, pois a localização física tem muito a dizer sobre a trajetória dos pacotes.

Vamos pegar apenas o *hostname* e IP do hop para análise, e dividir e conquistar!

1. `palantir (192.168.1.1)` — Meu roteador da rede local.
2. `10.18.128.1 (10.18.128.1)` — Como moro em um prédio, esse é provavelmente um aparelho da minha operadora de internet para distribuir o sinal dentro do condomínio. É possível que esse *hop* (nome usado para um ponto na rede) não exista caso você more em uma casa.
3. `c8bd5001.virtua.com.br (200.189.80.1)` — Esse é o

primeiro dispositivo de rede externo que estamos passando. É o dispositivo do meu **ISP** atual, a NET Virtua.

4. `embratel-T0-7-3-0-tacc01.spoph.embratel.net.br (200.178.127.29)`
5. `ebt-T0-5-0-9-tcore01.spo.embratel.net.br (200.230.158.82)`
6. `ebt-B10-tcore01.spo.embratel.net.br (200.230.0.2)` — Esses três são aparelhos da Embratel, que é outro ISP, em São Paulo.
7. `ebt-B21108-tcore01.rjo.embratel.net.br (200.230.251.209)`
8. `ebt-B10-tcore01.rjoen.embratel.net.br (200.230.252.157)`
9. `ebt-B10-tcore01.rjoen.embratel.net.br (200.230.252.157)`
10. `peer-T0-4-0-1-puacc01.rjoen.embratel.net.br (200.211.219.74)` — Agora a requisição foi para aparelhos da Embratel no Rio de Janeiro, passou por três aparelhos distintos e teve que fazer duas chamadas para um deles.
11. `gvt-te-0-6-0-7.rc02.rjo.gvt.net.br (177.205.9.31)` — Agora saímos da embratel e seguimos para outro ISP, que é a *GVT*, mas ainda no Rio de Janeiro.
12. `gvt-te-0-5-0-12.rc02.spo.gvt.net.br (177.99.249.230)` — Agora voltamos para São Paulo pela *GVT*.
13. `locaweb.static.gvt.net.br (177.135.153.234)` — Chegando ao link da *Locaweb* pela GVT.
14. `186.202.158.5 (186.202.158.5)`
15. `186.202.158.6 (186.202.158.6)` — Possivelmente dois switches de borda da infraestrutura interna da Locaweb.
16. `dist-aita20-a.noc.locaweb.com.br (186.202.191.88)` — Pelo nome `dist`, podemos assumir que esse é um switch de distribuição da infraestrutura da Locaweb, mas isso é

apenas uma suposição, pois pode ser outro dispositivo que não conseguimos identificar facilmente olhando de fora.

17. `177.153.1.102 (177.153.1.102)` — Esse é o servidor do Jelastic[22], onde o `desconstruindoaweb.com.br` está hospedado!

> **Resultados diferentes do traceroute**
>
> Dificilmente será possível conseguir os mesmos resultados caso esse comando seja executado novamente. Isso pode acontecer devido à localização física da conexão, se estiver sendo executada em uma cidade diferente, por exemplo. Mas não se restringe apenas a isso.
>
> A Locaweb pode alterar a forma como entrega a página para a internet sem que haja qualquer problema para o usuário final, mas que poderíamos ver no traceroute. Um exemplo desse tipo de modificação seria uma alteração no link de internet usada por eles.

Agora que finalmente chegamos ao servidor que nossa aplicação está hospedada, podemos estudar o que nos espera lá dentro. No próximo capítulo, veremos como os servidores web lidam com as requisições.

6.6 RESUMO

Depois de muito estudo, chegou o momento de sair para a internet. Em vez de sair via Ethernet, ou rede cabeada, vamos utilizar Wi-Fi. Nesse estudo, nós estamos usando uma placa da Intel que utiliza um driver chamado `iwlwifi` . Para que o sinal seja

enviado para o roteador, ele vai precisar passar por várias partes dentro do kernel, como: `socket`, `af_inet`, `tcp`, `ip`, driver da placa e firmware da placa.

Considerando que agora a informação vai efetivamente sair do computador local, ele pode trafegar tanto por uma rede pública e insegura como por uma rede segura configurada com WPA2, por exemplo. O 4-way handshake do WPA2 já foi feito no momento em que o cliente se conectou à rede. Logo, a conexão está estabelecida e pronta para que os dados sejam enviados de forma segura via ondas de rádio.

Isso faz com que outros computadores não consigam decifrar os dados que estão trafegando sem que eles tenham acesso a senha da rede sem fio. A conexão vai trafegar via ondas de rádio brigando com vários aparelhos que ficam na mesma frequência até chegar ao roteador que enviará os pacotes para a internet.

A requisição sai da rede local e atravessa dois estados brasileiros e 17 hops, ou saltos entre dispositivos de rede, até chegar a um dos computadores do Jelastic da Locaweb, onde está hospedado.

6.7 REFERÊNCIAS

1. Especificação da Ethernet — http://standards.ieee.org/about/get/802/802.3.html
2. `ethtool` no kernel — https://www.kernel.org/pub/software/network/ethtool/
3. Especificação do Wi-Fi (802.11) — http://standards.ieee.org/about/get/802/802.11.html
4. Especificação do 802.11ac — http://standards.ieee.org/getieee802/download/802.11ac-

2013.pdf

5. Referência da placa wireless no site da Intel — http://www.intel.com/content/www/us/en/wireless-products/dual-band-wireless-ac-7260-bluetooth.html

6. Paper sobre a saída de um pacote do kernel — http://www.hsnlab.hu/twiki/pub/Targyak/Mar11Cikkek/Network_stack.pdf

7. Código do `socket` no kernel — https://github.com/torvalds/linux/blob/master/net/socket.c

8. Código do `AF_INET` no kernel — https://github.com/torvalds/linux/blob/master/net/ipv4/af_inet.c

9. Código do `TCP` no kernel — https://github.com/torvalds/linux/blob/master/net/ipv4/tcp.c

10. Código do `IP` no kernel — https://github.com/torvalds/linux/blob/master/net/ipv4/ip_output.c

11. Código do driver da placa wireless `iwlwifi` no kernel — https://github.com/torvalds/linux/tree/master/drivers/net/wireless/intel/iwlwifi

12. *RFC826* e a definição do protocolo ARP — https://tools.ietf.org/html/rfc826

13. White paper da Cisco sobre interferência em Wi-Fi — http://www.cisco.com/c/en/us/products/collateral/wireless/spectrum-expert-wi-fi/prod_white_paper0900aecd807395a9.html

14. Paper sobre latência no uso de CSMA/CA —

http://www.iestcfa.org/bestpaper/etfa08/FH_ET.pdf

15. Paper sobre o uso de wireless `g` e `b` em conjunto — http://www.cise.ufl.edu/~helmy/papers/Shao-Cheng-IPCCC-published.pdf

16. Modo de captura para wireless no Wireshark — https://wiki.wireshark.org/CaptureSetup/WLAN

17. Site do `aircrack-ng` — https://www.aircrack-ng.org/

18. Implementação do WPA2 no 802.11 — http://standards.ieee.org/getieee802/download/802.11i-2004.pdf

19. Documento da incorporação do WPA2 no 802.11 — https://standards.ieee.org/findstds/standard/802.11-2007.html

20. Seção **11.4.3** sobre `CTR_with_CBC-MAC_Protocol` — http://standards.ieee.org/getieee802/download/802.11-2012.pdf

21. Seção **4.5.4.4** sobre `Data_confidentiality` — http://standards.ieee.org/getieee802/download/802.11-2012.pdf

22. Jelastic da Locaweb — http://www.locaweb.com.br/cloud/jelastic/

Capítulo 7

SERVIDOR WEB

Após trafegar por toda a internet desde sair do nosso roteador Wi-Fi, os pacotes começam a chegar ao servidor de destino. Como vimos em capítulos anteriores, há um servidor web aguardando essa conexão. Neste capítulo, vamos entender como a requisição passa por ele e é entregue para o nosso framework de estudo.

Primeiramente, estudaremos o que quer dizer a palavra *servidor web*, usada para dois casos diferentes aqui no Brasil. Depois veremos o que é um servidor de aplicação, para que serve e qual a sua relação com o servidor web.

7.1 O SERVIDOR FÍSICO

No primeiro uso da palavra *servidor web*, vamos nos referir ao computador disponível na infraestrutura de hospedagem que o `desconstruindoaweb.com.br` está hospedado. Nesse estudo, estamos usando um produto vendido pela Locaweb[1] chamado Jelastic.

O Jelastic é um produto de PaaS (**P**latform **a**s **a** **S**ervice), portanto, ele provê a plataforma pré-provisionada para que o desenvolvedor não tenha tanto trabalho para configurar o servidor da aplicação. A empresa do Jelastic faz parcerias com empresas do mundo para prover sua plataforma em várias localizações, e a Locaweb é uma dessas empresas[2].

Outras formas de hospedar a aplicação

Apesar de hospedarmos no Jelastic, há vários outros PaaS disponíveis no mercado. Um deles é o Heroku (https://heroku.com), que provê um serviço de criação de aplicações bem simples via linha de comando. Eles ficaram bem famosos na comunidade Ruby pela facilidade de publicar uma aplicação Rails, além de ter uma versão gratuita de hospedagem.

Poderíamos seguir também utilizando uma versão de IaaS, ou infraestrutura como serviço (do inglês, **I**nfrastructure **a**s **a** **S**ervice), na qual receberíamos um servidor básico e teríamos de configurá-lo para atender as necessidades da nossa aplicação. Entre os produtos que oferecem essa modalidade, temos o serviço EC2 da Amazon, servidores virtuais da Linode ou até o serviço de cloud da própria Locaweb.

Como vamos utilizar o produto via Locaweb, podemos assumir que nosso servidor estará dentro da infraestrutura de datacenter deles. Apesar de já termos visto parte da rede de uma forma prática no capítulo anterior via `traceroute`, não comentamos sobre o datacenter.

Figura 7.1: Datacenter da Locaweb. Fonte: http://www.tecmundo.com.br/servidor/41528-fortaleza-tecnologica-visitamos-o-datacenter-da-locaweb.htm

Relembrando uma parte do nosso `traceroute` do capítulo anterior, temos:

```
14  186.202.158.5 (186.202.158.5)  39.474 ms
15  186.202.158.6 (186.202.158.6)  40.475 ms
16  dist-aita20-a.noc.locaweb.com.br (186.202.191.88)  39.462 ms
17  177.153.1.102 (177.153.1.102)  35.057 ms
```

Por algum tempo, vamos estar em uma área especulativa, ou seja, não vamos saber exatamente como a infraestrutura de redes funciona dentro do datacenter da Locaweb. Então, vamos supor algumas informações.

Esses são os aparelhos que parecem estar dentro do datacenter da Locaweb, pois aparecem justamente após sair dos domínios da GVT. É muito difícil saber o que são cada um deles sem algum documento oficial da empresa, mas podemos supor que a Locaweb segue alguns padrões de estrutura de rede e utilizar os dados que temos para esse estudo.

O número **14** é provavelmente um daqueles grandes switches, chamados de *switch core*, feitos para estar na frente de todo o tráfego e aguentar uma quantidade absurda de dados que vem da internet, por exemplo. O número **15** se parece bastante com o primeiro, mas possivelmente é um switch de distribuição que recebe do core e distribui para outros switches abaixo dele.

Agora chegamos a um que possui um domínio, o número **16**. Por ter um domínio, podemos pensar que ele é um servidor de firewall ou até mesmo um servidor agindo como um aparelho de rede com *Software Defined Network*, implementando regras de switch. Só o fato de ele ter um domínio não diz muita coisa, pois eles podem criar domínios para qualquer aparelho sem gerar qualquer problema, portanto ainda estamos no modo de especulação.

Ao sair do datacenter, saímos também das especulações e voltamos aos estudos com dados concretos

Depois de passar por todos estes equipamentos de rede, o pacote vai chegar ao número 17, que é o servidor do `desconstruindoaweb.com.br` . Esse servidor não é um computador comum como estamos acostumados a ver por aí. Ele é feito exclusivamente para ser montado em um *rack*, que é uma estrutura feita para acomodar servidores da melhor forma possível em um datacenter.

Figura 7.2: Servidores montados no rack dentro do datacenter da Locaweb. Fonte: http://www.tecmundo.com.br/servidor/41528-fortaleza-tecnologica-visitamos-o-datacenter-da-locaweb.htm

Esse servidor está usando a infraestrutura de virtualização do Jelastic, que por sua vez usa um software chamado Virtuozzo[3] para gerenciamento de servidores virtuais e *containers*.

Tanto containers como servidores virtuais são formas de utilizar um servidor físico como se fosse vários servidores separados. A aplicação reconhece o ambiente como um servidor único, mas na verdade está isolada do hardware de verdade por uma camada de software que limita os recursos.

Servidor virtual

O conceito de virtualização é quando um sistema chamado *hypervisor* trabalha entre o sistema operacional e o hardware físico. O trabalho do hypervisor é criar servidores virtuais cujo sistema operacional se comporte como se estivesse sendo executado sobre um servidor físico.

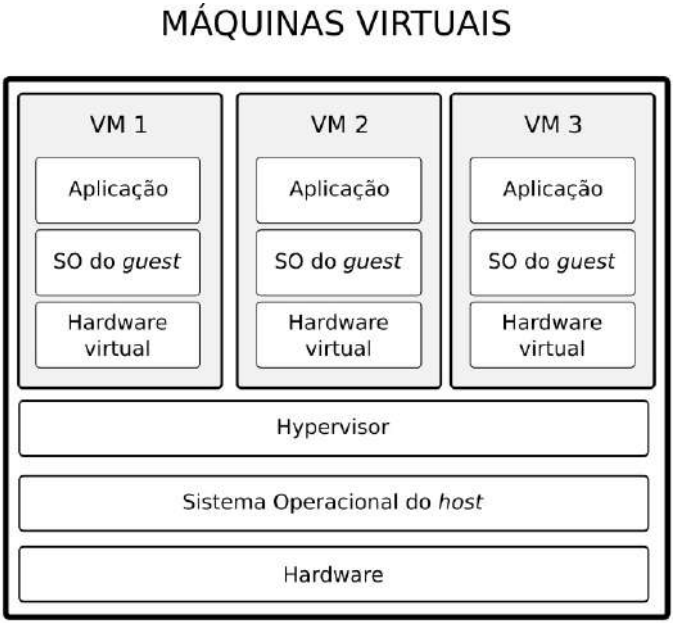


Figura 7.3: A arquitetura de servidores virtuais

Como exemplo para ilustrar a virtualização, vamos imaginar um servidor físico de 16 CPUs e 16 GB de memória rodando XenServer[4] hypervisor. Dentro desse servidor, teremos 10 servidores virtuais, sendo que cada um deles será criado com 1 GB de memória e 1 **v**irtual **CPU**, ou vCPU. Nesse caso, teremos 10 servidores executando dentro de um único hardware, sendo que cada um deles acha que possui apenas 1 GB de memória e 1 CPU disponível para serem utilizados.

O servidor virtual é conhecido como *guest*, e o servidor que está provendo o hardware para os servidores virtuais é conhecido como *host*. O sistema operacional que será executado no *guest* vai assumir que está sozinho em um servidor físico, portanto, tudo vai funcionar como se isso estivesse acontecendo. O hypervisor fará todo o trabalho necessário para traduzir as chamadas de sistemas feitas para o hardware virtual, tratando-as e enviando para o hardware

físico da melhor forma possível.

Container

Após o lançamento do Docker[5], os containers ficaram famosos. A ideia de container não é algo novo, eles existem desde o século passado. A diferença dessa vez é que o Docker conseguiu juntar as várias tecnologias usadas para construir um container em uma única ferramenta, transformando o que ainda era obscuro em algo acessível para todos.

A premissa básica do container é utilizar o mesmo *kernel* do sistema operacional, apenas isolando os processos e o sistema de arquivos.

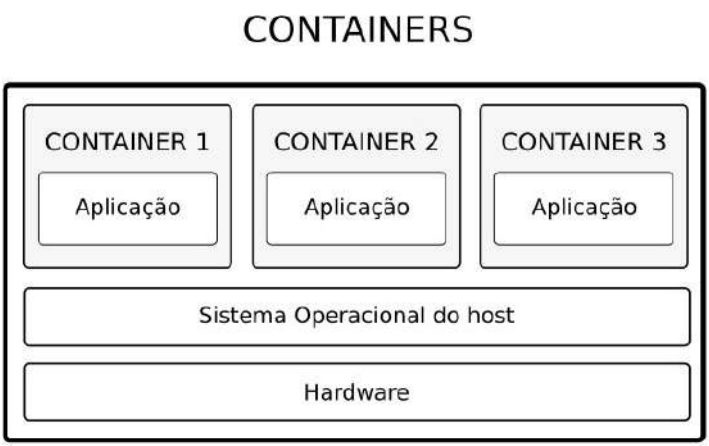


Figura 7.4: A arquitetura de containers

Como é possível utilizar um sistema de arquivos isolado dentro de um container, nós podemos até mesmo usar uma outra distribuição GNU/Linux. Podemos, por exemplo, utilizar um container com uma imagem de CentOS Linux dentro de um *host* Debian Linux, por exemplo.

As novas versões do *kernel* do Linux suportam vários mecanismos para isolamento de processos, rede e outras coisas. Isso possibilita a sistemas como o Docker criar processos que são executados no servidor *host* com acessos limitados.

Utilizando essas técnicas, é possível ter um servidor rodando Debian Linux e provisionar vários containers com CentOS, executando o gerenciador de pacotes `yum` e instalando softwares específicos dele dentro de um sistema de arquivos isolado. Bem legal, não?

Para o mundo fora do servidor, esses containers não existem, eles existirão apenas dentro do servidor hospedeiro, ou *host*. O servidor hospedeiro trata os recursos do container como se fossem dele. O Jelastic usa containers para provisionar seus *cloudlets*, portanto nosso estudo continua seguindo essa vertente.

7.2 O SOFTWARE

Nesse contexto, o *servidor web* é o software responsável por receber a requisição que chegou ao sistema operacional e processar para entregar para a próxima camada. Essa próxima camada pode ser a aplicação ou outro software, dependendo da configuração.

Quando dissemos *"receber a requisição que chegou ao sistema operacional"*, podemos assumir que o computador vai receber as informações na placa de rede, passar para o kernel do sistema operacional, tratar os pacotes, reconhecer os protocolos etc. Esse processo é algo parecido com uma versão reversa do que estudamos sobre a saída de pacotes do sistema operacional, no capítulo anterior.

A partir desse momento, vamos nos referir a servidor web como sendo o software que recebe a requisição do sistema operacional,

assim evitamos confusões. A principal função desse software é ficar aguardando conexões, analisar o conteúdo da requisição HTTP que chegar, e entregar o conteúdo solicitado da melhor maneira possível.

Os servidores web podem ser usados para responder diretamente para o cliente que está requisitando a conexão, ou encaminhar a conexão para uma segunda camada. No segundo caso, o servidor web está agindo somente como um *proxy reverso*. Nesse modo, ele fica exposto para a internet e encaminha as requisições para a camada de baixo. Os processos do *servidor de aplicação*, que veremos ainda neste capítulo, são um exemplo do que pode estar nessa camada de baixo.

Existem vários servidores web gratuitos, livres e pagos. Os mais utilizados na data de escrita deste livro, de acordo com a w3techs[6], são:

- Apache httpd (https://httpd.apache.org/)
- NGINX (https://nginx.org/en/)
- IIS (http://www.iis.net/)

Como estamos em um ambiente GNU/Linux, podemos descartar o uso do IIS. Usando o sistema PaaS do Jelastic, é possível escolher entre *Apache httpd*, conhecido apenas como **Apache**, e *NGINX*. Para esse nosso estudo, usaremos o NGINX para servir o site do `desconstruindoaweb.com.br` .

