

# Camada de aplicação

Aplicações de rede são a razão de ser de uma rede de computadores. Se não fosse possível inventar aplicações úteis, não haveria necessidade de projetar protocolos de rede para suportá-las. Desde o surgimento da Internet, foram criadas numerosas aplicações úteis e divertidas. Elas têm sido a força motriz por trás do sucesso da Internet, motivando pessoas em lares, escolas, governos e empresas a tornarem a rede uma parte integral de suas atividades diárias.

Entre as aplicações da Internet, estão as aplicações clássicas de texto, que se tornaram populares nas décadas de 1970 e 1980: correio eletrônico, acesso a computadores remotos, transferência de arquivo e grupos de discussão. Também há uma aplicação que alcançou estrondoso sucesso em meados da década de 1990: a World Wide Web, abrangendo a navegação na Web, busca e o comércio eletrônico. Desde o início do novo milênio, novas aplicações, altamente populares, continuam a emergir, incluindo voz sobre IP (VoIP) e videoconferência, como Skype, Facetime e Google Hangouts; vídeo gerado pelo usuário, como YouTube; filmes sob demanda, como Netflix; e jogos *multiplayer on-line*, como Second Life e World of Warcraft. Durante o mesmo período, assistimos à criação de uma nova geração de aplicações de redes sociais, como Facebook, Instagram e Twitter, que criaram redes humanas sobrepostas à rede de roteadores e enlaces de comunicação da Internet. Mais recentemente, com a chegada dos *smartphones* e a onipresença do acesso à Internet sem fio 4G/5G, houve uma explosão de aplicativos móveis baseados em localização, incluindo aplicativos populares de *check-in*, namoro e trânsito (como Yelp, Tinder e Waze), de pagamentos móveis (como WeChat e Apple Pay) e de mensagens (como WeChat e WhatsApp). É evidente que o surgimento de aplicações novas e interessantes para Internet não está desacelerando. Talvez alguns dos leitores deste texto criem a próxima geração de aplicações quentes para Internet!

Neste capítulo, estudaremos os aspectos conceituais e de implementação de aplicações de rede. Começaremos definindo conceitos fundamentais de camada de aplicação, incluindo serviços de rede exigidos por aplicações, clientes e servidores, processos e interfaces de camada de transporte. Vamos examinar detalhadamente várias aplicações de rede, entre elas a Web, *e-mail*, sistema de nomes de domínio (DNS, do inglês *domain name system*), distribuição de arquivos *peer-to-peer* (P2P) e *streaming* de vídeo. Em seguida, abordaremos o desenvolvimento de aplicações de rede por Protocolo de Controle de Transmissão (TCP, do inglês

*Transmission Control Protocol*) e também por Protocolo de Datagrama de Usuário (UDP, do inglês *User Datagram Protocol*). Em particular, vamos estudar a interface de *socket* e examinar algumas aplicações cliente-servidor simples em Python. Apresentaremos também vários exercícios divertidos e interessantes de programação de aplicações no final do capítulo.

A camada de aplicação é um lugar particularmente bom para iniciarmos o estudo de protocolos. É terreno familiar, pois conhecemos muitas das aplicações que dependem dos protocolos que estudaremos. Ela nos dará uma boa ideia do que são protocolos e nos apresentará muitos assuntos que encontraremos de novo quando estudarmos protocolos de camadas de transporte, de rede e de enlace.

## 2.1 PRINCÍPIOS DE APLICAÇÕES DE REDE

Suponha que você tenha uma grande ideia para uma nova aplicação de rede. Essa aplicação será, talvez, um grande serviço para a humanidade, ou agradará a seu professor, ou fará de você uma pessoa rica; ou apenas será divertido desenvolvê-la. Seja qual for sua motivação, vamos examinar agora como transformar a ideia em uma aplicação do mundo real.

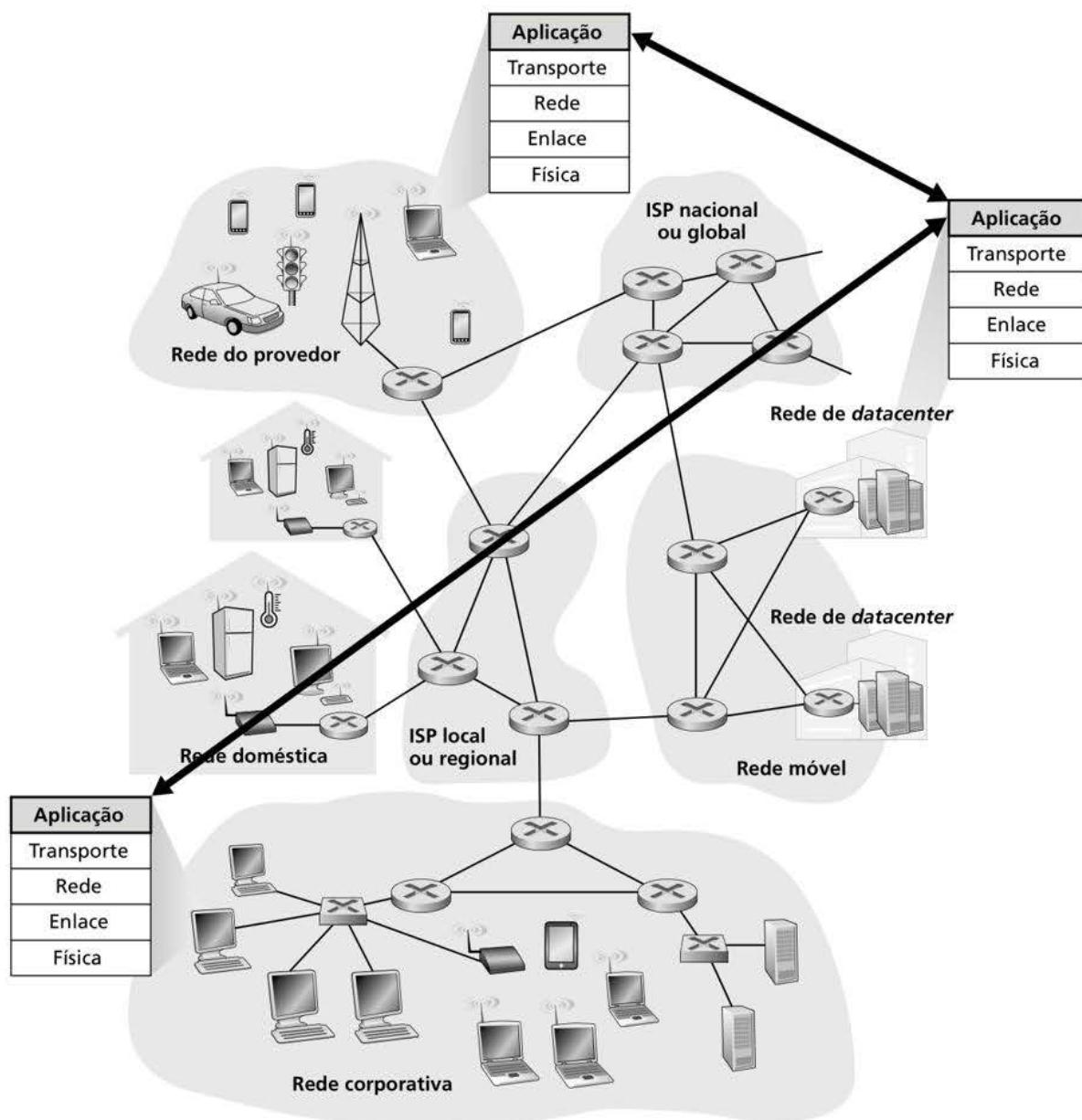
O núcleo do desenvolvimento de aplicação de rede é escrever programas que rodem em sistemas finais diferentes e se comuniquem entre si. Por exemplo, na aplicação Web, há dois programas distintos que se comunicam um com o outro: o do navegador, que roda no hospedeiro do usuário (computador de mesa, *notebook*, *tablet*, *smartphone* e assim por diante); e o do servidor Web, que roda em um servidor da rede. Outro exemplo é uma aplicação de vídeo sob demanda, como o Netflix (ver Seção 2.6); nela, um programa fornecido pela Netflix roda no *smartphone*, *tablet* ou computador do usuário; e um programa de servidor da Netflix roda no hospedeiro servidor da empresa. Os servidores muitas vezes (mas nem sempre) ficam alojados em um *datacenter*, como mostra a Figura 2.1.

Portanto, ao desenvolver sua nova aplicação, você precisará escrever um *software* que rode em vários sistemas finais. Esse *software* poderia ser criado, por exemplo, em C, Java ou Python. Importante: você não precisará escrever programas que executem nos elementos do núcleo de rede, como roteadores e comutadores da camada de enlace. Mesmo se quisesse, não poderia desenvolver programas para esses elementos. Como aprendemos no Capítulo 1 e mostramos na Figura 1.24, equipamentos de núcleo de rede não funcionam na camada de aplicação, mas em camadas mais baixas, em especial na de rede e abaixo dela. Esse projeto básico – a saber, confinar o *software* de aplicação nos sistemas finais –, como mostra a Figura 2.1, facilitou o desenvolvimento e a proliferação rápidos de uma vasta gama de aplicações de rede.

### 2.1.1 Arquiteturas de aplicação de rede

Antes de mergulhar na codificação do *software*, você deverá elaborar um plano geral para a arquitetura da sua aplicação. Tenha sempre em mente que a arquitetura de uma aplicação é bastante diferente da arquitetura da rede (p. ex., a arquitetura em cinco camadas da Internet que discutimos no Capítulo 1). Do ponto de vista do profissional que desenvolve a aplicação, a arquitetura de rede é fixa e provê um conjunto específico de serviços. Por outro lado, a **arquitetura da aplicação** é projetada pelo programador e determina como a aplicação é organizada nos vários sistemas finais. Ao escolher a arquitetura da aplicação, é provável que o programador aproveite uma das duas arquiteturas mais utilizadas em aplicações modernas de rede: cliente-servidor ou P2P.

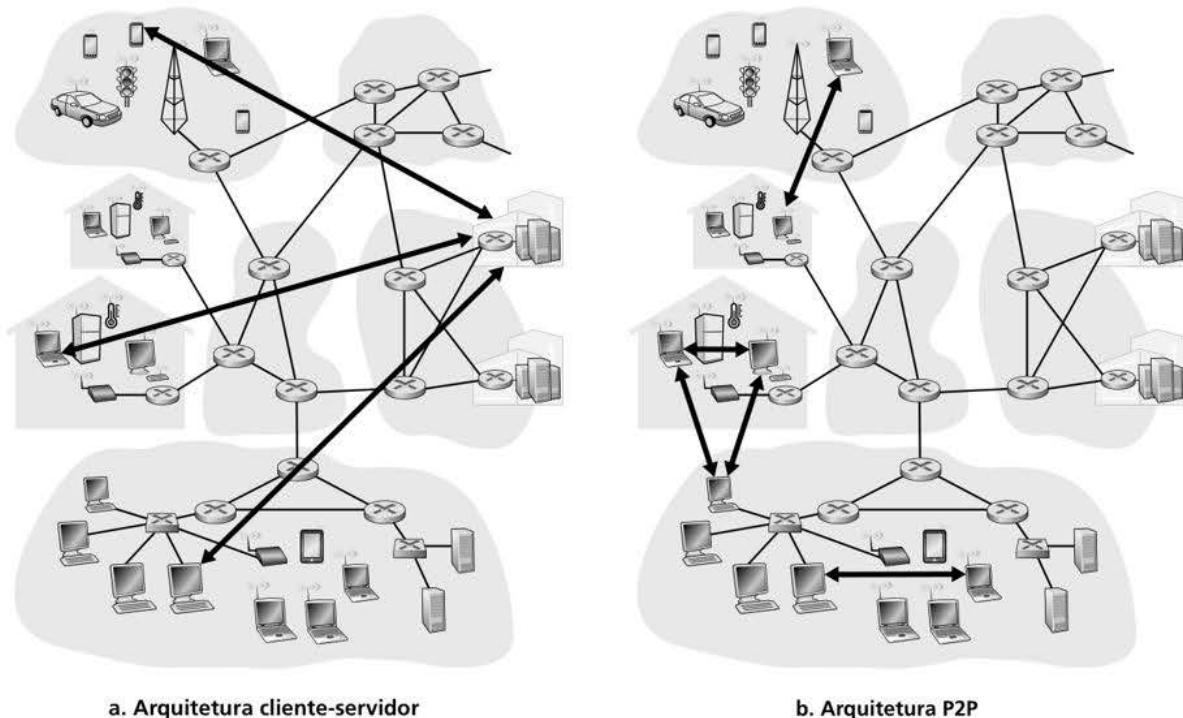
Em uma **arquitetura cliente-servidor**, há um hospedeiro sempre em funcionamento, denominado *servidor*, que atende a requisições de muitos outros hospedeiros, denominados *clientes*. Um exemplo clássico é a aplicação Web na qual um servidor Web que está sempre em funcionamento atende a solicitações de navegadores de hospedeiros clientes. Quando recebe uma requisição de um objeto de um hospedeiro cliente, um servidor Web responde



**Figura 2.1** A comunicação de uma aplicação de rede ocorre entre sistemas finais na camada de aplicação.

enviando o objeto solicitado. Observe que, na arquitetura cliente-servidor, os clientes não se comunicam diretamente uns com os outros; por exemplo, na aplicação Web, dois navegadores não se comunicam de modo direto. Outra característica dessa arquitetura é que o servidor tem um endereço fixo, bem conhecido, denominado endereço IP (que discutiremos em breve). Em razão dessa característica do servidor e pelo fato de ele estar sempre em funcionamento, um cliente sempre pode contatá-lo, enviando um pacote ao endereço do servidor. Algumas das aplicações mais conhecidas que empregam a arquitetura cliente-servidor são Web, FTP, Telnet e e-mail. Essa arquitetura é mostrada na Figura 2.2(a).

Em aplicações cliente-servidor, muitas vezes acontece de um único hospedeiro servidor ser incapaz de atender a todas as requisições de seus clientes. Por exemplo, um site popular de redes sociais pode ficar logo saturado se tiver apenas um servidor para atender a todas as solicitações. Por essa razão, um **datacenter**, acomodando um grande número de hospedeiros, é usado com frequência para criar um servidor virtual poderoso. Os serviços de Internet mais populares – como mecanismos de busca (p. ex., Google, Bing e Baidu), comércio via



**Figura 2.2** (a) Arquitetura cliente-servidor; (b) arquitetura P2P.

Internet (p. ex., Amazon, eBay e Alibaba), e-mail baseado na Web (p. ex., Gmail e Yahoo Mail), redes sociais (p. ex., Facebook, Instagram, Twitter e WeChat) – rodam em um ou mais *datacenters*. Conforme discutimos na Seção 1.3.3, a Google possui 19 *datacenters* distribuídos pelo mundo que, em conjunto, tratam de busca, YouTube, Gmail e outros serviços. Um *datacenter* pode ter centenas de milhares de servidores, que precisam ser alimentados e mantidos. Além disso, os provedores de serviços têm de pagar pelos custos de interconexão recorrente e largura de banda para o envio de dados a partir de seus *datacenters*.

Em uma **arquitetura P2P**, há uma dependência mínima (ou nenhuma) nos servidores dedicados nos *datacenters*. Em vez disso, a aplicação utiliza a comunicação direta entre duplas de hospedeiros conectados alternadamente, denominados *pares* (*peers*). Eles não são de propriedade dos provedores de serviço, mas são controlados por usuários de computadores de mesa e *notebooks*, cuja maioria se aloja em residências, universidades e escritórios. Como os pares se comunicam sem passar por nenhum servidor dedicado, a arquitetura é denominada par a par (P2P – *peer-to-peer*). Um exemplo de aplicação P2P popular é o BitTorrent, um aplicativo de compartilhamento de arquivos.

Uma das características mais marcantes da arquitetura P2P é sua **autoescalabilidade**. Por exemplo, em uma aplicação de compartilhamento de arquivos P2P, embora cada par gere uma carga de trabalho solicitando arquivos, também acrescenta capacidade de serviço ao sistema distribuindo arquivos a outros pares. As arquiteturas P2P também possuem uma boa relação custo-benefício, visto que, em geral, não requerem infraestrutura e largura de banda de servidor significativas (ao contrário de projetos cliente-servidor com *datacenters*). Entretanto, devido à sua estrutura altamente descentralizada, as aplicações P2P enfrentam desafios de segurança, desempenho e confiabilidade.

### 2.1.2 Comunicação entre processos

Antes de construir sua aplicação de rede, você também precisará ter um entendimento básico de como programas que rodam em vários sistemas finais comunicam-se entre si.

No jargão de sistemas operacionais, na verdade não são programas, mas **processos** que se comunicam. Um processo pode ser definido como um programa que está rodando dentro de um sistema final. Quando os processos rodam no mesmo sistema final, comunicam-se usando comunicação interprocessos, cujas regras são determinadas pelo sistema operacional do sistema final. Porém, neste livro, não estamos interessados na comunicação entre processos do mesmo hospedeiro, mas em como se comunicam os que rodam em sistemas finais *diferentes* (com sistemas operacionais potencialmente diferentes).

Os processos em dois sistemas finais diferentes se comunicam trocando **mensagens** por meio da rede de computadores. Um processo originador cria e envia mensagens para a rede; um processo destinatário recebe-as e responde, devolvendo outras. A Figura 2.1 mostra que processos se comunicam usando a camada de aplicação da pilha de cinco camadas da arquitetura.

### Processos clientes e processos servidores

Uma aplicação de rede consiste em pares de processos que enviam mensagens uns para os outros por meio de uma rede. Por exemplo, na aplicação Web, o processo navegador de um cliente troca mensagens com o de um servidor Web. Em um sistema de compartilhamento de arquivos P2P, um arquivo é transferido de um processo que está em um par para um que está em outro par. Para cada par de processos comunicantes, normalmente rotulamos um dos dois processos de **cliente** e o outro, de **servidor**. Na Web, um navegador é um processo cliente, e um servidor Web é um processo servidor. No compartilhamento de arquivos P2P, a entidade que envia o arquivo é rotulada de cliente e a que o recebe, de servidor.

Talvez você já tenha observado que, em algumas aplicações, tal como compartilhamento de arquivos P2P, um processo pode ser ambos, cliente e servidor. De fato, um processo em um sistema de compartilhamento de arquivos P2P pode carregar e descarregar arquivos. Mesmo assim, no contexto de qualquer sessão entre um par de processos, ainda podemos rotular um processo de cliente e o outro de servidor. Definimos os processos cliente e servidor como segue:

*No contexto de uma sessão de comunicação entre um par de processos, aquele que inicia a comunicação (i.e., o primeiro a contatar o outro no início da sessão) é rotulado de cliente. O que espera ser contatado para iniciar a sessão é o servidor.*

Na Web, um processo do navegador inicia o contato com um processo do servidor Web; por conseguinte, o processo do navegador é o cliente, e o do servidor Web é o servidor. No compartilhamento de arquivos P2P, quando o Par A solicita ao Par B o envio de um arquivo específico, o Par A é o cliente, enquanto o Par B é o servidor no contexto dessa sessão de comunicação. Quando não houver possibilidade de confusão, às vezes usaremos também a terminologia “lado cliente e lado servidor de uma aplicação”. No final deste capítulo, examinaremos passo a passo um código simples para ambos os lados de aplicações de rede: o lado cliente e o lado servidor.

### A interface entre o processo e a rede de computadores

Como dissemos anteriormente, a maioria das aplicações consiste em pares de processos comunicantes, e os dois processos de cada par enviam mensagens um para o outro. Qualquer mensagem enviada de um processo para outro tem de passar pela rede subjacente. Um processo envia mensagens para a rede e recebe mensagens dela através de uma interface de software denominada **socket**. Vamos considerar uma analogia que nos auxiliará a entender processos e *sockets*. Um processo é semelhante a uma casa, e seu *socket*, à porta da casa. Quando um processo quer enviar uma mensagem a outro processo em outro hospedeiro, ele empurra a mensagem pela porta (*socket*). O emissor admite que existe uma infraestrutura de transporte do outro lado de sua porta que transportará a mensagem pela rede até a porta do

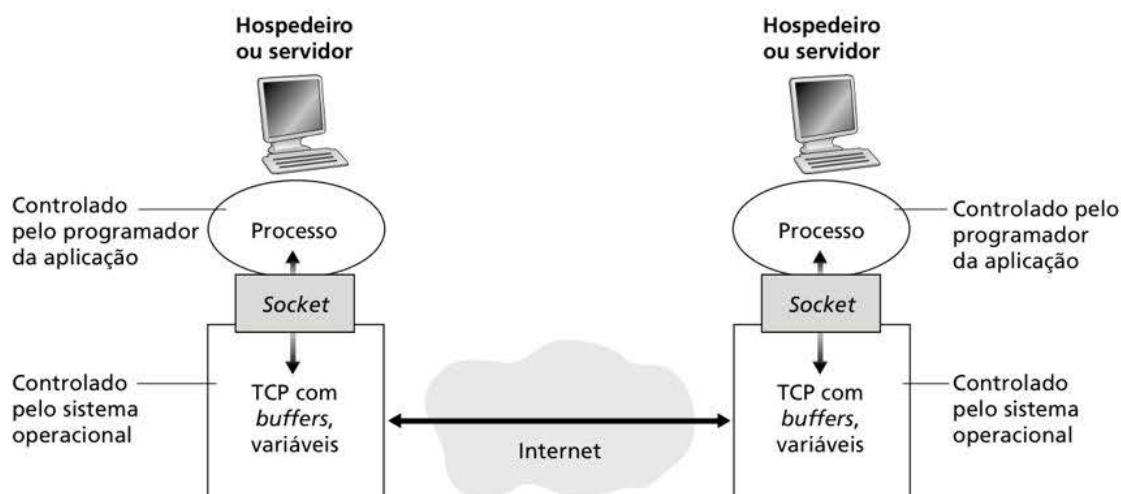
processo destinatário. Ao chegar ao hospedeiro destinatário, a mensagem passa pela porta (*socket*) do processo receptor, que então executa alguma ação sobre a mensagem.

A Figura 2.3 ilustra a comunicação por *socket* entre dois processos que se comunicam pela Internet. (A Figura 2.3 admite que o protocolo de transporte subjacente usado pelos processos é o TCP.) Como mostra essa figura, um *socket* é a interface entre a camada de aplicação e a de transporte dentro de um hospedeiro. É também denominado **interface de programação de aplicação (API)**, do inglês *application programming interface*) entre a aplicação e a rede, visto que é a interface de programação pela qual as aplicações de rede são criadas. O programador da aplicação controla tudo o que existe no lado da camada de aplicação do *socket*, mas tem pouco controle do lado da camada de transporte. Os únicos controles que o programador da aplicação tem do lado da camada de transporte são: (1) a escolha do protocolo de transporte e (2), talvez, a capacidade de determinar alguns parâmetros, tais como tamanho máximo de *buffer* e de segmentos (a serem abordados no Capítulo 3). Uma vez escolhido um protocolo de transporte (se houver escolha), o programador constrói a aplicação usando os serviços da camada de transporte oferecidos por esse protocolo. Examinaremos *sockets* mais detalhadamente na Seção 2.7.

### Endereçando processos

Para enviar correspondência postal a determinado destino, este precisa ter um endereço. De modo semelhante, para que um processo rodando em um hospedeiro envie pacotes a um processo rodando em outro hospedeiro, o receptor precisa ter um endereço. Para identificar o processo receptor, duas informações devem ser especificadas: (1) o endereço do hospedeiro e (2) um identificador que especifica o processo receptor no hospedeiro de destino.

Na Internet, o hospedeiro é identificado por seu **endereço IP**. Discutiremos os endereços IP com mais detalhes no Capítulo 4. Por enquanto, tudo o que precisamos saber é que um endereço IP é uma quantidade de 32 bits que podem identificar um hospedeiro de forma exclusiva. Além de saber o endereço do hospedeiro ao qual a mensagem é destinada, o processo de envio também precisa identificar o processo receptor (mais especificamente, o *socket* receptor) executando no hospedeiro. Essa informação é necessária porque, em geral, um hospedeiro poderia estar executando muitas aplicações de rede. Um **número de porta** de destino atende a essa finalidade. Aplicações populares receberam números de porta específicos. Por exemplo, um servidor Web é identificado pelo número de porta 80. Um processo servidor de correio (usando o protocolo SMTP) é identificado pelo número de porta 25. Uma lista dos números de porta conhecidos para todos os protocolos-padrão da



**Figura 2.3** Processos de aplicação, sockets e protocolo de transporte subjacente.

Internet poderá ser encontrada em <<http://www.iana.org>>. Vamos examinar os números de porta em detalhes no Capítulo 3.

### 2.1.3 Serviços de transporte disponíveis para aplicações

Lembre-se de que um *socket* é a interface entre o processo da aplicação e o protocolo de camada de transporte. A aplicação do lado remetente envia mensagens por meio do *socket*. Do outro lado, o protocolo de camada de transporte tem a responsabilidade de levar as mensagens pela rede até o *socket* do processo destinatário.

Muitas redes, inclusive a Internet, oferecem mais de um protocolo de camada de transporte. Ao desenvolver uma aplicação, você deve escolher um dos protocolos de camada de transporte disponíveis. Como fazer essa escolha? O mais provável é que você avalie os serviços e escolha o protocolo que melhor atenda às necessidades de sua aplicação. A situação é semelhante a decidir entre ônibus ou avião como meio de transporte entre duas cidades. Você tem de optar, e cada modalidade de transporte oferece serviços diferentes. (P. ex., o ônibus oferece a facilidade da partida e da chegada no centro da cidade, ao passo que o avião tem menor tempo de viagem.)

Quais são os serviços que um protocolo da camada de transporte pode oferecer às aplicações que o chamem? Podemos classificar, de maneira geral, os possíveis serviços segundo quatro dimensões: transferência confiável de dados, vazão, temporização e segurança.

#### Transferência confiável de dados

Como discutido no Capítulo 1, os pacotes podem se perder dentro de uma rede de computadores. Um pacote pode, por exemplo, esgotar um *buffer* em um roteador, ou ser descartado por um hospedeiro ou um roteador após alguns de seus *bits* terem sido corrompidos. Para muitas aplicações – como correio eletrônico, transferência de arquivo, acesso a hospedeiro remoto, transferências de documentos da Web e aplicações financeiras –, a perda de dados pode ter consequências devastadoras (no último caso, para o banco e para o cliente!). Assim, para suportar essas aplicações, algo deve ser feito para garantir que os dados enviados por uma extremidade da aplicação sejam transmitidos correta e completamente para a outra ponta. Se um protocolo fornecer um serviço de recebimento de dados garantido, ele fornecerá uma **transferência confiável de dados**. Um importante serviço que o protocolo da camada de transporte pode oferecer para uma aplicação é a transferência confiável de dados processo a processo. Quando um protocolo de transporte oferece esse serviço, o processo remetente pode apenas passar seus dados para um *socket* e saber com completa confiança que eles chegarão sem erro ao processo destinatário.

Quando um protocolo da camada de transporte não oferece uma transferência confiável de dados, os dados enviados pelo remetente talvez nunca cheguem ao destinatário. Isso pode ser aceitável para **aplicações tolerantes a perda**, em especial as de multimídia como áudio/vídeo em tempo real ou áudio/vídeo armazenado, que podem tolerar alguma perda de dados. Nessas aplicações, dados perdidos podem resultar em uma pequena falha durante a execução do áudio/vídeo – o que não é um prejuízo crucial.

#### Vazão

No Capítulo 1, apresentamos o conceito de vazão disponível, que, no contexto de sessão da comunicação entre dois processos ao longo de um caminho de rede, é a taxa pela qual o processo remetente pode enviar *bits* ao processo destinatário. Como outras sessões compartilharão a largura de banda no caminho da rede e seus dados estão indo e voltando, a vazão disponível pode oscilar com o tempo. Essas observações levam a outro serviço natural que um protocolo da camada de transporte pode oferecer, a saber, uma vazão disponível garantida a uma taxa específica. Com tal serviço, a aplicação pode solicitar uma vazão garantida

de  $r$  bits/s, e o protocolo de transporte garante, então, que a vazão disponível seja sempre  $r$  bits/s, pelo menos. Tal serviço de vazão garantida seria atraente para muitas aplicações. Por exemplo, se uma aplicação de telefonia por Internet codifica voz a 32 kbits/s, ela precisa enviar dados para a rede e fazer que sejam entregues na aplicação receptora à mesma taxa. Se o protocolo de transporte não puder fornecer essa vazão, a aplicação precisará codificar a uma taxa menor (e receber vazão suficiente para sustentar essa taxa de codificação mais baixa) ou então desistir, já que receber, digamos, metade da vazão de que precisa de nada ou pouco adianta para essa aplicação de telefonia por Internet. Aplicações que possuam necessidade de vazão definida são conhecidas como **aplicações sensíveis à largura de banda**. Muitas aplicações de multimídia existentes são sensíveis à largura de banda, embora algumas possam usar técnicas adaptativas para codificar vídeo ou voz digitalizada a uma taxa que corresponda à vazão disponível na ocasião.

Embora aplicações sensíveis à largura de banda possuam necessidades específicas de vazão, **aplicações elásticas** podem usar qualquer quantidade de transferência de dados mínima ou máxima que por acaso esteja disponível. Correio eletrônico, transferência de arquivos e transferências Web são todas aplicações elásticas. Claro, quanto mais vazão, melhor. Há um ditado que diz que “dinheiro nunca é demais”; nesse caso, podemos dizer que vazão nunca é demais!

### Temporização

Um protocolo da camada de transporte pode também oferecer garantias de temporização. Como nas garantias de vazão, as de temporização podem surgir em diversos aspectos e modos. Citamos como exemplo o fato de que cada *bit* que o remetente insere no *socket* chega ao *socket* destinatário em menos de 100 ms. Esse serviço seria atrativo para aplicações interativas em tempo real, como a telefonia por Internet, ambientes virtuais, teleconferência e jogos *multiplayer*, que exigem restrições de temporização no envio de dados para garantir eficácia (Gauthier, 1999; Ramjee, 1994). Longos atrasos na telefonia por Internet, por exemplo, tendem a resultar em pausas artificiais na conversação; em um jogo *multiplayer* ou ambiente virtual interativo, um longo atraso entre realizar uma ação e ver a reação do ambiente (p. ex., a reação de outro jogador na outra extremidade de uma conexão fim a fim) faz a aplicação parecer menos realista. Para aplicações que não são em tempo real, é sempre preferível um atraso menor a um maior, mas não há nenhuma limitação estrita aos atrasos fim a fim.

### Segurança

Por fim, um protocolo de transporte pode oferecer um ou mais serviços de segurança a uma aplicação. Por exemplo, no hospedeiro remetente, um protocolo de transporte é capaz de codificar todos os dados transmitidos pelo processo remetente, e, no hospedeiro destinatário, o protocolo da camada de transporte pode codificar os dados antes de enviá-los ao destinatário. Tal serviço pode oferecer sigilo entre os dois, mesmo que os dados sejam, de algum modo, observados entre os processos remetente e destinatário. Um protocolo de transporte consegue, além do sigilo, fornecer outros serviços de segurança, incluindo integridade dos dados e autenticação do ponto terminal, assuntos que serão abordados em detalhes no Capítulo 8.

#### 2.1.4 Serviços de transporte providos pela Internet

Até aqui, consideramos serviços de transportes que uma rede de computadores *poderia* oferecer em geral. Vamos agora nos aprofundar mais no assunto e analisar o tipo de suporte de aplicação provido pela Internet. A Internet (e, em um amplo sentido, as redes TCP/IP) disponibiliza dois protocolos de transporte para aplicações, o UDP e o TCP. Quando você (como um criador de aplicação) cria uma nova aplicação de rede para a Internet, uma das

primeiras decisões a ser tomada é usar o UDP ou o TCP. Cada um deles oferece um conjunto diferente de serviços para as aplicações solicitantes. A Figura 2.4 mostra os requisitos do serviço para algumas aplicações.

### Serviços do TCP

O modelo de serviço TCP inclui um serviço orientado à conexão e um serviço confiável de transferência de dados. Quando uma aplicação solicita o TCP como seu protocolo de transporte, recebe dele ambos os serviços.

- *Serviço orientado à conexão.* O TCP faz o cliente e o servidor trocarem informações de controle de camada de transporte *antes* que as mensagens de camada de aplicação comecem a fluir. Esse procedimento de apresentação, também chamado de *handshake*, alerta o cliente e o servidor, permitindo que eles se preparem para uma enxurrada de pacotes. Após a fase de apresentação, dizemos que existe uma **conexão TCP** entre os *sockets* dos dois processos. A conexão é *full-duplex* (simultânea), visto que os dois processos podem enviar mensagens um ao outro pela conexão ao mesmo tempo. Quando termina de enviar mensagens, a aplicação deve encerrar a conexão. No Capítulo 3, discutiremos em detalhes o serviço orientado para conexão e examinaremos como ele é implementado.
- *Serviço confiável de transporte.* Os processos comunicantes podem contar com o TCP para a entrega de todos os dados enviados sem erro e na ordem correta. Quando um lado da aplicação passa uma cadeia de *bytes* para dentro de um *socket*, pode contar com o TCP para entregar a mesma cadeia de dados ao *socket* receptor, sem falta de *bytes* nem *bytes* duplicados.

O TCP também inclui um mecanismo de controle de congestionamento, um serviço voltado ao bem-estar geral da Internet e não ao benefício direto dos processos comunicantes. O mecanismo de controle de congestionamento do TCP limita a capacidade de transmissão de um processo (cliente ou servidor) quando a rede está congestionada entre remetente e destinatário. Como veremos no Capítulo 3, o controle de congestionamento do TCP tenta limitar cada conexão do TCP à sua justa porção de largura de banda de rede.

### Serviços do UDP

O UDP é um protocolo de transporte simplificado, leve, com um modelo de serviço minimalista. É um serviço não orientado à conexão; portanto, não há apresentação antes que

Aplicação	Perda de dados	Vazão	Sensibilidade ao tempo
Transferência/download de arquivo	Sem perda	Elástica	Não
E-mail	Sem perda	Elástica	Não
Documentos Web	Sem perda	Elástica (alguns kbits/s)	Não
Telefonia via Internet/videoconferência	Tolerante à perda	Áudio: alguns kbits/s – 1Mbit/s Vídeo: 10 kbits/s – 5 Mbits/s	Sim: décimos de segundo
Streaming de áudio/vídeo armazenado	Tolerante à perda	Igual acima	Sim: alguns segundos
Jogos interativos	Tolerante à perda	Poucos kbits/s – 10 kbits/s	Sim: décimos de segundo
Mensagens de smartphone	Sem perda	Elástica	Sim e não

**Figura 2.4** Requisitos de aplicações de rede selecionadas.

## SEGURANÇA EM FOCO

### PROTEGENDO O TCP

Nem o TCP ou o UDP fornecem qualquer codificação – os dados que o processo remetente transfere para seu *socket* são os mesmos que percorrem a rede até o processo destinatário. Então, por exemplo, se o processo destinatário enviar uma senha em texto aberto (ou seja, não codificado) para seu *socket*, ela percorrerá por todos os enlaces entre o remetente e o destinatário, podendo ser analisada e descoberta em qualquer um dos enlaces intermediários. Uma vez que a privacidade e outras questões de segurança se tornaram importantes para muitas aplicações, a comunidade da Internet desenvolveu um aperfeiçoamento para o TCP, denominado **Segurança na Camada de Transporte (TLS)**, do inglês **Transport Layer Security**) (RFC 5246). O TCP aperfeiçoado com TLS não só faz tudo o que o TCP tradicional, como também oferece serviços importantes de segurança processo a processo, incluindo criptografia, integridade dos dados e autenticação do

ponto de chegada. Enfatizamos que a TLS não é um terceiro protocolo da Internet, no mesmo nível do TCP e do UDP, mas um aperfeiçoamento do TCP executado na camada de aplicação. Em particular, se uma aplicação quiser utilizar o serviço da TLS, é preciso incluir o código TLS (disponível na forma de classes e bibliotecas altamente otimizadas) da aplicação em ambas as partes cliente e servidor. A TLS possui sua própria API de *socket* que é semelhante à tradicional API de *socket* TCP. Quando uma aplicação utiliza o TLS, o processo remetente transfere dados em texto aberto para o *socket* TLS; no hospedeiro emissor, então, o TLS codifica os dados e os passa para o *socket* TCP. Os dados codificados percorrem a Internet até o *socket* TCP no processo destinatário. O *socket* destinatário passa os dados codificados à TLS, que os decodifica. Por fim, a TLS passa os dados em texto aberto por seu *socket* TLS até o processo destinatário. Abordaremos a TLS em mais detalhes no Capítulo 8.

os dois processos começem a se comunicar. O UDP provê um serviço não confiável de transferência de dados – isto é, quando um processo envia uma mensagem para dentro de um *socket* UDP, o protocolo *não* oferece garantias de que a mensagem chegará ao processo receptor. Além do mais, mensagens que chegam de fato ao processo receptor podem chegar fora de ordem.

O UDP não inclui um mecanismo de controle de congestionamento; portanto, um processo originador pode bombear dados para dentro de uma camada abaixo (a de rede) à taxa que quiser. (Observe, entretanto, que a vazão fim a fim real pode ser menor do que essa taxa em virtude da capacidade de transmissão limitada de enlaces intermediários ou pelo congestionamento.)

### Serviços não providos pelos protocolos de transporte da Internet

Organizamos os serviços do protocolo de transporte em quatro dimensões: transferência confiável de dados, vazão, temporização e segurança. Quais deles são providos pelo TCP e pelo UDP? Já vimos que o TCP fornece a transferência confiável de dados fim a fim, e sabemos também que ele pode ser facilmente aprimorado na camada de aplicação com o TLS para oferecer serviços de segurança. Mas em nossa breve descrição sobre o TCP e o UDP faltou mencionar as garantias de vazão e de temporização – serviços *não* fornecidos pelos protocolos de transporte da Internet de hoje. Isso significa que as aplicações sensíveis ao tempo, como a telefonia por Internet, não podem rodar na rede atual? A resposta decerto é negativa – a Internet tem recebido essas aplicações por muitos anos. Tais aplicações muitas vezes funcionam bem, por terem sido desenvolvidas para lidar, na medida do possível, com a falta de garantia. No entanto, mesmo o projeto inteligente possui suas limitações quando o atraso é excessivo, ou quando a vazão fim a fim é limitada. Em resumo, a Internet hoje pode oferecer serviços satisfatórios a aplicações sensíveis ao tempo, mas não garantias de temporização ou de largura de banda.

A Figura 2.5 mostra os protocolos de transporte usados por algumas aplicações populares da Internet. Vemos que e-mail, acesso a terminais remotos, a Web e transferência de arquivos usam o TCP. Essas aplicações escolheram o TCP principalmente porque ele oferece um serviço confiável de transferência de dados, garantindo que todos eles mais cedo ou mais tarde cheguem a seu destino. Como as aplicações de telefonia por Internet (como Skype) muitas vezes toleram alguma perda, mas exigem uma taxa mínima para que sejam eficazes, seus programadores em geral preferem rodá-las em cima do UDP, contornando assim o mecanismo de controle de congestionamento do TCP e evitando pacotes extras. Porém, como muitos *firewalls* são configurados para bloquear (quase todo) o tráfego UDP, as aplicações de telefonia por Internet quase sempre são projetadas para usar TCP como um apoio se a comunicação por UDP falhar.

## 2.1.5 Protocolos de camada de aplicação

Acabamos de aprender que processos de rede comunicam-se entre si enviando mensagens para dentro de *sockets*. Mas como essas mensagens são estruturadas? O que significam os vários campos nas mensagens? Quando os processos enviam as mensagens? Essas perguntas nos transportam para o mundo dos protocolos de camada de aplicação. Um **protocolo de camada de aplicação** define a maneira como processos de uma aplicação, que funcionam em sistemas finais diferentes, passam mensagens entre si. Em particular, um protocolo de camada de aplicação define:

- Os tipos de mensagens trocadas; por exemplo, de requisição e de resposta.
- A sintaxe dos vários tipos de mensagens, tais como os campos da mensagem e como os campos são delimitados.
- A semântica dos campos, isto é, o significado da informação nos campos.
- Regras para determinar quando e como um processo envia mensagens e responde a mensagens.

Alguns protocolos de camada de aplicação estão especificados em RFCs e, portanto, são de domínio público. Por exemplo, o protocolo de camada de aplicação da Web, HTTP (HyperText Transfer Protocol [RFC 7230]), está à disposição como um RFC. Se um programador de navegador seguir as regras do RFC do HTTP, o navegador estará habilitado a extrair páginas de qualquer servidor que também tenha seguido essas mesmas regras. Muitos outros protocolos de camada de aplicação são próprios e, de modo intencional, não estão disponíveis ao público; por exemplo, o Skype.

É importante distinguir aplicações de rede de protocolos de camada de aplicação, os quais são apenas um pedaço de aplicação de rede (embora muito importante, do nosso ponto de vista!). Examinemos alguns exemplos. A Web é uma aplicação cliente-servidor que

Aplicação	Protocolo de camada de aplicação	Protocolo de transporte subjacente
Correio eletrônico	SMTP (RFC 5321)	TCP
Acesso a terminal remoto	Telnet (RFC 854)	TCP
Web	HTTP 1.1 (RFC 7230)	TCP
Transferência de arquivo	FTP (RFC 959)	TCP
Streaming de multimídia	HTTP (p. ex., YouTube), DASH	TCP
Telefonia por Internet	SIP (RFC 3261), RTP (RFC 3550) ou proprietária (p. ex., Skype)	UDP ou TCP

**Figura 2.5** Aplicações populares da Internet, seus protocolos de camada de aplicação e seus protocolos de transporte subjacentes.

permite aos usuários obter documentos de servidores por demanda. A aplicação Web consiste em muitos componentes, entre eles um padrão para formato de documentos (i.e., HTML), navegadores Web (p. ex., Chrome e Microsoft Internet Explorer), servidores Web (p. ex., Apache e IIS da Microsoft) e um protocolo de camada de aplicação. O HTTP, protocolo de camada de aplicação da Web, define o formato e a sequência das mensagens que são passadas entre o navegador e o servidor. Assim, ele é apenas um pedaço (embora importante) da aplicação Web. Como outro exemplo, veremos na Seção 2.6 que o serviço de vídeo da Netflix também possui muitos componentes, incluindo servidores que armazenam e transmitem vídeos, outros servidores que gerenciam a cobrança e outras funções para o cliente, clientes (p. ex., o aplicativo da Netflix no seu *smartphone*, *tablet* ou computador) e um protocolo DASH no nível da aplicação que define o formato e a sequência das mensagens trocadas entre o servidor da Netflix e o cliente. Assim, o DASH é apenas um pedaço (embora importante) da aplicação da Netflix.

### 2.1.6 Aplicações de rede abordadas neste livro

Novas aplicações de Internet são desenvolvidas todos os dias. Em vez de tratarmos de um grande número dessas aplicações de maneira enciclopédica, preferimos focalizar um pequeno número de aplicações ao mesmo tempo importantes e populares. Neste capítulo, discutiremos cinco aplicações populares: a Web, o correio eletrônico, o serviço de diretório, *streaming* de vídeo e aplicações P2P. Discutiremos primeiro a Web não apenas porque ela é uma aplicação de imensa popularidade, mas também porque seu protocolo de camada de aplicação, HTTP, é simples e fácil de entender. Em seguida, discutiremos o correio eletrônico, a primeira aplicação de enorme sucesso da Internet. O correio eletrônico é mais complexo do que a Web, pois usa não somente um, mas vários protocolos de camada de aplicação. Após o e-mail, estudaremos o DNS, que provê um serviço de diretório para a Internet. A maioria dos usuários não interage diretamente com o DNS; em vez disso, eles o chamam indiretamente por meio de outras aplicações (inclusive a Web, a transferência de arquivos e o correio eletrônico). O DNS ilustra de maneira primorosa como um componente de funcionalidade do núcleo da rede (tradução de nome de rede para endereço de rede) pode ser implementado na camada de aplicação da Internet. Depois, discutiremos aplicações de compartilhamento de arquivos P2P e completamos nosso estudo sobre aplicações com uma discussão sobre o *streaming* de vídeo sob demanda, incluindo a distribuição de vídeo armazenado através de redes de distribuição de conteúdo.

## 2.2 A WEB E O HTTP

Até a década de 1990, a Internet era usada principalmente por pesquisadores, acadêmicos e estudantes universitários para efetuar *login* em hospedeiros remotos, transferir arquivos de hospedeiros locais para remotos e vice-versa, enviar e receber notícias e correio eletrônico. Embora essas aplicações fossem (e continuem a ser) de extrema utilidade, a Internet era desconhecida fora das comunidades acadêmicas e de pesquisa. Então, no início da década de 1990, entrou em cena uma nova aplicação importantíssima – a World Wide Web (Berners-Lee, 1994). A Web é a aplicação que chamou a atenção do público em geral. Ela transformou drasticamente a maneira como pessoas interagem dentro e fora de seus ambientes de trabalho. Alçou a Internet de apenas mais uma entre muitas para, na essência, a única rede de dados.

Talvez o que mais atraia a maioria dos usuários da Web é que ela funciona *sob demanda*. Usuários recebem o que querem, quando querem, o que é diferente da transmissão de rádio e de televisão, que obriga a sintonizar quando o provedor disponibiliza o conteúdo. Além de funcionar sob demanda, a Web tem muitas outras características maravilhosas

que as pessoas adoram. É muito fácil para qualquer indivíduo disponibilizar informações na Web – todo mundo pode se transformar em editor a um custo baixíssimo. Hiperlinks e buscadores nos ajudam a navegar pelo oceano de informações. Fotos e vídeos estimulam nossos sentidos. Formulários, JavaScript, vídeo e muitos outros recursos nos habilitam a interagir com páginas e *sites*. E a Web serve como uma plataforma para o YouTube, e-mail baseado na Web (como Gmail) e a maioria das aplicações móveis, incluindo Instagram e Google Maps.

### 2.2.1 Descrição geral do HTTP

O **HTTP**, o protocolo da camada de aplicação da Web, está no coração da Web e é definido no (RFC 1945), no (RFC 7230) e no (RFC 7540). O HTTP é executado em dois programas: um cliente e outro servidor. Os dois, executados em sistemas finais diferentes, conversam entre si por meio da troca de mensagens HTTP. O HTTP define a estrutura dessas mensagens e o modo como o cliente e o servidor as trocam. Antes de explicarmos em detalhes o HTTP, devemos revisar a terminologia da Web.

Uma **página Web** (também denominada documento) é constituída por objetos. Um **objeto** é apenas um arquivo – tal como um arquivo HTML, uma imagem JPEG, um arquivo de Javascript, uma folha de estilo CSS ou um clipe de vídeo – que se pode acessar com um único URL. A maioria das páginas Web é constituída por um **arquivo-base HTML** e diversos objetos referenciados. Por exemplo, se uma página contiver um texto HTML e cinco imagens JPEG, então ela terá seis objetos: o arquivo-base HTML e mais as cinco imagens. O arquivo-base HTML referencia os outros objetos na página com os URLs dos objetos. Cada URL tem dois componentes: o nome de hospedeiro (*hostname*) do servidor que abriga o objeto e o nome do caminho do objeto. Por exemplo, no URL

`http://www.someSchool.edu/someDepartment/picture.gif`

`www.someSchool.edu` é o nome de hospedeiro, e `/someDepartment/picture.gif` é o nome do caminho. Como **navegadores Web** (p. ex., Internet Explorer e Chrome) também executam o lado cliente do HTTP, usaremos as palavras *navegador* e *cliente* indiferentemente nesse contexto. Os **servidores Web**, que executam o lado servidor do HTTP, abrigam objetos Web, cada um endereçado por um URL. São servidores Web populares o Apache e o Microsoft Internet Information Server.

O HTTP define como os clientes requisitam páginas aos servidores e como eles as transferem aos clientes. Discutiremos em detalhes a interação entre cliente e servidor mais adiante, mas a ideia geral está ilustrada na Figura 2.6. Quando um usuário requisita uma página Web (p. ex., clica sobre um hiperlink), o navegador envia ao servidor mensagens de requisição HTTP para os objetos da página. O servidor recebe as requisições e responde com mensagens de resposta HTTP que contêm os objetos.

O HTTP usa o TCP como seu protocolo de transporte subjacente (em vez de rodar em cima do UDP). O cliente HTTP primeiro inicia uma conexão TCP com o servidor. Uma vez estabelecida, os processos do navegador e do servidor acessam o TCP por meio de suas interfaces de *socket*. Como descrito na Seção 2.1, no lado cliente, a interface *socket* é a porta entre o processo cliente e a conexão TCP; no lado servidor, ela é a porta entre o processo servidor e a conexão TCP. O cliente envia mensagens de requisição HTTP para sua interface *socket* e recebe mensagens de resposta HTTP de sua interface *socket*. De maneira semelhante, o servidor HTTP recebe mensagens de requisição de sua interface *socket* e envia mensagens de resposta para sua interface *socket*. Assim que o cliente envia uma mensagem para sua interface *socket*, a mensagem sai de suas mãos e passa a estar “nas mãos” do TCP. Lembre-se de que na Seção 2.1 dissemos que o TCP oferece ao HTTP um serviço confiável de transferência de dados, o que implica que toda mensagem de requisição HTTP emitida por um processo cliente chegará intacta ao servidor. De maneira semelhante, toda mensagem de



**Figura 2.6** Comportamento de requisição-resposta do HTTP.

resposta HTTP emitida pelo processo servidor chegará intacta ao cliente. Percebemos, nesse ponto, uma das grandes vantagens de uma arquitetura de camadas – o HTTP não precisa se preocupar com dados perdidos ou com detalhes de como o TCP se recupera da perda de dados ou os reordena dentro da rede. Essa é a tarefa do TCP e dos protocolos das camadas inferiores da pilha de protocolos.

É importante notar que o servidor envia ao cliente os arquivos solicitados sem armazenar qualquer informação de estado sobre o cliente. Se determinado cliente solicita o mesmo objeto duas vezes em um período de poucos segundos, o servidor não responde dizendo que acabou de enviá-lo; em vez disso, manda de novo o objeto, pois já esqueceu por completo o que fez antes. Como o servidor HTTP não mantém informação alguma sobre clientes, o HTTP é denominado um **protocolo sem estado**. Salientamos também que a Web usa a arquitetura de aplicação cliente-servidor, como descrito na Seção 2.1. Um servidor Web está sempre em funcionamento, tem um endereço IP fixo e atende requisições de potencialmente milhões de navegadores diferentes.

A versão original do HTTP é chamada de HTTP/1.0 e remonta ao início da década de 1990 (RFC 1945). Em 2020, a maioria das transações HTTP ocorria pelo HTTP/1.1 (RFC 7230). Contudo, os navegadores e servidores Web cada vez mais suportam também uma nova versão do HTTP, chamada de HTTP/2 (RFC 7540). No final desta seção, apresentaremos uma introdução ao HTTP/2.

### 2.2.2 Conexões persistentes e não persistentes

Em muitas aplicações da Internet, o cliente e o servidor se comunicam por um período prolongado, em que o cliente faz uma série de requisições e o servidor responde a cada uma. Dependendo da aplicação e de como ela está sendo usada, a série de requisições pode ser feita de forma consecutiva, periodicamente em intervalos regulares ou de modo esporádico. Quando a interação cliente-servidor acontece por meio de conexão TCP, o desenvolvedor da aplicação precisa tomar uma importante decisão – cada par de requisição/resposta deve ser enviado por uma conexão TCP *distinta* ou todas as requisições e suas respostas devem ser enviadas por uma *mesma* conexão TCP? Na primeira abordagem, a aplicação utiliza **conexões não persistentes**; e na segunda abordagem, **conexões persistentes**. Para entender melhor esse assunto, vamos analisar as vantagens e desvantagens das conexões não persistentes e das conexões persistentes no contexto de uma aplicação específica, o HTTP, que pode utilizar as duas. Embora o HTTP utilize conexões persistentes em seu modo padrão, os clientes e servidores HTTP podem ser configurados para utilizar a conexão não persistente.