Figura 7.5: Escolha do servidor web no Jelastic

7.3 NGINX

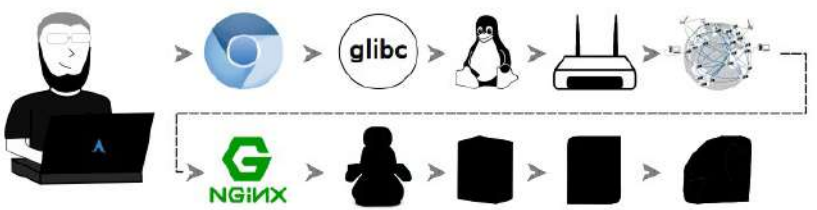


Figura 7.6: Chegando ao NGINX

O NGINX, que se pronuncia *engine X*, é um servidor web criado em 2004 para competir com o Apache. A motivação para a sua criação foi otimizar a quantidade de memória e CPU gastas para servir páginas web. Ele nasceu pouco depois da publicação do C10K[7], um *manifesto* que diz que *já está na hora dos servidores*

web conseguirem gerenciar dez mil requisições simultâneas, publicado por volta de 2003.

Nessa época, o Apache utilizava apenas o modelo de fork/threads para tratar as requisições recebidas, o que gera um gasto maior de memória e processamento. O NGINX veio com a ideia de usar uma quantidade pré-determinada de *workers* e o conceito *eventos* com processamento assíncrono e não bloqueante.

OUTRAS FORMAS DE PROCESSAR REQUISIÇÕES

Comentamos sobre *forks* e *threads*, mas não demos muitos detalhes sobre eles para focarmos no modelo de eventos. Entretanto, há vários outros modelos para lidar com as requisições. Aqui vão duas formas diferentes de processamento de requisições web que podem ser utilizadas ao usarmos o Apache:

- `prefork` — Carrega um processo para cada requisição, limitando a quantidade mínima e máxima de processos disponíveis. Cada processo vai atender uma requisição por vez, sem utilizar threads. Assim, é necessário configurar uma quantidade mínima e máxima razoável para a quantidade de requisições esperada.
- `multi-processing` — Cria uma quantidade de processos definida em configuração, e cada um deles vai possuir várias threads que serão responsáveis por atender as requisições. Com isso, cada processo poderá trabalhar em várias requisições ao mesmo tempo.

A arquitetura do NGINX

Ao executar o NGINX, ele criará um processo *master*, que não vai lidar com as requisições, mas será responsável por várias atividades secundárias. Entre essas atividades estão: ler o arquivo de configuração, lidar com arquivos de log, lidar com upgrades sem *downtime*, criar os processos *worker* e mantê-los, garantindo que estejam vivos e prontos para lidar com as novas conexões.

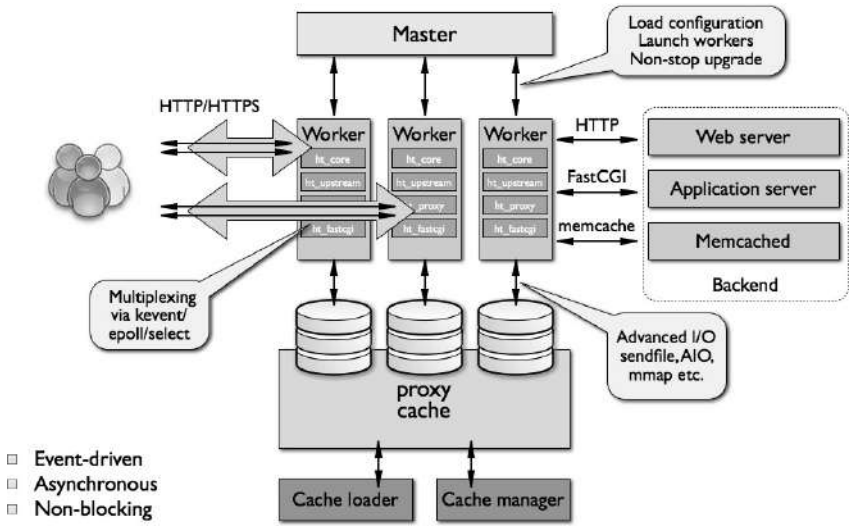


Figura 7.7: A arquitetura do NGINX. Fonte: http://www.aosabook.org/en/nginx.html, por Andrew Alexeev, disponível sob licença CC 3.0 Unported

Os processos *worker* criados pelo master são responsáveis por fazer o processamento das requisições que chegarem. Para ter uma ideia melhor de como tudo isso funciona, vamos executar o NGINX com o comando `strace`:

```
$ strace -f nginx
```

O argumento `-f` é para seguir também os processos filhos gerados via `fork` ou `clone`, que são syscalls para criação de processos. Após executar esse comando, podemos ver que temos

dois processos do NGINX sendo executados:

```
$ ps fexa -o pid,cmd | grep "[n]ginx:"

19897 nginx: master process /usr/sbin/nginx -c /etc/nginx/nginx.co
nf
19899  \_ nginx: worker process
```

A parte que importa desse comando são os PIDs **13205** e **13206**, referentes aos processos master e worker, respectivamente. Eles serão úteis para identificar os processos na saída do comando `strace`.

Após executar o `strace`, ele começará a imprimir na tela as syscalls que os processos estão executando. Como o resultado do comando `strace` é grande e não caberia bem como conteúdo para o livro, vamos usar apenas as partes importantes para fazer a análise.

O processo principal vai carregar todas as dependências e conseguir informações sobre o sistema operacional. Logo depois, ele lerá a configuração e usará a porta definida lá para fazer o `bind`. Para o nosso caso, a porta configurada foi a 80:

```
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 4
setsockopt(4, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
ioctl(4, FIONBIO, [1])                  = 0
bind(4, {sa_family=AF_INET, sin_port=htons(80), sin_addr=inet_addr
("0.0.0.0")}, 16) = 0
listen(4, 511)
```

Logo em seguida, ele cria o nosso processo de master utilizando a syscall `clone`, e o `strace` começa a seguir esse novo processo também:

```
clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID
|SIGCHLD, child_tidptr=0x7f5d04ca09d0) = 13205
strace: Process 13205 attached
```

Após criar o processo master, ele segue para os workers. No nosso caso, será apenas um único processo:

```
clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID
|SIGCHLD, child_tidptr=0x7f5d04ca09d0) = 13206
strace: Process 13206 attached
```

Daqui para a frente, é possível ver pelo PID, no começo da linha, que boa parte do trabalho em uma conexão é feita pelo worker, que está se preparando para receber os eventos das conexões:

```
[pid 13206] epoll_create(512)           = 7
[pid 13206] eventfd2(0, 0)              = 8
[pid 13206] epoll_ctl(7, EPOLL_CTL_ADD, 8, {EPOLLIN|EPOLLET, {u32=
4173138560, u64=94386079468160}}) = 0
[pid 13206] eventfd2(0, 0)              = 9
[pid 13206] ioctl(9, FIONBIO, [1])      = 0
[pid 13206] io_setup(32, [140037489303552]) = 0
[pid 13206] epoll_ctl(7, EPOLL_CTL_ADD, 9, {EPOLLIN|EPOLLET, {u32=
4173138208, u64=94386079467808}}) = 0
[pid 13206] mmap(NULL, 241664, PROT_READ|PROT_WRITE, MAP_PRIVATE|M
AP_ANONYMOUS, -1, 0) = 0x7f5d04ca7000
[pid 13206] brk(0x55d7fa4b9000)         = 0x55d7fa4b9000
[pid 13206] epoll_ctl(7, EPOLL_CTL_ADD, 4, {EPOLLIN|EPOLLRDHUP, {u
32=80375824, u64=140037489061904}}) = 0
[pid 13206] close(5)                    = 0
[pid 13206] epoll_ctl(7, EPOLL_CTL_ADD, 6, {EPOLLIN|EPOLLRDHUP, {u
32=80376056, u64=140037489062136}}) = 0
[pid 13206] epoll_wait(7,
```

Aqui está acontecendo a preparação para o *event loop*. Vamos estudar como isso acontece olhando cada uma dessas syscalls.

O event loop

O NGINX suporta vários métodos de processamento assíncrono, que depende do suporte do sistema operacional que ele está utilizando. De acordo com a documentação dos métodos de processamento de conexões do NGINX[8], nós poderíamos usar `select`, `poll` ou `epoll`. Cada uma dessas syscalls é utilizada para lidar com eventos de uma forma diferente, e cada uma tem a sua limitação.

Como é possível ver nas chamadas de sistema do final da seção

anterior, o worker está utilizando `epoll`. Isso acontece porque esse método é o mais eficiente para a tarefa quando se tem um kernel maior que 2.6. Usando o comando `uname`, podemos ver que o nosso kernel é maior que 2.6:

```
$ uname -r

2.6.32-042stab106.6
```

O `epoll` provê uma forma de lidar com eventos, em que um processo pode se ligar a um *file descriptor* e ser notificado sempre que há alguma novidade nele. A parte importante de quem usa esse modelo de eventos é não deixar que algo bloqueie o processo, pois, enquanto ele estiver bloqueado, ninguém mais receberá eventos.

Sabendo o que é o tal do `epoll`, vamos ver como o NGINX está preparando o *event loop* na prática, analisando os passos importantes do resultado do `strace` que vimos há pouco. A primeira coisa é iniciar um `epoll`, que é feito via syscall `epoll_create`:

```
[pid 13206] epoll_create(512) = 7
```

O retorno desse comando foi um *file descriptor* que vai ser referenciado pelo número **7**. Agora ele cria um objeto de `eventfd`, específico para eventos, e adiciona no `epoll` 7 criado há pouco.

```
[pid 13206] eventfd2(0, 0)              = 8
[pid 13206] epoll_ctl(7, EPOLL_CTL_ADD, 8, {EPOLLIN|EPOLLET, {u32=
4173138560,u64=94386079468160}}) = 0
```

Para finalmente executar um `epoll_wait`:

```
[pid 13206] epoll_wait(7,
```

Podemos ver que o `epoll_wait` está pela metade, pois nesse momento ele está dentro do *event loop* aguardando que haja alguma novidade no `epoll 7` que ele está monitorando. No momento de escrita deste livro, o site `desconstruindoweb.com.br` não foi

divulgado e não tem muitos acessos, portanto podemos ver a syscall aguardando novas conexões.

Recebendo a requisição do cliente

O `epoll_wait` continua aguardando que alguma conexão chegue ao NGINX, até que nós decidimos ir até o nosso navegador e digitar `https://desconstruindoaweb.com.br`.

Logo após apertar a tecla `Enter`, podemos ver o comando `strace` imprimindo mais syscalls no terminal. Novamente é muita coisa, mas vamos filtrar e olhar as mais importantes.

O NGINX verifica se é um arquivo estático (como um arquivo JavaScript ou CSS, que ele pode servir diretamente), e o serve caso seja. Caso seja um conteúdo dinâmico — ou seja, um arquivo que ele não possui pronto para entrega —, o NGINX encaminha para um *unix socket* para que um servidor de aplicação possa processar o request e construir uma resposta adequada.

Entraram dois conceitos novos na última frase: **unix socket** e **servidor de aplicação**. Vamos estudá-los com mais calma.

O **unix socket** é uma forma de comunicação interprocessos bastante usada quando há necessidade de troca de mensagens entre processos. Seu nome vem do inglês **I**nter **P**rocess **C**ommunication ou *IPC*. Ele funciona de uma forma parecida com os outros sockets utilizados para comunicação via redes que vimos nos capítulos anteriores, mas é usado apenas para comunicação local.

O **servidor de aplicação** é um servidor especializado em receber requisições HTTP e falar um idioma que a aplicação, ou o *framework* que ela utiliza, conheça e possa lidar de uma forma mais simples.

O servidor de aplicação já sabe HTTP, para que precisamos do servidor web?

Quando o servidor web funciona dessa maneira, ele se torna um *proxy reverso*. Essa técnica geralmente é utilizada para manter na linha de frente um servidor mais maduro e acostumado a lidar com o ambiente hostil da internet.

Apesar de o servidor de aplicação estar preparado para lidar com HTTP, ele geralmente não possui todos os mecanismos de proteção que servidores como o NGINX ou Apache possuem. Isso acontece devido a maioria dos servidores de aplicação assumirem que haverá um *proxy reverso* responsável por lidar com esses casos, portanto, eles implementam apenas o necessário.

Um exemplo de problema que pode ser gerado por expor o servidor de aplicação diretamente na internet são os clientes lentos. Esses clientes podem segurar a conexão, não deixando que o servidor continue o processamento e segurando o processo. Os clientes em redes de celular geralmente se comportam dessa forma devido à demora para entregar a resposta em uma rede com mais latência. Caso existam muitos clientes que estão em redes com latência alta ou simulam essa latência de propósito, será necessário ter um proxy reverso que consiga lidar com uma grande concorrência de I/O.

Podemos ver a comunicação do servidor web com o servidor de aplicação na saída do comando `strace` que executamos:

```
socket(PF_LOCAL, SOCK_STREAM, 0)        = 14
epoll_ctl(7, EPOLL_CTL_ADD, 14, {EPOLLIN|EPOLLOUT|EPOLLRDHUP|EPOLL
ET, {u32=4281800408, u64=140325158328024}}) = 0
```

```
connect(14, {sa_family=AF_LOCAL, sun_path="/tmp/passenger.b89vIib/
agents.s/core"}, 110) = 0
writev(14, [{"GET / HTTP/1.1\r\nConnection: clos"..., 1151}], 1) =
 1151
epoll_wait(7, {{EPOLLIN|EPOLLOUT|EPOLLHUP|EPOLLRDHUP, {u32=4281800
408, u64=140325158328024}}}, 512, 33646) = 1
recvfrom(14, "HTTP/1.1 200 OK\r\nStatus: 200 OK\r"..., 16384, 0, N
ULL, NULL) = 10340
write(8, "\27\3\3(\317\230_\36_9TJ\0162\322\241\224PT.\23\327\371\
265#if\277fY9\227"..., 10452) = 10452
recvfrom(14, "", 16384, 0, NULL, NULL)  = 0
close(14)
```

O socket `14` foi criado, adicionado ao `epoll` e conectado ao caminho `/tmp/passenger.b89vIib/agents.s/core`. O NGINX encaminha a conexão HTTP para o socket via `writev` e aguarda resposta com `epoll_wait`. Algum tempo depois, o servidor de aplicação responde com o status **200** do HTTP e com o conteúdo, que é escrito em outro *file descriptor* para ser devolvido para o cliente.

Esse código é parte do módulo do Phusion Passenger dentro do NGINX, que vamos estudar com mais detalhes na próxima seção.

Outras comunicações entre o servidor web e o servidor de aplicação

No caso que vimos há pouco, a comunicação acontece por meio de um unix socket, mas isso nem sempre acontece dessa forma. Uma das formas que é muito usada atualmente por ser recomendada pelo site *12factor* (http://12factor.net/) é a comunicação via TCP. Utilizando o *unicorn* como servidor de aplicação, essa configuração é feita por meio do parâmetro `listen` na configuração. Caso o servidor de aplicação seja o *puma*, podemos usar o parâmetro `-b` com o endereço TCP.

7.4 PHUSION PASSENGER, O SERVIDOR DE APLICAÇÃO

Após o NGINX lidar diretamente com a internet da melhor maneira possível, ele encaminha a requisição para uma das instâncias do servidor de aplicação.

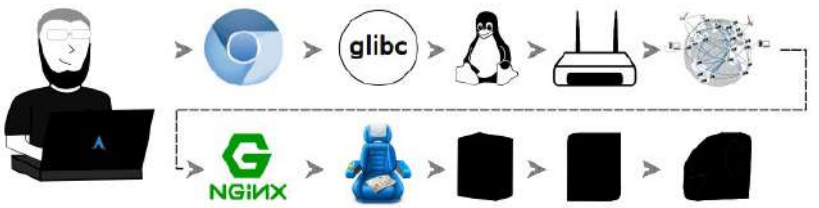


Figura 7.8: Phusion Passenger, o servidor de aplicação

O Phusion Passenger[9], que vamos chamar apenas de Passenger, é um software criado por uma empresa chamada Phusion[10]. Seu principal objetivo é trabalhar junto com o servidor web, no nosso caso o NGINX, para entregar o conteúdo da aplicação Ruby on Rails do `desconstruindoaweb.com.br` para o cliente HTTP, que nesse caso é o navegador.

A escolha do Phusion Passenger

Para quem conhece a comunidade Ruby, sabe que o Passenger não é a escolha número 1 dos rubistas há algum tempo. Ele perdeu popularidade para outros servidores de aplicações baseados em *Rack*, como *Unicorn* e *puma*, desde o seu lançamento. A Phusion afirma que a nova versão do Passenger, chamada de Raptor, tende a ser até 4 vezes mais rápida que o unicorn e até 2 vezes mais rápida que o puma[11].

Mesmo assim, ainda fica a dúvida: *se ele não é mais a escolha principal de quem está trabalhando com Rails, por que utilizá-lo?*. E a resposta é bem mais simples do que parece: ele foi a recomendação do provedor escolhido, e gasta bem menos recursos no ambiente deles, melhorando a performance e diminuindo os gastos com servidor. Mas, além disso, o Passenger possui uma ótima documentação, inclusive de implementações internas, que vai nos ajudar durante essa jornada.

Mesmo com o uso do Passenger para o nosso estudo, continuaremos fazendo comparação com outros servidores de aplicação para entender como as aplicações lidam com as requisições no geral.

Para começar nosso estudo, vamos utilizar o comando `curl` com bastante log para dar uma olhada no que uma requisição nos diz sobre ele:

```
$ curl -vvv http://desconstruindoaweb.com.br

* Rebuilt URL to: http://desconstruindoaweb.com.br/
*   Trying 177.153.1.102...
```

```
* Connected to desconstruindoaweb.com.br (177.153.1.102) port 80 (
#0)
> GET / HTTP/1.1
> Host: desconstruindoaweb.com.br
> User-Agent: curl/7.49.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/html; charset=utf-8
< Transfer-Encoding: chunked
< Connection: keep-alive
< Status: 200 OK
< Cache-Control: max-age=0, private, must-revalidate
< ETag: W/"f4f6f4cafb43228fb17c25ad738918c7"
< X-Frame-Options: SAMEORIGIN
< X-XSS-Protection: 1; mode=block
< X-Content-Type-Options: nosniff
< X-Runtime: 0.002855
< X-Request-Id: fc8c7ef7-3762-42e9-834c-d457ee9f5f4d
< Date: Sun, 26 Jun 2016 23:16:25 GMT
< X-Powered-By: Phusion Passenger 5.0.11
< Server: nginx + Phusion Passenger 5.0.11
```

A parte que nos interessa é essa:

```
X-Powered-By: Phusion Passenger 5.0.11
```

Este é um cabeçalho que o Passenger adiciona em todas as requisições e nos diz qual é a versão que está sendo executada. Como essa informação pode ser desabilitada via configuração, podemos também confirmar diretamente no servidor com:

```
$ passenger-config about version

Phusion Passenger 5.0.11
```

Isso nos diz que estamos utilizando a versão 5 do Passenger, também conhecida como *Raptor*. Podemos também olhar diretamente o código-fonte da versão no repositório do projeto no GitHub[12], se quisermos referências.

Configuração

O Passenger atua como um módulo nos dois principais

servidores web que temos atualmente para o ambiente GNU/Linux, o NGINX e o Apache. No nosso caso, o Passenger está instalado como um módulo do NGINX:

```
$ nginx -V

nginx version: nginx/1.8.0
built by gcc 4.8.3 20140911 (Red Hat 4.8.3-9) (GCC)
built with OpenSSL 1.0.1e-fips 11 Feb 2013
TLS SNI support enabled
configure arguments: --prefix=/opt/nginx-ruby-2.2.3 --with-http_ss
l_module --with-http_gzip_static_module --with-http_stub_status_mo
dule --with-cc-opt=-Wno-error --add-module=/usr/lib/rvm/gems/ruby-
2.2.3/gems/passenger-5.0.11/ext/nginx --user=nginx --group=nginx -
-conf-path=/etc/nginx/nginx.conf --error-log-path=/var/log/nginx/e
rror.log --http-log-path=/var/log/nginx/access.log --pid-path=/var
/run/nginx.pid --lock-path=/var/lock/subsys/nginx --with-http_ssl_
module --with-http_realip_module --with-http_flv_module --with-htt
p_sub_module --with-http_dav_module --with-file-aio --with-http_ad
dition_module --with-http_flv_module --with-http_mp4_module --with
-http_random_index_module --with-http_secure_link_module --with-ht
tp_geoip_module --with-http_sub_module --with-ipv6 --with-mail --w
ith-mail_ssl_module --add-module=/home/nginx-dav-ext-module-master
 --add-module=/home/headers-more-nginx-module-master --add-module=
/home/modsecurity-2.9.0/nginx/modsecurity
```

O NGINX, diferentemente do Apache, ainda não suporta módulos "linkados" dinamicamente, fazendo com que seja necessário recompilar o NGINX toda vez que um novo módulo for instalado. Apesar da quantidade grande de flags no comando `nginx -V`, a que nos interessa mesmo é a `--add-module=/usr/lib/rvm/gems/ruby-2.2.3/gems/passenger-5.0.11/ext/nginx`, que mostra o caminho do módulo.