## O HTTP com conexões não persistentes

Vamos percorrer as etapas da transferência de uma página de um servidor para um cliente para o caso de conexões não persistentes. Suponhamos que uma página consista em um arquivo-base HTML e em dez imagens JPEG e que todos esses 11 objetos residam no mesmo servidor. Suponha também que o URL para o arquivo-base HTTP seja

`http://www.someSchool.edu/someDepartment/home.index`

Eis o que acontece:

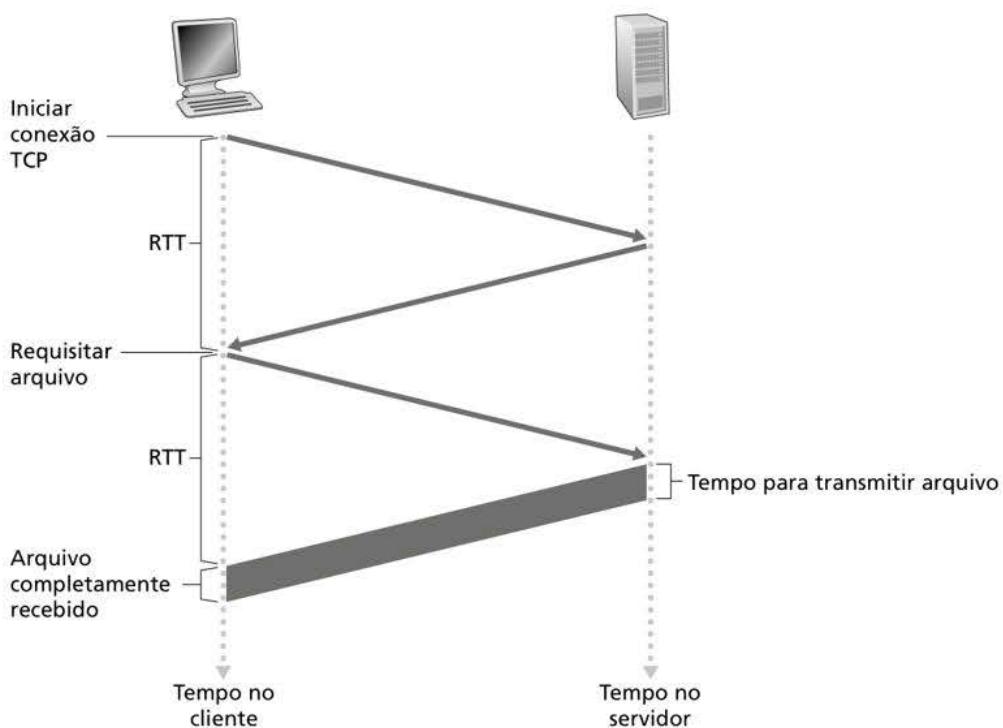
1. O processo cliente HTTP inicia uma conexão TCP para o servidor `www.someSchool.edu` na porta número 80, que é o número de porta padrão para o HTTP. Associados à conexão TCP, haverá um *socket* no cliente e um *socket* no servidor.
2. O cliente HTTP envia uma mensagem de requisição HTTP ao servidor por meio de seu *socket*. Essa mensagem inclui o nome do caminho `/someDepartment/home.index`. (Discutiremos mensagens HTTP mais detalhadamente logo adiante.)
3. O processo servidor HTTP recebe a mensagem de requisição por meio de seu *socket*, extrai o objeto `/someDepartment/home.index` de seu armazenamento (RAM ou disco), encapsula-o em uma mensagem de resposta HTTP e a envia ao cliente pelo *socket*.
4. O processo servidor HTTP ordena ao TCP que encerre a conexão TCP. (Mas, na realidade, o TCP só a encerrará quando tiver certeza de que o cliente recebeu a mensagem de resposta intacta.)
5. O cliente HTTP recebe a mensagem de resposta e a conexão TCP é encerrada. A mensagem indica que o objeto encapsulado é um arquivo HTML. O cliente extrai o arquivo da mensagem de resposta, analisa o arquivo HTML e encontra referências aos dez objetos JPEG.
6. As primeiras quatro etapas são repetidas para cada um dos objetos JPEG referenciados.

À medida que recebe a página Web, o navegador a apresenta ao usuário. Dois navegadores diferentes podem interpretar (i.e., exibir ao usuário) uma página de modos um pouco diferentes. O HTTP não tem a ver com o modo como uma página Web é interpretada por um cliente. As especificações do HTTP ([RFC 1945] e [RFC 7540]) definem apenas o protocolo de comunicação entre o programa cliente HTTP e o programa servidor HTTP.

As etapas apresentadas ilustram a utilização de conexões não persistentes, nas quais cada conexão TCP é encerrada após o servidor enviar um objeto – a conexão não persiste para outros objetos. O HTTP/1.0 emprega conexões TCP não persistentes. Note que cada conexão TCP transporta exatamente uma mensagem de requisição e uma mensagem de resposta. Assim, nesse exemplo, quando um usuário solicita a página Web, são geradas 11 conexões TCP.

Nos passos descritos, fomos intencionalmente vagos sobre se os clientes obtêm as dez JPEGs por meio de dez conexões TCP em série ou se algumas delas são recebidas por conexões TCP paralelas. Na verdade, usuários podem configurar navegadores modernos para controlar o grau de paralelismo. Os navegadores podem abrir múltiplas conexões TCP paralelas e solicitar partes diferentes de uma página Web através delas. Como veremos no próximo capítulo, a utilização de conexões paralelas reduz o tempo de resposta.

Antes de continuarmos, vamos fazer um rascunho de cálculo para estimar o tempo que transcorre entre a requisição e o recebimento de um arquivo-base HTTP por um cliente. Para essa finalidade, definimos o **tempo de viagem de ida e volta (RTT)**, do inglês *round-trip time*), ou seja, o tempo que leva para um pequeno pacote viajar do cliente ao servidor e de volta ao cliente. O RTT inclui atrasos de propagação de pacotes, de fila de pacotes em roteadores e comutadores intermediários e de processamento de pacotes. (Esses atrasos foram discutidos na Seção 1.4.) Considere, agora, o que acontece quando um usuário clica sobre um hiperlink. Como ilustrado na Figura 2.7, isso faz com que o navegador inicie uma conexão TCP entre ele e o servidor, o que envolve uma “apresentação de três vias” (*three-way handshake*) – o cliente



**Figura 2.7** Cálculo simples para o tempo necessário para solicitar e receber um arquivo HTML.

envia um pequeno segmento TCP ao servidor, este o reconhece e responde com um pequeno segmento ao cliente que, por fim, o reconhece novamente para o servidor. As duas primeiras partes da apresentação de três vias representam um RTT. Após concluir-las, o cliente envia a mensagem de requisição HTTP combinada com a terceira parte da apresentação de três vias (o reconhecimento) por meio da conexão TCP. Assim que a mensagem de requisição chega ao servidor, este envia o arquivo HTML por meio da conexão TCP. Essa requisição/resposta HTTP consome outro RTT. Assim, de modo aproximado, o tempo total de resposta são dois RTTs mais o tempo de transmissão do arquivo HTML no servidor.

### O HTTP com conexões persistentes

Conexões não persistentes têm algumas desvantagens. Primeiro, uma nova conexão deve ser estabelecida e mantida para *cada objeto solicitado*. Para cada conexão, devem ser alocados *buffers* TCP e conservadas variáveis TCP tanto no cliente quanto no servidor. Isso pode sobrecarregar seriamente o servidor Web, que poderá estar processando requisições de centenas de diferentes clientes ao mesmo tempo. Segundo, como acabamos de descrever, cada objeto sofre dois RTTs – um RTT para estabelecer a conexão TCP e outro para solicitar e receber um objeto.

Em conexões persistentes HTTP/1.1, o servidor deixa a conexão TCP aberta após enviar resposta. Requisições e respostas subsequentes entre os mesmos cliente e servidor podem ser enviadas por meio da mesma conexão. Em particular, uma página Web inteira (no exemplo anterior, o arquivo-base HTML e as dez imagens) pode ser enviada mediante uma única conexão TCP persistente. Além do mais, várias páginas residindo no mesmo servidor podem ser enviadas ao mesmo cliente por uma única conexão TCP persistente. Essas requisições por objetos podem ser feitas em sequência sem ter de esperar por respostas a requisições pendentes (paralelismo ou *pipelining*). Em geral, o servidor HTTP fecha uma conexão quando ela não é usada durante certo tempo (um intervalo de temporização configurável). Quando o servidor recebe requisições consecutivas, os objetos são enviados de

forma ininterrupta. O modo default do HTTP usa conexões persistentes com paralelismo. Faremos uma comparação quantitativa entre os desempenhos de conexões persistentes e não persistentes nos exercícios de fixação dos Capítulos 2 e 3. Aconselhamos o leitor interessado a consultar Heidemann (1997), Nielsen (1997) e o RFC 7540.

### 2.2.3 Formato da mensagem HTTP

As especificações do HTTP (RFC 1945; RFC 7230; RFC 7540) incluem as definições dos formatos das mensagens HTTP. Há dois tipos delas: de requisição e de resposta, ambas discutidas a seguir.

#### Mensagem de requisição HTTP

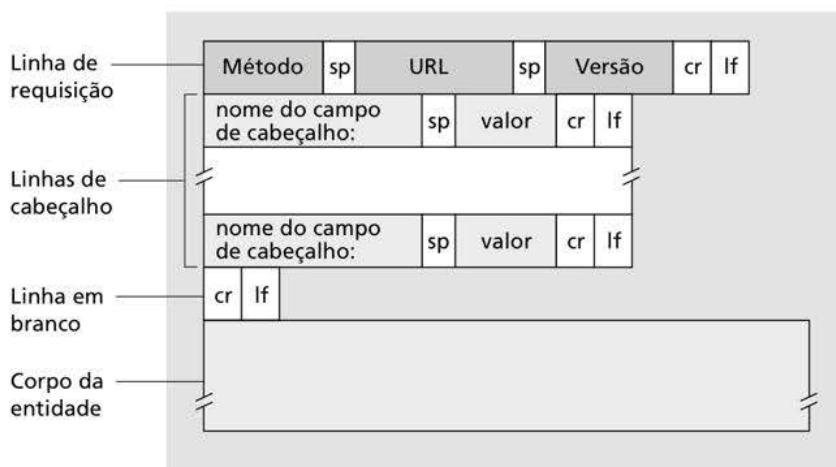
Apresentamos a seguir uma mensagem de requisição HTTP típica:

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

Podemos aprender bastante examinando essa simples mensagem de requisição. Primeiro, vemos que ela está escrita em texto ASCII comum, de modo que pode ser lida por qualquer um que conheça o linguajar dos computadores. Segundo, vemos que ela é constituída por cinco linhas, cada qual seguida de um *carriage return* e *line feed* (fim de linha). A última linha é seguida de um comando adicional de *carriage return* e *line feed*. Embora essa específica mensagem de requisição tenha cinco linhas, uma mensagem de requisição pode ter muitas mais e também menos do que isso, podendo conter até mesmo uma única linha. A primeira linha é denominada **linha de requisição**; as subsequentes são denominadas **linhas de cabeçalho**. A linha de requisição tem três campos: o do método, o do URL e o da versão do HTTP. O campo do método pode assumir vários valores diferentes, entre eles GET, POST, HEAD, PUT e DELETE. A grande maioria das mensagens de requisição HTTP emprega o método GET, o qual é usado quando o navegador requisita um objeto e este é identificado no campo do URL. Nesse exemplo, o navegador está requisitando o objeto /somedir/page.html. O campo da versão é autoexplicativo. Nesse exemplo, o navegador executa a versão HTTP/1.1.

Vamos agora examinar as linhas de cabeçalho do exemplo. A linha de cabeçalho Host:www.someschool.edu especifica o hospedeiro no qual o objeto reside. Talvez você ache que ela é desnecessária, pois já existe uma conexão TCP para o hospedeiro. Mas, como veremos na Seção 2.2.5, a informação fornecida pela linha de cabeçalho do hospedeiro é exigida por *caches proxy* da Web. Ao incluir a linha de cabeçalho Connection: close, o navegador está dizendo ao servidor que não quer usar conexões persistentes; quer que o servidor feche a conexão após o envio do objeto requisitado. A linha de cabeçalho User-agent: especifica o agente de usuário, isto é, o tipo de navegador que está fazendo a requisição ao servidor. Neste caso, o agente de usuário é o Mozilla/5.0, um navegador Firefox. Essa linha de cabeçalho é útil porque, na verdade, o servidor pode enviar versões diferentes do mesmo objeto a tipos diferentes de agentes de usuário. (Cada versão é endereçada pelo mesmo URL.) Por fim, o cabeçalho Accept-language: mostra que o usuário prefere receber uma versão em francês do objeto se este existir no servidor; se não existir, o servidor deve enviar a versão padrão. O cabeçalho Accept-language: é apenas um dos muitos de negociação de conteúdo disponíveis no HTTP.

Após examinar um exemplo, vamos agora analisar o formato geral de uma mensagem de requisição, ilustrado na Figura 2.8. Vemos que tal formato é muito parecido com nosso exemplo anterior. Contudo, você provavelmente notou que, após as linhas de cabeçalho



**Figura 2.8** Formato geral de uma mensagem de requisição HTTP.

(e após a linha adicional com *carriage return* e *line feed*), há um “corpo de entidade”. O corpo de entidade fica vazio com o método GET, mas é utilizado com o método POST. Um cliente HTTP em geral usa o método POST quando o usuário preenche um formulário – por exemplo, quando fornece palavras de busca a um *site* buscador. Com uma mensagem POST, o usuário continua solicitando uma página Web ao servidor, mas seu conteúdo depende do que ele escreveu nos campos do formulário. Se o valor do campo de método for POST, então o corpo de entidade conterá o que o usuário digitou nos campos do formulário.

Seríamos omissos se não mencionássemos que uma requisição gerada com um formulário não utiliza necessariamente o método POST. Ao contrário, formulários HTML costumam empregar o método GET e incluem os dados digitados (nos campos do formulário) no URL requisitado. Por exemplo, se um formulário usar o método GET, tiver dois campos e as entradas desses dois campos forem *monkeys* e *bananas*, então a estrutura do URL será [www.somesite.com/animalsearch?monkeys&bananas](http://www.somesite.com/animalsearch?monkeys&bananas). Ao navegar normalmente pela Web, você talvez já tenha notado URLs extensos como esse.

O método HEAD é semelhante ao GET. Quando um servidor recebe uma requisição com o método HEAD, responde com uma mensagem HTTP, mas deixa de fora o objeto requisitado. Esse método é usado com frequência pelos programadores de aplicação para depuração. O método PUT é muito usado junto com ferramentas de edição da Web. Permite que um usuário carregue um objeto para um caminho (diretório) específico em um servidor Web específico. O método PUT também é usado por aplicações que precisam carregar objetos para servidores Web. O método DELETE permite que um usuário, ou uma aplicação, elimine um objeto em um servidor Web.

### Mensagem de resposta HTTP

Apresentamos a seguir uma mensagem de resposta HTTP típica. Essa mensagem poderia ser a resposta ao exemplo de mensagem de requisição que acabamos de discutir.

```

HTTP/1.1 200 OK
Connection: close
Date: Tue, 18 Aug 2015 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 18 Aug 2015 15:11:03 GMT
Content-Length: 6821
Content-Type: text/html
(dados dados dados dados dados ...)

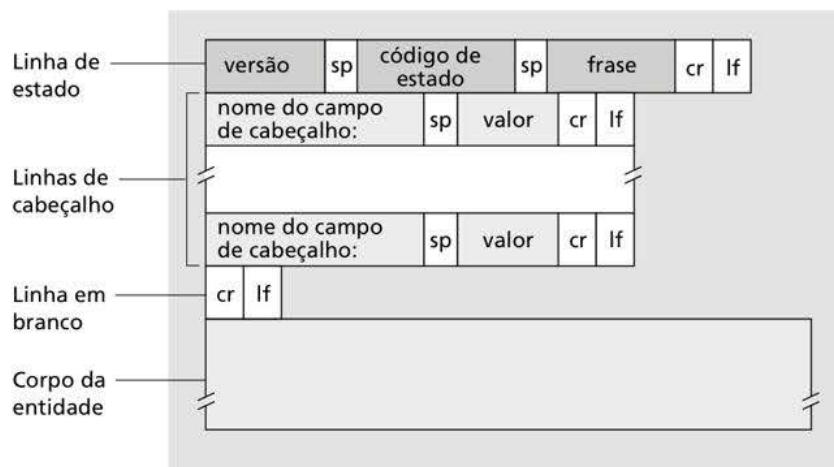
```

Vamos examinar com cuidado essa mensagem de resposta. Ela tem três seções: uma **linha inicial** ou *linha de estado*, seis **linhas de cabeçalho** e, em seguida, o **corpo da entidade**, que é a parte principal da mensagem – contém o objeto solicitado (representado por dados dados dados dados ...). A linha de estado tem três campos: o de versão do protocolo, um código de estado e uma mensagem de estado correspondente. Neste exemplo, ela mostra que o servidor está usando o HTTP/1.1 e que está tudo OK (i.e., o servidor encontrou e está enviando o objeto solicitado).

Agora, vamos ver as linhas de cabeçalho. O servidor usa `Connection: close` para informar ao cliente que fechará a conexão TCP após enviar a mensagem. A linha de cabeçalho `Date:` indica a hora e a data em que a resposta HTTP foi criada e enviada pelo servidor. Note que esse não é o horário em que o objeto foi criado nem o de sua modificação mais recente; é a hora em que o servidor extraiu o objeto de seu sistema de arquivos, inseriu-o na mensagem de resposta e a enviou. A linha de cabeçalho `Server:` mostra que a mensagem foi gerada por um servidor Web Apache, semelhante à linha de cabeçalho `User-agent:` na mensagem de requisição HTTP. A linha de cabeçalho `Last-Modified:` indica a hora e a data em que o objeto foi criado ou sofreu a última modificação. Esse cabeçalho, que logo estudaremos em mais detalhes, é fundamental para fazer *cache* do objeto, tanto no cliente local quanto em servidores de *cache* da rede (também conhecidos como servidores *proxy*). A linha de cabeçalho `Content-Length:` indica o número de *bytes* do objeto que está sendo enviado, e a linha de cabeçalho `Content-Type:` mostra que o objeto presente no corpo da mensagem é um texto HTML. (O tipo do objeto é oficialmente indicado pelo cabeçalho `Content-Type:`, e não pela extensão do arquivo.)

Após examinar um exemplo, vamos agora analisar o formato geral de uma mensagem de resposta, ilustrado na Figura 2.9. Esse formato geral de mensagem de resposta condiz com o exemplo anterior. Mas falemos um pouco mais sobre códigos de estado e suas frases, que indicam o resultado da requisição. Eis alguns códigos de estado e frases associadas comuns:

- 200 OK: requisição bem-sucedida e a informação é entregue com a resposta.
- 301 Moved Permanently: objeto requisitado foi removido em definitivo; novo URL é especificado no cabeçalho `Location:` da mensagem de resposta. O *software* do cliente recuperará automaticamente o novo URL.
- 400 Bad Request: código genérico de erro que indica que a requisição não pôde ser entendida pelo servidor.
- 404 Not Found: o documento requisitado não existe no servidor.
- 505 HTTP Version Not Supported: a versão do protocolo HTTP requisitada não é suportada pelo servidor.



**Figura 2.9** Formato geral de uma mensagem de resposta HTTP.

Você gostaria de ver uma mensagem de resposta HTTP real? É muito recomendável e muito fácil! Primeiro, dê um comando Telnet em seu servidor favorito. Em seguida, digite uma mensagem de requisição de uma linha solicitando algum objeto abrigado no servidor. Por exemplo, se você tem acesso a um *prompt* de comando, digite:

```
telnet gaia.cs.umass.edu 80
GET /kurose_ross/interactive/index.php HTTP/1.1
Host: gaia.cs.umass.edu
```

(Pressione duas vezes a tecla “Enter” após digitar a última linha.) Essa sequência de comandos abre uma conexão TCP para a porta número 80 do hospedeiro `gaia.cs.umass.edu` e, em seguida, envia a mensagem de requisição HTTP. Deverá aparecer uma mensagem de resposta que inclui o arquivo-base HTML dos exercícios de fixação interativos deste livro. Se preferir apenas ver as linhas da mensagem HTTP e não receber o objeto em si, substitua GET por HEAD.

Nesta seção, discutimos várias linhas de cabeçalho que podem ser usadas em mensagens de requisição e de resposta HTTP. A especificação do HTTP define muitas outras linhas de cabeçalho que podem ser inseridas por navegadores, servidores Web e servidores de *cache* da rede. Vimos apenas um pouco do total de linhas de cabeçalho. Examinaremos mais algumas a seguir e mais um pouco quando discutirmos armazenagem Web na Seção 2.2.5. Uma discussão muito abrangente e fácil de ler sobre o protocolo HTTP, seus cabeçalhos e códigos de estado pode ser encontrada em Krishnamurthy (2001).

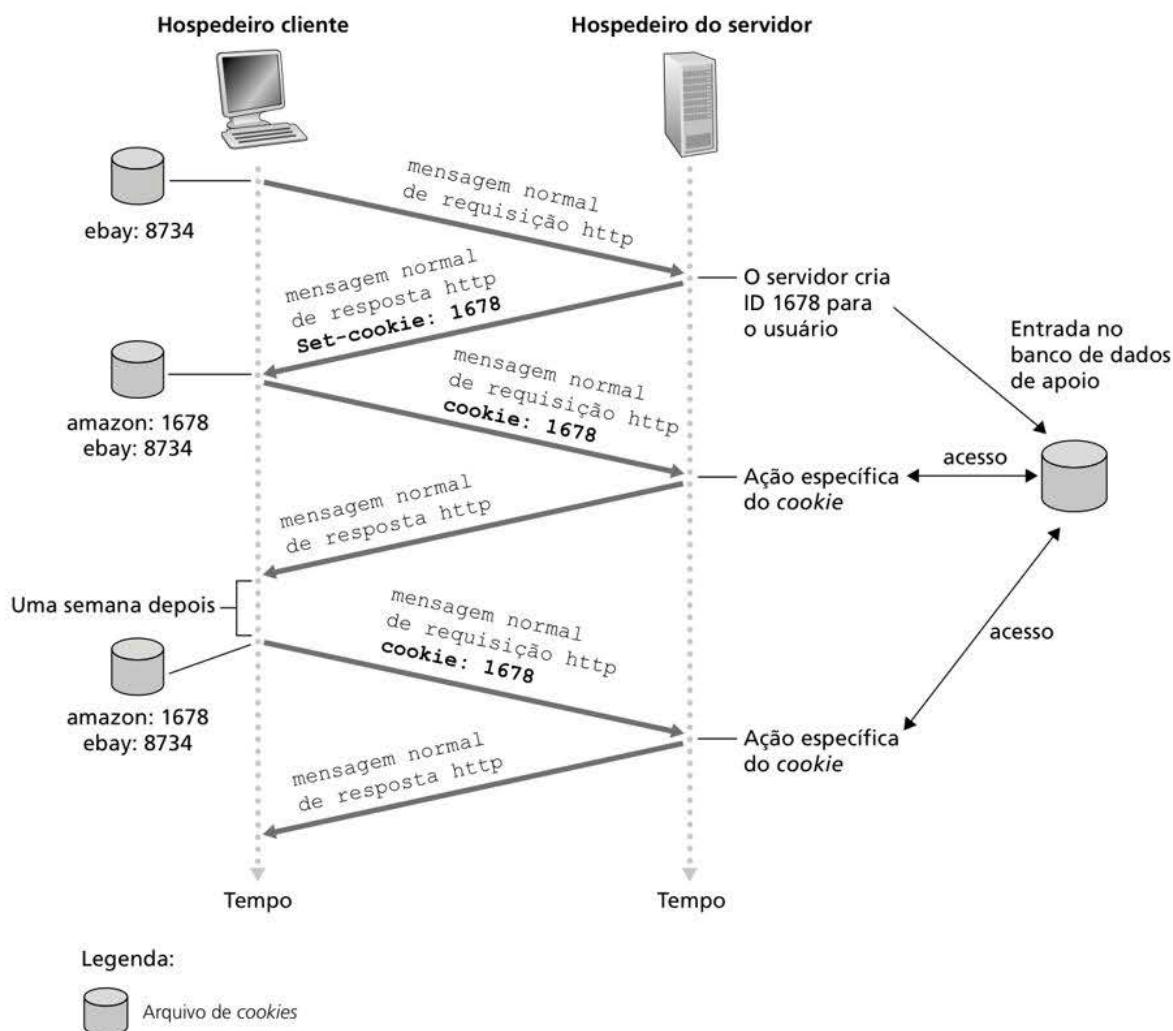
Como um navegador decide quais linhas de cabeçalho serão incluídas em uma mensagem de requisição? Como um servidor Web decide quais linhas de cabeçalho serão incluídas em uma mensagem de resposta? Um navegador vai gerar linhas de cabeçalho em função de seu tipo e versão, da configuração do usuário para o navegador e se o navegador tem uma versão do objeto em *cache*, possivelmente desatualizada. Servidores Web se comportam de maneira semelhante: há diferentes produtos, versões e configurações, e todos influenciam as linhas de cabeçalho que são incluídas nas mensagens de resposta.

## 2.2.4 Interação usuário-servidor: *cookies*

Mencionamos anteriormente que um servidor HTTP não tem estado, o que simplifica o projeto do servidor e vem permitindo que engenheiros desenvolvam servidores Web de alto desempenho que podem manipular milhares de conexões TCP simultâneas. No entanto, é sempre bom que um *site* identifique usuários, seja porque o servidor deseja restringir acesso, seja porque quer apresentar conteúdo em função da identidade do usuário. Para essas finalidades, o HTTP usa *cookies*. *Cookies*, definidos no (RFC 6265), permitem que *sites* monitorem seus usuários. Hoje, a maioria dos *sites* comerciais utiliza *cookies*.

Como ilustrado na Figura 2.10, a tecnologia dos *cookies* tem quatro componentes: (1) uma linha de cabeçalho de *cookie* na mensagem de resposta HTTP; (2) uma linha de cabeçalho de *cookie* na mensagem de requisição HTTP; (3) um arquivo de *cookie* mantido no sistema final do usuário e gerenciado pelo navegador do usuário; (4) um banco de dados de apoio no *site*. Utilizando a Figura 2.10, vamos esmiuçar um exemplo de como os *cookies* são usados. Suponha que Susan, que sempre acessa a Web usando o Internet Explorer de seu PC, acesse o Amazon.com pela primeira vez, e que, no passado, ela já tenha visitado o *site* da eBay. Quando a requisição chega ao servidor da Amazon, ele cria um número de identificação exclusivo e uma entrada no seu banco de dados de apoio, que é indexado pelo número de identificação. Então, o servidor da Amazon responde ao navegador de Susan, incluindo na resposta HTTP um cabeçalho *Set-cookie*: que contém o número de identificação. Por exemplo, a linha de cabeçalho poderia ser:

```
Set-cookie: 1678
```



**Figura 2.10** Mantendo o estado do usuário com *cookies*.

Quando recebe a mensagem de resposta HTTP, o navegador de Susan vê o cabeçalho `Set-cookie:` e, então, anexa uma linha ao arquivo especial de *cookies* que ele gerencia. Essa linha inclui o nome de hospedeiro do servidor e seu número de identificação no cabeçalho. Observe que o arquivo de *cookie* já possui uma entrada para o eBay, pois Susan já visitou esse *site* no passado. Toda vez que ela requisita uma página enquanto navega pelo *site* da Amazon, seu navegador consulta seu arquivo de *cookies*, extrai seu número de identificação para esse *site* e insere na requisição HTTP uma linha de cabeçalho de *cookie* que inclui o número de identificação. Especificamente, cada uma de suas requisições HTTP ao servidor da Amazon inclui a linha de cabeçalho:

`Cookie: 1678`

Dessa maneira, o servidor da Amazon pode monitorar a atividade de Susan em seu *site* e, embora não saiba necessariamente que o nome dela é Susan, sabe com exatidão quais páginas o usuário 1678 visitou, em qual ordem e em quais horários! Então, pode utilizar *cookies* para oferecer um serviço de carrinho de compra – a Amazon pode manter uma lista de todas as compras de Susan, de modo que ela possa pagar por todas elas ao mesmo tempo, no final da sessão.

Se Susan voltar ao *site* da Amazon, digamos, uma semana depois, seu navegador continuará a inserir a linha de cabeçalho *Cookie*: 1678 nas mensagens de requisição. A Amazon pode recomendar produtos com base nas páginas que Susan visitou anteriormente. Se ela também se registrar no *site* – fornecendo seu nome completo, endereço de e-mail, endereço postal e informações de cartão de crédito –, a Amazon pode incluir essas informações no banco de dados e, assim, associar o nome de Susan com seu número de identificação (e com todas as páginas que ela consultou em suas visitas anteriores). É assim que a Amazon e outros *sites* de comércio eletrônico oferecem “compras com um só clique” – quando quiser comprar um item em uma visita posterior, Susan não precisará digitar de novo seu nome, número de cartão de crédito, nem endereço.

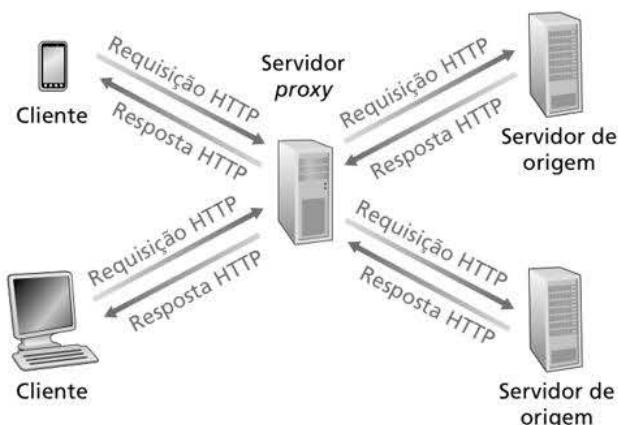
Essa discussão nos mostrou que *cookies* podem ser usados para identificar um usuário. Quando visitar um *site* pela primeira vez, um usuário pode fornecer dados de identificação (possivelmente seu nome). No decorrer das próximas sessões, o navegador passa um cabeçalho de *cookie* ao servidor durante todas as visitas subsequentes ao *site*, identificando, desse modo, o usuário ao servidor. Assim, vemos que os *cookies* podem ser usados para criar uma camada de sessão de usuário sobre HTTP sem estado. Por exemplo, quando um usuário acessa uma aplicação de e-mail baseada na Web (como o Hotmail), o navegador envia informações de *cookie* ao servidor, permitindo que o servidor identifique o usuário por meio da sessão deste com a aplicação.

Embora *cookies* quase sempre simplifiquem a experiência de compra pela Internet, continuam provocando muita controvérsia porque também podem ser considerados invasão da privacidade do usuário. Como acabamos de ver, usando uma combinação de *cookies* e informações de conta fornecidas pelo usuário, um *site* pode ficar sabendo muita coisa sobre esse usuário e, potencialmente, vender essas informações para algum terceiro.

## 2.2.5 Caches Web

Um **cache Web** – também denominado **servidor proxy** – é uma entidade da rede que atende requisições HTTP em nome de um servidor Web de origem. O *cache* Web tem seu próprio disco de armazenagem e mantém, dentro dele, cópias de objetos recentemente requisitados. Como ilustrado na Figura 2.11, o navegador de um usuário pode ser configurado de modo que todas as suas requisições HTTP sejam dirigidas primeiro ao *cache* Web. Uma vez que esteja configurado um navegador, cada uma das requisições de um objeto que o navegador faz é primeiro dirigida ao *cache* Web. Como exemplo, suponha que um navegador esteja requisitando o objeto <http://www.someschool.edu/campus.gif>. Eis o que acontece:

1. O navegador estabelece uma conexão TCP com o *cache* Web e envia a ele uma requisição HTTP para o objeto.



**Figura 2.11** Clientes requisitando objetos por meio de um cache Web.

2. O *cache* Web verifica se tem uma cópia do objeto armazenada localmente. Se tiver, envia o objeto ao navegador do cliente, dentro de uma mensagem de resposta HTTP.
3. Se não tiver o objeto, o *cache* Web abre uma conexão TCP com o servidor de origem, isto é, com `www.someschool.edu`. Então, envia uma requisição HTTP do objeto para a conexão TCP. Após recebê-la, o servidor de origem envia o objeto ao *cache* Web, dentro de uma resposta HTTP.
4. Quando recebe o objeto, o *cache* Web guarda uma cópia em seu armazenamento local e envia outra, dentro de uma mensagem de resposta HTTP, ao navegador do cliente (pela conexão TCP existente entre o navegador do cliente e o *cache* Web).

Note que um *cache* é, ao mesmo tempo, um servidor e um cliente. Quando recebe requisições de um navegador e lhe envia respostas, é um servidor. Quando envia requisições para um servidor de origem e recebe respostas dele, é um cliente.

Em geral, é um Provedor de Serviços de Internet (ISP, do inglês *Internet Service Provider*) que compra e instala um *cache* Web. Por exemplo, uma universidade poderia instalar um *cache* na rede de seu campus e configurar todos os navegadores apontando para esse *cache*. Ou um importante ISP residencial (como a Comcast) poderia instalar um ou mais *caches* em sua rede e configurar antecipadamente os navegadores que fornece apontando para os *caches* instalados.

O *cache* na Web tem tido ampla utilização na Internet por duas razões. Primeiro, porque pode reduzir substancialmente o tempo de resposta para a requisição de um cliente, em particular se o gargalo da largura de banda entre o cliente e o servidor de origem for muito menor do que aquele entre o cliente e o *cache*. Se houver uma conexão de alta velocidade entre o cliente e o *cache*, como em geral é o caso, e se este tiver o objeto requisitado, então ele poderá entregar com rapidez o objeto ao cliente. Segundo, como logo ilustraremos com um exemplo, *caches* Web podem reduzir de modo substancial o tráfego no enlace de acesso de uma instituição qualquer à Internet. Com a redução do tráfego, a instituição (p. ex., uma empresa ou uma universidade) não precisa ampliar sua largura de banda tão rapidamente, o que diminui os custos. Além disso, *caches* Web podem reduzir bastante o tráfego na Internet como um todo, melhorando, assim, o desempenho para todas as aplicações.

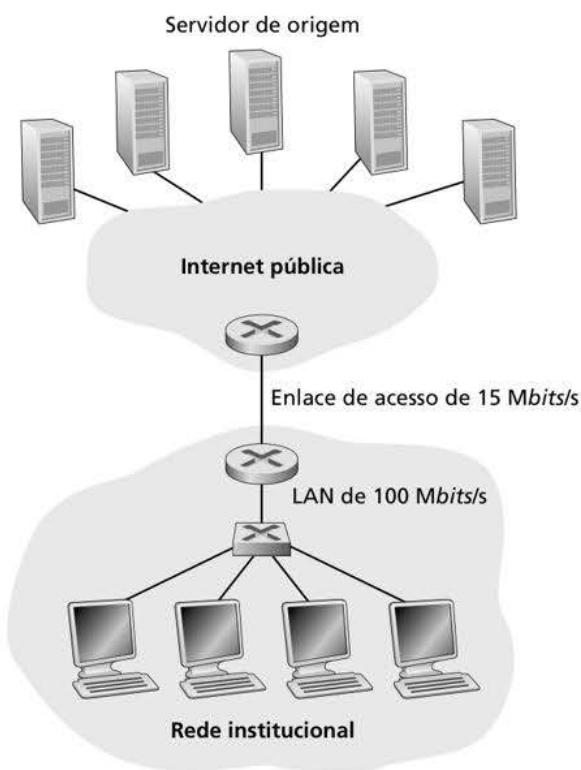
Para entender melhor os benefícios dos *caches*, vamos considerar um exemplo no contexto da Figura 2.12. Essa figura mostra duas redes: uma rede institucional e o resto da Internet pública. A rede institucional é uma rede local (LAN, do inglês *local area network*) de alta velocidade. Um roteador da rede institucional e um roteador da Internet estão ligados por um enlace de 15 Mbit/s. Os servidores de origem estão todos ligados à Internet, mas localizados pelo mundo todo. Suponha que o tamanho médio do objeto seja 1 Mbit/s e que a taxa média de requisição dos navegadores da instituição até os servidores de origem seja de 15 requisições por segundo. Imagine também que o tamanho das mensagens de requisição HTTP seja insignificante e, portanto, elas não criem tráfego nas redes ou no enlace de acesso (do roteador da instituição até o da Internet). Suponha ainda que o tempo entre o envio de uma requisição HTTP (dentro de um datagrama IP) pelo roteador do lado da Internet do enlace de acesso mostrado na Figura 2.12 e o recebimento da resposta (em geral, dentro de muitos datagramas IPs) seja de 2 segundos, em média. Esse último atraso é denominado informalmente “atraso da Internet”.

O tempo de resposta total – isto é, aquele transcorrido entre a requisição de um objeto feita pelo navegador e o recebimento dele – é a soma do atraso da LAN, do atraso de acesso (i.e., o atraso entre os dois roteadores) e do atraso da Internet. Vamos fazer agora um cálculo bastante rudimentar para estimar esse atraso. A intensidade de tráfego na LAN (veja a Seção 1.4.2) é

$$(15 \text{ requisições/s}) \cdot (1 \text{ Mbit/s/requisição}) / (100 \text{ Mbit/s}) = 0,15$$

ao passo que a intensidade de tráfego no enlace de acesso (do roteador da Internet ao da instituição) é

$$(15 \text{ requisições/s}) \cdot (1 \text{ Mbit/s/requisição}) / (15 \text{ Mbit/s}) = 1$$



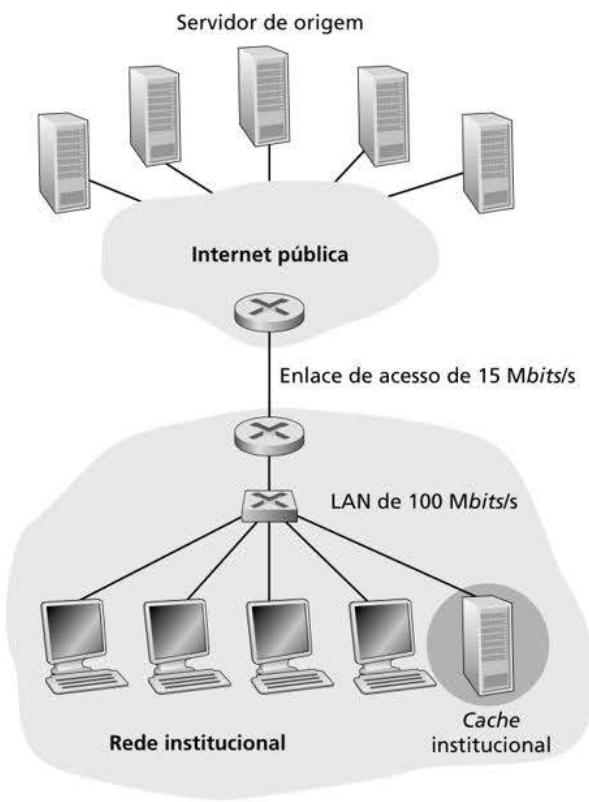
**Figura 2.12** Gargalo entre uma rede institucional e a Internet.

Uma intensidade de tráfego de 0,15 em uma LAN resulta em, no máximo, dezenas de milisegundos de atraso; por conseguinte, podemos desprezar o atraso da LAN. Contudo, como discutimos na Seção 1.4.2, à medida que a intensidade de tráfego se aproxima de 1 (como é o caso do enlace de acesso da Figura 2.12), o atraso em um enlace se torna muito grande e cresce sem limites. Assim, o tempo médio de resposta para atender requisições será da ordem de minutos, se não for maior, o que é inaceitável para os usuários da instituição. Obviamente, algo precisa ser feito.

Uma possível solução seria aumentar a velocidade de acesso de 15 Mbits/s para, digamos, 100 Mbits/s. Isso reduziria a intensidade de tráfego no enlace de acesso a 0,15, o que se traduziria em atrasos desprezíveis entre os dois roteadores. Nesse caso, o tempo total de resposta seria de mais ou menos 2 segundos, isto é, o atraso da Internet. Mas essa solução também significa que a instituição tem de atualizar seu enlace de acesso de 15 Mbits/s para 100 Mbits/s, o que pode ser muito dispendioso.

Considere agora a solução alternativa de não atualizar o enlace de acesso, mas, em vez disso, instalar um *cache* Web na rede institucional. Essa solução é ilustrada na Figura 2.13. A taxa de resposta local – a fração de requisições atendidas por um *cache* – em geral fica na faixa de 0,2 a 0,7 na prática. Para fins de ilustração, vamos supor que a taxa de resposta local do *cache* dessa instituição seja 0,4. Como os clientes e o *cache* estão conectados à mesma LAN de alta velocidade, 40% das requisições serão atendidas quase de imediato pelo *cache*, digamos, em 10 milissegundos. Mesmo assim, os demais 60% das requisições ainda precisam ser atendidos pelos servidores de origem. Mas com apenas 60% dos objetos requisitados passando pelo enlace de acesso, a intensidade de tráfego neste diminui de 1,0 para 0,6. Em geral, uma intensidade de tráfego menor do que 0,8 corresponde a um atraso pequeno, digamos, dezenas de milissegundos, no caso de um enlace de 15 Mbits/s. Esse atraso é desprezível se comparado aos 2 segundos do atraso da Internet. Dadas essas considerações, o atraso médio é, portanto,

$$0,4 \cdot (0,01 \text{ s}) + 0,6 \cdot (2,01 \text{ s})$$



**Figura 2.13** Acrescentando um cache à rede institucional.

que é ligeiramente maior do que 1,2 segundo. Assim, essa segunda solução resulta em tempo de resposta até menor do que o da primeira e não requer que a instituição atualize seu enlace com a Internet. Evidentemente, a instituição terá de comprar e instalar um *cache* Web. Mas esse custo é baixo – muitos *caches* usam softwares de domínio público que rodam em PCs baratos.

Com o uso de **redes de distribuição de conteúdo** (CDNs, do inglês *content distribution networks*), *caches* Web estão cada vez mais desempenhando um papel importante na Internet. Uma empresa de CDN instala muitos *caches* geograficamente dispersos pela Internet, localizando assim grande parte do tráfego. Existem CDNs compartilhadas (como Akamai e Limelight) e CDNs dedicadas (como Google e Netflix). Discutiremos as CDNs em mais detalhes na Seção 2.6.

### GET condicional

Embora possa reduzir os tempos de resposta do ponto de vista do usuário, fazer *cache* introduz um novo problema – a cópia de um objeto existente no *cache* pode estar desatualizada. Em outras palavras, o objeto abrigado no servidor pode ter sido modificado desde a data em que a cópia entrou no *cache* no cliente. Felizmente, o HTTP tem um mecanismo que permite que um *cache* verifique se seus objetos estão atualizados. Esse mecanismo é denominado **GET condicional (conditional GET)** (RFC 7232). Uma mensagem de requisição HTTP é denominada uma mensagem GET condicional se (1) usar o método GET e (2) possuir uma linha de cabeçalho *If-Modified-Since*:

Para ilustrar como o GET condicional funciona, vamos examinar um exemplo. Primeiro, um *cache proxy* envia uma mensagem de requisição a um servidor em nome de um navegador requisitante:

```
GET /fruit/kiwi.gif HTTP/1.1
Host: www.exotiquecuisine.com
```

Segundo, o servidor Web envia ao *cache* uma mensagem de resposta com o objeto requisitado:

```
HTTP/1.1 200 OK
Date: Sat, 3 Oct 2015 15:39:29
Server: Apache/1.3.0 (Unix)
Last-Modified: Wed, 9 Sep 2015 09:23:24
Content-Type: image/gif

(dados dados dados dados dados ...)
```

O *cache* encaminha o objeto ao navegador requisitante, mas também o guarda em sua memória *cache* local. O importante é que ele também guarda, junto com o objeto, a data da última modificação. Terceiro, uma semana depois, outro navegador requisita ao *cache* o mesmo objeto, que ainda está ali. Como esse objeto pode ter sido modificado no servidor na semana anterior, o navegador realiza uma verificação de atualização emitindo um GET condicional. Especificamente, o *cache* envia:

```
GET /fruit/kiwi.gif HTTP/1.1
Host: www.exotiquecuisine.com
If-modified-since: Wed, 9 Sep 2015 09:23:24
```

Note que o valor da linha de cabeçalho *If-modified-since*: é idêntico ao da linha de cabeçalho *Last-Modified*: que foi enviada pelo servidor há uma semana. Esse GET condicional está dizendo ao servidor para enviar o objeto somente se ele tiver sido modificado desde a data especificada. Suponha que o objeto não tenha sofrido modificação desde 9 set. 2015 09:23:24. Então, em quarto lugar, o servidor Web envia uma mensagem de resposta ao *cache*:

```
HTTP/1.1 304 Not Modified
Date: Sat, 10 Oct 2015 15:39:29
Server: Apache/1.3.0 (Unix)

(corpo de mensagem vazio)
```

Vemos que, em resposta ao GET condicional, o servidor ainda envia uma mensagem de resposta, mas não inclui nela o objeto requisitado, o que apenas desperdiçaria largura de banda e aumentaria o tempo de resposta do ponto de vista do usuário, em particular se o objeto fosse grande. Note que, na linha de estado dessa última mensagem de resposta está inserido 304 *Not Modified*, que informa ao *cache* que ele pode seguir e transmitir ao navegador requisitante a cópia do objeto que está contida nele (no *cache proxy*).

## 2.2.6 HTTP/2

O HTTP/2 (RFC 7540), padronizado em 2015, foi a primeira nova versão do HTTP desde o HTTP/1.1, padronizado em 1997. Desde a padronização, o HTTP/2 decolou, com mais de 40% dos maiores 10 milhões de sites suportando HTTP/2 em 2020 (W3Techs). A maioria dos navegadores, incluindo Google Chrome, Internet Explorer, Safari, Opera e Firefox, também suportam HTTP/2.

Os principais objetivos do HTTP/2 são reduzir a latência percebida ao possibilitar a multiplexação de solicitações e respostas em uma *única* conexão TCP, permitir a priorização de solicitações, que o servidor envie informações não solicitadas (*push* do servidor) e oferecer compressão mais eficiente dos campos de cabeçalho HTTP. O HTTP/2 não altera os métodos, códigos de *status*, URLs ou campos de cabeçalho do HTTP. Em vez disso, o HTTP/2 muda a maneira como os dados são formatados e transportados entre o cliente e o servidor.

Para motivar a necessidade do HTTP/2, lembre-se que o HTTP/1.1 usa conexões TCP persistentes, que permite que uma página Web seja enviada de um servidor para o cliente por uma única conexão TCP. Por ter uma única conexão TCP por página Web, o número de *Head of Line blocking* no servidor é reduzido, e cada página Web transportada recebe o seu quinhão da largura de banda da rede (como discutido abaixo). Mas os desenvolvedores de navegadores Web logo descobriram que enviar todos os objetos em uma página Web por uma única conexão TCP possui um problema de **bloqueio de cabeça de fila (HOL)**, do inglês *Head of Line blocking*). Para entender o bloqueio HOL, considere uma página Web que inclui uma página de base HTML, um clipe de vídeo grande próximo à parte superior da página e muitos objetos pequenos abaixo do vídeo. Além disso, suponha que há um enlace congestionado de baixa a média velocidade (p. ex., um enlace sem fio de baixa velocidade) no caminho entre o servidor e o cliente. Usando uma única conexão TCP, o clipe de vídeo demorará muito para atravessar o gargalo, ao passo que os muitos objetos pequenos são atrasados enquanto esperam o clipe de vídeo; ou seja, o clipe de vídeo na cabeça de fila bloqueia os objetos pequenos atrás de si. Os navegadores HTTP/1.1 normalmente contornam esse problema com a abertura de múltiplas conexões TCP paralelas, pelas quais os objetos na mesma página Web são enviados em paralelo ao navegador. Dessa forma, os objetos menores podem chegar e ser apresentados pelo navegador muito mais rapidamente, o que reduz o atraso percebido pelo usuário.

O controle de congestionamento do TCP, discutido em detalhes no Capítulo 3, também oferece aos navegadores um incentivo não intencional para usar múltiplas conexões TCP paralelas em vez de uma única conexão persistente. A muito grosso modo, o controle de congestionamento do TCP pretende dar a cada conexão TCP que compartilha um enlace congestionado uma porção igual da largura de banda disponível daquele enlace; assim, se há  $n$  conexões TCP operando através de um enlace congestionado, cada uma recebe aproximadamente  $1/n$  da largura de banda. Com a abertura de múltiplas conexões TCP paralelas para transportar uma única página Web, o navegador pode “trapacear” e se apropriar de uma porção maior da largura de banda do enlace. Muitos navegadores HTTP/1.1 abrem até seis conexões TCP paralelas para contornar o problema do bloqueio HOL e também para obter mais banda.

Um dos principais objetivos do HTTP/2 é se livrar (ou, pelo menos, reduzir o número) das conexões TCP paralelas para transportar uma única página Web. Além de reduzir o número de HOL que precisam ser abertos e mantidos nos servidores, isso também permite que o controle de congestionamento do TCP opere como pretendido. No entanto, com apenas uma conexão TCP para transportar uma página Web, o HTTP/2 exige a aplicação de mecanismos muito bem pensados para evitar o problema do bloqueio HOL.

## Enquadramento HTTP/2

A solução do HTTP/2 para o bloqueio HOL é dividir cada mensagem em quadros menores e intercalar as mensagens de solicitação e resposta na mesma conexão TCP. Para entender isso melhor, considere mais uma vez o exemplo de uma página Web composta por um único clipe de vídeo grande e, digamos, oito objetos menores. Assim, o servidor receberá nove solicitações simultâneas de qualquer navegador que desejar visualizar essa página Web. Para cada uma delas, o servidor precisará enviar nove mensagens de resposta HTTP concorrentes ao navegador. Suponha que todos os quadros tenham comprimento fixo, que o clipe de vídeo seja composto por 1.000 quadros, e que cada objeto menor seja composto por dois quadros. Com a intercalação dos quadros, após enviar um quadro do clipe de vídeo, o primeiro quadro de cada um dos objetos menores é enviado. Depois, após enviar o segundo quadro do clipe de vídeo, o último quadro de cada um dos objetos menores é enviado. Dessa forma, todos os objetos menores são transmitidos após o envio de 18 quadros no total. Sem o uso de intercalação, os objetos menores seriam enviados apenas após o envio de 1.016 quadros. Dessa forma, o mecanismo de enquadramento HTTP/2 pode reduzir significativamente o atraso percebido pelo usuário.

A capacidade de dividir uma mensagem HTTP em quadros independentes, intercalá-los e então remontá-los na outra ponta é absolutamente a melhoria mais importante do HTTP/2.

O enquadramento é realizado pela subcamada de enquadramento do protocolo HTTP/2. Quando um servidor quer enviar uma resposta HTTP, esta é processada pela subcamada de enquadramento, na qual é dividida em quadros. O campo de cabeçalho da resposta se torna um quadro, e o corpo da mensagem é dividido para mais quadros adicionais. A seguir, os quadros da mensagem são intercalados na subcamada de enquadramento no servidor com os quadros de outras respostas e enviados por uma única conexão TCP persistente. À medida que chegam no cliente, os quadros são remontados para formar as mensagens de resposta originais na subcamada de enquadramento e então processados pelo navegador da maneira tradicional. Da mesma forma, as solicitações HTTP do cliente são divididas em quadros e intercaladas.

Além de dividir cada mensagem HTTP em quadros independentes, a subcamada de enquadramento também realiza a codificação binária dos quadros. Os protocolos com dados binários são analisados de forma mais eficiente, o que leva a quadros ligeiramente menores, e são menos sujeitos a erros.

### Priorização da mensagem de resposta e *push* do servidor

A priorização de mensagens permite que os desenvolvedores customizem a prioridade relativa das solicitações para melhor otimizar o desempenho da aplicação. Como vimos, a subcamada de enquadramento organiza as mensagens em fluxos de dados paralelos destinados ao mesmo solicitante. Quando envia solicitações concorrentes a um servidor, o cliente pode priorizar as respostas que está solicitando com a designação de um peso de 1 a 256 para cada mensagem. Quanto mais alto o número, maior a prioridade. Usando esses pesos, o servidor pode enviar os primeiros quadros das respostas com a maior prioridade. Além disso, o cliente também informa a dependência de cada mensagem em relação às outras ao especificar a identidade da mensagem da qual ela depende.