RVM

É possível notar, pelo caminho até o código-fonte do módulo que o Jelastic usa RVM (http://rvm.io) para manter várias versões de Ruby no mesmo ambiente. O RVM altera o PATH do sistema para adicionar alguns binários alternativos para o Ruby, fugindo do padrão de gerenciamento de pacotes do sistema operacional.

O usuário ganha na facilidade de instalação e utilização de múltiplas versões de Ruby, porém perde a atualização oficial de pacotes que as distribuições disponibilizam. No caso do Jelastic, o uso de várias versões de forma programática é mais simples de ser executado via RVM.

Apesar do módulo já estar instalado, ele precisa também ser **habilitado** na configuração do NGINX. Olhando o arquivo de configuração chamado `nginx.conf` (comentários e linhas em branco removidas), temos:

```
$ cat /etc/nginx/nginx.conf

worker_processes  1;

events {
    worker_connections  1024;
}

http {
     server_tokens off ;
     include       mime.types;
     default_type  application/octet-stream;
     sendfile        on;
     keepalive_timeout  65;
     include  app_servers/nginx-passenger.conf;
     include /etc/nginx/conf.d/*.conf;
}
```

Vamos dar uma olhada nesses arquivos mencionados com `include`, começando com o `nginx-passenger.conf`:

```
$ cat /etc/nginx/app_servers/nginx-passenger.conf

    passenger_root /usr/lib/rvm/gems/current-gems/gems/passenger-v
ersion;
    passenger_ruby /usr/lib/rvm/wrappers/current-wrapper/ruby;
    include /etc/nginx/ruby.env ; # rails_env production;

    server {
        listen       *:80;
        server_name  _;
        root /var/www/webroot/ROOT/public ;
        passenger_enabled on;

        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }

    }
```

Nesse arquivo, o diretório do módulo é especificado com o comando `passenger_root`, e o Ruby utilizado é especificado com `passenger_ruby`. No bloco chamado `server`, o Passenger é habilitado com o comando `passenger_enable on`.

Nos arquivos presentes no diretório `conf.d`, temos apenas a configuração do SSL/TLS:

```
$ cat /etc/nginx/conf.d/*.conf
server {
        listen       443;
        server_name  _;

        ssl                  on;
        ssl_certificate      /var/lib/jelastic/SSL/jelastic.chain;
        ssl_certificate_key  /var/lib/jelastic/SSL/jelastic.key;
        ssl_session_timeout  5m;

        ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers ...
        ssl_prefer_server_ciphers   on;
        ssl_session_cache shared:SSL:10m;
```

```
        passenger_enabled on;
        passenger_env_var HTTPS on;
        passenger_set_header X_CLIENT_DN $ssl_client_s_dn;
        passenger_set_header X_CLIENT_VERIFY $ssl_client_verify;
        root   /var/www/webroot/ROOT/public;
}
```

Os arquivos `/var/lib/jelastic/SSL/jelastic.chain` e `/var/lib/jelastic/SSL/jelastic.key` são os aqueles que estudamos no *capítulo 5*, sobre HTTPS e segurança, agora sendo usados na prática. O primeiro é a cadeia de certificados, contendo os três que o `letsencrypt.org` nos forneceu. O segundo é a chave privada da conexão, que deve estar sempre muito bem guardada.

Configuração de outros servidores de aplicação

Ao usar *Unicorn*, por exemplo, a configuração tende a usar TCP ou unix socket, e o NGINX vai servir como proxy reverso. Para que isso aconteça, além de fazer uma configuração parecida com a que estudamos, também é necessário configurar explicitamente o `upstream` no NGINX:

```
upstream desconstruindoaweb {
       server 127.0.0.1:5055;
}
```

Considere que o unicorn estará ouvindo por TCP na porta 5055.

No caso de aplicações PHP com `php5-fpm`, por exemplo, não será necessário usar um `upstream`, e a configuração pode ir diretamente no `location` da configuração do NGINX:

```
location ~ [^/]\.php(/|$) {
   fastcgi_split_path_info ^(.+?\.php)(/.*)$;
   if (!-f $document_root$fastcgi_script_name) {
       return 404;
   }

   fastcgi_pass unix:/var/run/php5-fpm.sock;
   fastcgi_index index.php;
   include fastcgi_params;
}
```

Essa configuração vai usar o módulo de `fastCGI` do NGINX para encaminhar as requisições para um servidor de `fastcgi`, que no caso é o `php-fpm`. Apesar de a arquitetura ser diferente, lá está a configuração do unix socket, assim como é feito com os outros.

Arquitetura

Como vimos na seção sobre o NGINX, há um módulo do Passenger no servidor web que faz a comunicação com o servidor de aplicação. Esse módulo foi o responsável pela execução das syscalls que enviam informações para um unix socket no final da seção sobre NGINX.

> **Onde está a configuração dos unix sockets?**
>
> Diferente de outros servidores de aplicação, o Passenger não provê uma forma de configurarmos como será a troca de mensagens entre os processos internos. Como os unix sockets são mais performáticos do que TCP local, os desenvolvedores do Passenger o deixaram como a única opção de troca de mensagens local, utilizando uma forma "performática por padrão", nas palavras deles.

Mas, além do módulo, ainda há alguns outros processos que garantem o funcionamento e resiliência do Passenger.

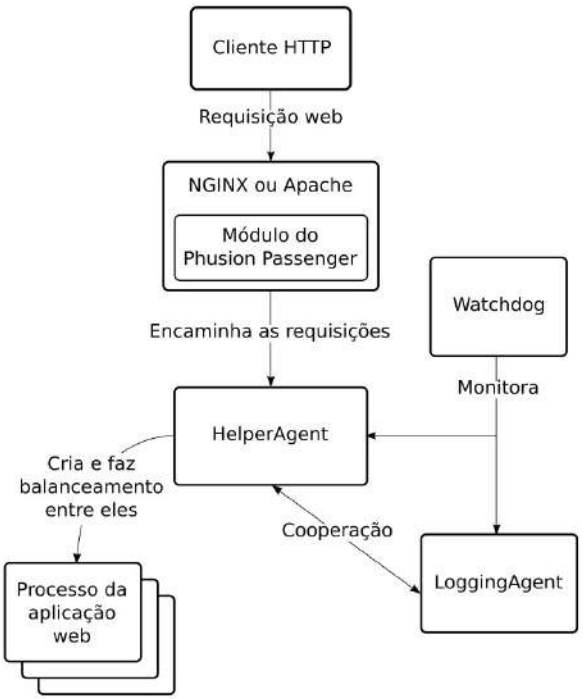


Figura 7.9: Arquitetura do Passenger. Baseado na imagem disponível em: https://www.phusionpassenger.com/documentation/Design%20and%20Architecture.html

É possível ver esses processos em execução no servidor. Vamos usar um comando `sed` depois da `|` (barra vertical) só para melhorar a diagramação do conteúdo no livro, mas não é necessário para obter o resultado do comando:

```
$ ps xufa | grep "[P]assenger" | sed -r 's/.*:[0-9]{2}\s(.*)/\1/g'

Passenger watchdog
 \_ Passenger core
 |   \_ Passenger AppPreloader: /var/www/webroot/ROOT
 |       \_ Passenger RubyApp: /var/www/webroot/ROOT/public (produ
ction)
 \_ Passenger ust-router
Passenger RubyApp: /var/www/webroot/ROOT/public (production)
```

O processo de `watchdog` é responsável por manter todos os outros processos funcionando. Caso aconteça algum problema, ele

vai reiniciá-los ou criar novos processos. Já o processo `ust-router` não consome recursos, pois a função de *routing* não está habilitada no Passenger.

`Passenger AppPreloader` é o processo que faz o cache de memória da nossa aplicação. Por usá-lo, o Passenger não precisa fazer tudo o que é necessário para ter a aplicação funcionando toda vez que precisar de um novo processo, pois bastará fazer um `fork()` para conseguir um novo processo idêntico a esse.

O processo `core` e o `Passenger RubyApp` são os que realmente trabalham nas requisições. O `core` possui um subsistema chamado *Request Handler*[13], responsável por receber a requisição do servidor web e falar com o módulo `Application Pool`. Esse módulo é responsável pelos processos da aplicação e devolve para qual processo `core` deve encaminhar essa requisição.

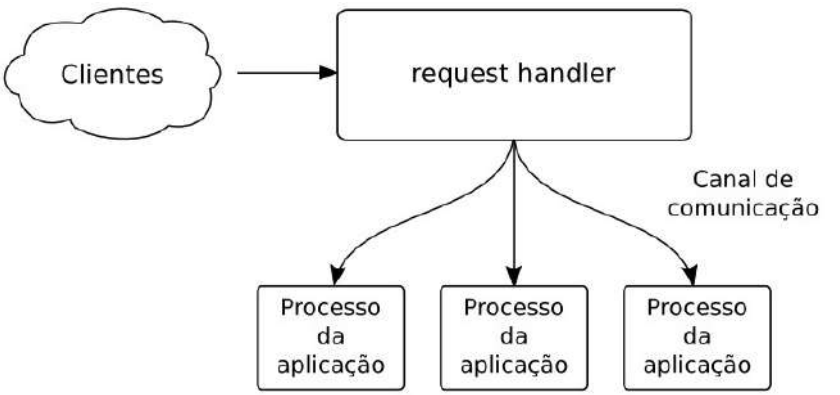


Figura 7.10: Comunicação com os processos. Baseado na imagem disponível em: http://www.rubyraptor.org/pointer-tagging-linked-string-hash-tables-turbocaching-and-other-raptor-optimizations/

O `Passenger RubyApp`, visto como resultado do comando `ps`, é o processo que possui a aplicação do `desconstruindoaweb.com.br`, e que vai receber a conexão para processar.

Recebendo a requisição do NGINX

É possível ver a conexão chegando no processo `Passenger RubyApp` usando o nosso velho conhecido comando `strace`. A primeira coisa que vemos é o servidor aguardando para aceitar uma conexão no unix socket do `core` com a syscall `accept4`:

```
[pid 32124] accept4(8,
```

Ao acessar a URL *https://desconstruindoaweb.com.br*, a syscall `accept4` se completa:

```
[pid 32124] accept4(8, {sa_family=AF_LOCAL, NULL}, [2], SOCK_CLOEX
EC) = 13
```

Ao se completar, ela gera um *file descriptor* com o número **13**, que será usado durante a conexão. Logo em seguida, ele tenta ler esse file descriptor:

```
[pid 32124] read(13, "REQUEST_URI\0/\0PATH_INFO\0/\0SCRIPT"..., 11
20) = 1120
```

Essa string, presente na syscall `read`, contém os dados da conexão. Esses dados estão utilizando um protocolo chamado SCGI[14], que é usado pelo NGINX e pelo próprio Passenger para a comunicação entre os processos.

Após algumas syscalls, é possível ver a resposta já processada pela aplicação sendo enviada de volta pelo mesmo unix socket:

```
[pid 32124] writev(13, [{"HTTP/1.1 200 Whatever\r\n", 23}, {"X-Fra
me-Options", 15}, {": ", 2}, {"SAMEORIGIN", 10}, {"\r\n", 2}, {"X-
XSS-Protection", 16}, {": ", 2}, {"1; mode=block", 13}, {"\r\n", 2
}, {"X-Content-Type-Options", 22}, {": ", 2}, {"nosniff", 7}, {"\r
\n", 2}, {"Strict-Transport-Security", 25}, {": ", 2}, {"max-age=3
1536000; includeSubDoma"..., 35}, {"\r\n", 2}, {"Content-Type", 12
}, {": ", 2}, {"text/html; charset=utf-8", 24}, {"\r\n", 2}, {"ETa
g", 4}, {": ", 2}, {"W/\"6e08a4f9906888f72c5ee4df4aa8f"..., 36}, {
"\r\n", 2}, {"Cache-Control", 13}, {": ", 2}, {"max-age=0, private
, must-revalid"..., 35}, {"\r\n", 2}, {"Set-Cookie", 10}, {": ", 2
}, {"_desconstruindo-a-web-hotsite_se"..., 342}, ...], 43) = 777
[pid 32124] ppoll([{fd=13, events=POLLOUT}], 1, NULL, NULL, 8) = 1
```

```
 ([{fd=13, revents=POLLOUT}])
[pid 32124] writev(13, [{"250d", 4}, {"\r\n", 2}, {"<!doctype html
>\n<!--[if IE]><![e"..., 9485}, {"\r\n", 2}], 4) = 9493
[pid 32124] write(13, "0\r\n\r\n", 5)   = 5
```

Depois de enviar tudo o que tinha de ser enviado, o servidor fecha a conexão e o socket, começando o processo novamente:

```
[pid 32124] shutdown(13, SHUT_WR)       = 0
[pid 32124] close(13)                   = 0
[pid 32124] accept4(8,
```

A syscall `accept4` fica incompleta como quando começamos o processo, até que uma nova requisição seja feita.

Apesar de ter chegado ao fim da conexão, ainda não descobrimos como que essa conexão chegou até a nossa aplicação e retornou aquele resultado escrito via `writev`. No próximo capítulo, veremos a ligação entre o servidor de aplicação, o Rack, o framework e o código da aplicação.

7.5 RESUMO

O servidor web pode significar tanto o servidor físico, que está dentro de um datacenter plugado à internet, quanto o software, que está instalado e responde as requisições. Há vários softwares que fazem o trabalho de servidor web e, entre esses vários, nós decidimos escolher o NGINX para esse estudo. O NGINX utiliza um processo master e vários processos worker, sendo que estes últimos são os responsáveis por responder as requisições que chegam para o servidor web.

Os workers do NGINX trabalham com o modelo de eventos, no qual cada processo atende as requisições de uma forma assíncrona com apenas uma thread. Para ser assíncrono, ele utiliza o modelo de eventos que o sistema operacional disponibiliza, chamado `epoll`.

Ao receber uma requisição e perceber que não é um conteúdo

estático, o NGINX se comporta como um proxy reverso e usa um unix socket para se comunicar com o servidor de aplicação.

O servidor de aplicação escolhido para esse estudo é o Phusion Passenger, que será o responsável por receber o HTTP e falar com a aplicação. Ele usa vários processos, sendo os mais importantes para esse fluxo o `core` e o `Passenger RubyApp`.

Por utilizar o Passenger, um módulo é adicionado ao NGINX para que a comunicação com o servidor de aplicação seja feita. O módulo se comunica com o processo `core`, que por sua vez se comunica com a aplicação para conseguir o resultado esperado.

7.6 REFERÊNCIAS

1. Site da Locaweb — https://locaweb.com.br
2. Informações da Locaweb como provedor de Jelastic — https://jelastic.cloud/details/locaweb
3. Site do Virtuozzo — https://virtuozzo.com
4. XenServer — http://xenserver.org/
5. Docker — https://www.docker.com/
6. Pesquisa dos servidores web mais utilizados na w3techs — https://w3techs.com/technologies/overview/web_server/all
7. O problema c10k — http://www.kegel.com/c10k.html
8. Modos de processamento de conexões do NGINX — http://nginx.org/en/docs/events.html
9. Phusion Passenger — https://www.phusionpassenger.com/
10. Phusion, a empresa criadora do Passenger —

https://phusion.nl/

11. Site da versão 5 do Passenger, conhecida como Raptor — http://www.rubyraptor.org

12. Código da versão *5.0.11* do Passenger, que estamos estudando — https://github.com/phusion/passenger/tree/release-5.0.11

13. Documentação de Design e arquitetura do Passenger — https://www.phusionpassenger.com/documentation/Design%20and%20Architecture.html#_request_handling

14. Descrição do protocolo SCGI — https://python.ca/scgi/protocol.txt

Capítulo 8

O FRAMEWORK E A APLICAÇÃO

O servidor web recebeu a conexão da internet e trabalhou como um proxy reverso, encaminhando a requisição para o servidor de aplicação. Após uma pequena espera, o servidor de aplicação responde e o servidor web encaminha a resposta para o cliente. Entre o servidor de aplicação e a aplicação, há algumas camadas de ligação, incluindo o framework usado para criá-la.

Em situações comuns, os desenvolvedores entendem basicamente como usar o framework e começam o desenvolvimento, mas esse não é o nosso caso. Nesse capítulo, vamos mais a fundo para descobrir como o código de uma aplicação conversa com o código do framework.

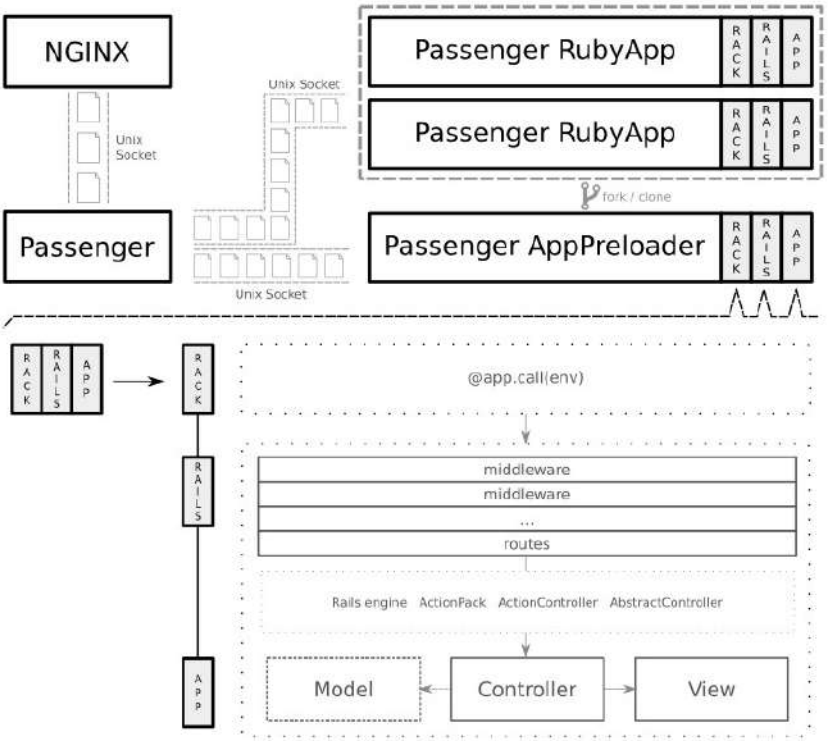


Figura 8.1: Do Passenger até a aplicação

Esteja preparado para um capítulo mais denso, pois nesse vamos estudar mais código do que o normal. A *figura 8.1* mostra o processo completo e pode ser utilizada como guia, se necessário.

8.1 CONHECENDO O RACK

Para entender como o servidor de aplicação chega até a aplicação, estudaremos o que é o Rack e como ele funciona.

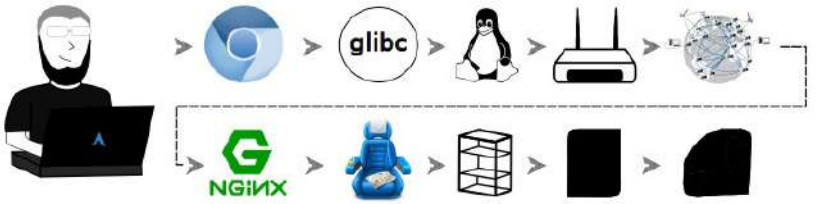


Figura 8.2: A jornada pelo Rack

No mundo Ruby, existem diversos servidores de aplicação disponíveis para uso. Também há vários frameworks web que podem ser utilizados, apesar do Ruby on Rails ser o mais popular.