Outro recurso do HTTP/2 é a capacidade do servidor de enviar múltiplas respostas para uma única solicitação do cliente. Em outras palavras, além da resposta à solicitação original, o servidor pode *transmitir (push)* objetos adicionais para o cliente, sem que este tenha que solicitar cada um deles. Isso é possível porque a página de base indica os objetos que serão necessários para apresentar completamente a página Web. Assim, em vez de esperar as solicitações HTTP referentes a tais objetos, o servidor pode analisar a página em HTML, identificar os objetos necessários e enviá-los para o cliente *antes de receber solicitações explícitas referentes a tais objetos*. O *push* do servidor elimina a latência adicional causada pela espera por essas solicitações.

### HTTP/3

O QUIC, discutido no Capítulo 3, é um novo protocolo de “transporte” implementado na camada de aplicação sobre o protocolo UDP básico. O QUIC possui diversos recursos desejáveis para o HTTP, tais como multiplexação de mensagens (intercalação), controle por fluxos e estabelecimento de conexões de baixa latência. O HTTP/3 é mais um protocolo HTTP projetado para operar no QUIC. Em 2020, o HTTP/3 é descrito em padrões propostos (os chamados *Internet drafts*) e ainda não foi completamente padronizado. Muitos dos recursos do HTTP/2 (como a intercalação de mensagens) são incorporados pelo QUIC, o que permite que o HTTP/3 tenha um projeto mais simples.

## 2.3 CORREIO ELETRÔNICO NA INTERNET

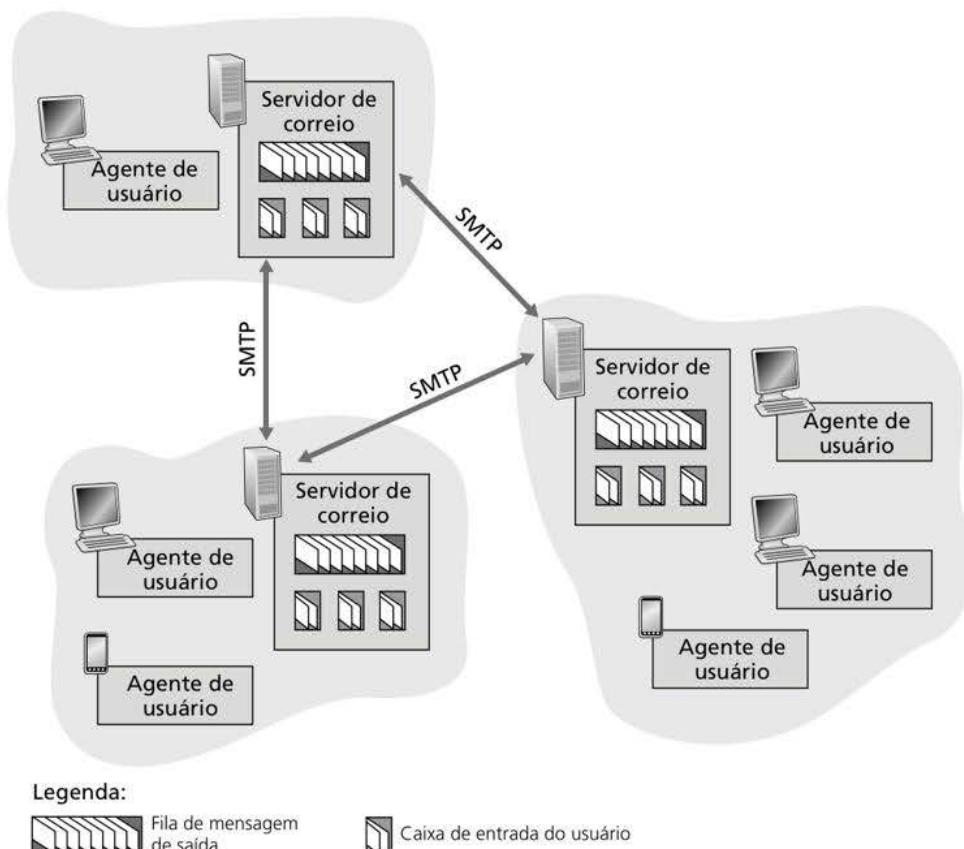
O correio eletrônico existe desde o início da Internet. Era uma das aplicações mais populares quando a Internet ainda estava na infância (Segaller, 1998), e ficou mais elaborada e poderosa ao longo dos anos. É uma das aplicações mais importantes e de maior uso na Internet.

Tal como o correio normal, o e-mail é um meio de comunicação assíncrono – as pessoas enviam e recebem mensagens quando for conveniente para elas, sem ter de estar coordenadas com o horário das outras. Ao contrário do correio normal, que anda a passos lentos, o eletrônico é rápido, fácil de distribuir e barato. O correio eletrônico moderno tem muitas características poderosas, incluindo mensagens com anexos, hiperlinks, textos formatados em HTML e fotos embutidas.

Nesta seção, examinaremos os protocolos de camada de aplicação que estão no cerne do correio eletrônico da Internet. Mas antes de entrarmos nessa discussão, vamos tomar uma visão geral do sistema de correio da Internet e de seus componentes principais.

A Figura 2.14 apresenta uma visão do sistema de correio da Internet. Vemos, por esse diagrama, que há três componentes principais: **agentes de usuário**, **servidores de correio** e o **SMTP** (do inglês *Simple Mail Transfer Protocol – Protocolo de Transferência de Correio Simples*). Descreveremos agora cada um deles no contexto de um remetente, Alice, enviando uma mensagem de e-mail para um destinatário, Bob. Agentes de usuário permitem que usuários leiam, respondam, encaminhem, salvem e componham mensagens. Microsoft Outlook, Apple Mail, o Gmail baseado na Web e o aplicativo do Gmail em um *smartphone* são alguns desses agentes, entre tantos outros. Quando Alice termina de compor sua mensagem, seu agente de usuário envia a mensagem para seu servidor de correio, onde a mensagem é colocada em sua fila de mensagens de saída. Quando Bob quer ler uma mensagem, seu agente de usuário apanha a mensagem de sua caixa de correio, em seu servidor de correio.

Servidores de correio formam o núcleo da infraestrutura do e-mail. Cada destinatário, como Bob, tem uma **caixa postal** localizada em um desses servidores. A de Bob administra e guarda as mensagens que foram enviadas a ele. Uma mensagem típica inicia sua jornada no agente de usuário do remetente, vai até seu servidor de correio e viaja até o do



**Figura 2.14** Uma visão de alto nível do sistema de e-mail da Internet.

destinatário, onde é depositada na caixa postal. Quando Bob quer acessar as mensagens de sua caixa postal, o servidor de correio que contém sua caixa postal o autentica (com nome de usuário e senha). O servidor de correio de Alice também deve cuidar das falhas no servidor de correio de Bob. Se o servidor de correio dela não puder entregar a correspondência ao dele, manterá a mensagem em uma **fila de mensagens** e tentará transferi-la mais tarde. Em geral, novas tentativas serão feitas a cada 30 minutos mais ou menos; se não obtiver sucesso após alguns dias, o servidor removerá a mensagem e notificará o remetente (Alice) por meio de uma mensagem de correio.

O SMTP é o principal protocolo de camada de aplicação do correio eletrônico da Internet. Usa o serviço confiável de transferência de dados do TCP para transferir mensagens do servidor de correio do remetente para o do destinatário. Como acontece com a maioria dos protocolos de camada de aplicação, o SMTP tem dois lados: um lado cliente, que funciona no servidor de correio do remetente, e um lado servidor, que funciona no servidor de correio do destinatário. Ambos funcionam em todos os servidores de correio. Quando um servidor de correio envia correspondência para outros, age como um cliente SMTP. Quando o servidor de correio recebe correspondência de outros, age como um servidor SMTP.

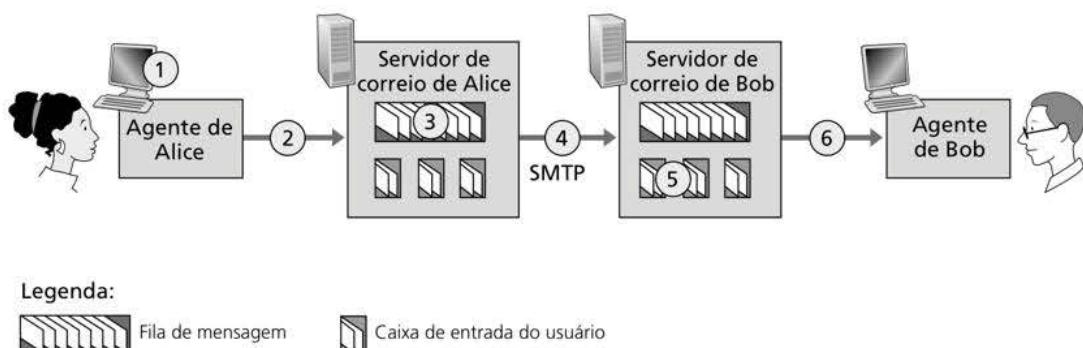
### 2.3.1 SMTP

O SMTP, definido no RFC 5321, está no cerne do correio eletrônico da Internet. Como já dissemos, esse protocolo transfere mensagens de servidores de correio remetentes para servidores de correio destinatários. O SMTP é muito mais antigo que o HTTP. (O RFC original do SMTP data de 1982, e ele já existia muito antes disso.) Embora tenha inúmeras qualidades maravilhosas, como evidencia sua ubiquidade na Internet, o SMTP é uma tecnologia antiga que possui certas características arcaicas. Por exemplo, restringe o corpo (e não apenas o cabeçalho) de todas as mensagens de correio ao simples formato ASCII de 7 bits. Essa restrição tinha sentido no começo da década de 1980, quando a capacidade de transmissão era escassa e ninguém enviava correio eletrônico com anexos volumosos nem arquivos grandes com imagens, áudio ou vídeo. Mas, hoje, na era da multimídia, a restrição do ASCII de 7 bits é um tanto incômoda – exige que os dados binários de multimídia sejam codificados em ASCII antes de serem enviados pelo SMTP, e que a mensagem correspondente em ASCII seja decodificada novamente para o sistema binário depois do transporte pelo SMTP. Lembre-se da Seção 2.2, na qual dissemos que o HTTP não exige que os dados de multimídia sejam codificados em ASCII antes da transferência.

Para ilustrar essa operação básica do SMTP, vamos percorrer um cenário comum. Suponha que Alice queira enviar a Bob uma simples mensagem ASCII.

1. Alice chama seu agente de usuário para e-mail, fornece o endereço de Bob (p. ex., `bob@someschool.edu`), compõe uma mensagem e instrui o agente de usuário a enviá-la.
2. O agente de usuário de Alice envia a mensagem para seu servidor de correio, onde ela é colocada em uma fila de mensagens.
3. O lado cliente do SMTP, que funciona no servidor de correio de Alice, vê a mensagem na fila e abre uma conexão TCP para um servidor SMTP, que funciona no servidor de correio de Bob.
4. Após alguns procedimentos iniciais de apresentação (*handshaking*), o cliente SMTP envia a mensagem de Alice pela conexão TCP.
5. No servidor de correio de Bob, o lado servidor do SMTP recebe a mensagem e a coloca na caixa postal dele.
6. Bob chama seu agente de usuário para ler a mensagem quando for mais conveniente para ele.

Esse cenário está resumido na Figura 2.15.



**Figura 2.15** Alice envia uma mensagem a Bob.

É importante observar que o SMTP em geral não usa servidores de correio intermediários para enviar correspondência, mesmo quando os dois servidores estão localizados em lados opostos do mundo. Se o servidor de Alice está em Hong Kong, e o de Bob, em St. Louis, a conexão TCP é uma conexão direta entre os servidores em Hong Kong e St. Louis. Em particular, se o servidor de correio de Bob não estiver em funcionamento, a mensagem permanece no de Alice esperando por uma nova tentativa – a mensagem não é colocada em nenhum servidor de correio intermediário.

Vamos agora examinar mais de perto como o SMTP transfere uma mensagem de um servidor de correio remetente para um servidor de correio destinatário. Veremos que o protocolo SMTP tem muitas semelhanças com protocolos usados na interação humana cara a cara. Primeiro, o cliente SMTP (que funciona no hospedeiro do servidor de correio remetente) faz o TCP estabelecer uma conexão na porta 25 com o servidor SMTP (que funciona no hospedeiro do servidor de correio destinatário). Se o servidor não estiver em funcionamento, o cliente tenta de novo mais tarde. Uma vez estabelecida a conexão, o servidor e o cliente trocam alguns procedimentos de apresentação de camada de aplicação – exatamente como os seres humanos, que costumam se apresentar antes de transferir informações, clientes e servidores SMTP também se apresentam antes de transferir informações. Durante essa fase, o cliente SMTP indica os endereços de e-mail do remetente (a pessoa que gerou a mensagem) e do destinatário. Assim que o cliente e o servidor SMTP terminam de se apresentar, o cliente envia a mensagem. O SMTP pode contar com o serviço confiável de transferência de dados do TCP para entregar a mensagem ao servidor sem erros. Então, o cliente repetirá esse processo, na mesma conexão TCP, se houver outras mensagens a enviar ao servidor; caso contrário, dará uma instrução ao TCP para encerrar a conexão.

Vamos analisar um exemplo de troca de mensagens entre um cliente (C) e um servidor SMTP (S). O nome do hospedeiro do cliente é `crepes.fr` e o nome do hospedeiro do servidor é `hamburger.edu`. As linhas de texto ASCII iniciadas com `C:` são exatamente as linhas que o cliente envia para dentro de seu *socket* TCP, e as iniciadas com `S:` são exatamente as linhas que o servidor envia para dentro de seu *socket* TCP. A transcrição a seguir começa assim que a conexão TCP é estabelecida:

```

S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr ... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?

```

```
C: How about pickles?  
C: .  
S: 250 Message accepted for delivery  
C: QUIT  
S: 221 hamburger.edu closing connection
```

Neste exemplo, o cliente enviou uma mensagem (“Do you like ketchup? How about pickles?”) do servidor de correio `crepes.fr` ao servidor de correio `hamburger.edu`. Como parte do diálogo, o cliente emitiu cinco comandos: HELO (uma abreviação de HELLO), MAIL FROM, RCPT TO, DATA e QUIT. Esses comandos são autoexplicativos. O cliente também enviou uma linha consistindo em um único ponto final, que indica o final da mensagem para o servidor. (No jargão ASCII, cada mensagem termina com CRLF.CRLF, em que CR significa *carriage return* e LF significa *line feed*.) O servidor emite respostas a cada comando, e cada resposta tem uma codificação de resposta e algumas explicações (opcionais) em inglês. Mencionamos aqui que o SMTP usa conexões persistentes: se o servidor de correio remetente tiver diversas mensagens para enviar ao mesmo servidor de correio destinatário, poderá enviar todas pela mesma conexão TCP. Para cada mensagem, o cliente inicia o processo com um novo MAIL FROM: `crepes.fr`, indica o final da mensagem com um ponto final isolado e emite QUIT somente após todas as mensagens terem sido enviadas.

Recomendamos vivamente que você utilize o Telnet para executar um diálogo direto com um servidor SMTP. Para fazer isso, digite

```
telnet serverName 25
```

em que `serverName` é o nome de um servidor de correio local. Ao fazer isso, você está apenas estabelecendo uma conexão TCP entre seu hospedeiro local e o servidor de correio. Após digitar essa linha, você deverá receber imediatamente do servidor a resposta 220. Digite, então, os comandos HELO, MAIL FROM, RCPT TO, DATA, CRLF.CRLF e QUIT nos momentos apropriados. Também recomendamos que você faça a Tarefa de Programação 3 no final deste capítulo. Nela, você construirá um agente de usuário simples que executa o lado cliente do SMTP. Esse agente permitirá que você envie uma mensagem de e-mail a um destinatário qualquer, por meio de um servidor de correio local.

### 2.3.2 Formatos de mensagem de correio

Quando Alice escreve uma carta a Bob e a envia pelo correio normal, ela pode incluir todos os tipos de informações periféricas no cabeçalho da carta, como seu próprio endereço, o endereço de Bob e a data. De modo semelhante, quando uma mensagem de e-mail é enviada, um cabeçalho contendo informações periféricas antecede o corpo da mensagem em si. Essas informações periféricas estão contidas em uma série de linhas de cabeçalho definidas no RFC 5322. As linhas de cabeçalho e o corpo da mensagem são separados por uma linha em branco (i.e., por CRLF). O RFC 5322 especifica o formato exato das linhas de cabeçalho das mensagens, bem como suas interpretações semânticas. Como acontece com o HTTP, cada linha de cabeçalho contém um texto legível, consistindo em uma palavra-chave seguida de dois-pontos e de um valor. Algumas palavras-chave são obrigatórias, e outras, opcionais. Cada cabeçalho deve ter uma linha de cabeçalho `From:` e uma `To:` e pode incluir também uma `Subject:`, bem como outras opcionais. É importante notar que essas linhas de cabeçalho são *diferentes* dos comandos SMTP que estudamos na Seção 2.3.1 (ainda que contenham algumas palavras em comum, como *from* e *to*). Os comandos daquela seção faziam parte do protocolo de apresentação SMTP; as linhas de cabeçalho examinadas nesta seção fazem parte da própria mensagem de correio.

Um cabeçalho de mensagem típico é semelhante a:

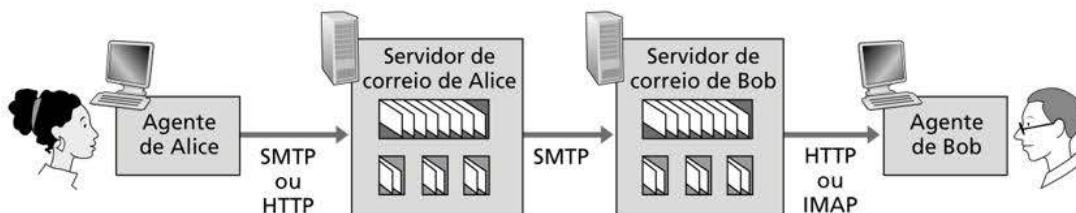
```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Searching for the meaning of life.
```

Após o cabeçalho da mensagem, vem uma linha em branco e, em seguida, o corpo da mensagem (em ASCII). Você pode usar o Telnet para enviar a um servidor de correio uma mensagem que contenha algumas linhas de cabeçalho, inclusive `Subject:`. Para tal, utilize o comando `telnet serverName 25`, como discutido na Seção 2.3.1.

### 2.3.3 Protocolos de acesso ao correio

Quando o SMTP entrega a mensagem do servidor de correio de Alice ao de Bob, ela é colocada na caixa postal de Bob. Dado que Bob (o destinatário) executa seu agente de usuário em seu hospedeiro local (p. ex., *smartphone* ou PC), é natural que ele considere a instalação de um servidor de correio também em seu hospedeiro local. Adotando essa abordagem, o servidor de correio de Alice dialogaria diretamente com o PC de Bob. Porém, há um problema com essa abordagem. Lembre-se de que um servidor de correio gerencia caixas postais e executa os lados cliente e servidor do SMTP. Se o servidor de correio de Bob residisse em seu PC local, este teria de ficar sempre em funcionamento e ligado na Internet para poder receber novas correspondências que poderiam chegar a qualquer hora, o que não é prático para muitos usuários. Em vez disso, um usuário típico executa um agente de usuário no hospedeiro local, mas acessa sua caixa postal armazenada em um servidor de correio compartilhado que está sempre em funcionamento. Esse servidor de correio é compartilhado com outros usuários.

Agora, vamos considerar o caminho que uma mensagem percorre quando é enviada de Alice para Bob. Acabamos de aprender que, em algum ponto do percurso, a mensagem de e-mail precisa ser depositada no servidor de correio de Bob. Essa tarefa poderia ser realizada simplesmente fazendo o agente de usuário de Alice enviar a mensagem diretamente ao servidor de correio de Bob. Contudo, em geral, o agente de usuário do remetente não dialoga diretamente com o servidor de correio do destinatário. Em vez disso, como mostra a Figura 2.16, o agente de usuário de Alice usa SMTP ou HTTP para enviar a mensagem de e-mail a seu servidor de correio. Em seguida, esse servidor usa SMTP (como um cliente SMTP) para retransmitir a mensagem de e-mail ao servidor de correio de Bob. Por que esse procedimento em duas etapas? Primordialmente porque, sem a retransmissão pelo servidor de correio de Alice, o agente de usuário dela não dispõe de nenhum recurso para tratar o caso de um servidor de correio de destinatário que não pode ser alcançado. Se Alice primeiro depositar o e-mail em seu próprio servidor de correio, este pode tentar, várias vezes, enviar a mensagem ao servidor de correio de Bob, digamos, a cada 30 minutos, até que esse servidor entre em operação. (E, se o servidor de correio de Alice não estiver funcionando, ela terá o recurso de se queixar ao administrador do seu sistema!)



**Figura 2.16** Protocolos de e-mail e suas entidades comunicantes.

Mas ainda falta uma peça do quebra-cabeça! De que forma um destinatário como Bob, que executa um agente de usuário em seu PC local, obtém suas mensagens que estão em um servidor de correio? Note que o agente de usuário de Bob não pode usar SMTP para obter as mensagens porque essa operação é de recuperação (*pull*), e o SMTP é um protocolo de envio (*push*).

Hoje, Bob tem dois modos comuns de recuperar seu e-mail de um servidor de correio. Se Bob usa e-mail baseado na Web ou um aplicativo para *smartphone* (como o Gmail), o agente do usuário utiliza HTTP para recuperar o e-mail de Bob. Nesse caso, o servidor de correio de Bob precisa ter uma interface HTTP além de uma interface SMTP (para se comunicar com o servidor de correio de Alice). O método alternativo, geralmente usado com clientes de correio como o Microsoft Outlook, é usar o **IMAP** (do inglês **Internet Mail Access Protocol – Protocolo de Acesso à Mensagem da Internet**), definido no RFC 3501. Ambas as abordagens, HTTP e IMAP, permitem que Bob gerencie pastas, que são mantidas no seu servidor de correio. Bob pode mover as mensagens para as pastas que cria, apagar mensagens, marcar mensagens como importantes e assim por diante.

## 2.4 DNS: O SERVIÇO DE DIRETÓRIO DA INTERNET

Nós, seres humanos, podemos ser identificados por diversas maneiras. Por exemplo, podemos ser identificados pelo nome que aparece em nossa certidão de nascimento, pelo número do RG ou da carteira de motorista. Embora cada um desses números possa ser usado para identificar pessoas, em um dado contexto um pode ser mais adequado que outro. Por exemplo, os computadores da Receita Federal preferem usar o número do CPF (de tamanho fixo) ao nome que consta em nossa certidão de nascimento. Por outro lado, pessoas comuns preferem nosso nome de batismo, mais fácil de lembrar, ao número do CPF. (Você lá consegue se imaginar dizendo: “Oi, meu nome é 132.679.875. Este é meu marido, 178.871.146”?)

Assim como seres humanos podem ser identificados de muitas maneiras, o mesmo acontece com hospedeiros da Internet. Um identificador é seu **nome de hospedeiro (hostname)**. Nomes de hospedeiro – como [www.facebook.com](http://www.facebook.com), [www.google.com](http://www.google.com) e [gaia.cs.umass.edu](http://gaia.cs.umass.edu) – são fáceis de lembrar e, portanto, apreciados pelos seres humanos. Todavia, eles fornecem pouca – se é que alguma – informação sobre a localização de um hospedeiro na Internet. (Um nome como [www.eurecom.fr](http://www.eurecom.fr), que termina com o código do país `.fr`, nos informa que o hospedeiro deve estar na França, mas não diz muito mais do que isso.) Além disso, como nomes de hospedeiros podem consistir em caracteres alfanuméricos de comprimento variável, seriam difíceis de ser processados por roteadores. Por essas razões, hospedeiros também são identificados pelo que denominamos **endereços IP**.

Discutiremos endereços IP mais detalhadamente no Capítulo 4, mas é importante falar um pouco sobre eles agora. Um endereço IP é constituído por 4 *bytes*, e sua estrutura hierárquica é rígida. Ele é algo como 121.7.106.83, no qual cada ponto separa um dos *bytes* expressos em notação decimal de 0 a 255. Um endereço IP é hierárquico porque, ao examiná-lo da esquerda para a direita, obtemos gradativamente mais informações específicas sobre onde o hospedeiro está localizado na Internet (i.e., em qual rede, entre as muitas que compõem a rede de redes). De maneira semelhante, quando examinamos um endereço postal de cima para baixo, obtemos informações cada vez mais específicas sobre a localização do destinatário.

### 2.4.1 Serviços fornecidos pelo DNS

Acabamos de ver que há duas maneiras de identificar um hospedeiro – por um nome de hospedeiro e por um endereço IP. As pessoas preferem o identificador nome de hospedeiro por ser mais fácil de lembrar, ao passo que roteadores preferem endereços IP de comprimento

fixo e estruturados hierarquicamente. Para conciliar essas preferências, é necessário um serviço de diretório que traduza nomes de hospedeiro para endereços IP. Esta é a tarefa principal do DNS (do inglês **domain name system – sistema de nomes de domínio**) da Internet. O DNS é (1) um banco de dados distribuído executado em uma hierarquia de **servidores de DNS**, e (2) um protocolo de camada de aplicação que permite que hospedeiros consultem o banco de dados distribuído. Os servidores DNS são muitas vezes máquinas UNIX que executam o *software* BIND (Berkeley Internet Name Domain) (BIND, 2020). O protocolo DNS utiliza UDP e usa a porta 53.

O DNS costuma ser empregado por outras entidades da camada de aplicação – inclusive HTTP e SMTP – para traduzir nomes de hospedeiros fornecidos por usuários para endereços IP. Como exemplo, considere o que acontece quando um navegador (i.e., um cliente HTTP), que executa na máquina de algum usuário, requisita o URL `www.someschool.edu/index.html`. Para que a máquina do usuário possa enviar uma mensagem de requisição HTTP ao servidor Web `www.someschool.edu`, ela precisa primeiro obter o seu endereço IP. Isso é feito da seguinte maneira:

1. A própria máquina do usuário executa o lado cliente da aplicação DNS.
2. O navegador extrai o nome de hospedeiro, `www.someschool.edu`, do URL e passa o nome para o lado cliente da aplicação DNS.
3. O cliente DNS envia uma consulta contendo o nome do hospedeiro para um servidor DNS.
4. O cliente DNS por fim recebe uma resposta, que inclui o endereço IP correspondente ao nome de hospedeiro.
5. Tão logo o navegador receba o endereço do DNS, pode abrir uma conexão TCP com o processo servidor HTTP localizado na porta 80 naquele endereço IP.