> **O NOME "FRAMEWORK"**
>
> O meio acadêmico geralmente traduz a palavra *framework* para "arcabouço". Neste livro, usaremos a palavra em inglês por ser a mais utilizada na prática.
>
> O significado para esse caso é o mesmo para ambos: um conjunto de código inicial que vai ser a base para você construir a sua aplicação. Alguns exemplos: Ruby on Rails para o Ruby, Django para o Python, Laravel para o PHP etc.

Cada servidor de aplicação precisa conversar com cada um dos frameworks para que a conexão seja encaminhada com sucesso. Esse problema precisava ser resolvido para evitar a duplicação de esforço e código.

Nessa época, nasceu o *Rack*[1], que tem como objetivo ser a interface entre os frameworks e os servidores de aplicação.

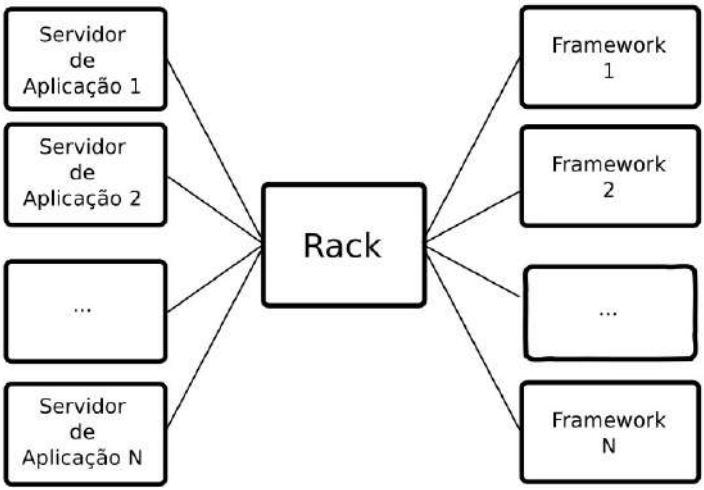


Figura 8.3: Rack como interface entre frameworks e servidores de aplicação. Baseado em: https://www.phusionpassenger.com/documentation/Design%20and%20Architecture.html#_about_rack

Utilizando o Rack

O Rack define uma API bem sucinta, bastando que o desenvolvedor do framework entregue um objeto que responda ao método `call`. Um exemplo de uma aplicação é uma com Rack válida:

```
app = Proc.new do |env|
  ['200', {'Content-Type' => 'text/html'}, ['request body']]
end
```

O servidor de aplicação só precisa entender essa API para se comunicar com todos os frameworks que a implementam. Atualmente, há vários frameworks web para aplicações Ruby e servidores de aplicação compatíveis com Rack, fazendo dessa integração o novo padrão para as coisas novas. O Rails, que será nosso framework de estudo, é totalmente compatível com Rack desde a versão 3.

Configuração

Por padrão, as aplicações Rack devem usar um arquivo chamado `config.ru` na raiz da aplicação. Ele é utilizado para executar uma aplicação, sendo o primeiro arquivo a ser procurado caso o comando `rackup`, disponibilizado pelo Rack, seja executado.

Esse arquivo possui uma chamada para o comando `run` em uma aplicação Rack. Utilizando um exemplo parecido com o que usamos na seção anterior, temos a seguinte execução de uma aplicação Rack básica:

```
run proc {|env| [200, {'Content-Type' => 'text/plain'}, ["request
body"]] }
```

Vamos executar o comando `rackup` nesse diretório para ver o resultado:

```
$ rackup

[2016-08-05 08:44:41] INFO  WEBrick 1.3.1
[2016-08-05 08:44:41] INFO  ruby 2.3.1 (2016-04-26) [x86_64-linux]
[2016-08-05 08:44:41] INFO  WEBrick::HTTPServer#start: pid=19967 p
ort=9292
```

Esse comando carregou o arquivo `config.ru` e disponibilizou uma aplicação na porta `9292`. Essa aplicação escreve `request body` quando o endereço `http://localhost:9292` for acessado.

8.2 DO PASSENGER AO RACK

Agora que conhecemos o básico do Rack, vamos estudar como o Passenger se conecta com ele para executar a aplicação. Como vimos no capítulo anterior, o Passenger tem vários processos, mas o único que entrega as requisições para a aplicação é o `Passenger RubyApp`. Ele vai se conectar ao Rack e carregá-lo juntamente com o Rails e o código da aplicação.

Para entender esse processo, vamos olhar o código do Passenger na tag `5.0.11` que foi adicionada como referência no capítulo

passado.

O Passenger possui duas formas de carregar a aplicação, sendo que em ambas há um processo `Spawner` responsável por criar os processos de aplicação. Cada um dos modos lida com a criação dos processos de uma forma diferente:

1. O processo `Spawner` cria um novo processo com a aplicação completa. Esse novo processo reporta que está pronto para receber requisições assim que terminar de ser carregado. Nesse caso, uma nova aplicação será carregada todas as vezes que um processo for necessário. Esse modo é chamado de `direct` na configuração do Passenger.

2. O processo `Spawner` cria um novo processo com a aplicação completa. Esse novo processo, assim como no modo `direct`, reporta que está pronto assim que terminar de ser carregado. Mas em vez de reportar que está pronto para receber requisições web, ele reporta para o `Spawner` que está pronto para receber requisições de **criação de novos processos**. Todas as cópias criadas por ele são reportadas para o processo `Spawner`, que utilizará essa informação para enviar as requisições web diretamente para eles. Esse modo é chamado de `smart` na configuração do Passenger.

Como vimos no capítulo anterior, nossa aplicacão tem um processo chamado `Passenger AppPreloader`, que vamos chamar apenas de `Preloader`. Esse processo é responsável por executar o modo `smart`, portanto, continuaremos nossos estudos seguindo essa linha.

Como estamos usando a implementação de `smart spawning` [2], o Passenger vai executar um processo intermediário chamado `SpawnPreparer`, que checará alguns valores e criará um novo processo utilizando o arquivo `helper-scripts/rack-`

`preloader.rb` [3].

Esse arquivo é responsável por carregar a aplicação. Podemos ver no final dele a ordem de execução dos métodos para executar essa tarefa:

```
handshake_and_read_startup_request
init_passenger
preload_app
if PreloaderSharedHelpers.run_main_loop(options) == :forked
  handler = negotiate_spawn_command
  handler.main_loop
  handler.cleanup
  LoaderSharedHelpers.after_handling_requests
end
```

No primeiro método, ele faz o `handshake` com o processo de Spawner via `STDIN` e `STDOUT`, também conhecidos como entrada e saída padrão, ou *Standard Input* e *Standard Output*. Nessa comunicação, o processo `Preloader` recebe as informações referentes ao Passenger e a aplicação, utilizando um protocolo curto criado pelos desenvolvedores da Phusion.

O handshake começa com o Loader enviando a mensagem `!> I have control 1.0`, que deve ser respondida com `!> You have control 1.0` pelo processo Spawner. Depois disso, o Spawner envia diversas linhas com o conteúdo descrito por `chave: valor` com as informações do Passenger, terminando o envio com uma linha vazia.

Ele então faz a inicialização do Passenger por meio do método `init_passenger` e executa o método responsável por fazer o `Preload` da aplicação, que está presente no método `preload_app`. Nesse método, temos o seguinte código, responsável por carregar o arquivo do Rack:

```
rackup_file = options["startup_file"] || "config.ru"
rackup_code = ::File.open(rackup_file, 'rb') do |f|
  f.read
```

```
end
@@app = eval("Rack::Builder.new {( #{rackup_code}\n )}.to_app",
  TOPLEVEL_BINDING, rackup_file)
```

Nesse código, o arquivo `config.ru` da aplicação é encontrado e lido para uma variável chamada `rackup_code`. Esta é utilizada como argumento para o comando `Rack::Builder.new`, para carregar a aplicação.

Agora que o Passenger tem a aplicação carregada, ele pode carregar outros processos baseados nela. Esse modo de criação de processos economiza tempo, por não precisar carregar a aplicação novamente, e memória, por utilizar melhor a funcionalidade de *copy-on-write*.

O *copy-on-write* é uma funcionalidade que está implementada no sistema operacional e é usada sempre que um processo novo é criado, copiando a memória apenas quando for necessário. O que o Passenger faz é se preocupar em usar essa funcionalidade da forma mais performática possível.

O método `SharedHelpers.run_main_loop`, que é o próximo da linha de chamadas, cria um unix socket para comunicação com o Spawner e reporta que está pronto para receber novas conexões. Para reportar isso, ele utiliza os mesmos comandos usados no handshake com o Passenger:

```
puts "!> Ready"
puts "!> socket: unix:#{socket_filename}"
puts "!> "
```

Com o socket criado, o `Preloader` aguarda o comando `spawn` para criar novos processos. Com a chegada desse comando, o `Preloader` cria um novo processo baseado nele mesmo e retorna `:forked`, continuando o fluxo que vimos no início.

O novo processo criado vai seguir o mesmo fluxo de handshake, anunciando que está preparado para receber requisições no método

`negotiate_spawn_command` . Em seguida, ele chama o método `main_loop` do handler para entrar no *loop* principal que recebe as conexões.

O Passenger usa processos e threads para processar as requisições recebidas. Esse processo é iniciado no arquivo `lib/phusion_passenger/request_handler.rb` [4].

Após recebida, a requisição é processada e encaminhada para a aplicação Rack. Podemos ver a conexão do Passenger com o Rack no arquivo `thread_handler_extension` [5], que é um dos arquivos incluídos `thread_handler.rb` , responsável por fazer o controle das threads.

Vamos dar uma olhada no código que faz essa ligação:

```
begin
  status, headers, body = @app.call(env)
rescue => e
  # (...)
  PhusionPassenger.log_request_exception(env, e)
  return false
end
```

Após executar o comando `@app.call(env)` , a requisição será encaminhada para o Rack e executada pelo nosso framework de estudo, o Ruby on Rails.

8.3 O RUBY ON RAILS

Agora estamos chegando perto da aplicação do `desconstruindoaweb.com.br` e vamos entender como o Rack interage com o nosso servidor de estudo, o Ruby on Rails.

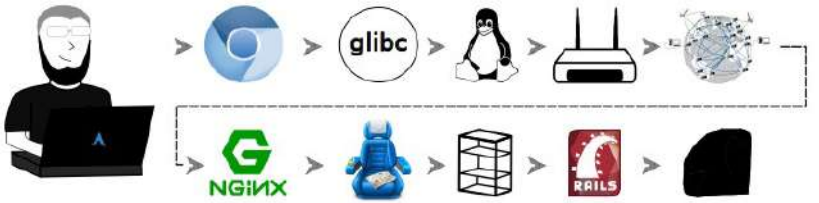


Figura 8.4: Uma jornada pelo Rails

O Ruby on Rails[6], conhecido popularmente como *Rails*, é um framework para desenvolvimento de aplicações web. Ele foi lançado em 2005 e popularizou ideias como *migrations* para o banco de dados e convenção sobre configuração, que mudaram a forma de se ver frameworks web na época. Ainda hoje, ele é considerado por muitos como o responsável pela popularidade do Ruby como linguagem.

O framework foi feito para facilitar a criação de aplicações web completas, e atualmente possui uma vasta quantidade de bibliotecas compatíveis que ajudam a fazer as tarefas corriqueiras das aplicações web atuais. Alguns exemplos de tarefas que possuem ferramentas já estabelecidas: autenticação, autorização, formulários de cadastro, upload de imagens, compressão de arquivos CSS/JavaScript e testes automatizados.

Ele utiliza o modelo *MVC*, acrônimo para **M**odel **V**iew **C**ontroller. Este instrui a separação da camada regras de negócio (*model*), camada de visualização (*view*) e camada de controle (*controller*), que faz a ligação entre as outras duas. Nesse estudo, vamos ver exemplos da camada de *controller* e *view*, deixando a camada de modelos de lado devido à natureza da aplicação.

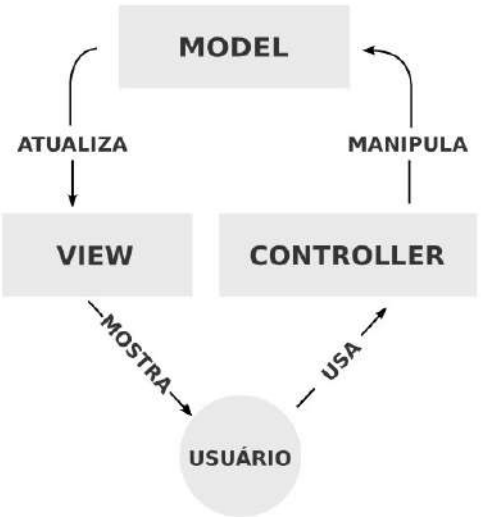


Figura 8.5: O modelo MVC. Baseado no trabalho de RegisFrey, em https://commons.wikimedia.org/w/index.php?curid=10298177

O conceito de *convention over configuration* (ou convenção sobre configuração) foi uma das ideias que mais ajudou na popularização do Rails devido à facilidade que ela trazia para começar a desenvolver. Esse conceito traz a ideia de assumir um padrão em vez de fazer o desenvolvedor ter de configurar para usar. Isso traz a vantagem de não precisar conhecer todas as funcionalidades para começar a desenvolver e, aos poucos, ir adequando o framework às suas necessidades.

Para o nosso estudo, usaremos o código da aplicação do `desconstruindoaweb.com.br`, que está utilizando Rails 5. O código é open source e está disponível no GitHub[7]. Por estar usando a versão 5 do Rails, vamos utilizar também o código do Rails nessa versão[8] como referência.

A conexão do Rails com o Rack

O Rails ficou totalmente compatível com o Rack a partir da

versão 3, sendo que, desde essa versão, toda nova aplicação possui um arquivo `config.ru`. Por utilizar uma versão maior que a 3, podemos ver o arquivo `config.ru` na raiz do projeto do `desconstruindoaweb.com.br`:

```
# This file is used by Rack-based servers to start the application.

require_relative 'config/environment'
run Rails.application
```

Como vimos enquanto estudávamos o Rack, esse é o arquivo que é carregado pelo Passenger para fazer a conexão com o framework e, consequentemente, com a aplicação.

Vamos inspecionar o conteúdo de `Rails.application` para entender melhor:

```
$ rails runner "p Rails.application" 2>/dev/null
#<DesconstruindoAWeb::Application:0x0000000304deb0 @_all_autoload_
paths=...
```

Esse é o objeto da nossa aplicação, que é um `Rails::Application` via herança:

```
# https://github.com/PotHix/desconstruindoaweb.com.br/blob/master/
config/application.rb#L18
module DesconstruindoAWeb
  class Application < Rails::Application
    # ...
  end
end
```

O `Rails::Application`, por sua vez, é uma `Engine` no Rails. Por esse motivo, ele responde ao método `call`, se tornando uma aplicação Rack. Podemos ver isso no código do Rails:

```
# https://github.com/rails/rails/blob/v5.0.0/railties/lib/rails/en
gine.rb#L520

# Define the Rack API for this engine.
def call(env)
  req = build_request env
  app.call req.env
```

```
end
```

Além disso, o Rails também utiliza uma grande quantidade de *middlewares* do Rack para lidar com os vários passos da conexão.

Os middlewares

Às vezes, a palavra *middleware* é traduzida como *mediador*. O que ele faz é ficar no meio do caminho entre duas camadas executando uma tarefa. No caso do Rack, os middlewares são plugáveis, possibilitando que eles sejam encadeados. Cada middleware faz uma parte do trabalho, podendo ou não alterar dados na conexão atual.

O Rails possui vários middlewares, que funcionam como microaplicações utilizando o conceito de encadeamento para atender uma requisição. Para ver como é a cadeia de middlewares do `desconstruindoaweb.com.br` , vamos usar o comando `rails middleware` :

```
$ bin/rails middleware

use Rack::Sendfile
use ActionDispatch::Static
use ActionDispatch::Executor
use ActiveSupport::Cache::Strategy::LocalCache::Middleware
use Rack::Runtime
use Rack::MethodOverride
use ActionDispatch::RequestId
use Rails::Rack::Logger
use ActionDispatch::ShowExceptions
use WebConsole::Middleware
use ActionDispatch::DebugExceptions
use ActionDispatch::RemoteIp
use ActionDispatch::Reloader
use ActionDispatch::Callbacks
use ActionDispatch::Cookies
use ActionDispatch::Session::CookieStore
use ActionDispatch::Flash
use Rack::Head
use Rack::ConditionalGet
use Rack::ETag
```

```
run DesconstruindoAWeb::Application.routes
```

> **Middleware de SSL**
>
> No *capítulo 1*, vimos o HSTS, e mencionamos que nossa aplicação não estava usando-o para que pudéssemos testar tanto HTTP como HTTPS. Para utilizar, precisamos ativar a opção `force_ssl` do Rails. Quando essa opção é ativada, um novo middleware é adicionado à lista: `ActionDispatch::SSL`.
>
> Esse middleware lida com o redirecionamento da requisição atual para HTTPS e adiciona o cabeçalho do HSTS para que novas requisições usem o protocolo.

Como exemplo, podemos usar o `ActionDispatch::RequestId` que faz parte do `ActionDispatch` e tem como finalidade gerar um número aleatório para a requisição. Vamos estudá-lo mais de perto:

```
# https://github.com/rails/rails/blob/v5.0.0/actionpack/lib/action
_dispatch/middleware/request_id.rb#L17
class RequestId
  X_REQUEST_ID = "X-Request-Id".freeze # :nodoc:

  def initialize(app)
    @app = app
  end

  def call(env)
    req = ActionDispatch::Request.new env
    req.request_id = make_request_id(req.x_request_id)
    @app.call(env).tap { |_status, headers, _body| headers[X_REQUE
ST_ID] = req.request_id }
  end
  # ...
end
```

Essa é a estrutura de um middleware do Rack. Ele recebe a aplicação quando é iniciado e responde pelo método `call(env)`. O `env` é um hash com todas as informações sobre a conexão atual, incluindo as que foram adicionadas por middlewares executados antes dele.

Nesse código, ele chama o método `call` para passar a requisição para o próximo middleware e aguarda até a resposta. Quando a resposta é recebida, ele executa o método `tap` para ter acesso ao conteúdo e utiliza apenas o `Hash` de cabeçalhos para incluir uma nova chave. Essa chave se chama `X-Request-Id` e possui o número aleatório referente a essa requisição.

Podemos ver esse cabeçalho na prática quando fazemos uma requisição para o `desconstruindoaweb.com.br`:

```
$ curl -v http://desconstruindoaweb.com.br |& grep Request
< X-Request-Id: 1f844f1e-d176-4584-a629-75fa887b928a
```

CURIOSIDADE SOBRE O COMANDO CURL

Nesse comando, estamos utilizando o comando `curl` com o argumento `-v` para mostrar, além do conteúdo da página, todos os cabeçalhos enviados e recebidos. Como esse valor vai para a saída de erro padrão, também conhecida com `STDERR`, precisamos usar esse truque do `|&` para conseguir utilizar o comando `grep` para filtrá-los.

Esse é apenas um dos middlewares, sendo provavelmente o mais simples deles. Por padrão, o Rails adiciona middlewares para executar diversos tipos de tarefas. Mas não é só um privilégio do Rails, pois o desenvolvedor também pode adicionar middlewares customizados, bastando adicioná-los ao processo na configuração

principal da aplicação.

O sistema de middlewares é uma forma poderosa e extensível que vem sendo utilizada em aplicações web há muitos anos.