Vemos, por esse exemplo, que o DNS adiciona mais um atraso – às vezes substancial – às aplicações de Internet que o usam. Felizmente, como discutiremos mais adiante, o endereço IP procurado quase sempre está no cache de um servidor DNS “próximo”, o que ajuda a reduzir o tráfego DNS na rede, bem como o atraso médio do DNS.

O DNS provê alguns outros serviços importantes além da tradução de nomes de hospedeiro para endereços IP:

- **Apelidos (aliasing) de hospedeiro.** Um hospedeiro com nome complicado pode ter um ou mais apelidos. Um nome como `relay1.west-coast.enterprise.com` pode ter, por exemplo, dois apelidos, como `enterprise.com` e `www.enterprise.com`. Nesse caso, o nome de hospedeiro `relay1.west-coast.enterprise.com` é denominado **nome canônico**. Apelidos, quando existem, são em geral mais fáceis de lembrar do que o nome canônico. O DNS pode ser chamado por uma aplicação para obter o nome canônico correspondente a um apelido fornecido, bem como para obter o endereço IP do hospedeiro.
- **Apelidos de servidor de correio.** Por razões óbvias, é adequado que endereços de e-mail sejam fáceis de lembrar. Por exemplo, se Bob tem uma conta no Yahoo Mail, seu endereço de e-mail pode ser simplesmente `bob@yahoo.com`. Contudo, o nome de hospedeiro do servidor do Yahoo Mail é mais complicado e muito mais difícil de lembrar do que apenas `yahoo.com` (p. ex., o nome canônico pode ser algo parecido com `relay1.west-coast.yahoo.com`). O DNS pode ser chamado por uma aplicação de correio para obter o nome canônico a partir de um nome de domínio, bem como o endereço IP do hospedeiro. Na verdade, o registro MX (veja adiante) permite que o servidor de correio e o servidor Web de uma empresa tenham nomes (apelidos) idênticos; por exemplo, o servidor Web e o servidor de correio de uma empresa podem ambos ser denominados `enterprise.com`.
- **Distribuição de carga.** O DNS também é usado para realizar distribuição de carga entre servidores replicados, tais como os servidores Web replicados. *Sites* movimentados, como `cnn.com`, são replicados em vários servidores, cada qual rodando em um sistema

## PRINCÍPIOS NA PRÁTICA

### FUNÇÕES CRÍTICAS DE REDE VIA PARADIGMA CLIENTE-SERVIDOR

Assim como HTTP, FTP e SMTP, o DNS é um protocolo da camada de aplicação, já que (1) roda entre sistemas finais comunicantes usando o paradigma cliente-servidor e (2) depende de um protocolo de transporte fim a fim subjacente para transferir mensagens DNS entre sistemas finais comunicantes. Em outro sentido, contudo, o papel do DNS é bastante diferente das aplicações Web, da transferência de arquivo e do e-mail. Ao contrário delas, o DNS não é uma aplicação

com a qual o usuário interage diretamente. Em vez disso, fornece uma função interna da Internet – a saber, a tradução de nomes de hospedeiro para seus endereços IP subjacentes, para aplicações de usuário e outros softwares da Internet. Notamos, na Seção 1.2, que grande parte da complexidade da arquitetura da Internet está localizada na “periferia” da rede. O DNS, que executa o processo crucial de tradução de nome para endereço usando clientes e servidores localizados nas bordas da rede, é mais um exemplo dessa filosofia de projeto.

final e com um endereço IP diferentes. Assim, no caso de servidores Web replicados, um *conjunto* de endereços IP fica associado a um único apelido de hospedeiro e contido no banco de dados do DNS. Quando clientes consultam um nome mapeado para um conjunto de endereços, o DNS responde com o conjunto inteiro de endereços IP, mas faz um rodízio da ordem deles dentro de cada resposta. Como um cliente em geral envia sua mensagem de requisição HTTP ao endereço IP que ocupa o primeiro lugar no conjunto, o rodízio de DNS distribui o tráfego entre os servidores replicados. O rodízio de DNS também é usado para e-mail, de modo que vários servidores de correio podem ter o mesmo apelido. Além disso, empresas distribuidoras de conteúdo, como a Akamai, usam o DNS de maneira mais sofisticada (Dilley, 2002) para prover distribuição de conteúdo na Web (veja Seção 2.6.3).

O DNS está especificado no RFC 1034 e no RFC 1035 e atualizado em diversos RFCs adicionais. É um sistema complexo e, neste livro, apenas mencionamos os aspectos fundamentais de sua operação. O leitor interessado pode consultar os RFCs citados, o livro escrito por Albitz e Liu (Albitz, 1993) e os artigos de Mockapetris (1988), que apresenta uma retrospectiva e uma ótima descrição do que e do porquê do DNS, e Mockapetris (2005).

### 2.4.2 Visão geral do modo de funcionamento do DNS

Apresentaremos, agora, uma visão panorâmica do modo de funcionamento do DNS. Nossa discussão focalizará o serviço de tradução de nome de hospedeiro para endereço IP.

Suponha que certa aplicação (como um navegador Web ou um cliente de correio), que executa na máquina de um usuário, precise traduzir um nome de hospedeiro para um endereço IP. A aplicação chamará o lado cliente do DNS, especificando o nome de hospedeiro que precisa ser traduzido. (Em muitas máquinas UNIX, `gethostbyname()` é a chamada de função que uma aplicação invoca para realizar a tradução.) A partir daí, o DNS do hospedeiro do usuário assume o controle, enviando uma mensagem de consulta para a rede. Todas as mensagens de consulta e de resposta do DNS são enviadas dentro de datagramas UDP à porta 53. Após um atraso na faixa de milissegundos a segundos, o DNS no hospedeiro do usuário recebe uma mensagem de resposta DNS fornecendo o mapeamento desejado, que é, então, passado para a aplicação que está interessada. Portanto, do ponto de vista dessa aplicação, que está na máquina do cliente, o DNS é uma caixa-preta que provê um serviço de tradução simples e direto. Mas, na realidade, a caixa-preta que executa o serviço é complexa, consistindo em um grande número de servidores DNS distribuídos ao redor do mundo, bem como em um protocolo de camada de aplicação que especifica como se comunicam os servidores DNS e os hospedeiros que fazem a consulta.

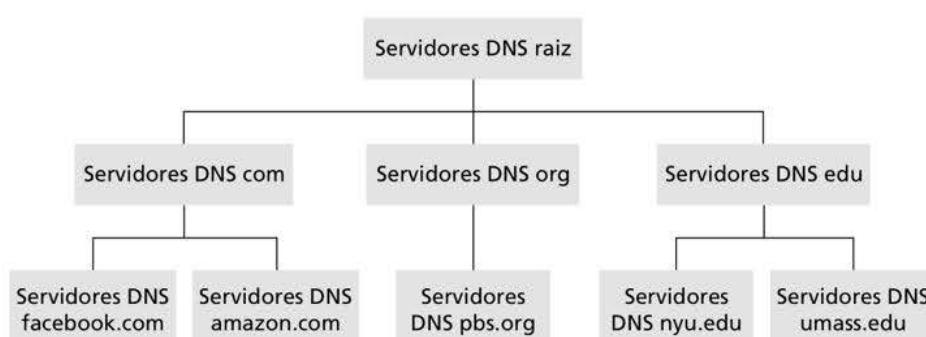
Um arranjo simples para DNS seria ter um servidor DNS contendo todos os mapeamentos. Nesse projeto centralizado, os clientes apenas dirigiriam todas as consultas a esse único servidor DNS, que responderia diretamente aos clientes que estão fazendo a consulta. Embora a simplicidade desse arranjo seja atraente, ele não é adequado para a Internet de hoje com seu vasto (e crescente) número de hospedeiros. Entre os problemas de um arranjo centralizado, estão:

- **Um único ponto de falha.** Se o servidor DNS quebrar, a Internet inteira quebrará!
- **Volume de tráfego.** Um único servidor DNS teria de manipular todas as consultas DNS (para todas as requisições HTTP e mensagens de e-mail geradas por centenas de milhões de hospedeiros).
- **Banco de dados centralizado distante.** Um único servidor DNS nunca poderia estar “próximo” de todos os clientes que fazem consultas. Se colocarmos o único servidor DNS na cidade de Nova York, todas as buscas provenientes da Austrália terão de viajar até o outro lado do globo, talvez por linhas lentas e congestionadas, o que pode resultar em atrasos significativos.
- **Manutenção.** O único servidor DNS teria de manter registros de todos os hospedeiros da Internet. Esse banco de dados não só seria enorme, mas também precisaria ser atualizado frequentemente para atender a todos os novos hospedeiros.

Resumindo, um banco de dados centralizado em um único servidor DNS simplesmente *não é escalável*. Por conseguinte, o DNS é distribuído por conceito de projeto. Na verdade, ele é um ótimo exemplo de como um banco de dados distribuído pode ser executado na Internet.

### Um banco de dados distribuído e hierárquico

Para tratar da questão da escala, o DNS usa um grande número de servidores, organizados de maneira hierárquica e distribuídos por todo o mundo. Nenhum servidor DNS isolado tem todos os mapeamentos para todos os hospedeiros da Internet. Em vez disso, os mapeamentos são distribuídos pelos servidores DNS. Como uma primeira aproximação, há três classes de servidores DNS: raiz, de domínio de alto nível (TLD, do inglês *top-level domain*) e servidores DNS autoritativos – organizados em uma hierarquia, como mostra a Figura 2.17. Para entender como essas três classes interagem, suponha que um cliente DNS queira determinar o endereço IP para o nome de hospedeiro `www.amazon.com`. Como uma primeira aproximação, ocorrerão os seguintes eventos. Primeiro, o cliente contatará um dos servidores raiz, que retornará endereços IP dos servidores TLD para o domínio de alto nível `.com`. Então, o cliente contatará um desses servidores TLD, que retornará o endereço IP de um servidor autoritativo para `amazon.com`. Por fim, o cliente contatará um dos servidores autoritativos para `amazon.com`, que retornará o endereço IP para o nome de hospedeiro `www.amazon.com`.



**Figura 2.17** Parte da hierarquia de servidores DNS.

Mais adiante, analisaremos em detalhes esse processo de consulta DNS. Mas, primeiro, vamos examinar mais de perto as três classes de servidores DNS:

- **Servidores DNS raiz.** Existem mais de 1.000 instâncias de servidores-raiz espalhadas pelo mundo, como mostra a Figura 2.18. Esses servidores raiz são cópias de 13 servidores-raiz diferentes, administrados por 12 organizações e coordenados através da Internet Assigned Numbers Authority (IANA, 2020). A lista completa de servidores-raiz, assim como das organizações que os administram e seus endereços IP, se encontra em (Root Servers, 2020). Os servidores-raiz fornecem os endereços IP dos servidores TLD.
- **Servidores DNS de domínio de alto nível (TLD).** Para cada um dos domínios de alto nível (p. ex., com, org, net, edu e gov) e todos os domínios de alto nível de países (p. ex., uk, fr, ca, br e jp) existe um servidor (ou *cluster* de servidores) TLD. A empresa Verisign Global Registry Services mantém os servidores TLD para o domínio de alto nível com, e a Educause mantém os servidores TLD para o domínio de alto nível edu. A infraestrutura de rede por trás de um TLD pode ser grande e complexa; para um panorama sobre o trabalho da Verisign, consulte (Osterweil, 2012). Para uma lista de todos os domínios de alto nível, veja (TLD list, 2020). Os servidores TLD fornecem os endereços IP dos servidores DNS autoritativos.
- **Servidores DNS autoritativos.** Toda organização que tiver hospedeiros que possam ser acessados publicamente na Internet (como servidores Web e servidores de correio) deve fornecer registros DNS também acessíveis publicamente que mapeiem os nomes desses hospedeiros para endereços IP. Um servidor DNS autoritativo de uma organização abriga esses registros. Uma organização pode preferir executar seu próprio servidor DNS autoritativo para abrigar esses registros ou pagar para armazená-los em um servidor DNS autoritativo de algum provedor de serviço. A maioria das universidades e empresas de grande porte executa e mantém seus próprios servidores DNS primário e secundário (*backup*) autoritativos.

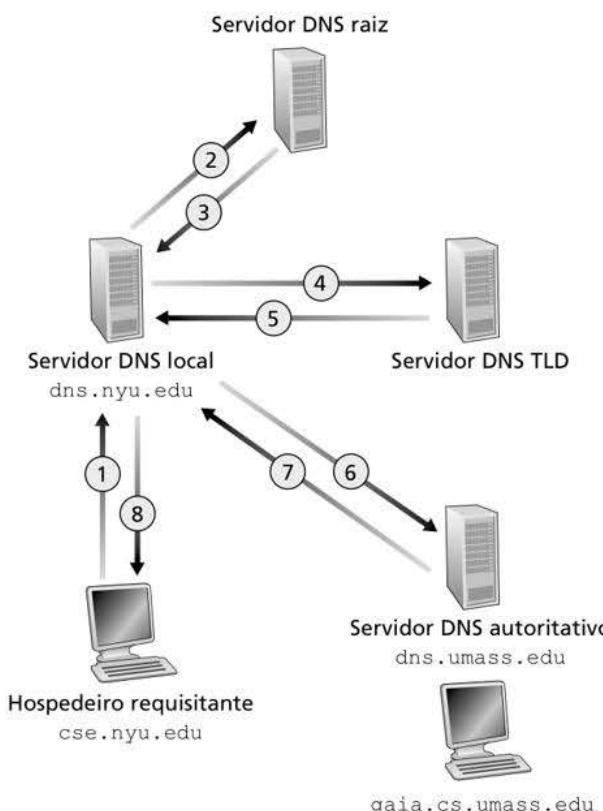
Os servidores DNS raiz, TLD e autoritativo pertencem à hierarquia de servidores DNS, como mostra a Figura 2.17. Há mais um tipo importante de DNS, denominado **servidor DNS local**, que não pertence, estritamente, à hierarquia de servidores, mas, mesmo assim, é central para a arquitetura DNS. Cada ISP – como um ISP residencial ou institucional – tem um servidor DNS local (também denominado servidor DNS *default*). Quando um hospedeiro se conecta com um ISP, este fornece os endereços IP de um ou mais de seus servidores



**Figura 2.18** Servidores DNS raiz em 2020.

DNS locais (em geral por DHCP, que será discutido no Capítulo 4). Determinar o endereço IP do seu servidor DNS local é fácil: basta acessar as janelas de estado da rede no Windows ou UNIX. O servidor DNS local de um hospedeiro costuma estar “próximo” dele. No caso de um ISP institucional, pode estar na mesma LAN do hospedeiro; já no caso de um ISP residencial, em geral o servidor DNS está separado do hospedeiro por não mais do que alguns roteadores. Quando um hospedeiro faz uma consulta ao DNS, ela é enviada ao servidor DNS local, que age como *proxy* e a retransmite para a hierarquia do servidor DNS, como discutiremos mais detalhadamente a seguir.

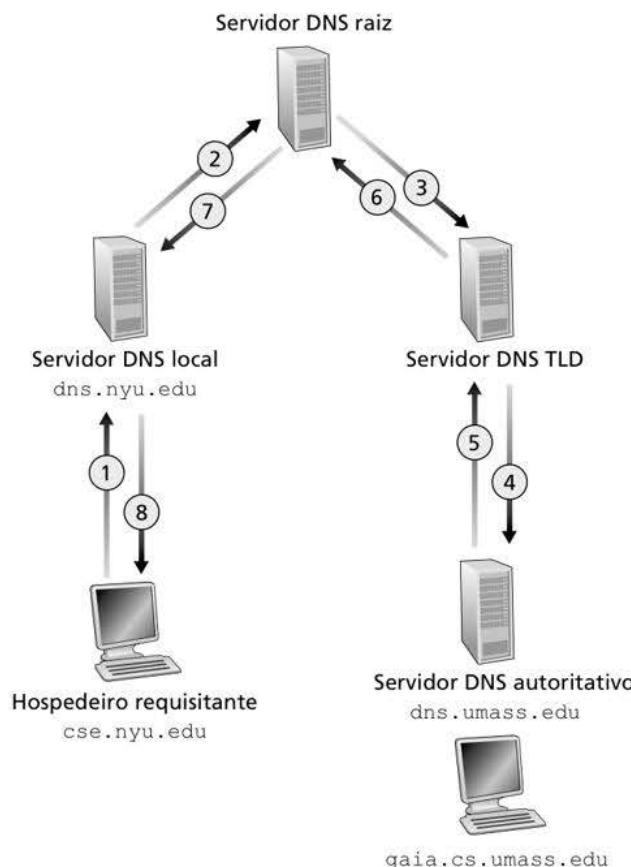
Vamos examinar um exemplo simples. Suponha que o hospedeiro `cse.nyu.edu` deseje o endereço IP de `gaia.cs.umass.edu`. Imagine também que o servidor DNS local da NYU para `cse.nyu.edu` seja denominado `dns.nyu.edu`, e que um servidor DNS autoritativo para `gaia.cs.umass.edu` seja denominado `dns.umass.edu`. Como mostra a Figura 2.19, o hospedeiro `cse.nyu.edu` primeiro envia uma mensagem de consulta DNS a seu servidor DNS local `dns.nyu.edu`. A mensagem de consulta contém o nome de hospedeiro a ser traduzido, isto é, `gaia.cs.umass.edu`. O servidor DNS local transmite a mensagem de consulta a um servidor DNS raiz, que identifica o sufixo `edu` e retorna ao servidor DNS local uma lista de endereços IP contendo servidores TLD responsáveis por `edu`. Então, o servidor DNS local retransmite a mensagem de consulta a um desses servidores TLD que, por sua vez, identifica o sufixo `umass.edu` e responde com o endereço IP do servidor DNS autorizado para a University of Massachusetts, a saber, `dns.umass.edu`. Por fim, o servidor DNS local reenvia a mensagem de consulta diretamente a `dns.umass.edu`, que responde com o endereço IP de `gaia.cs.umass.edu`. Note que, nesse exemplo, para poder obter o mapeamento para um único nome de hospedeiro, foram enviadas oito mensagens DNS: quatro mensagens de consulta e quatro mensagens de resposta! Em breve veremos como o *cache* de DNS reduz esse tráfego de consultas.



**Figura 2.19** Interação dos diversos servidores DNS.

Nosso exemplo anterior considerou que o servidor TLD conhece o servidor DNS autoritativo para o nome de hospedeiro, o que em geral não acontece. Ele pode conhecer apenas um servidor DNS intermediário que, por sua vez, conhece o servidor DNS autoritativo para o nome de hospedeiro. Por exemplo, suponha de novo que a Universidade de Massachusetts tenha um servidor DNS para a universidade denominado dns.umass.edu. Imagine também que cada departamento da universidade tenha seu próprio servidor DNS, e que cada servidor DNS departamental seja um servidor DNS autoritativo para todos os hospedeiros do departamento. Nesse caso, quando o servidor DNS intermediário dns.umass.edu receber uma consulta para um hospedeiro cujo nome termina com cs.umass.edu, ele retornará a dns.nyu.edu o endereço IP de dns.cs.umass.edu, que tem autoridade para todos os nomes de hospedeiro que terminam com cs.umass.edu. Então, o servidor DNS local dns.nyu.edu enviará a consulta ao servidor DNS autoritativo, que retornará o mapeamento desejado para o servidor DNS local e que, por sua vez, o repassará ao hospedeiro requisitante. Nesse caso, serão enviadas dez mensagens DNS no total!

O exemplo mostrado na Figura 2.19 usa **consultas recursivas** e **consultas iterativas**. A consulta enviada de cse.nyu.edu para dns.nyu.edu é recursiva, visto que pede a dns.nyu.edu que obtenha o mapeamento em seu nome. Mas as três consultas subsequentes são iterativas, visto que todas as respostas são retornadas diretamente a dns.nyu.edu. Em teoria, qualquer consulta DNS pode ser iterativa ou recursiva. Por exemplo, a Figura 2.20 mostra uma cadeia de consultas DNS na qual todas são recursivas. Na prática, as consultas em geral seguem o padrão mostrado na Figura 2.19: a consulta do hospedeiro requisitante ao servidor DNS local é recursiva, e todas as outras são iterativas.



**Figura 2.20** Consultas recursivas em DNS.

## Cache DNS

Até aqui, nossa discussão ignorou o **cache DNS**, uma característica muito importante do sistema DNS. Na realidade, o DNS explora extensivamente o *cache* para melhorar o desempenho quanto ao atraso e reduzir o número de mensagens DNS que transmite pela Internet. A ideia por trás do *cache* DNS é muito simples. Em uma cadeia de consultas, quando um servidor DNS recebe uma resposta DNS (contendo, p. ex., o mapeamento de um nome de hospedeiro para um endereço IP), pode fazer *cache* das informações da resposta em sua memória local. Por exemplo, na Figura 2.19, toda vez que o servidor DNS local dns.nyu.edu recebe uma resposta de algum servidor DNS, pode fazer *cache* de qualquer informação contida na resposta. Se um par nome de hospedeiro/endereço IP estiver no *cache* de um servidor DNS e outra consulta chegar ao mesmo servidor para o mesmo nome de hospedeiro, o servidor DNS poderá fornecer o endereço IP desejado, mesmo que não tenha autoridade para esse nome. Já que hospedeiros e mapeamentos entre hospedeiros e endereços IP não são, de modo algum, permanentes, após um período (quase sempre dois dias), os servidores DNS descartam as informações armazenadas em seus *caches*.

Como exemplo, imagine que um hospedeiro apricot.nyu.edu consulte dns.nyu.edu para o endereço IP da máquina cnn.com. Além disso, suponha que algumas horas mais tarde outra máquina da NYU, digamos, kiwi.nyu.edu, também consulte dns.nyu.edu para o mesmo nome de hospedeiro. Em razão do *cache*, o servidor local poderá imediatamente retornar o endereço IP de cnn.com a esse segundo hospedeiro requisitante, sem ter de consultar quaisquer outros servidores DNS. Um servidor DNS local também pode fazer *cache* de endereços IP de servidores TLD, permitindo, assim, que servidores DNS locais evitem os servidores DNS raiz em uma cadeia de consultas. Na verdade, devido aos *caches*, os servidores-raiz são contornados para praticamente todas as consultas DNS.

### 2.4.3 Registros e mensagens DNS

Os servidores DNS que juntos executam o banco de dados distribuído do DNS armazenam **registros de recursos (RR)** que fornecem mapeamentos de nomes de hospedeiros para endereços IP. Cada mensagem de resposta DNS carrega um ou mais registros de recursos. Nesta seção e na subsequente, apresentaremos uma breve visão geral dos registros de recursos e mensagens DNS. Para mais detalhes, consulte Albitz (1993) ou os RFCs 1034 e 1035.

Um registro de recurso é uma tupla de quatro elementos que contém os seguintes campos:

(Name, Value, Type, TTL)

TTL é o tempo de vida útil do registro de recurso; determina quando um recurso deve ser removido de um *cache*. Nos exemplos de registros dados a seguir, ignoramos o campo TTL. Os significados de Name e Value dependem de Type:

- Se Type=A, então Name é um nome de hospedeiro e Value é o endereço IP para o nome de hospedeiro. Assim, um registro Type A fornece o mapeamento-padrão entre nomes de hospedeiros e endereços IP. Como exemplo, (relay1.bar.foo.com, 145.37.93.126, A) é um registro com Type igual a A.
- Se Type=NS, então Name é um domínio (como foo.com) e Value é o nome de um servidor DNS autoritativo que sabe como obter os endereços IP para hospedeiros do domínio. Esse registro é usado para encaminhar consultas DNS ao longo da cadeia de consultas. Como exemplo, (foo.com, dns.foo.com, NS) é um registro com Type igual a NS.
- Se Type=CNAME, então Value é um nome canônico de hospedeiro para o apelido de hospedeiro contido em Name. Esse registro pode fornecer aos hospedeiros consultantes

o nome canônico correspondente a um apelido de hospedeiro. Como exemplo, (`foo.com, relay1.bar.foo.com, CNAME`) é um registro CNAME.

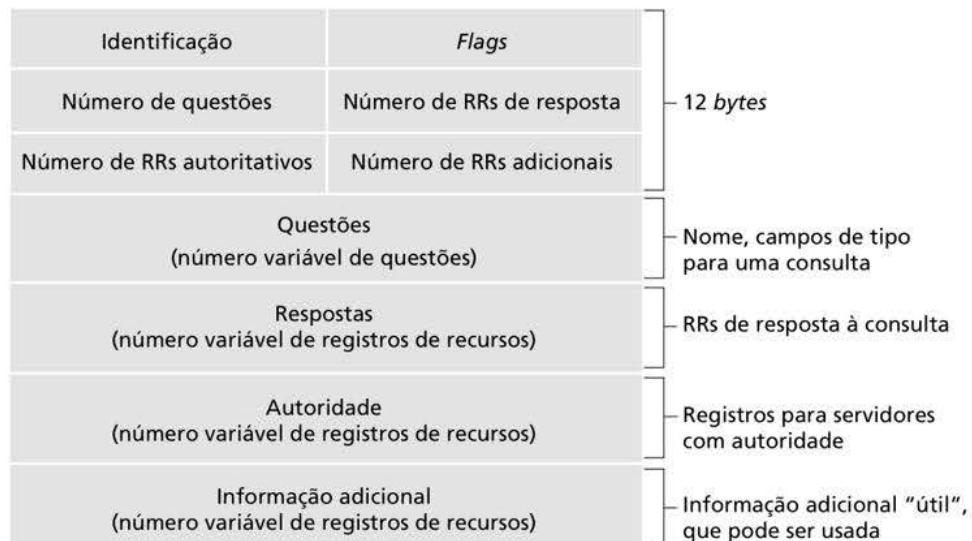
- Se `Type=MX`, então `Value` é o nome canônico de um servidor de correio cujo apelido de hospedeiro está contido em `Name`. Como exemplo, (`foo.com, mail.bar.foo.com, MX`) é um registro MX. Registros MX permitem que os nomes de hospedeiros de servidores de correio tenham apelidos simples. Note que, usando o registro MX, uma empresa pode ter o mesmo apelido para seu servidor de arquivo e para um de seus outros servidores (tal como seu servidor Web). Para obter o nome canônico do servidor de correio, um cliente DNS consultaria um registro MX; para obter o nome canônico do outro servidor, o cliente DNS consultaria o registro CNAME.