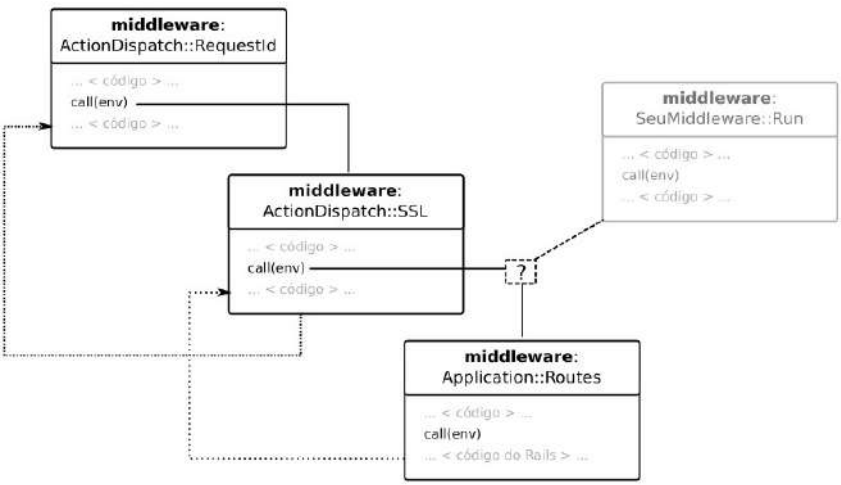


Figura 8.6: O processo dos middlewares

Chegando nas rotas

O último dos middlewares da lista é o `DesconstruindoAWeb::Application.routes`, responsável pelas rotas da aplicação. Vamos inspecioná-lo para obter mais informações:

```
$ rails runner "p DesconstruindoAWeb::Application.routes"

#<ActionDispatch::Routing::RouteSet:0x0000000368e6a8>
```

Ele é um objeto do tipo `ActionDispatch::Routing::RouteSet`, que também é um middleware. Se olharmos no código do Rails, veremos que ele possui um método `initialize` e um método `call`. Para entender o que ele faz, vamos olhar o método `call`:

```
# https://github.com/rails/rails/blob/v5.0.0/actionpack/lib/action
```

```
_dispatch/routing/route_set.rb#L725
def call(env)
  req = make_request(env)
  req.path_info = Journey::Router::Utils.normalize_path(req.path_i
nfo)
  @router.serve(req)
end
```

Vamos utilizar a gem `byebug`, que vem com o Rails 5 por padrão, para inspecionar o valor do parâmetro `env` que o Rack envia:

```
# => (byebug) env.keys

["GATEWAY_INTERFACE", "PATH_INFO", "QUERY_STRING", "REMOTE_ADDR",
"REMOTE_HOST", "REQUEST_METHOD", "REQUEST_URI", "SCRIPT_NAME", "SE
RVER_NAME", "SERVER_PORT", "SERVER_PROTOCOL", "SERVER_SOFTWARE", "
HTTP_HOST", "HTTP_CONNECTION", "HTTP_UPGRADE_INSECURE_REQUESTS", "
HTTP_USER_AGENT", "HTTP_ACCEPT", "HTTP_ACCEPT_ENCODING", "HTTP_ACC
EPT_LANGUAGE", "HTTP_COOKIE", "rack.version", "rack.input", "rack.
errors", "rack.multithread", "rack.multiprocess", "rack.run_once",
"rack.url_scheme", "rack.hijack?", "rack.hijack", "rack.hijack_io"
, "HTTP_VERSION", "REQUEST_PATH", "action_dispatch.parameter_filte
r", "action_dispatch.redirect_filter", "action_dispatch.secret_tok
en", "action_dispatch.secret_key_base", "action_dispatch.show_exce
ptions", "action_dispatch.show_detailed_exceptions", "action_dispa
tch.logger", "action_dispatch.backtrace_cleaner", "action_dispatch
.key_generator", "action_dispatch.http_auth_salt", "action_dispatc
h.signed_cookie_salt", "action_dispatch.encrypted_cookie_salt", "a
ction_dispatch.encrypted_signed_cookie_salt", "action_dispatch.coo
kies_serializer", "action_dispatch.cookies_digest", "action_dispat
ch.routes", "ROUTES_12445380_SCRIPT_NAME", "ORIGINAL_FULLPATH", "O
RIGINAL_SCRIPT_NAME", "action_dispatch.request_id", "action_dispat
ch.remote_ip", "rack.session", "rack.session.options"]
```

Como vimos na seção anterior, o `env` é um hash com todas as informações sobre a conexão atual. Portanto, ele possui várias informações sobre cabeçalhos do HTTP, como `HTTP_ACCEPT` e `HTTP_ACCEPT_LANGUAGE`, além de outras informações internas sobre a conexão.

Como o `DesconstruindoAWeb::Application.routes` é o último middleware, é necessário que ele comece a processar a requisição. Vamos buscar a precedência desse router:

```
$ rails runner "p DesconstruindoAWeb::Application.routes.instance_
variable_get(:"@router").class"

ActionDispatch::Journey::Router
```

Ele é um `ActionDispatch::Journey::Router` e seu método chamado `serve` é o responsável por enviar as informações da requisição para o sistema de rotas da aplicação.

```
# https://github.com/rails/rails/blob/v5.0.0/actionpack/lib/action
_dispatch/journey/router.rb#L37
def serve(req)
  # mais 11 linhas de código acima
  req.path_parameters = set_params.merge parameters

  status, headers, body = route.app.serve(req)
  # mais 12 linhas de código abaixo
end
```

Vamos utilizar a gem `byebug` novamente para inspecionar alguns valores:

```
# => (byebug) req
#<ActionDispatch::Request:0x007fd610339a30 @env={"GATEWAY_INTERFAC
E"=>"CGI/1.1" ...

# => (byebug) route
#<ActionDispatch::Journey::Route:0x00000004e3ba58 @name="root", @a
pp=...

# => (byebug) route.app
#<ActionDispatch::Routing::RouteSet::Dispatcher:0x00000004e3bbe8 @
raise_on_name_error=true>

# => (byebug) req.path_parameters
{:controller=>"hotsite", :action=>"index"}
```

Os objetos são bem grandes e foi necessário adicionar `...` para que eles pudessem se adequar ao livro, mas ainda assim é possível ver as informações que precisamos. O objeto `req` possui as informações sobre a requisição, o `route` é a rota raiz, que estamos usando ao acessar *https://desconstruindoaweb.com.br/*, e o `route.app` é um `ActionDispatch::Routing::RouteSet::Dispatcher`.

O `req.path_parameters` foi a action e o controller que ele encontrou baseando-se nas rotas. Agora que ele já tem a informação do controller, o objeto `req` também consegue acesso a classe dele:

```
# => (byebug) req.controller_class
HotsiteController
```

Essa classe é usada no `ActionDispatch::Routing::RouteSet::Dispatcher` logo após a execução do método `serve`. Vamos analisar as partes importantes do código:

```
# https://github.com/rails/rails/blob/v5.0.0/actionpack/lib/action
_dispatch/routing/route_set.rb#L32
def serve(req)
  params     = req.path_parameters
  controller = controller req
  res        = controller.make_response! req
  dispatch(controller, params[:action], req, res)

  # ...
end

def controller(req)
  req.controller_class
end
```

O método `serve` precisa juntar as informações sobre o controller, a action, a requisição e a resposta para enviar para o método `dispatch`. Em nosso caso, os valores serão: `HotsiteController` como controller, `"index"` como action, a conexão que estamos acompanhando há um tempo como `req`, e um novo objeto `ActionDispatch::Response` como resposta.

Esses valores são encaminhados para o método `dispatch` da classe `ActionDispatch::Routing::RouteSet`. Esse método será responsável por encaminhar os mesmos parâmetros para o método `dispatch` do nosso controller:

```
# https://github.com/rails/rails/blob/v5.0.0/actionpack/lib/action
_dispatch/routing/route_set.rb#L49
```

```
def dispatch(controller, action, req, res)
  controller.dispatch(action, req, res)
end
```

Chegando ao controller

Descobrimos que o método `dispatch` está sendo executado no nosso controller, mas sabemos que não adicionamos nenhum método com esse nome lá. Isso está com cara de *magia* do Rails e, como não gostamos de ficar sem entender esses truques, vamos atrás do método `dispatch`.

Se inspecionarmos aquele `controller` utilizando o `byebug`, podemos conseguir os módulos e classes que têm relação com o nosso controller e que possuem o método `dispatch`. Para isso, vamos pegar os *ancestrais* do nosso *controller* e *selecionar* apenas os que respondem para o método `dispatch`:

```
# => (byebug) controller.ancestors.select{|ctrlr| ctrlr.respond_to
?(:dispatch) }

[HotsiteController, ApplicationController, ActionController::Base,
ActionController::Metal]
```

O controller `HotsiteController` herda os métodos do `ApplicationController`, que por sua vez herda de `ActionController::Base`, que herda de `ActionController::Metal`.

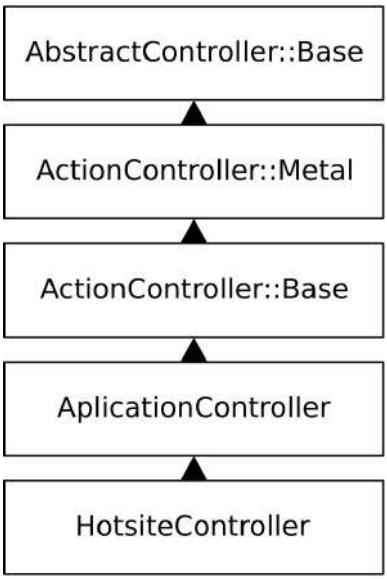


Figura 8.7: Herança do controller do desconstruindoaweb no Rails

Vamos ver a implementação do método `dispatch` no `ActionController::Metal`:

```
# https://github.com/rails/rails/blob/v5.0.0/actionpack/lib/action
_controller/metal.rb#L187
def dispatch(name, request, response) #:nodoc:
  set_request!(request)
  set_response!(response)
  process(name)
  request.commit_flash
  to_a
end
```

A parte que nos interessa nesse código é a chamada do método `process`, que passa como parâmetro o nome da *action*. Essa action será mapeada para um método do controller. O método `process` não está implementado no `ActionController::Metal`, mas sim em uma camada acima chamada de `AbstractController`.

Podemos ver a implementação do método `process` no `AbstractController::Base`:

```
# https://github.com/rails/rails/blob/v5.0.0/actionpack/lib/abstra
ct_controller/base.rb#L117
def process(action, *args)
  @_action_name = action.to_s

  unless action_name = _find_action_name(@_action_name)
    raise ActionNotFound, "The action '#{action}' could not be fou
nd for #{self.class.name}"
  end

  @_response_body = nil

  process_action(action_name, *args)
end
```

Ele faz a verificação de existência da action e, caso exista, ele chama o método `process_action` :

```
# https://github.com/rails/rails/blob/v5.0.0/actionpack/lib/abstra
ct_controller/base.rb#L117
def process_action(method_name, *args)
  send_action(method_name, *args)
end

alias send_action send
```

Por incrível que pareça, é só uma execução do método `send` passando a *action* que tem o mesmo nome do método que definimos no nosso controller. O método `send` recebe um *Symbol*, ou uma *String* do Ruby, e executa o método daquele nome. No caso dessa requisição, estamos fazendo a chamada do método `index` no controller `HotsiteController` .

A renderização da *view*

O controller da aplicação é o responsável por fazer a ligação da camada de *model* e *view* do MVC. Caso não exista chamadas para a camada de model, o controller apenas renderiza a camada de view diretamente. O `desconstruindoaweb.com.br` não possui models devido a natureza da aplicação, portanto, essa camada não vai fazer parte do nosso estudo.

UM POUCO SOBRE A CAMADA DE MODEL

A camada de *model* é responsável por conter os objetos de domínio da aplicação. Por padrão, o Rails mantém nessa camada os objetos que representam as entidades do banco de dados.

O seu estudo demandaria um capítulo a parte. Nele, poderíamos estudar desde a montagem da query no `ActiveRecord` até os dados retornados do banco de dados como objeto.

Vimos na introdução ao Ruby on Rails neste capítulo que ele usa um conceito de convenção sobre configuração. Agora veremos um exemplo de como isso funciona.

O Rails assume que, caso não tenha uma configuração especificada para renderizar um determinado arquivo, ele vai renderizá-lo baseado no controller e action recebidos. No nosso caso, estamos acessando a raiz do projeto, que está mapeada nas rotas:

```
# https://github.com/PotHix/desconstruindoaweb.com.br/blob/master/
config/routes.rb#L6
root 'hotsite#index'
```

Quando acessamos o site, vamos diretamente para o controller `HotsiteController` na action `index`. Sabendo disso, o Rails carregará automaticamente o template que está em `app/views/hotsite/index.html.erb`, caso não tenha nada especificado.

Vimos há pouco que o método `send_action` é executado para chegar até o método do nosso controller, mas não existe apenas um

`send_action` nessa cadeia de objetos. Se procurarmos no código do Rails pelos arquivos que contêm referência para a definição do método `send_action`:

```
$ grep "def.*send_action" --include \*.rb -rl

actionpack/lib/action_controller/metal/basic_implicit_render.rb
```

Encontraremos uma definição dele no arquivo `basic_implicit_render.rb` também. A implementação é pequena:

```
# https://github.com/rails/rails/blob/v5.0.0/actionpack/lib/action
_controller/metal/basic_implicit_render.rb
module ActionController
  module BasicImplicitRender # :nodoc:
    def send_action(method, *args)
      super.tap { default_render unless performed? }
    end

    def default_render(*args)
      head :no_content
    end
  end
end
```

Vamos dar uma olhada nos `ancestors` do nosso controller como fizemos na seção passada, mas dessa vez sem nenhum filtro:

```
# => (byebug) controller.ancestors
[HotsiteController, #<Module:0x007f64fc368c70>, ApplicationControl
ler, #<Module:0x007f64fc356110>, #<Module:0x000000030b9598>, #<Mod
ule:0x000000030b95c0>, ActionController::Base, ActionDispatch::Rou
ting::RouteSet::MountedHelpers, ActionController::ParamsWrapper, A
ctionController::Instrumentation, ActionController::Rescue, Action
Controller::HttpAuthentication::Token::ControllerMethods, ActionCo
ntroller::HttpAuthentication::Digest::ControllerMethods, ActionCon
troller::HttpAuthentication::Basic::ControllerMethods, ActionContr
oller::DataStreaming, ActionController::Streaming, ActionControlle
r::ForceSSL, ActionController::RequestForgeryProtection, AbstractC
ontroller::Callbacks, ActiveSupport::Callbacks, ActionController::
FormBuilder, ActionController::Flash, ActionController::Cookies, A
ctionController::StrongParameters, ActiveSupport::Rescuable, Actio
nController::ImplicitRender, ActionController::BasicImplicitRender
, ActionController::MimeResponds, AbstractController::Caching, Abs
```

```
tractController::Caching::ConfigMethods, AbstractController::Cachi
ng::Fragments, ActionController::Caching, ActionController::EtagWi
thTemplateDigest, ActionController::ConditionalGet, ActionControll
er::Head, ActionController::Renderers::All, ActionController::Rend
erers, ActionController::Rendering, ActionView::Layouts, ActionVie
w::Rendering, ActionController::Redirecting, ActiveSupport::Benchm
arkable, AbstractController::Logger, ActionController::UrlFor, Abs
tractController::UrlFor, ActionDispatch::Routing::UrlFor, ActionDi
spatch::Routing::PolymorphicRoutes, ActionController::Helpers, Abs
tractController::Helpers, AbstractController::AssetPaths, Abstract
Controller::Translation, AbstractController::Rendering, ActionView
::ViewPaths, #<Module:0x007f64fc0b37f0>, ActionController::Metal,
AbstractController::Base, ActiveSupport::Configurable, ActiveSuppo
rt::ToJsonWithActiveSupportEncoder, Object, PP::ObjectMixin, Activ
eSupport::Dependencies::Loadable, V8::Conversion::Object, JSON::Ex
t::Generator::GeneratorMethods::Object, ActiveSupport::Tryable, Ke
rnel, BasicObject]

# => (byebug) controller.ancestors.size
66
```

Esse é um objeto bem grande, contendo **66** ancestrais! Podemos ver nessa lista que o `ActionController::ImplicitRender` aparece várias camadas antes do `AbstractController::Base`, que é onde está implementado a versão que vimos para o controller. Por estar antes na hierarquia de chamadas, o `ActionController::ImplicitRender` será chamado primeiro.

Ele vai chamar o método `super` para passar o controle para o próximo da hierarquia e, na volta, vai executar o `default_render` se for necessário.

Uma pergunta comum que surge neste momento é:

```
Mas não era o ActionController::BasicImplicitRender que implementa
va esse     método?
```

Sim, ele implementa, mas é uma versão básica que é usada no `ActionController::ImplicitRender`. Esse render possui a implementação do `default_render` que é chamado no `send_action`:

```
# https://github.com/rails/rails/blob/v5.0.0/actionpack/lib/action
```

```
_controller/metal/implicit_render.rb#L32
include BasicImplicitRender

def default_render(*args)
  if template_exists?(action_name.to_s, _prefixes, variants: reque
st.variant)
    render(*args)
    # ...
  end
end
```

Esse é o método que vai executar a renderização da view com os parâmetros necessários.

8.4 A APLICAÇÃO

Depois de passar por todo o código do Rails, a requisição chega no controller da aplicação do `desconstruindoaweb.com.br`.

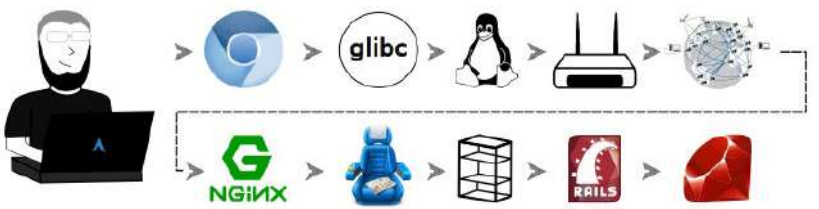


Figura 8.8: Enfim o código da aplicação

O controller da nossa aplicação é bem simples:

```
# https://github.com/PotHix/desconstruindoaweb.com.br/blob/master/
app/controllers/hotsite_controller.rb
class HotsiteController < ApplicationController
  def index
  end
end
```

Esse controller possui apenas um método cuja finalidade é entregar a página inicial da aplicação. Como o nome do método é `index`, ele renderizará o arquivo `app/views/hotsite/index.html.erb` por padrão.

O arquivo `index.html.erb` contém a parte interna da página que se juntará ao arquivo padrão de layout para montar a página completa. O arquivo padrão de layout fica em `app/views/layouts/application.html.erb` e possui o código `yield`, que vai receber o conteúdo da view e renderizá-lo nesse lugar.

POR QUE USAR RAILS NESSE CASO?

Não faz sentido usar Rails se o site é bem pequeno, e um conteúdo estático resolveria.

Sim, é verdade. Existem ferramentas mais simples que o Rails para fazer essa aplicação, diminuindo o custo com servidor e fazendo a manutenção do site ser mais simples. Duas dessas ferramentas que poderiam ser utilizadas para essa finalidade são o Middleman (http://middlemanapp.com) e o Jekyll (http://jekyllrb.com). Ambas resolvem muito bem o problema e têm custo de manutenção baixo.

O Rails foi usado nesse caso por motivos didáticos. Utilizando uma aplicação bem simples como essa, podemos estudar o framework sem nos preocuparmos com a complexidade da aplicação.

Os arquivos `.erb` possuem código Ruby entre as tags `<%` e `%>`. Essas tags são interpretadas e o código Ruby é executado para retornar o que deve ir para o cliente. Um exemplo disso é o código `javascript_include_tag` que está presente no arquivo `application.html.erb`:

```
# https://github.com/PotHix/desconstruindoaweb.com.br/blob/master/
app/views/layouts/application.html.erb#L45
```

```
<%= javascript_include_tag 'application' %>
```

Ele vai ser interpretado e se tornará uma tag HTML parecida com essa na página final:

```
<script src="/assets/application-aab96e7e33efb6dec3505a3a6b3ce02b1
d184dea269f6a0f7d88e96f8f0be70f.js"></script>
```

> **O PROCESSO DE COMPRESSÃO DE ASSETS**
>
> A tag `javascript_include_tag` inclui o arquivo que foi gerado no processo de compilação de assets. Esse processo usa todos os arquivos descritos em > `application.css` e `application.js`, fazendo uma união e compactação e gerando esse arquivo longo e com nome único.
>
> Esse processo é muito útil atualmente, pois ajuda na utilização de cache agressivo e menor transferência de arquivos, impactando diretamente no carregamento do site. O problema da transferência vai ser resolvido quando a migração dos sites para HTTP/2 estiver completa, pois ele utiliza multiplexação para enviar vários arquivos pela mesma conexão.