Se um servidor DNS tiver autoridade para determinado nome de hospedeiro, então conterá um registro Type A para o nome de hospedeiro. (Mesmo que não tenha autoridade, o servidor DNS pode conter um registro Type A em seu *cache*.) Se um servidor não tiver autoridade para um nome de hospedeiro, conterá um registro Type NS para o domínio que inclui o nome e um registro Type A que fornece o endereço IP do servidor DNS no campo `Value` do registro NS. Como exemplo, suponha que um servidor TLD edu não tenha autoridade para o hospedeiro `gaia.cs.umass.edu`. Nesse caso, esse servidor conterá um registro para um domínio que inclui o hospedeiro `gaia.cs.umass.edu`, por exemplo (`umass.edu, dns.umass.edu, NS`). O servidor TLD edu conterá também um registro Type A, que mapeia o servidor DNS `dns.umass.edu` para um endereço IP, por exemplo (`dns.umass.edu, 128.119.40.111, A`).

## Mensagens DNS

Abordamos anteriormente nesta seção mensagens de consulta e de resposta DNS, que são as duas únicas espécies de mensagem DNS. Além disso, tanto as mensagens de consulta como as de resposta têm o mesmo formato, como ilustra a Figura 2.21. A semântica dos vários campos de uma mensagem DNS é a seguinte:

- Os primeiros 12 *bytes* formam a *seção de cabeçalho*, que tem vários campos. O primeiro campo é um número de 16 *bits* que identifica a consulta. Esse identificador é copiado para a mensagem de resposta a uma consulta, permitindo que o cliente combine respostas recebidas com consultas enviadas. Há várias *flags* no campo de *flag*. Uma *flag* de consulta/resposta de 1 *bit* indica se a mensagem é uma consulta (0) ou uma resposta (1).



**Figura 2.21** Formato da mensagem DNS.

Uma *flag* de autoridade de 1 bit é marcada em uma mensagem de resposta quando o servidor DNS é um servidor autoritativo para um nome consultado. Uma *flag* de recursão desejada de 1 bit é estabelecida quando um cliente (hospedeiro ou servidor DNS) quer que um servidor DNS proceda recursivamente sempre que não tem o registro. Um campo de recursão disponível de 1 bit é marcado em uma resposta se o servidor DNS suporta recursão. No cabeçalho, há também quatro campos de “tamanho”. Eles indicam o número de ocorrências dos quatro tipos de seção de dados que se seguem ao cabeçalho.

- A *seção de questão* contém informações sobre a consulta que está sendo feita. Essa seção inclui (1) um campo de nome que contém o nome que está sendo consultado e (2) um campo de tipo que indica o tipo da pergunta que está sendo feita sobre o nome – por exemplo, um endereço de hospedeiro associado a um nome (Type A) ou o servidor de correio para um nome (Type MX).
- Em uma resposta de um servidor DNS, a *seção de resposta* contém os registros de recursos para o nome que foi consultado originalmente. Lembre-se de que em cada registro de recurso há o Type (p. ex., A, NS, CNAME e MX), o Value e o TTL. Uma resposta pode retornar vários RRs, já que um nome de hospedeiro pode ter diversos endereços IP (p. ex., para servidores Web replicados, como já discutimos anteriormente nesta seção).
- A *seção de autoridade* contém registros de outros servidores autoritativos.
- A *seção adicional* contém outros registros úteis. Por exemplo, o campo resposta em uma resposta a uma consulta MX conterá um registro de recurso que informa o nome canônico de um servidor de correio. A seção adicional conterá um registro Type A que fornece o endereço IP para o nome canônico do servidor de correio.

Você gostaria de enviar uma mensagem de consulta DNS direto de sua máquina a algum servidor DNS? Isso pode ser feito facilmente com o **programa nslookup**, que está disponível na maioria das plataformas Windows e UNIX. Por exemplo, se um hospedeiro executar Windows, abra o *Prompt* de comando e chame o programa nslookup apenas digitando “nslookup”. Depois de chamar o programa, você pode enviar uma consulta DNS a qualquer servidor DNS (raiz, TLD ou autoritativo). Após receber a mensagem de resposta do servidor DNS, o nslookup apresentará os registros incluídos na resposta (em formato que pode ser lido normalmente). Como alternativa para executar nslookup na sua própria máquina, você pode visitar um dos muitos *sites* que permitem o emprego remoto do programa. (Basta digitar “nslookup” em um buscador e você será levado a um desses *sites*.) O Wireshark lab DNS, ao final deste capítulo, lhe permitirá explorar o DNS com muito mais detalhes.

## Inserindo registros no banco de dados do DNS

A discussão anterior focalizou como são extraídos registros do banco de dados DNS. É possível que você esteja se perguntando como os registros entraram no banco de dados em primeiro lugar. Vamos examinar como isso é feito no contexto de um exemplo específico. Imagine que você acabou de criar uma nova empresa muito interessante denominada Network Utopia. A primeira coisa que você certamente deverá fazer é registrar o nome de domínio networkutopia.com em uma entidade registradora. Uma **entidade registradora** é uma organização comercial que verifica se o nome de domínio é exclusivo, registra-o no banco de dados do DNS (como discutiremos mais adiante) e cobra uma pequena taxa por seus serviços. Antes de 1999, uma única entidade registradora, a Network Solutions, detinha o monopólio do registro de nomes para os domínios .com, .net e .org. Mas agora existem muitas entidades registradoras credenciadas pela Internet Corporation for Assigned Names and Numbers (ICANN) competindo por clientes. Uma lista completa das entidades credenciadas está disponível em <http://www.internic.net>.

Ao registrar o nome de domínio networkutopia.com, você também precisará informar os nomes e endereços IP dos seus servidores DNS com autoridade, primários e secundários. Suponha que os nomes e endereços IP sejam dns1.networkutopia.com,

`dns2.networkutopia.com`, `212.2.212.1` e `212.212.212.2`. A entidade registradora ficará encarregada de providenciar a inserção dos registros Type NS e Type A nos servidores TLD do domínio `.com` para cada um desses dois servidores de nomes autorizados. Em especial, para o servidor primário com autoridade para `networkutopia.com`, a autoridade registradora inseriria no sistema DNS os dois registros de recursos a seguir:

```
(networkutopia.com, dns1.networkutopia.com, NS)
(dns1.networkutopia.com, 212.212.212.1, A)
```

Não se esqueça de providenciar também a inserção em seus servidores de nomes com autoridade do registro de recurso Type A para seu servidor Web `www.networkutopia.com` e o registro de recurso Type MX para seu servidor de correio `mail.networkutopia.com`. (Até há pouco tempo, o conteúdo de cada servidor DNS era configurado estaticamente, p. ex., a partir de um arquivo de configuração criado por um administrador de sistema. Mais recentemente, foi acrescentada ao protocolo DNS uma opção UPDATE, que permite que dados sejam dinamicamente acrescentados no banco de dados ou apagados deles por

## SEGURANÇA EM FOCO

### VULNERABILIDADES DO DNS

Vimos que o DNS é um componente fundamental da infraestrutura da Internet, com muitos serviços importantes – incluindo a Web e o e-mail – simplesmente incapazes de funcionar sem ele. Dessa maneira, perguntamos: como o DNS pode ser atacado? O DNS é um alvo fácil, esperando para ser derrubado, e levaria com ele a maioria das aplicações da Internet?

O primeiro tipo de ataque que vem à mente é a inundação na largura de banda DDoS (veja a Seção 1.6) contra servidores DNS. Por exemplo, um atacante pode tentar enviar para cada servidor DNS raiz uma inundação de pacotes, fazendo a maioria das consultas DNS legítimas nunca ser respondida. Um ataque DDoS em larga escala contra servidores DNS raiz aconteceu em 21 de outubro de 2002. Os atacantes utilizaram um *botnet* para enviar inúmeras mensagens ICMP ping para os 13 servidores DNS raiz. (As mensagens ICMP são discutidas na Seção 5.6. Por enquanto, basta saber que os pacotes ICMP são tipos especiais de datagramas IP.) Felizmente, esse ataque em larga escala causou um dano mínimo, tendo pouco ou nenhum impacto na experiência dos usuários com a Internet. Os atacantes obtiveram êxito ao direcionar centenas de pacotes aos servidores raiz. Mas muitos dos servidores DNS raiz foram protegidos por filtros de pacotes, configurados para sempre bloquear todas as mensagens ICMP ping encaminhadas aos servidores raiz. Assim, esses servidores protegidos foram poupadinhos e funcionaram normalmente. Além disso, a maioria dos servidores DNS locais armazena os endereços IP dos servidores de domínio de

nível superior, permitindo que o processo de consulta ultrapasse com frequência os servidores DNS raiz.

Um ataque DDoS potencialmente mais eficaz contra o DNS é enviar uma inundação de consultas DNS aos servidores de domínio de alto nível, por exemplo, para todos os que lidam com o domínio `.com`. É mais difícil filtrar as consultas DNS direcionadas aos servidores DNS; e os servidores de domínio de alto nível não são ultrapassados com tanta facilidade quanto os raiz. Um ataque desse tipo ocorreu contra a Dyn, uma provedora de serviços de domínio de alto nível, em 21 de outubro de 2016. Esse ataque DDoS foi realizado por meio de uma grande quantidade de solicitações de consulta DNS de uma *botnet* composta por cerca de 100 mil dispositivos da “Internet das Coisas”, como impressoras, câmeras de IP, portões residenciais e babás eletrônicas, infectados com o malware Mirai. Amazon, Twitter, Netflix, Github e Spotify foram prejudicados durante quase um dia inteiro.

O DNS poderia ser atacado de outras maneiras. Em um ataque *man-in-the-middle*, o atacante intercepta consultas do hospedeiro e retorna respostas falsas. No ataque de envenenamento, o atacante envia respostas falsas a um servidor DNS, fazendo-o armazenar os registros falsos em sua cache. Ambos os ataques podem ser utilizados, por exemplo, para redirecionar um usuário da Web inocente ao site do atacante. As DNS Security Extensions (DNSSEC) (Gieben 2004; RFC 4033) foram projetadas e implementadas para proteção contra essas oportunidades de ataque. O DNSSEC, uma versão segura do DNS, resolve muitos desses possíveis ataques e está se popularizando na Internet.

meio de mensagens DNS. O [RFC 2136] e o [RFC 3007] especificam atualizações dinâmicas do DNS.)

Quando todas essas etapas estiverem concluídas, o público em geral poderá visitar o site de “networkutopia” e enviar e-mails aos empregados da empresa. Vamos concluir nossa discussão do DNS verificando que essa afirmação é verdadeira, o que também ajudará a solidificar aquilo que aprendemos sobre o DNS. Suponha que Alice, que está na Austrália, queira consultar a página [www.networkutopia.com](http://www.networkutopia.com). Como discutimos, seu hospedeiro primeiro enviará uma consulta DNS a seu servidor de nomes local, que então contatará um servidor TLD do domínio com. (O servidor de nomes local também terá de contatar um servidor de nomes raiz caso não tenha em seu *cache* o endereço de um servidor TLD com.) Esse servidor TLD contém os registros de recursos Type NS e Type A já citados, porque a entidade registradora já os tinha inserido em todos os servidores TLD do domínio com. O servidor TLD com envia uma resposta ao servidor de nomes local de Alice, contendo os dois registros de recursos. Então, o servidor de nomes local envia uma consulta DNS a 212.212.212.1, solicitando o registro Type A correspondente a [www.networkutopia.com](http://www.networkutopia.com). Este registro oferece o endereço IP do servidor Web desejado, digamos, 212.212.71.4, que o servidor local de nomes transmite para o hospedeiro de Alice. Agora, o navegador de Alice pode iniciar uma conexão TCP com o hospedeiro 212.212.71.4 e enviar uma requisição HTTP pela conexão. Ufa! Acontecem muito mais coisas do que percebemos quando navegamos na Web!

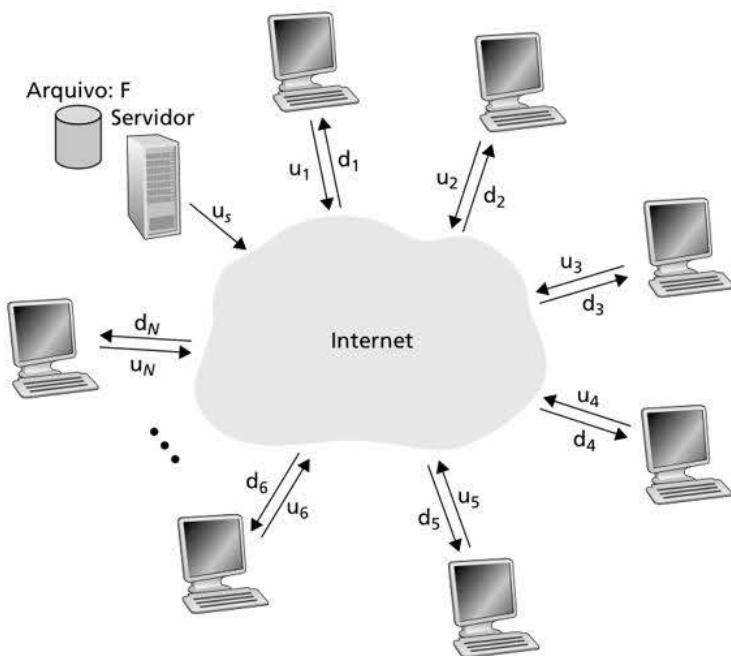
## 2.5 DISTRIBUIÇÃO DE ARQUIVOS P2P

Todas as aplicações descritas neste capítulo até agora – inclusive a Web, e-mail e DNS – empregam arquiteturas cliente-servidor com dependência significativa em servidores de infraestrutura que sempre permanecem ligados. Como consta na Seção 2.1.1, com uma arquitetura P2P, há dependência mínima (quando muito) de servidores de infraestrutura que permanecem sempre ligados. Em vez disso, duplas de hospedeiros间断地连结, chamados pares, comunicam-se direto entre si. Os pares não são de propriedade de um provedor de serviços, mas correspondem a *desktops* e *notebooks* controlados por usuários.

Nesta seção, consideraremos uma aplicação bastante natural do P2P, a saber, a distribuição de um grande arquivo a partir de um único servidor a um grande número de hospedeiros (chamados pares). O arquivo pode ser uma nova versão do sistema operacional Linux, um *patch* de *software* para um sistema operacional ou aplicação, um arquivo de vídeo MPEG. Em uma distribuição de arquivo cliente-servidor, o servidor deve enviar uma cópia para cada par – colocando um enorme fardo sobre o servidor e consumindo grande quantidade de banda deste. Na distribuição de arquivos P2P, cada par pode redistribuir qualquer parte do arquivo recebido para outros pares, auxiliando, assim, o servidor no processo de distribuição. Desde 2020, o protocolo de distribuição de arquivos P2P mais popular é o BitTorrent. Desenvolvido originalmente por Bram Cohen, há agora muitos diferentes clientes independentes de BitTorrent conformes ao protocolo do BitTorrent, assim como há diversos clientes de navegadores Web conformes ao protocolo HTTP. Nesta subseção, examinaremos primeiro a autoescalabilidade de arquiteturas P2P no contexto de distribuição de arquivos. Então, descreveremos o BitTorrent em certo nível de detalhes, destacando suas características mais importantes.

### Escalabilidade de arquiteturas P2P

Para comparar arquiteturas cliente-servidor com arquiteturas P2P, e ilustrar a inerente autoescalabilidade de P2P, consideraremos um modelo quantitativo simples para a distribuição de um arquivo a um conjunto fixo de pares para ambos os tipos de arquitetura. Conforme demonstrado na Figura 2.22, o servidor e os pares são conectados por enlaces de acesso da Internet. A taxa de *upload* do enlace de acesso do servidor é denotada por  $u_s$ ,



**Figura 2.22** Um esquema ilustrativo de distribuição de arquivo.

e a de *upload* do enlace de acesso do par  $i$  é denotada por  $u_i$ , e a taxa de *download* do enlace de acesso do par  $i$  é denotada por  $d_i$ . O tamanho do arquivo a ser distribuído (em bits) é indicado por  $F$ , e o número de pares que querem obter uma cópia do arquivo, por  $N$ . O **tempo de distribuição** é o tempo necessário para que todos os  $N$  pares obtenham uma cópia do arquivo. Em nossa análise do tempo de distribuição a seguir, tanto para a arquitetura cliente-servidor como para a arquitetura P2P, fazemos a hipótese simplificada (e, em geral, precisa [Akella, 2003]) de que o núcleo da Internet tem largura de banda abundante, o que implica que todos os gargalos encontram-se nas redes de acesso. Suponha também que o servidor e os clientes não participam de nenhuma outra aplicação de rede, para que toda sua largura de banda de acesso de *upload* e *download* possa ser totalmente dedicada à distribuição do arquivo.

Determinemos primeiro o tempo de distribuição para a arquitetura cliente-servidor, que indicaremos por  $D_{cs}$ . Na arquitetura cliente-servidor, nenhum dos pares auxilia na distribuição do arquivo. Fazemos as observações a seguir:

- O servidor deve transmitir uma cópia do arquivo a cada um dos  $N$  pares. Assim, o servidor deve transmitir  $NF$  bits. Como a taxa de *upload* do servidor é de  $u_s$ , o tempo para distribuição do arquivo deve ser de pelo menos  $NF/u_s$ .
- Suponha que  $d_{\min}$  indique a taxa de *download* do par com menor taxa de *download*, ou seja,  $d_{\min} = \min\{d_1, d_2, \dots, d_N\}$ . O par com a menor taxa de *download* não pode obter todos os bits  $F$  do arquivo em menos de  $F/d_{\min}$  segundos. Assim, o tempo de distribuição mínimo é de pelo menos  $F/d_{\min}$ .

Reunindo essas observações, temos:

$$D_{cs} \geq \max\left\{\frac{NF}{u_s}, \frac{F}{d_{\min}}\right\}.$$

Isso proporciona um limite inferior para o tempo mínimo de distribuição para a arquitetura cliente-servidor. Nos problemas do final do capítulo, você deverá demonstrar que o servidor pode programar suas transmissões de forma que o limite inferior seja sempre

alcançado. Portanto, consideraremos esse limite inferior fornecido antes como o tempo real de distribuição, ou seja,

$$D_{cs} = \max \left\{ \frac{NF}{u_s}, \frac{F}{d_{\min}} \right\} \quad (2.1)$$

Vemos, a partir da Equação 2.1, que para  $N$  grande o suficiente, o tempo de distribuição cliente-servidor é dado por  $NF/u_s$ . Assim, o tempo de distribuição aumenta linearmente com o número de pares  $N$ . Portanto, por exemplo, se o número de pares de uma semana para a outra for multiplicado por mil, de mil para um milhão, o tempo necessário para distribuir o arquivo para todos os pares aumentará mil vezes.

Passemos agora para uma análise semelhante para a arquitetura P2P, em que cada par pode auxiliar o servidor na distribuição do arquivo. Em particular, quando um par recebe alguns dados do arquivo, ele pode usar sua própria capacidade de *upload* para redistribuir os dados a outros pares. Calcular o tempo de distribuição para a arquitetura P2P é, de certa forma, mais complicado do que para a arquitetura cliente-servidor, visto que o tempo de distribuição depende de como cada par distribui parcelas do arquivo aos outros pares. Não obstante, uma simples expressão para o tempo mínimo de distribuição pode ser obtida (Kumar, 2006). Para essa finalidade, faremos as observações a seguir:

- No início da distribuição, apenas o servidor tem o arquivo. Para levar esse arquivo à comunidade de pares, o servidor deve enviar cada *bit* do arquivo pelo menos uma vez para seu enlace de acesso. Assim, o tempo de distribuição mínimo é de pelo menos  $F/u_s$ . (Diferente do esquema cliente-servidor, um *bit* enviado uma vez pelo servidor pode não precisar ser enviado novamente, visto que os pares podem redistribuir entre si esse *bit*.)
- Assim como na arquitetura cliente-servidor, o par com a menor taxa de *download* não pode obter todos os *bits*  $F$  do arquivo em menos de  $F/d_{\min}$  segundos. Assim, o tempo mínimo de distribuição é de pelo menos  $F/d_{\min}$ .
- Por fim, observemos que a capacidade de *upload* total do sistema como um todo é igual à taxa de *upload* do servidor mais as taxas de *upload* de cada par, ou seja,  $u_{\text{total}} = u_s + u_1 + \dots + u_N$ . O sistema deve entregar (fazer o *upload* de)  $F$  *bits* para cada um dos  $N$  pares, entregando assim um total de  $NF$  *bits*. Isso não pode ser feito em uma taxa mais rápida do que  $u_{\text{total}}$ . Assim, o tempo mínimo de distribuição é também de pelo menos  $NF/(u_s + u_1 + \dots + u_N)$ .

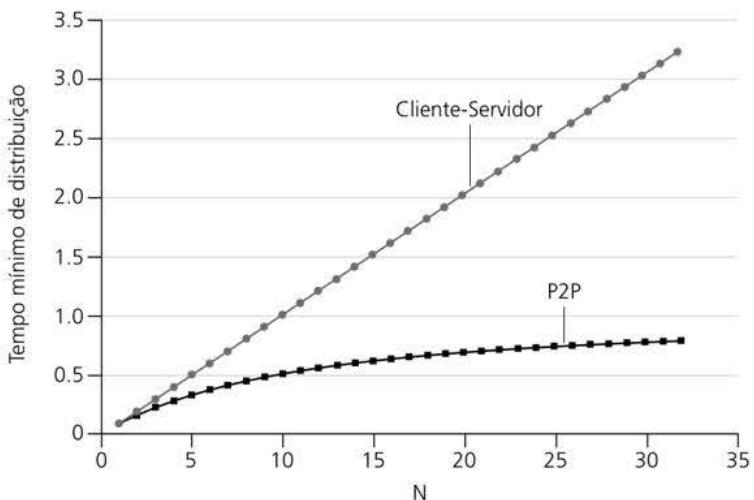
Juntando as três observações, obtemos o tempo mínimo de distribuição para P2P, indicado por  $D_{\text{P2P}}$ :

$$D_{\text{P2P}} \geq \max \left\{ \frac{F}{u_s}, \frac{F}{d_{\min}}, \frac{NF}{u_s + \sum_{i=1}^N u_i} \right\} \quad (2.2)$$

A Equação 2.2 fornece um limite inferior para o tempo mínimo de distribuição para a arquitetura P2P. Ocorre que, se imaginarmos que cada par pode redistribuir um *bit* assim que o recebe, há um esquema de redistribuição que, de fato, alcança esse limite inferior (Kumar, 2006). (Provaremos um caso especial desse resultado nos exercícios.) Na realidade, quando blocos do arquivo são redistribuídos, em vez de *bits* individuais, a Equação 2.2 serve como uma boa aproximação do tempo mínimo real de distribuição. Assim, peguemos o limite inferior fornecido pela Equação 2.2 como o tempo mínimo real de distribuição, que é:

$$D_{\text{P2P}} = \max \left\{ \frac{F}{u_s}, \frac{F}{d_{\min}}, \frac{NF}{u_s + \sum_{i=1}^N u_i} \right\} \quad (2.3)$$

A Figura 2.23 compara o tempo mínimo de distribuição para as arquiteturas cliente-servidor e P2P, pressupondo que todos os pares têm a mesma taxa de *upload*  $u$ . Na Figura 2.23,



**Figura 2.23** Tempo de distribuição para arquiteturas P2P e cliente-servidor.

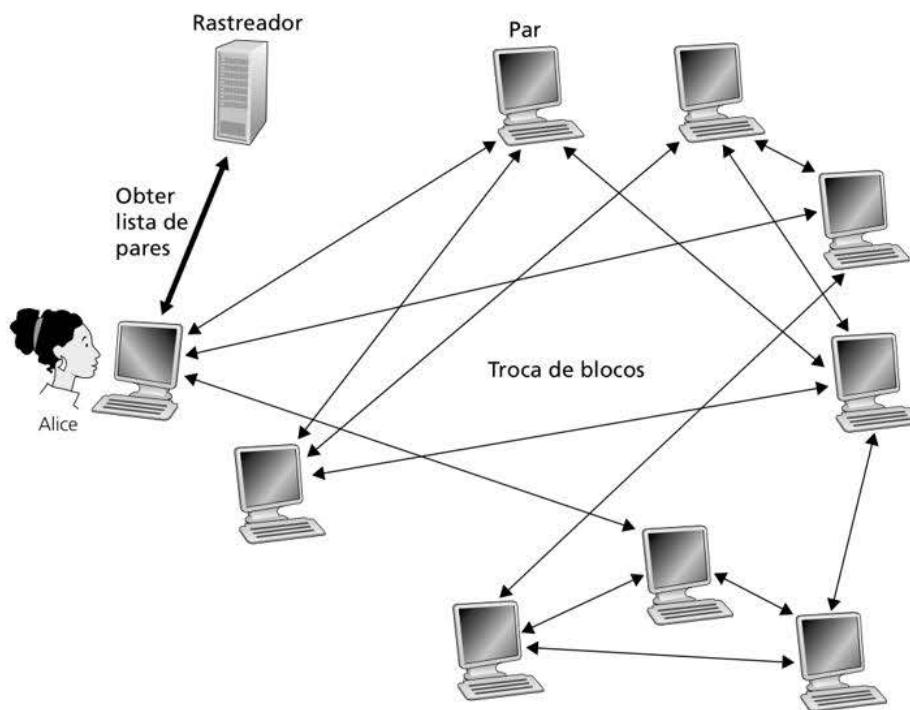
definimos que  $F/u = 1$  hora,  $u_s = 10u$  e  $d_{\min} \geq u_s$ . Assim, um par pode transmitir todo o arquivo em 1 hora, sendo a taxa de transmissão do servidor 10 vezes a taxa de *upload* do par, e (para simplificar) as taxas de *download* de par são definidas grandes o suficiente de forma a não ter efeito. Vemos na Figura 2.23 que, para a arquitetura cliente-servidor, o tempo de distribuição aumenta linearmente e sem limite, conforme cresce o número de pares. No entanto, para a arquitetura P2P, o tempo mínimo de distribuição não é apenas sempre menor do que o tempo de distribuição da arquitetura cliente-servidor; é também de menos do que 1 hora para qualquer número de pares  $N$ . Assim, aplicações com a arquitetura P2P podem ter autoescalabilidade. Tal escalabilidade é uma consequência direta de pares sendo redistribuidores, bem como consumidores de *bits*.