Nesse momento, todos os códigos necessários foram executados nas camadas de model, controller e view. Com isso, o HTML está pronto para ser enviado de volta para o cliente.

8.5 O RETORNO PARA O NAVEGADOR

A requisição é finalmente processada depois de passar pelas inúmeras camadas que mencionamos até este momento. Cada um dos passos que estudamos mostrava a entrada e saída dos dados, mas vamos juntar tudo o que vimos e acompanhar a trajetória deles

saindo do servidor e voltando para o cliente.

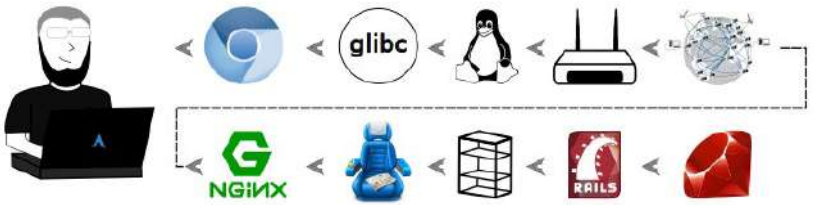


Figura 8.9: A viagem de volta para o navegador

O Rails já tem tudo o que precisa para responder o protocolo do Rack. Se olharmos no `Journey::Router`, que define o método `serve` visto no começo do estudo sobre o Rails, podemos ver onde o contrato com o Rack é respondido:

```
# https://github.com/rails/rails/blob/v5.0.0/actionpack/lib/action
_dispatch/journey/router.rb#L48
return [status, headers, body]
```

Vamos conferir mais de perto para ver do que esse objeto é composto:

```
# => (byebug) [status, headers, body]
[200, {"X-Frame-Options"=>"SAMEORIGIN", "X-XSS-Protection"=>"1; mo
de=block", "X-Content-Type-Options"=>"nosniff", "Content-Type"=>"t
ext/html; charset=utf-8"}, "<!doctype html>\n<!--[if IE]><![endif]
-->\n<html lang=\"pt-br\" ..."]

# => (byebug) body.class
ActionDispatch::Response::RackBody
```

O status de retorno do *HTTP* foi 200, o que nos diz que a requisição foi um sucesso. Apenas os cabeçalhos `X-Frame-Options`, `X-XSS-Protection`, `X-Content-Type-Options` e `Content-Type` foram adicionados pelo Rails até o momento nessa requisição. O corpo da requisição é o HTML que esperamos, mas dentro de um objeto `RackBody` do `ActionDispatch::Response`.

Se olharmos a resposta no navegador, veremos que há vários

outros cabeçalhos como resposta. Esse fato diz que ainda há muita coisa para acontecer com essa requisição.

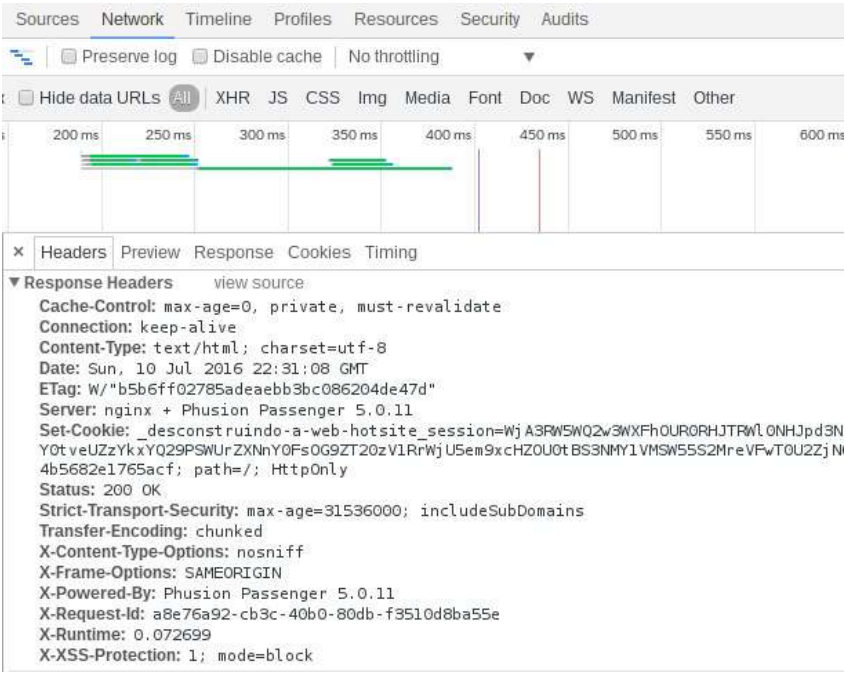


Figura 8.10: Cabeçalhos na resposta da requisição para o desconstruindoaweb.com.br

Um exemplo de adição de cabeçalhos está no middleware `Rack::ETag` do Rack. Esse Middleware é adicionado pelo Rails como parte do processamento de uma requisição. Para ver o que ele faz, vamos dar uma olhada no código do Rack:

```
# https://github.com/rack/rack/blob/master/lib/rack/etag.rb#L14

class ETag
  ETAG_STRING = Rack::ETAG
  DEFAULT_CACHE_CONTROL = "max-age=0, private, must-revalidate".fr
eeze

  def initialize(app, no_cache_control = nil, cache_control = DEFA
ULT_CACHE_CONTROL)
    @app = app
    @cache_control = cache_control
```

```
    @no_cache_control = no_cache_control
  end

  def call(env)
    status, headers, body = @app.call(env)

    if etag_status?(status) && etag_body?(body) && !skip_caching?(
headers)
      original_body = body
      # ...
```

A classe `ETag` implementa o construtor e o método `call`, que são o contrato de um middleware. A parte que nos interessa está no método `call`, onde podemos ver que ele primeiro repassa a chamada para o próximo middleware via `@app.call(env)`, para depois fazer o que precisa fazer. Esse é o motivo de não termos a requisição completa, pois ainda há código a ser executado antes dela ser entregue.

A requisição volta para o Passenger no arquivo `thread_handler_extension.rb` que, como vimos no começo deste capítulo, estava aguardando o retorno da conexão:

```
# https://github.com/phusion/passenger/blob/release-5.0.11/lib/phu
sion_passenger/rack/thread_handler_extension.rb#L94
begin
  status, headers, body = @app.call(env)
rescue => e
```

O código vai continuar o processamento e chamar o método `process_body`:

```
# https://github.com/phusion/passenger/blob/release-5.0.11/lib/phu
sion_passenger/rack/thread_handler_extension.rb#L143
process_body(env, connection, socket_wrapper, status.to_i, is_head
_request, headers, body)
```

O método `process_body` será o responsável por escrever de volta no socket via `writev`:

```
# https://github.com/phusion/passenger/blob/release-5.0.11/lib/phu
sion_passenger/rack/thread_handler_extension.rb#L281
case message_length_type
```

```
when :content_length
  if body.is_a?(Array)
    connection.writev2(headers_output, body)
  else
    connection.writev(headers_output)
    body.each do |part|
      connection.write(part.to_s)
    end
  end
```

Como vimos no capítulo passado, os dados que vão seguir via `writev` são os dados da requisição:

```
[pid 32124] writev(13, [{"HTTP/1.1 200 Whatever\r\n", 23}, {"X-Fra
me-Options", 15}, {": ", 2}, {"SAMEORIGIN", 10}, {"\r\n", 2}, {"X-
XSS-Protection", 16}, {": ", 2}, {"1; mode=block", 13}, {"\r\n", 2
}, {"X-Content-Type-Options", 22}, {": ", 2}, {"nosniff", 7}, {"\r
\n", 2}, {"Strict-Transport-Security", 25}, {": ", 2}, {"max-age=3
1536000; includeSubDoma"..., 35}, {"\r\n", 2}, {"Content-Type", 12
}, {": ", 2}, {"text/html; charset=utf-8", 24}, {"\r\n", 2}, {"ETa
g", 4}, {": ", 2}, {"W/\"6e08a4f9906888f72c5ee4df4aa8f"..., 36}, {
"\r\n", 2}, {"Cache-Control", 13}, {": ", 2}, {"max-age=0, private
, must-revalid"..., 35}, {"\r\n", 2}, {"Set-Cookie", 10}, {": ", 2
}, {"_desconstruindo-a-web-hotsite_se"..., 342}, ...], 43) = 777
[pid 32124] ppoll([{fd=13, events=POLLOUT}], 1, NULL, NULL, 8) = 1
 ([{fd=13, revents=POLLOUT}])
[pid 32124] writev(13, [{"250d", 4}, {"\r\n", 2}, {"<!doctype html
>\n<!--[if IE]><![e"..., 9485}, {"\r\n", 2}], 4) = 9493
[pid 32124] write(13, "0\r\n\r\n", 5)   = 5
```

O NGINX recebe a conexão do Passenger e escreve no socket que estava aguardando a resposta da conexão. Como o navegador enviou no cabeçalho `Connection: keep-alive`, o NGINX manterá a conexão aberta por mais um tempo para receber novas requisições sem precisar refazer o processo de *three-way handshake* novamente.

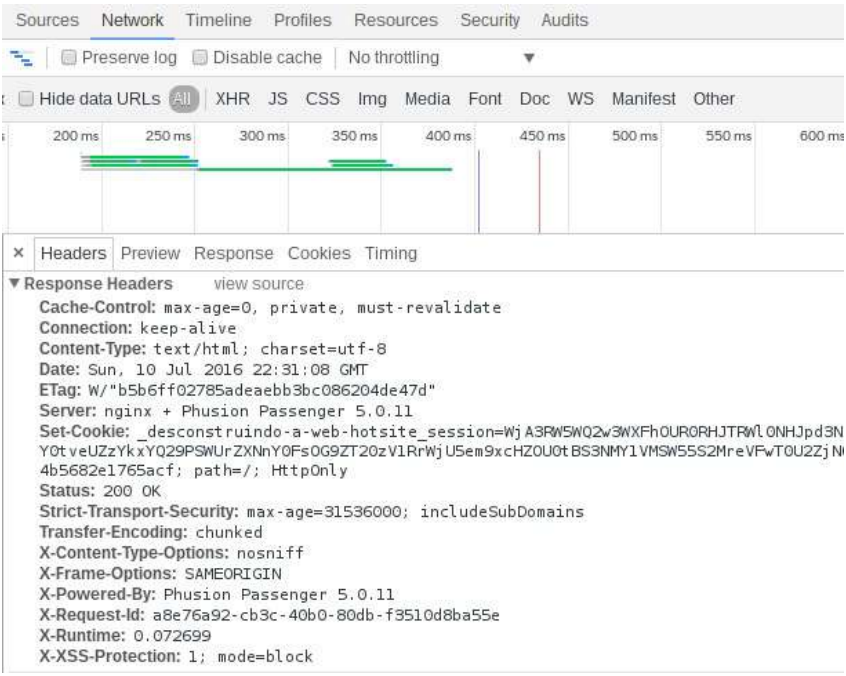


Figura 8.11: Cabeçalhos do envio da requisição para o desconstruindoaweb.com.br

Os dados vão sair do servidor seguindo os passos da nossa conhecida *camada Ozzy*, até passar pela placa de rede e sair do datacenter da nossa empresa de hospedagem. Os dados farão um caminho parecido com o que vimos na saída do comando `traceroute` no *capítulo 6*, passando pelos dispositivos de rede da empresa de hospedagem, ISP deles, nosso ISP, até chegar ao roteador local. Localmente, a conexão vai brigar com outros dispositivos da frequência 2.4GHz enquanto viaja pelas ondas de rádio do Wi-Fi até o computador local.

Já no computador local, a requisição será recebida na placa de rede wireless e seguirá caminho pela *camada Ozzy* local. O pacote vai ser montado até virar a chamada HTTP que o navegador estava esperando. Esse pacote vai ser interpretado para se tornar as informações que serão renderizadas na tela.

O processo de interpretação do que vem via HTTP não é algo simples. No próximo capítulo, estudaremos o que o navegador precisa fazer para mostrar a página que estamos esperando ver.

8.6 RESUMO

O Rack faz a conexão entre o servidor de aplicação e o framework. O Passenger, que é o servidor de aplicação que estamos estudando, usa um sistema que carrega a aplicação em um processo separado e faz clones dele para criar novos, ganhando tempo, memória e processamento. Dentro desses processos, ele utiliza um sistema de threads para melhorar a quantidade de requisições que um processo pode receber. Esse sistema de threads fala com a aplicação por meio da interface do Rack, que por sua vez fala com o framework da aplicação.

Estamos usando o Ruby on Rails, conhecido como Rails, como framework para a aplicação. O Rails usa o Rack para fazer a interface entre a aplicação e o servidor de aplicação. Uma conexão recebida via Rack vai passar pelo conjunto de middlewares do Rack que o Rails possui, sendo que o último deles é uma ligação com o sistema de rotas do Rails.

O Rails usa o sistema MVC, portanto, o sistema de rotas faz a ligação do que foi pedido na requisição, com o método do *controller* que deve ser executado para esse caso. Junto com esse código, também é executado uma chamada para renderizar a *view*, que executará todos os códigos Ruby que estão dentro dos arquivos `html.erb`. O controller faz as chamadas para códigos que estão no *model*, ou em outras bibliotecas de código, e faz a interface deles com o que precisa ser feito na view.

Com as *views* processadas, a requisição começa a fazer o caminho de volta para o cliente que a solicitou. O primeiro passo é

sair do Rails, passando pelos middlewares que ainda estavam aguardando para adicionar informações na resposta, para depois chegar ao Rack. O servidor de aplicação, que estava aguardando um retorno, recebe o resultado da requisição, seguindo a estrutura do Rack, e faz as transformações necessárias para enviar para o servidor web via unix socket.

O servidor web recebe a conexão e escreve no socket do cliente que estava aguardando a resposta. Essa requisição vai passar por toda a *camada Ozzy* do servidor para sair pela placa de rede. Vai sair pelo datacenter da empresa de hospedagem, passando pelos dispositivos de rede até chegar ao roteador local, onde será enviado via ondas de rádio para o computador que fez a requisição.

O computador local vai receber o sinal, que passará pela *camada Ozzy* local, até se tornar um pacote HTTP formado. Esse pacote será entregue ao navegador para que o conteúdo final seja interpretado e mostrado ao usuário final.

8.7 REFERÊNCIAS

1. Rack — http://rack.github.io/
2. O sistema de criação de processos do Passenger — https://www.phusionpassenger.com/library/indepth/ruby/spawn_methods/#the-smart-spawning-method
3. Código do Passenger responsável pela criação de novos processos — https://github.com/phusion/passenger/blob/release-5.0.11/helper-scripts/rack-preloader.rb
4. Código do Passenger que recebe as requisições — https://github.com/phusion/passenger/blob/release-5.0.11/lib/phusion_passenger/request_handler.rb#L175

5. Código de threads que recebe as conexões — https://github.com/phusion/passenger/blob/release-5.0.11/lib/phusion_passenger/rack/thread_handler_extension.rb#L94

6. Site oficial do Ruby on Rails — http://rubyonrails.org/

7. Código do `desconstruindoaweb.com.br` — https://github.com/PotHix/desconstruindoaweb

8. Código da versão do Rails que estamos utilizando para estudo — https://github.com/rails/rails/tree/v5.0.0

Capítulo 9

DE VOLTA AO NAVEGADOR

Depois de viajar na velocidade da luz por cabos de fibra ótica e passar por diversos dispositivos de rede, a requisição chegou ao destino, foi processada e voltou como o navegador esperava. Alguns podem achar que a nossa jornada termina aqui, afinal, o navegador já recebeu a resposta da requisição que pediu, mas ainda há muito a se fazer.

Neste capítulo, continuaremos o fluxo que começamos no início da nossa jornada. Vamos estudar a partir do momento que o navegador recebe resposta, passando pelo processamento dos arquivos recebidos, até o momento em que a página está completamente carregada na tela do usuário.

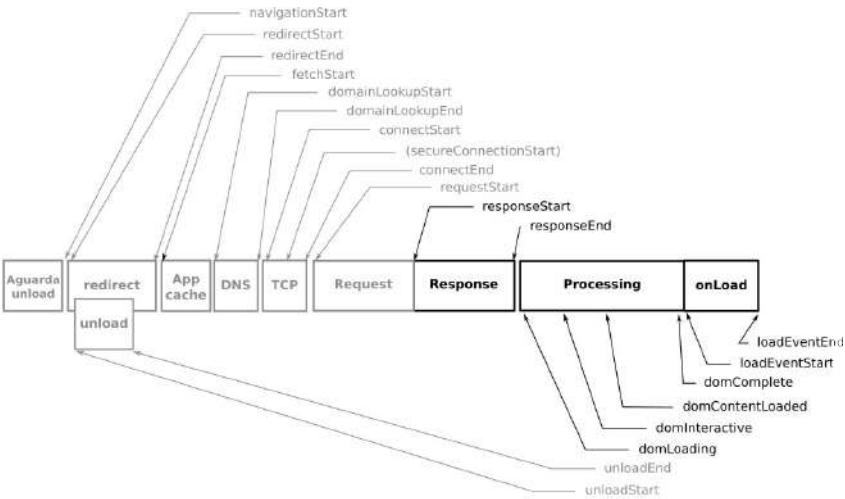


Figura 9.1: A resposta da requisição no navegador. Baseado na imagem disponível em https://www.w3.org/TR/navigation-timing

9.1 O RECEBIMENTO DOS DADOS

O computador local vai receber os dados que vieram da *camada Ozzy*, e encaminhará o pacote HTTP para o navegador. Ao receber esse pacote, o navegador começa o seu processamento para extrair as informações necessárias para construir a página.

O pacote HTTP de resposta vai trazer as informações como status da resposta, cookies e cabeçalhos. Podemos vê-los facilmente com o comando `curl`, utilizando o comando `grep` para filtrar apenas os dados que vieram no retorno:

```
$ curl -vvv https://desconstruindoaweb.com.br/ |& grep "< "

< HTTP/1.1 200 OK
< Content-Type: text/html; charset=utf-8
< Transfer-Encoding: chunked
< Connection: keep-alive
< Status: 200 OK
< Cache-Control: max-age=0, private, must-revalidate
< Strict-Transport-Security: max-age=31536000; includeSubDomains
< ETag: W/"494584c41c6c929cdc582758a5fce1c0"
< X-Frame-Options: SAMEORIGIN
```

```
< X-XSS-Protection: 1; mode=block
< X-Content-Type-Options: nosniff
< X-Runtime: 0.003341
< X-Request-Id: 70c7d61d-f6f5-451d-a294-04aa5c102d52
< Date: Thu, 14 Jul 2016 13:02:24 GMT
< Set-Cookie: _desconstruindo-a-web-hotsite_session=3Ss4StR1nG03hc
0MpL3t4M3nT30AL3aToHr1400C3g3l401PV3VyL1R6N3hJbjQ3V2dkMXZJV01jalBt
YkdBc0E1bXRLTDgvRmhkUFc4OTMwUG1iV3hBcDhSAIxDWXpJaUFxM2Z4MWFubS8rQU
9lRC3g3l4rN3FCVmtxYXZmZ2U5Nm9pR4nD0MJzZFkvQ2J1aWxLSEVKWEFBPT0P0Th1
XtLUJucEJhVGgwQ1liUGw1d1VkNFowRFE9PQ%3D%3D--9eb3e18489ae4185379cf6
dd7e5371f656e5c245; path=/; HttpOnly
< X-Powered-By: Phusion Passenger 5.0.11
< Server: nginx + Phusion Passenger 5.0.11
<
```

Além dos cabeçalhos e status, o pacote também retorna o conteúdo da página HTML que requisitamos, obviamente:

```
$ curl https://desconstruindoaweb.com.br/ 2>/dev/null | head -n5

<!doctype html>
<!--[if IE]><![endif]-->
<html lang="pt-br" xmlns:og="http://ogp.me/ns#" itemscope itemtype
="http://schema.org/Book">
<head>
  <meta charset="utf-8">
```

Como vimos no resultado do comando `curl`, esses valores chegam em texto para o navegador. Cabe ao navegador entender esse texto e transformá-lo nos elementos visuais que estamos acostumados.

A partir de agora, vamos ver a continuação do que vimos no *capítulo 1*. No começo do nosso estudo, nós vimos como o navegador faz para enviar os pacotes para fora do computador local. Agora que estamos na fase final, veremos novamente essa infraestrutura, mas do ponto de vista de um pacote que está chegando.

O Chromium vai usar novamente sua implementação do protocolo HTTP, que fica na parte de rede[1], para fazer a interpretação do protocolo e transformar o texto em uma estrutura

conhecida. Ao transformar os dados recebidos em informações conhecidas, o navegador interpretará os cabeçalhos e executará as ações que eles demandam.

Quando estudamos a estrutura do Chromium para enviar os dados para a rede, vimos que antes de ir buscar conteúdo na rede, ele primeiro verifica se esse conteúdo já não está disponível no cache que ele possui em disco. Agora que o pacote está voltando da internet, o navegador sabe que não havia cache para o conteúdo ou para parte dele. Isso quer dizer que as classes de `HttpCache` vão fazer o trabalho de guardá-los em disco, respeitando os cabeçalhos `Cache-Control`, `Expires` e `Etag`.

Ao receber os dados da classe de cache e passar pelas diversas classes do código de rede do Chromium, a *thread* de *I/O* vai encaminhar os dados recebidos para a próxima camada.

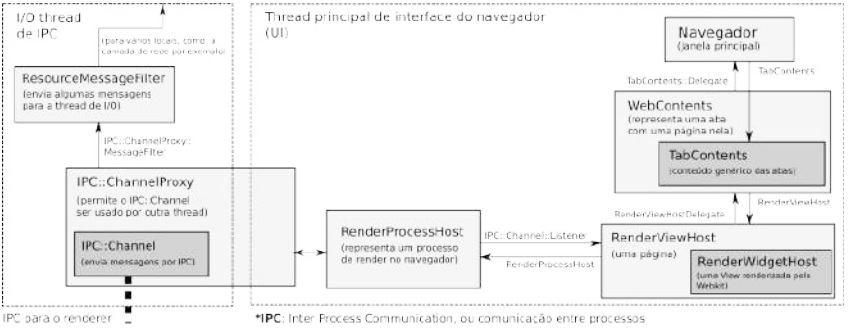


Figura 9.2: Saindo da thread de I/O e indo para a renderização. Fonte: http://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome

Assim que essa camada recebe o primeiro pacote da thread de I/O, ela começa a preparar o navegador para o carregamento da página e processo de renderização.

9.2 A RENDERING ENGINE

O processo de renderização é feito por um sistema chamado *rendering engine*. O trabalho desse sistema é receber o que está vindo da camada de rede, interpretar e transformar em elementos visuais que serão mostrados na tela.

Para fazer isso, a rendering engine passa pelos passos de parse de HTML para construção da DOM tree, construção da Render tree, Layout da Render tree e Painting da Render tree. Vamos passar por cada um deles durante este capítulo, enquanto estudamos a rendering engine do Chromium.

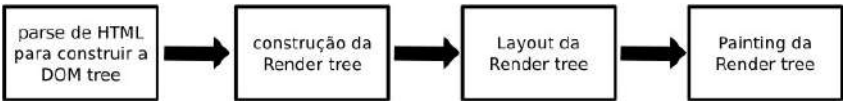


Figura 9.3: Fluxo de uma engine de rendering. Baseado no trabalho da Tali Garsiel: http://taligarsiel.com/Projects/howbrowserswork1.htm

O Chromium utiliza uma rendering engine chamada Blink[2], que é baseada no Webkit[3].

Curiosidades sobre a Blink

O Chromium começou utilizando apenas o Webkit como rendering engine, pois ele atendia bem as necessidades do projeto. Com o tempo, o Chromium começou a usar um sistema diferente de arquitetura, que dependia da utilização de vários processos se comunicando entre si. Esse tipo de implementação não é comum para o Webkit, e muito código estava sendo desenvolvido para manter a implementação do Chromium.

O fato de manter várias arquiteturas estava aumentando a complexidade no desenvolvimento de ambos. Com isso, em 2013, a equipe do Chromium decidiu fazer um *fork* do projeto, continuando o desenvolvimento de sua própria maneira, assim dando início a uma nova rendering engine, a Blink[4].

A equipe do Chromium possui várias demandas de modificação na engine Blink, mas o fluxo principal ainda é parecido com o do Webkit. As principais modificações são referentes ao código multiprocessos que o Chromium usa e às otimizações de códigos antigos do Webkit. Esses códigos mais antigos não podiam ser modificados devido a compatibilidade com sistemas mais antigos que o Webkit precisa manter.

Em vários pontos, usaremos as documentações e diagramas do Webkit como exemplo, graças a essa semelhança.

9.3 PARSE DE HTML

A primeira coisa que acontece é o carregamento dos dados.

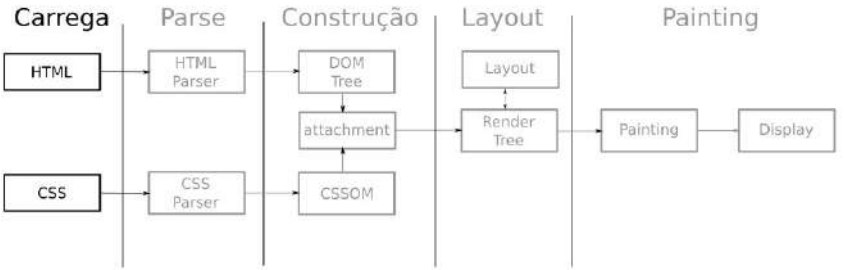


Figura 9.4: Carregamento dos dados

A rendering engine não aguarda os dados chegarem completamente para começar a renderização. De acordo com a pesquisa da Tali Garsiel[5], os dados são recebidos pela rendering engine em pedaços de 8K. Assim que os primeiros pacotes chegam, eles são passados diretamente para o processo de *parse*.

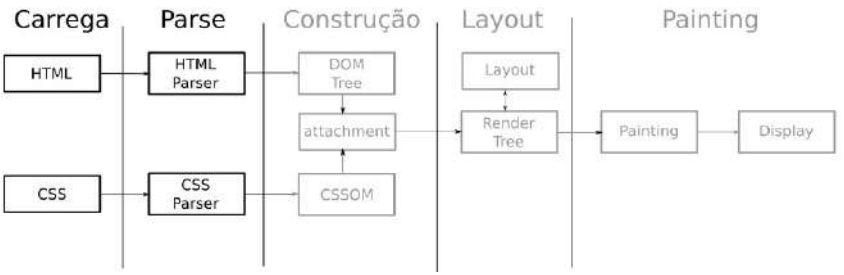


Figura 9.5: Parse dos dados

Como já foi mencionado, o arquivo HTML que a nossa requisição para o `desconstruindoaweb.com.br` está retornando é apenas texto. O navegador não pode jogar esse texto para o usuário final, ele precisa interpretar e aplicar todas as regras para que a página se transforme no que acostumados a ver. Para que ele possa entender cada elemento da página, o navegador precisa entender o que o texto recebido significa.

O processo de parse do HTML é especificado[6] pela W3C, para que todos os navegadores possam implementar o mesmo

comportamento.

> **Curiosidade sobre a especificação**
>
> No passado, cada navegador fazia a sua própria implementação do HTML. Isso tornava a internet um ambiente hostil para se desenvolver. Quem utiliza a internet há algum tempo, deve lembrar do problema que era usar um navegador diferente do Internet Explorer 6 na época.
>
> O Internet Explorer 6, ou *IE6* para os íntimos, fazia parse de HTML do seu próprio jeito. Isso causava uma dificuldade enorme para os desenvolvedores de aplicações para a internet, pois uma aplicação dificilmente funcionava bem em mais de um navegador sem muito esforço.
>
> Ao perceber esse problema, começaram as iniciativas para definir um padrão que todos os navegadores pudessem seguir para facilitar a vida dos desenvolvedores. Os membros da **w3.org** são desenvolvedores de várias empresas que desenvolvem navegadores, sendo que a organização tem como principal atividade a padronização dessas implementações.

A especificação do HTML define um fluxo para o processamento e mostra como devem ser tratados alguns casos especiais. Nesse fluxo, os dados são recebidos via rede, que no nosso caso é a thread de I/O, e encaminhados para a próxima fase. Nessa fase, o conteúdo passa pelo decodificador de bytes e pelo pré-processador, que são responsáveis por transformar os bytes em caracteres e remover o que não deveria estar lá, como o `CRLF` , por exemplo[7].

Agora que a informação já pode ser lida com mais facilidade, ela é encaminhada para o criador de tokens, conhecido como *tokenizer*, em inglês. Ele é responsável por identificar cada tag ou texto dentro do conjunto de caracteres e fazer a *construção da árvore*. A especificação do HTML mostra como cada token deve ser identificado[8] e como eles devem ser inseridos na árvore[9].

O processo de *construção da árvore* é responsável por consertar documentos HTML mal formados ou incompletos. Um documento HTML que não possua a tag `<head>`, por exemplo, vai receber uma ao passar pela máquina de estados que está implementada no processo de parse.

Há também os casos de páginas que injetam HTML utilizando a função `document.write()` dentro de uma tag `<script>`. Quando isso acontece, o retorno gerado por essa função será usado como entrada para o pré-processador, que fará o mesmo processo feito para os dados recebidos via rede.

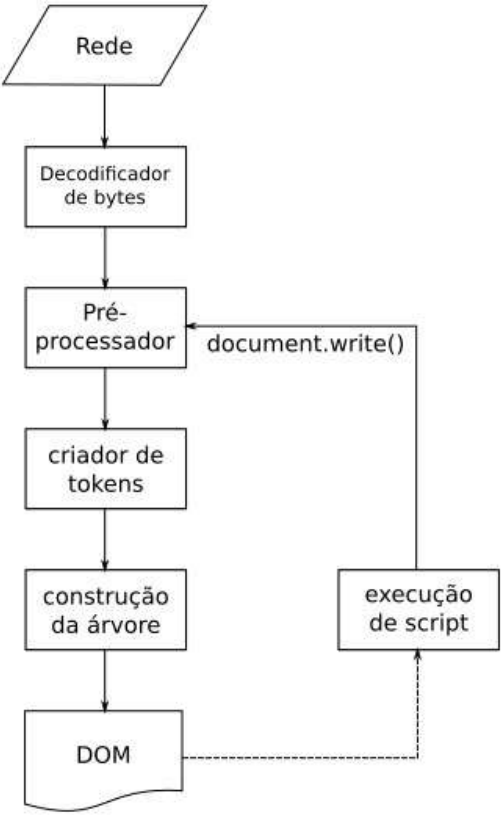


Figura 9.6: Processo de parse e document.write(). Baseado em: https://www.w3.org/TR/html5/syntax.html#overview-of-the-parsing-model

A estrutura criada a partir do processo de criação da árvore é chamada de *DOM Tree*. O acrônimo **DOM** siginifica ***D**ocument **O**bject **M**odel*, e é traduzido como *modelo de objeto de documentos*. A tradução literal é um tanto quanto estranha, portanto chamaremos apenas de DOM. Esse modelo tem como finalidade representar o HTML em forma de árvore, para possibilitar a manipulação dos elementos isoladamente. Por esse motivo, chamamos de *DOM Tree*, ou árvore do DOM.

Em vez de utilizar o código HTML complicado do `desconstruindoaweb.com.br` , vamos usar um código pequeno

como exemplo:

```
<html>
  <head>
    <link href="style.css" rel="stylesheet">
  </head>
  <body>
    <p>O <span>grande</span> livro!</p>
    <div><img src="livro.jpg"></div>
  </body>
</html>
```

Esse HTML vai gerar uma árvore pequena, tendo a tag `html` como nó principal e a tag `head` e a `body` como filhas. No `head`, teremos apenas a tag `link` e, no body, teremos uma tag `p` e `div` com seus filhos. Os textos que estão dentro da tag `p` e da tag `span` são representados como nó também, mas um nó de texto.

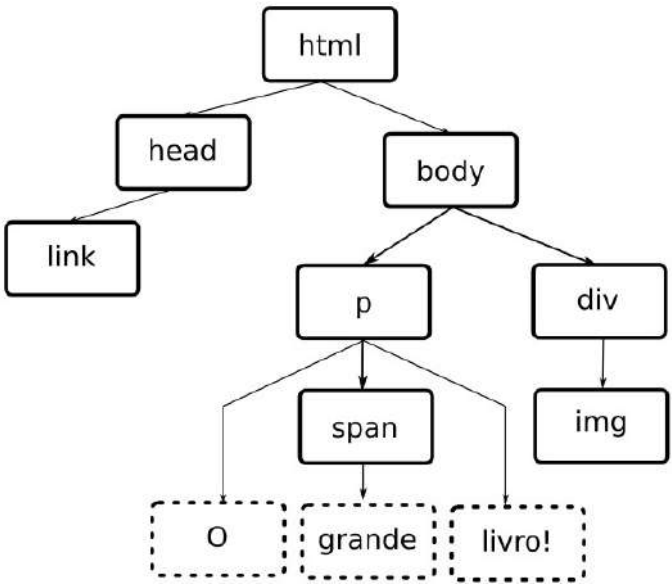


Figura 9.7: DOM Tree gerada a partir do HTML de exemplo. Baseado no exemplo do Ilya Grigorik: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model

9.4 PARSE DE CSS

Assim como o HTML, o CSS não pode ser jogado como texto para o usuário final. Ele precisa ser transformado em uma estrutura que o navegador possa manipular e entender as regras lá definidas.

Para entender o que um texto com CSS significa, o navegador utiliza um sistema de tokens parecido com o do HTML. Os dados chegam, são transformados em caracteres, passam pelo processo de transformação em tokens e geram um modelo de árvore.

O processo de criação de tokens é mostrado como *regular expressions*, que também conhecidas como *regex*. A definição pode ser vista na documentação de syntax[10] e de gramática[11] do CSS, caso esteja curioso em ver o conteúdo completo.

Um exemplo real de definição:

```
nmstart        [_a-z]|{nonascii}|{escape}
nmchar        [_a-z0-9-]|{nonascii}|{escape}
comment        \/\*[^*]*\*+([^/*][^*]*\*+)*\/
ident        -?{nmstart}{nmchar}*
name        {nmchar}+

"#"{name}        {return HASH;}
```

Nesse exemplo, está descrito como fazer parse do comentário (*comment*), identificadores (*ident*) que podem usados para classes e nomes (*name*) que são utilizados para `id`s e podem ser referenciados por `#`.

O modelo de árvore que é gerado depois da criação dos tokens é chamado de CSSOM[12], que significa ***CSS Object Model***, ou modelo de objetos CSS, em português. Ele é uma árvore assim como a DOM Tree, o que facilita para o navegador no momento de aplicar as regras.

Novamente, vamos evitar o CSS grande do `desconstruindoaweb.com.br` e utilizar uma versão pequena para fins de estudo:

```
body { font-size: 16px }
p { font-weight: bold }
span { color: red }
p span { display: none }
img { float: right }
```

Esse CSS gerará uma árvore de CSSOM, que chamaremos de **CSSOM Tree**, aplicando os estilos a cada nó da árvore, para que seja possível usar no momento de juntar com a DOM Tree. No CSS, as regras são hierárquicas, portanto, uma regra aplicada na tag `<body>` é aplicada também para todos os elementos filhos dela. Essa regra pode ser sobreposta por eles, por isso a árvore é tão importante.

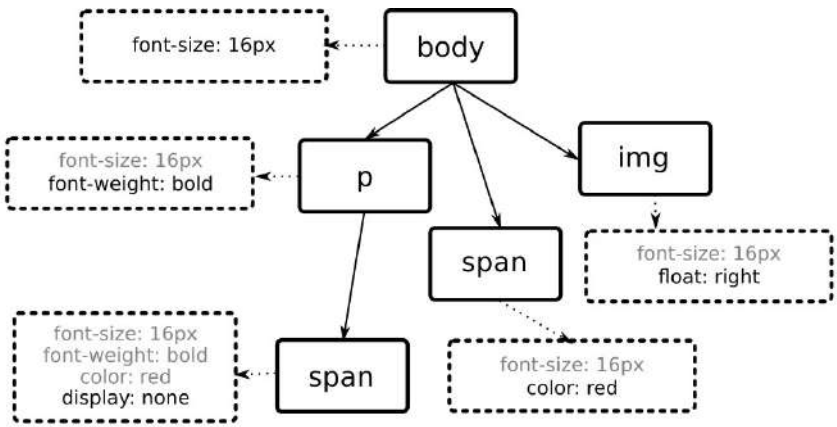


Figura 9.8: Representação da CSSOM Tree. Baseado no exemplo do Ilya Grigorik: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model

9.5 CARREGANDO ARQUIVOS EXTERNOS

Durante o processo de *parse*, o navegador pode encontrar referências para arquivos externos como arquivos JavaScript dentro da tag `<script>`, arquivos CSS dentro da tag `<link>`, ou imagens dentro da tag `<img>`. Há outros tipos de arquivos externos, mas vamos focar apenas nesses que são os mais comuns de se achar.

Alguns navegadores, como é o caso do Chromium, implementam uma otimização para carregar tudo o que pode ser carregado assincronamente antes de começar o parse. É como se fosse um "pré-parse", limitado a entender apenas o que pode ser computado assincronamente. Esse processo é executado paralelamente com o parse principal e é chamado de *speculative parsing*.

Alguns passos já possuem cache, evitando que algumas informações precisem ser processadas novamente. Apesar disso, cada um dos arquivos externos encontrados nesse passo vão passar por um processo bem parecido com o que estudamos até este momento no livro. Eles vão precisar de DNS e de um pacote HTTP, vão sair do sistema operacional, passar pela rede etc.

Imagens

As tags `<img>` são identificadas no HTML e suas respectivas imagens baixadas imediatamente. O download das imagens não interfere no parse da página, mas os navegadores têm um limite de quantos arquivos eles conseguem baixar ao mesmo tempo do mesmo domínio. No caso do Chromium, esse número é 6 por padrão, assim como a grande maioria dos outros navegadores.

Figura 9.9: Número de conexões simultâneas por domínio no Chromium. Teste feito em: http://www.browserscope.org/network/test

Nota sobre o HTTP/2

Com o HTTP/2, o problema de download de vários arquivos do mesmo domínio está resolvido. O HTTP/1.1 abre uma nova conexão para cada uma das requisições necessárias para construir a página. Já o HTTP/2 utiliza uma única conexão e faz multiplexação de todos os arquivos por ela. Com isso, o problema de quantidade de conexões por domínio desaparece.

Uma boa prática é manter os arquivos estáticos, como imagens e scripts, em um servidor de **CDN**. Uma CDN, ou **C**ontent **D**elivery **N**etwork, é uma rede de entrega de conteúdo que está distribuída em várias partes do mundo para servir o conteúdo de um local mais próximo. Utilizando uma CDN, a aplicação ganha mais 6 requisições paralelas no HTTP/1.1, e ameniza o problema com a velocidade da luz. A luz viaja a uma velocidade conhecida e, quanto mais longe estão os dados, mais latência terá e mais tempo o usuário terá de esperar.

JavaScript

Quando a tag `<script>` é encontrada, a página para de carregar até que os dados referentes a ela sejam recebidos e executados. Isso é necessário pois esses scripts geralmente alteram o DOM, o que causa problemas no processo de parse. Essa tag pode possuir código diretamente nela, como vimos no exemplo do `document.write()`, mas também pode conter um arquivo.

Em caso de arquivo externo, o navegador vai parar o parse da página até que o arquivo seja baixado e executado. Por esse motivo, há recomendações de deixar os arquivos JavaScript sempre no final

da página, de preferência utilizando JavaScript não obstrutivo[13].

As novas versões do HTML já especificam algumas formas de lidar com esse problema[14], usando `async` ou `defer`. Para scripts que não dependem de um estado específico de carregamento do DOM, o atributo `async` pode ser utilizado. Usando esse atributo, o navegador vai continuar o processo de parse enquanto baixa o arquivo JavaScript, executando-o assim que ele estiver disponível.

Caso o código JavaScript a ser baixado dependa de um estado final do DOM, o atributo `defer` pode ser usado. Com esse atributo, o script será baixado assincronamente, mas só será executado quando o DOM estiver completo.

CSS

Os arquivos CSS que estão em tags `<link>` são carregados de forma assíncrona. Esse prática é utilizada porque conceitualmente os arquivo CSS não alteram a árvore do DOM, logo, não faz sentido parar de fazer o parse enquanto eles não estão disponíveis.

É uma boa prática manter essa tag dentro da tag `<head>` para evitar que o usuário final veja uma página sem estilos por muito tempo. Porém, tudo dependerá do tempo de carregamento do arquivo.

9.6 CONSTRUÇÃO DA RENDER TREE

A etapa de construção é o momento que a DOM Tree e a CSSOM Tree vão ser combinadas para formar a Render Tree.

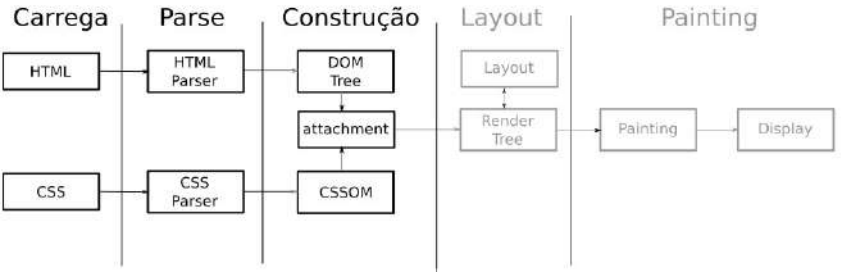


Figura 9.10: Etapa de Construção

O processo de criação da Render Tree também é conhecido como *attachment* no Webkit e na Blink. Ele é conhecido dessa maneira devido ao fato de chamar o método `attach()` nos elementos do DOM que serão inseridos.

A Render Tree é uma árvore que representa o que será mostrado na tela do navegador. O resultado dessa árvore será uma combinação dos elementos do DOM com os elementos do CSSOM, culminando nos elementos que devem ser mostrados.

Utilizando os exemplos de HTML e CSS que vimos até agora, juntamente com as suas respectivas árvores geradas, vamos ver como fica a Render Tree na *Figura 9.11*.

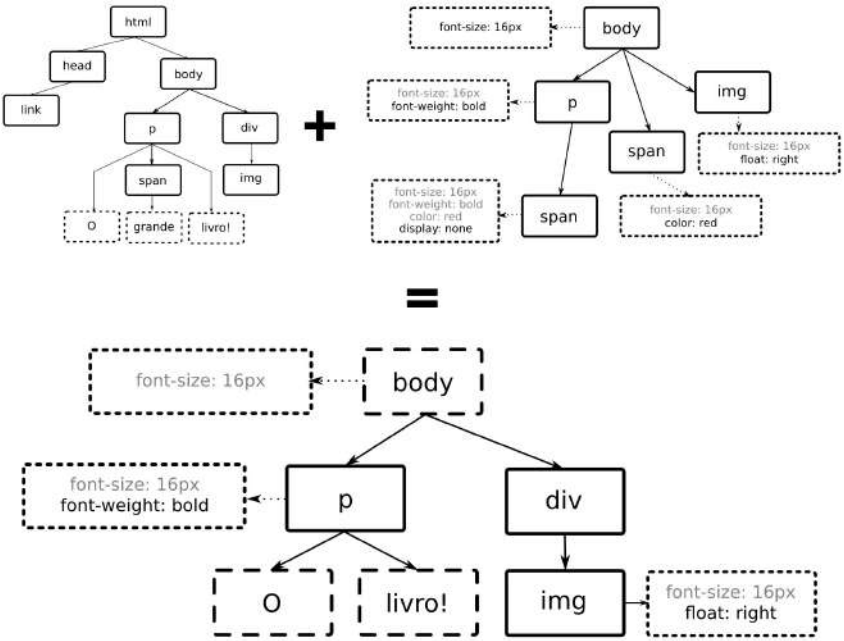


Figura 9.11: Render Tree baseada nos exemplos

A Render Tree contém apenas os elementos que devem ser mostrados na tela. No nosso exemplo, o elemento `<span>` não está presente na Render Tree. Isso acontece porque ele tem estilo `display: none`, que é responsável por fazer um elemento não fazer parte da renderização. Outros elementos de controle, como a tag `<head>` e as suas tags filhas, também não fazem parte da Render Tree por não preencherem o layout da página.

9.7 LAYOUT DA RENDER TREE

O processo de layout, também conhecido como *reflow*, é o processo de posicionar os elementos da Render Tree.

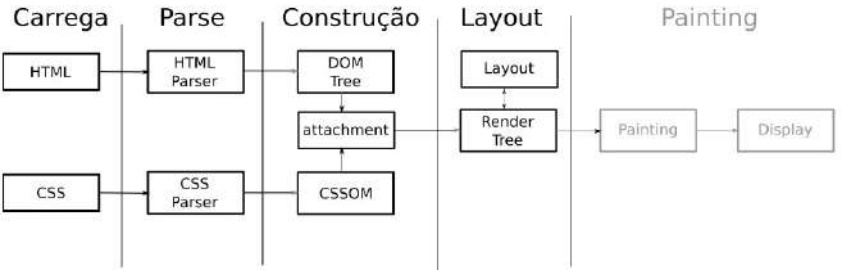


Figura 9.12: Layout

Os elementos visíveis foram adicionados na Render Tree e estão dispostos de uma forma hierárquica, juntamente com todas as informações necessárias sobre seus estilos. Apesar disso, ainda não está definido onde cada elemento deve aparecer na tela e qual tamanho ele terá. Esse é o trabalho principal do processo de layout: fazer a definição do posicionamento exato de cada elemento.

O navegador começa a calcular a partir do primeiro elemento, e segue recursivamente para os elementos filhos. Para saber as dimensões que ele deve utilizar para alocar os elementos na tela, o navegador utiliza o *viewport*, que representa a parte visível da tela.

Durante esse processo, são aplicadas as regras definidas por CSS para a página. Estas usam medidas relativas por padrão, portanto, se um elemento diz que vai ocupar 50% do espaço, ele está se referindo a metade do espaço que o elemento pai ocupa.

Ao passar pelo processo de layout, todas as regras são analisadas e convertidas para posições absolutas na tela do usuário. O navegador utiliza as coordenadas do plano cartesiano para posicionar os elementos — aqueles famigerados *x,y* que já vimos em vários lugares, começando com o ponto 0,0.

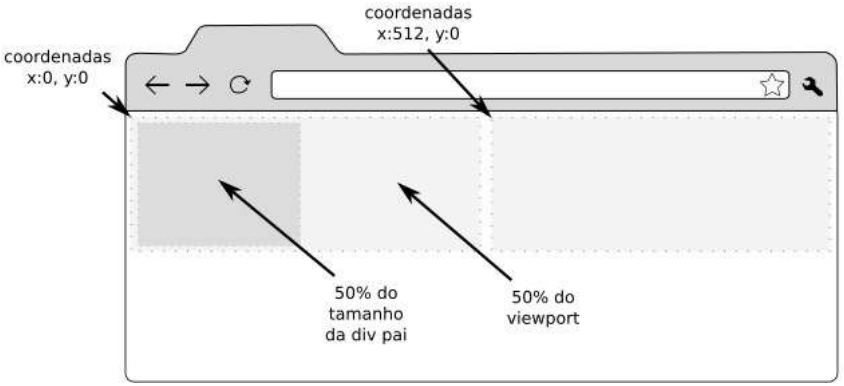


Figura 9.13: Posicionamento no navegador

Se houver qualquer modificação após a árvore estar pronta, o processo de layout será executado para organizar os elementos que terão suas posições alteradas. Os elementos alterados serão marcados para atualização, que será feita pelo processo de *painting*.

9.8 PAINTING

O que encerra o nosso estudo é a parte de *painting*, responsável por mostrar o conteúdo para o usuário.

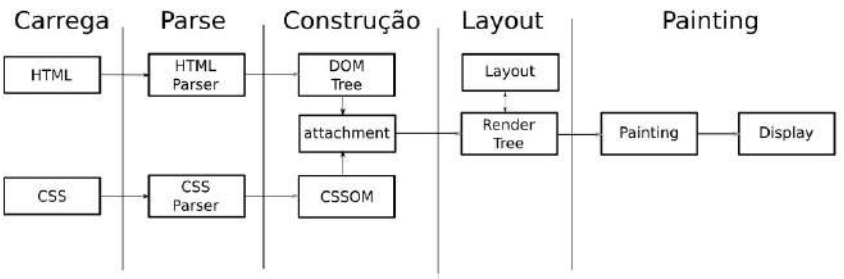


Figura 9.14: Painting

O processo de painting é responsável por navegar pela árvore organizada pelo processo de layout e criar os objetos visuais na tela. Esse processo pode ser incremental, onde apenas uma parte da tela

precisa ser alterada, ou total, onde o documento inteiro será mostrado na tela.

No caso da nossa requisição, acontecerá o painting do documento completo, por ser a primeira vez que o conteúdo será mostrado. Caso algum código JavaScript modifique algum elemento que tenha impacto visual na tela, haverá um *repaint* local. Um *repaint* é o processo de mostrar aquele elemento visualmente novamente.

Se um código JavaScript fizer uma modificação no tamanho dos elementos, ou o usuário redimensionar a janela do navegador, haverá em *reflow* seguido de um *repaint*. Nesse caso, o processo de layout precisará fazer uma modificação local e chamar o processo de painting novamente para também fazer uma modificação local. Esse processo é fonte de possíveis problemas de performance no navegador, portanto, é sempre importante evitar ao máximo os reflows e os repaintings.

O processo de paint e repaint na prática

Podemos ver o processo de painting quando executamos o Chromium com a flag `--show-paint-rects`. Para executá-lo e fazer os testes no `desconstruindoaweb.com.br`, podemos usar o seguinte comando:

```
chromium --incognito --show-paint-rects https://desconstruind
oaweb.com.br
```

Esse comando abrirá apenas uma aba, em modo incógnito, com a funcionalidade de "painting visual" habilitada. Com isso, é possível ver o primeiro painting do navegador, juntamente com todos os repaintings que forem necessários durante a navegação. Os paintings são representados por quadrados verdes e acontecem em toda mudança de conteúdo do navegador.

O processo de painting é responsável por saber qual elemento deve ser mostrado primeiro. A especificação do CSS define[15] qual a ordem que o navegador deve mostrar os elementos baseando-se no atributo `z-index`, que pode ser definido via CSS.

Com a ordem definida, o Chromium usa a GPU do computador para fazer a renderização do conteúdo na tela do usuário, caso ela esteja disponível. Se olharmos os processos que o Chromium cria, vemos que alguns deles são referentes à renderização via GPU:

```
$ ps xo cmd | grep chromium | grep gpu

/usr/lib/chromium/chromium --type=gpu-process --channel=29173.1.87
5868289 --window-depth=24 --supports-dual-gpus=false --gpu-driver-
bug-workarounds=4,5,19,46,54 --disable-gl-extensions=GL_ARB_occlus
ion_query GL_ARB_occlusion_query2 GL_ARB_timer_query GL_EXT_timer_
query --gpu-vendor-id=0x8086 --gpu-device-id=0x0a16 --gpu-driver-v
```

```
endor=Mesa --gpu-driver-version=11.2.2 --v8-natives-passed-by-fd -
-v8-snapshot-passed-by-fd
/usr/lib/chromium/chromium --type=gpu-broker
```

No caso caso do GNU/Linux, ele está utilizando a *libmesa*[16] como interface para o hardware 3D. Os processos de Render e GPU se comunicam para enviar cada parte do conteúdo para a tela do usuário, completando o carregamento da página.

9.9 TUDO PRONTO, EM MENOS DE UM SEGUNDO!

Ufa! Depois de centenas de páginas lidas, a requisição está finalmente completa e a página do *desconstruindoaweb.com.br* está totalmente carregada.

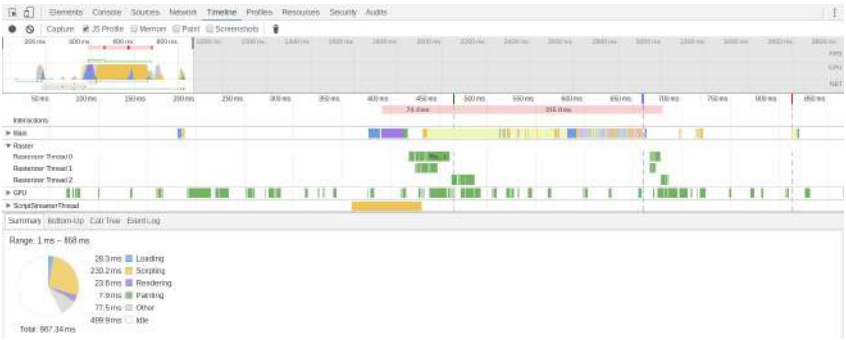


Figura 9.15: Resumo do processo do navegador

Ao utilizar a ferramenta de *timeline* do Developer Tools do Chromium[17], podemos ver que, apesar do processo completo ter sido longo para o nosso estudo, ele levou menos de um segundo para estar completo.

Tanto as medidas de performance quanto o tempo de carregamento vão variar de acordo com o ambiente que o estudo está sendo executado, mas ainda assim a mensagem é válida:

```
Entender um segundo de um processo pode levar um tempo incalculáve
l.
```

9.10 RESUMO

Depois de enviar a requisição e aguardar por algumas frações de segundos, o navegador começa a receber os dados que foram requisitados para o *desconstruindoaweb.com.br*. Metadados como `ETag` e `Cache-Control` são extraídos dos cabeçalhos do pacote HTTP, juntamente com o corpo do pacote que possui o HTML.

As informações extraídas do pacote HTTP são encaminhadas para a Blink, que é a *rendering engine* do Chromium. Ela começa o trabalho de entender o HTML e transformá-lo no que o usuário espera. O primeiro passo é fazer o *parse* do HTML, transformando-o de um simples texto em uma estrutura de árvore chamada de *DOM*. O mesmo acontece para os estilos CSS, mas esses criam uma segunda árvore, chamada CSSOM.

Com as árvores do DOM e CSSOM prontas, o navegador cria uma terceira baseando-se nelas. Essa terceira árvore se chama Render Tree e contém os elementos que vão ser mostrados na tela. Utilizando essa árvore, o navegador faz o processo de layout para dizer exatamente onde cada elemento deve estar na tela do dispositivo.

Com todos os elementos devidamente posicionados, o navegador faz o processo de *painting*. Esse processo é responsável por criar os elementos visualmente, entregando o conteúdo que o usuário estava aguardando.

9.11 REFERÊNCIAS

1. Código de rede do Chromium — https://chromium.googlesource.com/chromium/src/net/

2. Blink, a *rendering engine* do Chromium — http://chromium.org/blink

3. Site oficial do Webkit — https://webkit.org

4. Motivos que levaram a criação da Blink — http://www.chromium.org/blink/developer-faq#TOC-Why-is-Chrome-spawning-a-new-browser-engine-

5. Informação sobre chunks de 8KB, no artigo da Tali Garsiel — http://taligarsiel.com/Projects/howbrowserswork1.htm#Rendering_engines

6. Processo de parse do HTML — https://www.w3.org/TR/html5/syntax.html#parsing

7. Remoção de `CRLF` no pré-processador — https://www.w3.org/TR/html5/syntax.html#preprocessing-the-input-stream

8. Identificação de tokens na especificação do HTML — https://www.w3.org/TR/html5/syntax.html#tokenization

9. Inserção de tokens na árvore do DOM — https://www.w3.org/TR/html5/syntax.html#tree-construction

10. Especificação de syntax do CSS — https://www.w3.org/TR/CSS2/syndata.html#tokenization

11. Especificação de gramática do CSS — https://www.w3.org/TR/CSS2/grammar.html#scanner

12. Introdução ao CSSOM no W3C — https://www.w3.org/TR/cssom-1/#introduction

13. JavaScript não obstrutivo no wiki da W3C — https://www.w3.org/wiki/The_principles_of_unobtrusive_Jav

aScript

14. Definição do `async` e `defer` na W3C — https://www.w3.org/TR/html5/scripting-1.html#attr-script-defer
15. Order de painting na especificação do CSS e zindex — https://www.w3.org/TR/CSS22/zindex.html#painting-order
16. Site da libmesa — http://mesa3d.org
17. Ferramenta de timeline do Developer Tools do Chromium — https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/timeline-tool

Capítulo 10

ALÉM DESSA REQUISIÇÃO WEB

Figura 10.1: "Don't say: We have come now to the end..." — Into the west

Nestas mais de duzentas páginas, nós conseguimos remover as sombras que estavam na frente de algumas tecnologias, assim como foi acontecendo com o mapa da nossa jornada. Com esses conhecimentos, teremos mais informações para analisar o cenário macro e entender as conexões das micropeças que o compõem.

Podemos usar como analogia um mecânico que ouve o som do motor e diz: *"Ah! Isso é carburador sujo"*. Ele nem sequer olhou as peças, mas sabe bem onde começar a busca, pois entende as conexões das pequenas peças e sabe qual o impacto delas no todo.

As tecnologias que estudamos já não parecem um completo mistério para nós, mas ainda há várias outras que podem estar no

nosso dia a dia e não foram estudadas durante nossa jornada. Vamos conhecer alguns próximos passos que podem ser tomados e como atacar algumas perguntas que podem aparecer.

10.1 OUTRAS TECNOLOGIAS

Para vários leitores, ainda haverá uma pergunta ao final deste livro:

```
Eu uso a tecnologia X e estudamos Y no livro, como faço?
```

Muitas das coisas que estudamos vão servir para quase todos os casos, como a rede ou o navegador. Apesar de haver diferenças, essas tecnologias ainda são parecidas conceitualmente e vão ser o suficiente para entender o contexto geral. Já outras tecnologias, como o *Rack* por exemplo, são bem específicas do conjunto de tecnologias que estudamos.

Este livro ajuda a entender o contexto geral de uma requisição web estudando um conjunto de tecnologias. A partir desse estudo, é possível separar uma parte da requisição e estudar isoladamente outra tecnologia que faça o mesmo trabalho.

Um exemplo é o caso de utilizar um *framework* diferente como o Sinatra (http://sinatrarb.com), em que todo o estudo seria o mesmo apenas alterando o *framework*. No caso da utilização de outra linguagem de programação, e consequentemente outro framework, é possível que também seja necessário estudar um servidor de aplicação diferente, mas ainda assim uma boa parte do estudo não precisa ser alterado para entender o contexto geral.

Onde encontrar material

Como é possível ver nas referências de cada capítulo, boa parte do material é escrito em inglês. Logo, conhecimento no idioma é

primordial.

Muitos sistemas grandes possuem documentos de design (ou *design documents*, em inglês). Esses documentos explicam como o software foi criado e por que algumas decisões foram tomadas. O Chromium é um exemplo de software que possui tais documentos.

Softwares que possuem foco em performance, como o Phusion Passenger, costumam escrever documentações explicando cada uma das decisões que fazem a performance aumentar em X porcento. Esse tipo de documento é um ótimo lugar para começar, pois ajuda a entender por que uma implementação foi escolhida para determinada funcionalidade.

Em alguns casos, mesmo que o software seja grande ou focado em performance, ele não possui esse tipo de documentação. Nesses casos, a melhor forma ainda é olhar o código-fonte e, se possível, utilizar debuggers ou o nosso velho amigo `strace`, para entender o que está acontecendo. Essa é a forma mais demorada de se entender, mas é garantia de novos conhecimentos sobre linguagens, arquitetura, código aberto e comunidades.

10.2 DISCUSSÃO E APRENDIZADO

A Casa do Código provê um fórum no qual os autores (inclusive eu!) e outros leitores discutem questões sobre os livros. Lá pode ser um bom lugar para perguntar sobre alguma dúvida do conteúdo do livro, ou colocar dicas sobre um ponto de partida a partir deste conteúdo.

O fórum está disponível em: http://forum.casadocodigo.com.br.

10.3 CONSIDERAÇÕES FINAIS

O conteúdo deste livro é um trabalho que estava em projeto há vários anos, mas só agora surgiu o tempo e oportunidade para criar este material. Espero que suas expectativas tenham sido atendidas e que, a partir de agora, encontrar problemas em uma aplicação web, ou até mesmo navegar na internet, nunca mais seja a mesma coisa. :)

Boa sorte!