### BitTorrent

O BitTorrent é um protocolo P2P popular para distribuição de arquivos (Chao, 2011). No jargão do BitTorrent, a coleção de todos os pares que participam da distribuição de um determinado arquivo é chamada de *torrent*. Os pares em um *torrent* fazem o *download* de *blocos* de tamanho igual do arquivo entre si, com um tamanho típico de bloco de 256 KBytes. Quando um par entra em um *torrent*, ele não tem nenhum bloco. Com o tempo, ele acumula mais blocos. Enquanto ele faz o *download* de blocos, faz também *uploads* de blocos para outros pares. Uma vez que um par adquire todo o arquivo, ele pode (de forma egoísta) sair do *torrent* ou (de forma altruísta) permanecer e continuar fazendo o *upload* a outros pares. Além disso, qualquer par pode sair do *torrent* a qualquer momento com apenas um subconjunto de blocos, e depois voltar.

Observemos agora, mais atentamente, como opera o BitTorrent. Como é um protocolo e sistema complicado, descreveremos apenas seus mecanismos mais importantes, ignorando alguns detalhes; isso nos permitirá ver a floresta a partir das árvores. Cada *torrent* tem um nó de infraestrutura chamado *rastreador*. Quando um par chega em um *torrent*, ele se registra com o rastreador e periodicamente informa ao rastreador que ainda está lá. Dessa forma, o rastreador mantém um registro dos pares que participam do *torrent*. Um determinado *torrent* pode ter menos de dez ou mais de mil pares participando a qualquer momento.

Como demonstrado na Figura 2.24, quando um novo par, Alice, chega, o rastreador seleciona aleatoriamente um subconjunto de pares (para dados concretos, digamos que sejam 50) do conjunto de pares participantes, e envia os endereços IP desses 50 pares a Alice. Com a lista de pares, ela tenta estabelecer conexões TCP simultâneas com todos. Chamaremos todos os pares com quem Alice consiga estabelecer uma conexão TCP de “pares vizinhos”. (Na Figura 2.24, Alice é representada com apenas três pares vizinhos. Normalmente, ela



**Figura 2.24** Distribuição de arquivos com o BitTorrent.

teria muito mais.) Com o tempo, alguns desses pares podem sair e outros pares (fora dos 50 iniciais) podem tentar estabelecer conexões TCP com Alice. Portanto, os pares vizinhos de um par podem flutuar com o tempo.

A qualquer momento, cada par terá um subconjunto de blocos do arquivo, com pares diferentes com subconjuntos diferentes. De tempos em tempos, Alice pedirá a cada um de seus pares vizinhos (nas conexões TCP) a lista de quais blocos eles têm. Caso Alice tenha  $L$  vizinhos diferentes, ela obterá  $L$  listas de blocos. Com essa informação, Alice emitirá solicitações (novamente, nas conexões TCP) de blocos que ela não tem.

Portanto, a qualquer momento, Alice terá um subconjunto de blocos e saberá quais blocos seus vizinhos têm. Com essa informação, ela terá duas decisões importantes a fazer. Primeiro, quais blocos deve solicitar de início a seus vizinhos, e, segundo, a quais vizinhos deve enviar blocos solicitados. Ao decidir quais blocos solicitar, Alice usa uma técnica chamada *rarest first* (o mais raro primeiro). A ideia é determinar, entre os blocos que ela não tem, quais são os mais raros entre seus vizinhos (ou seja, os blocos que têm o menor número de cópias repetidas) e então solicitar esses blocos mais raros primeiro. Dessa forma, os blocos mais raros são redistribuídos mais depressa, procurando (grosso modo) equalizar os números de cópias de cada bloco no *torrent*.

Para determinar a quais pedidos atender, o BitTorrent usa um algoritmo de troca inteligente. A ideia básica é Alice dar prioridade aos vizinhos que estejam fornecendo seus dados *com a maior taxa*. Especificamente, para cada vizinho, Alice mede de maneira contínua a taxa em que recebe *bits* e determina os quatro pares que lhe fornecem na taxa mais alta. Então, ela reciprocamente envia blocos a esses mesmos quatro pares. A cada 10 segundos, ela recalcula as taxas e talvez modifique o conjunto de quatro pares. No jargão do BitTorrent, esses quatro pares são chamados de **unchoked (não sufocado)**. É importante informar que a cada 30 segundos ela também escolhe um vizinho adicional ao acaso e envia blocos a ele. Chamaremos o vizinho escolhido de Bob. No jargão de BitTorrent, Bob é chamado de **optimistically unchoked (otimisticamente não sufocado)**. Como Alice envia dados a Bob, ela pode se tornar um dos quatro melhores transmissores para Bob, caso em que ele começaria a enviar dados para Alice. Caso a taxa em que Bob envie dados seja alta

o suficiente, ele pode, em troca, tornar-se um dos quatro melhores transmissores para Alice. Em outras palavras, a cada 30 segundos, Alice escolherá ao acaso um novo parceiro de troca e começará a trocar dados com ele. Caso os dois pares estejam satisfeitos com a troca, eles colocarão um ao outro nas suas listas de quatro melhores pares e continuarão a troca até que um dos pares encontre um parceiro melhor. O efeito é que pares capazes de fazer *optimistically unchoked* em taxas compatíveis tendem a se encontrar. A seleção aleatória de vizinho também permite que novos pares obtenham blocos, de forma que possam ter algo para trocar. Todos os pares vizinhos, além desses cinco pares (quatro pares “*top*” e um em experiência), estão “sufocados”, ou seja, não recebem nenhum bloco de Alice. O BitTorrent tem diversos mecanismos interessantes não discutidos aqui, incluindo pedaços (miniblocos), *pipelining* (tubulação), primeira seleção aleatória, modo *endgame* (fim de jogo) e *anti-snubbing* (antirrejeição) (Cohen, 2003).

O mecanismo de incentivo para troca descrito costuma ser chamado de *tit-for-tat* (olho por olho) (Chen, 2003). Demonstrou-se que esse esquema de incentivo pode ser burlado (Liogkas, 2006; Locher, 2006; Piatek, 2008). Não obstante, o ecossistema do BitTorrent é muito bem-sucedido, com milhões de pares simultâneos compartilhando arquivos ativamente em centenas de milhares de *torrents*. Caso o BitTorrent tivesse sido projetado sem o *tit-for-tat* (ou uma variante), mas com o restante da mesma maneira, ele talvez nem existisse mais, visto que a maioria dos usuários são pessoas que apenas querem obter as coisas de graça (Saroiu, 2002).

Para encerrar nossa discussão sobre P2P, mencionamos brevemente outra aplicação da tecnologia, a saber, a tabela *hash* distribuída (DHT, do inglês *distributed hash table*). Uma tabela *hash* distribuída é um banco de dados simples, com os registros distribuídos pelos pares em um sistema P2P. As DHTs foram implementadas em larga escala (p. ex., BitTorrent) e foram objeto de muitas pesquisas.

## 2.6 STREAMING DE VÍDEO E REDES DE DISTRIBUIÇÃO DE CONTEÚDO

De acordo com muitas estimativas, o *streaming* de vídeo (incluindo Netflix, YouTube e Amazon Prime) representa cerca de 80% de todo o tráfego na Internet em 2020 (Cisco, 2020). Esta seção apresenta uma visão geral sobre como os serviços populares de *streaming* de vídeo são implementados na Internet atual. Veremos que são implementados usando protocolos de nível de aplicação e servidores que atuam, em certos aspectos, como um *cache*.

### 2.6.1 Vídeo de Internet

No *streaming* de vídeo armazenado, o meio subjacente é o vídeo pré-gravado, como um filme, um seriado, um evento esportivo pré-gravado ou um vídeo pré-gravado gerado pelo usuário (como aqueles vistos no YouTube). Esses vídeos são colocados em servidores, e os usuários enviam solicitações aos servidores para verem os vídeos *sob demanda*. Muitas empresas de Internet oferecem hoje *streaming* de vídeo, incluindo Netflix, YouTube (Google), Amazon e TikTok.

Mas antes de nos lançarmos em uma discussão sobre *streaming* de vídeo, devemos entender um pouco melhor a mídia de vídeo em si. Um vídeo é uma sequência de imagens, em geral exibidas a uma velocidade constante, por exemplo, 24 ou 30 imagens por segundo. Uma imagem não compactada, codificada digitalmente, consiste em uma matriz de *pixels*, com cada *pixel* codificado em uma série de *bits* para representar luminosidade e cor. Uma característica importante do vídeo é que ele pode ser compactado, compensando assim a qualidade com a taxa de *bits*. Os algoritmos de compactação de hoje, prontos para uso,

podem compactar um vídeo basicamente para qualquer taxa de *bits* desejada. É claro que, quanto mais alta a taxa de *bits*, melhor a qualidade da imagem e, em geral, melhor a experiência de exibição do usuário.

De uma perspectiva de redes, provavelmente a característica mais saliente do vídeo é a sua alta taxa de *bits*. Vídeos de Internet compactados costumam variar de 100 kbits/s para os vídeos de baixa qualidade até mais de 4 Mbits/s para o *streaming* de filmes em alta definição; o *streaming* de vídeo em qualidade 4K supõe uma taxa de *bits* de mais de 10 Mbits/s. Tudo isso pode se traduzir em uma quantidade enorme de tráfego e armazenamento de dados, especialmente para vídeos de alta qualidade. Por exemplo, um único vídeo de 2 Mbits/s com 67 minutos de duração consome 1 gigabyte de armazenamento e tráfego. A medida de desempenho mais importante para o *streaming* de vídeo é, de longe, a vazão fim a fim média. De modo a permitir reprodução contínua, a rede deve oferecer uma vazão média à aplicação de *streaming* que seja igual ou maior à taxa de *bits* do vídeo compactado.

Também podemos usar a compactação para criar múltiplas versões do mesmo vídeo, cada uma em um nível de qualidade diferente. Por exemplo, podemos usar a compactação para criar, digamos, três versões do mesmo vídeo, nas taxas de 300 kbits/s, 1 Mbit/s e 3 Mbits/s. Os usuários podem decidir qual versão eles querem ver como uma função da largura de banda disponível. Os usuários com conexões de alta velocidade com a Internet escolheriam a de 3 Mbits/s; os que assistem ao vídeo por 3G com um *smartphone* poderiam escolher a versão de 300 kbits/s.

## 2.6.2 Streaming por HTTP e DASH

No *streaming* por HTTP, o vídeo é apenas armazenado em um servidor HTTP como um arquivo comum com um URL específico. Quando um usuário quer assistir a esse vídeo, ele estabelece uma conexão TCP com o servidor e realiza um comando HTTP GET para aquele URL. O servidor envia então o arquivo de vídeo, dentro de uma mensagem de resposta HTTP, tão depressa quanto os protocolos de rede subjacentes e as condições de congestionamento de tráfego permitem. No cliente, os *bytes* são armazenados em um *buffer* de aplicação cliente. Uma vez que o número de *bytes* neste *buffer* exceder um limite predeterminado, a aplicação cliente inicia uma reprodução – mas especificamente, ela captura quadros do vídeo do *buffer* da aplicação cliente, descompacta os quadros e os apresenta na tela do usuário. Dessa forma, a aplicação de *streaming* está reproduzindo o vídeo à medida que o recebe e guardando no *buffer* os quadros correspondentes às partes posteriores do vídeo.

Apesar de o *streaming* por HTTP, como descrito no parágrafo anterior, estar sendo amplamente utilizado na prática (p. ex., pelo YouTube desde o seu início), ele possui uma grande deficiência: todos os clientes recebem a mesma codificação do vídeo, apesar das grandes variações na quantidade de largura de banda disponível para o cliente, tanto para diferentes clientes quanto ao longo do tempo para um mesmo cliente. Isto levou ao desenvolvimento de um novo tipo de *streaming* baseado em HTTP, normalmente chamado de **Streaming Adaptativo Dinamicamente sobre HTTP (DASH)**, do inglês *Dynamic Adaptive Streaming over HTTP*). Pelo DASH, o vídeo é codificado em muitas versões diferentes, cada qual com uma taxa de *bits* e um diferente nível de qualidade. O cliente requisita dinamicamente, dessas diferentes versões, trechos de alguns segundos de segmentos do vídeo. Quando a quantidade de largura de banda disponível é alta, o cliente em geral seleciona trechos de uma versão que possui uma taxa alta; e quando a largura de banda disponível é baixa, ele naturalmente seleciona de uma versão cuja taxa é baixa. O cliente seleciona diferentes trechos um de cada vez com mensagens de requisição HTTP GET (Akhshabi, 2011).

O DASH permite aos clientes com diferentes taxas de acesso à Internet usufruir de *streaming* de vídeo a diferentes taxas de codificação. Clientes com conexões 3G lentas podem receber uma versão com baixa taxa de *bits* (e baixa qualidade), e clientes com conexões por fibra ótica podem receber versões de alta qualidade. O uso de DASH permite a um cliente se adaptar à largura de banda disponível, se a largura de banda fim a fim mudar durante

a sessão. Esta funcionalidade é particularmente importante para usuários de dispositivos móveis, que com frequência experimentam flutuações na largura de banda disponível, conforme se movimentam de uma estação rádio-base para outra.

Com o DASH, cada versão do vídeo é armazenada em um servidor HTTP, cada uma com uma diferente URL. O servidor HTTP também possui um **arquivo de manifesto**, que provê um URL para cada versão junto com a sua taxa de *bits*. Primeiro, o cliente requisita o arquivo de manifesto e identifica as várias versões. O cliente, então, seleciona um trecho de cada vez, especificando um URL e um intervalo de *bytes* em uma mensagem de requisição HTTP GET para cada trecho. Enquanto estiver baixando trechos, o cliente também mede a largura de banda de recepção e executa um algoritmo de determinação de taxa para selecionar o próximo trecho a ser requisitado. Naturalmente, se o cliente possui muito vídeo em seu *buffer* e se a largura de banda de recepção que foi medida for alta, ele escolherá um trecho de uma versão com taxa alta associada. E, claro, se o cliente possui pouco vídeo em seu *buffer* e a sua largura de banda de recepção for baixa, escolherá um trecho de uma versão de taxa baixa. Portanto, DASH permite ao cliente alternar livremente em diferentes níveis de qualidade.

### 2.6.3 Redes de distribuição de conteúdo

Muitas companhias de vídeo na Internet têm distribuído multi-Mbits/s de *streams* sob demanda para milhões de usuários diariamente. O YouTube, por exemplo, com uma biblioteca de centenas de milhões de vídeos, distribui centenas de milhões de *streams* de vídeos para usuários ao redor do mundo inteiro, todos os dias. Distribuir todo esse tráfego para locais ao redor do mundo inteiro enquanto provê reprodução contínua e alta interatividade claramente representa uma tarefa desafiadora.

Para uma empresa de vídeos na Internet, talvez a abordagem mais simples e direta para prover serviços de *streaming* de vídeo seja construir um único *datacenter* gigante, armazenar ali todos os vídeos e realizar o *streaming* dos vídeos diretamente do *datacenter* para os clientes ao redor do mundo. No entanto, existem três grandes problemas com essa abordagem. Primeiro, se o cliente estiver muito longe do *datacenter*, os pacotes que vão do servidor para o cliente atravessarão muitos enlaces de comunicação, e também possivelmente passarão por muitos ISPs, com alguns destes ISPs talvez localizados em diferentes continentes. Se um dos enlaces fornecer uma taxa de transferência menor que a de consumo do vídeo, a taxa de transferência fim a fim será também abaixo da de consumo, resultando, para o usuário, em atrasos incômodos por congestionamento. (Lembre-se, do Capítulo 1, que uma taxa de transferência fim a fim de um *streaming* é controlada pela taxa de transferência de seu enlace de gargalo.) A probabilidade de isso acontecer cresce na mesma proporção em que o número de enlaces no caminho fim a fim aumenta. Uma segunda desvantagem é que um vídeo popular será possivelmente enviado muitas vezes pelos mesmos enlaces de comunicação. Isto não apenas é um desperdício de largura de banda de rede, mas a própria companhia de vídeo pela Internet estará pagando ao seu ISP (conectado com o *datacenter*) por enviar os *mesmos bytes* pela Internet muitas e muitas vezes. Um terceiro problema com essa solução é que um único *datacenter* representa um único ponto de falha – se o *datacenter* ou seus enlaces para a Internet caírem, ele não será capaz de distribuir *nenhum* vídeo por *streaming*.

Para enfrentar o desafio de distribuição de uma quantidade maciça de dados de vídeo para usuários espalhados ao redor do mundo, quase todas as maiores companhias de *streaming* de vídeo fazem uso de **CDNs**. Uma CDN gerencia servidores em múltiplas localidades distribuídas geograficamente, armazena cópias dos vídeos (e outros tipos de conteúdos Web, incluindo documentos, imagens e áudio) em seus servidores, e tenta direcionar cada requisição do usuário para uma localidade CDN que proporcionará a melhor experiência para o usuário. A CDN pode ser uma **CDN privada**, isto é, que pertence ao próprio provedor de conteúdo; por exemplo, a CDN da Google distribui vídeos do YouTube e outros tipos de conteúdo. A CDN também pode ser uma **CDN de terceiros**, que distribui conteúdo em nome de múltiplos provedores de conteúdo; a Akamai, a Limelight e a Level-3 operam CDNs de terceiros, por exemplo.

Uma visão geral e fácil de ler sobre as CDNs modernas se encontra em Leighton (2009) e em Nygren (2010).

Em geral, as CDNs adotam uma entre as duas filosofias de instalação de servidores (Huang, 2008):

- **Enter deep.** Uma filosofia, que começou a ser praticada pela Akamai, é *entrar profundamente* (*enter deep*) dentro das redes de acesso dos Provedores de Serviço de Internet pela implantação de *clusters* de servidores no acesso de ISPs por todo o mundo. (Redes de acesso são descritas na Seção 1.3.) A Akamai usa esta abordagem com *clusters* de servidores em milhares de locais. O objetivo é conseguir proximidade com os usuários finais, melhorando assim o atraso percebido pelo usuário e a taxa de transferência pela diminuição do número de enlaces e roteadores entre o usuário final e o servidor de CDN do qual ele recebe conteúdo. Em razão desse projeto altamente distribuído, a tarefa de manter e gerenciar os agrupamentos se torna desafiadora.
- **Bring home.** Uma segunda filosofia de projeto, adotada pela Limelight e por muitas outras companhias de CDN, é *trazer para dentro de casa os ISPs*, construindo *clusters* enormes, mas em um número menor (p. ex., dezenas) de lugares. Em vez de entrar nos ISPs de acesso, estas CDNs normalmente colocam seus *clusters* em pontos de troca da Internet (IXPs – ver Seção 1.3). Em comparação com a filosofia de projeto *Enter deep*, o projeto *Bring home* em geral resulta em menos desperdício com gerenciamento e manutenção, mas com um maior atraso e menores taxas de transferência para os usuários finais.

Uma vez que os seus *clusters* estejam operando, a CDN replica conteúdo através deles. A CDN pode não querer pôr uma cópia de todos os vídeos em cada *cluster*, já que alguns vídeos são vistos raramente ou são populares só em alguns países. De fato, muitas CDNs não enviam vídeos para seus *clusters*, mas, em vez disso, usam uma estratégia por demanda simples: se um cliente requisita um vídeo de um *cluster* que não o está armazenando, o *cluster* recupera o vídeo (de um repositório central ou de outro *cluster*) e armazena uma cópia localmente enquanto transmite o vídeo para o cliente ao mesmo tempo. Similarmente aos *caches* Web (ver Seção 2.2.5), quando a memória do *cluster* fica cheia, ele remove vídeos que não são requisitados com frequência.

## Operação da CDN

Tendo identificado as duas principais técnicas de implantação de uma CDN, vamos agora nos aprofundar em como uma CDN opera. Quando um navegador em um hospedeiro do usuário recebe a instrução para recuperar um vídeo específico (identificado por um URL), a CDN tem de interceptar a requisição a fim de: (1) determinar um *cluster* de servidor de CDN apropriado para o cliente naquele momento, e (2) redirecionar a requisição do cliente para um servidor naquele *cluster*. Falaremos, em linhas gerais, como uma CDN pode determinar um *cluster* apropriado. Antes, no entanto, vamos examinar os mecanismos que são usados para interceptar e redirecionar uma requisição.

A maioria das CDNs utiliza o DNS para interceptar e redirecionar requisições; uma discussão interessante sobre esse uso do DNS pode ser vista em Vixie (2009). Consideremos um simples exemplo para ilustrar como o DNS normalmente é envolvido. Suponha que um provedor de conteúdo, NetCinema, utiliza outra companhia de CDN, KingCDN, para distribuir seus vídeos para os seus consumidores. Nas páginas de Internet do NetCinema, cada vídeo está associado a um URL que inclui a palavra “vídeo” e um identificador único para o vídeo em si; por exemplo *Transformers 7* pode ser associado a <<http://video.netcinema.com/6Y7B23V>>. Então, acontecem seis passos, como mostra a Figura 2.25.

1. O usuário visita a página da Internet no NetCinema.
2. Quando o usuário clica no link <<http://video.netcinema.com/6Y7B23V>>, o hospedeiro do usuário envia uma consulta de DNS para [video.netcinema.com](http://video.netcinema.com).

## ESTUDO DE CASO

### INFRAESTRUTURA DE REDE DA GOOGLE

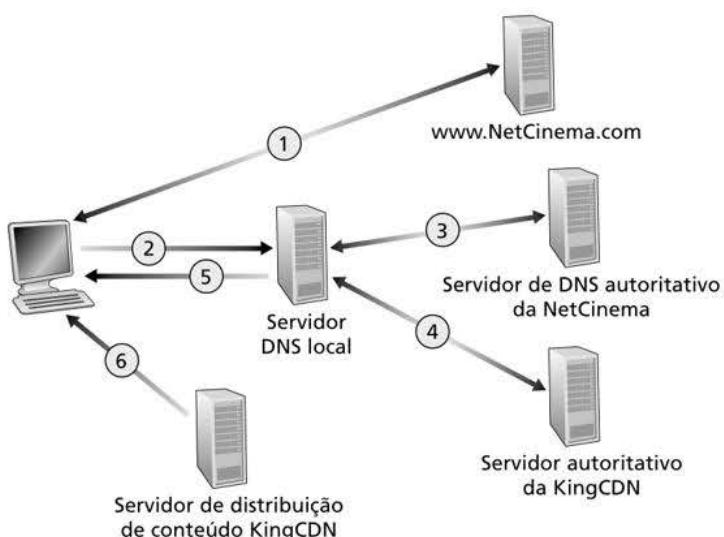
Para suportar sua vasta lista de serviços – incluindo busca, Gmail, calendário, vídeos do YouTube, mapas, documentos e redes sociais –, a Google implementou uma extensa rede privada e infraestrutura de CDN. A infraestrutura de CDN da Google possui três camadas de agrupamentos de servidores:

- Dezenove “mega datacenters” na América do Norte, Europa e Ásia (Google Locations, 2020), com cada datacenter contendo algo na ordem de 100 mil servidores. Esses mega datacenters são responsáveis por servir conteúdo dinâmico (e muitas vezes personalizado), incluindo resultados de busca e mensagens de Gmail.
- Com cerca de 90 clusters em IXPs espalhados pelo mundo, com cada cluster consistindo em centenas de servidores (Adhikari, 2011a) (Google CDN, 2020). Esses agrupamentos são responsáveis por servir conteúdo estático, incluindo vídeos do YouTube.
- Muitas centenas de clusters localizados dentro de um ISP de acesso são “Enter deep”. Este tipo de cluster consiste em dezenas de servidores dentro

de um único armário. Os servidores *Enter deep* realizam a divisão de TCP (veja a Seção 3.7) e servem conteúdo estático (Chen, 2011), incluindo as partes estáticas das páginas da Internet que incluem resultados de busca.

Todos esses datacenters e localizações de clusters de servidores estão dispostos juntos em rede com a rede privada da Google. Quando um usuário efetua um pedido de busca, geralmente esse pedido é enviado primeiro do ISP local para o cache de um servidor *enter deep* próximo de onde o conteúdo estático é recuperado; embora forneça o conteúdo estático ao cliente, este cache próximo também encaminha a consulta pela rede privada da Google para um dos mega datacenters, de onde o resultado da busca personalizada é obtido. Para um vídeo do YouTube, o vídeo em si pode vir de um dos caches *bring home*, enquanto partes da página Web ao redor do vídeo podem vir do cache *enter deep* nas vizinhanças, e os anúncios ao redor do vídeo podem vir dos datacenters. Em suma, exceto pelos ISPs locais, os serviços na nuvem da Google são, em grande parte, fornecidos por uma infraestrutura de rede que é independente da Internet pública.

3. O Servidor de DNS Local (LDNS, do inglês *Local DNS Server*) do usuário retransmite a consulta de DNS a um servidor DNS autoritativo para NetCinema, o qual encontra a palavra “video” no nome do hospedeiro video.netcinema.com. Para “entregar” a consulta de DNS à KingCDN, em vez de um endereço IP, o servidor de DNS autoritativo do NetCinema retorna ao LDNS um nome de hospedeiro no domínio da KingCDN, por exemplo, a1105.kingcdn.com.



**Figura 2.25** DNS redireciona uma requisição do usuário para um servidor de CDN.

4. Desse ponto em diante, a consulta de DNS entra na infraestrutura da DNS privada da KingCDN. O LDNS do usuário, então, envia uma segunda consulta, agora para a1105. kingcdn.com, e o sistema de DNS da KingCDN por fim retorna os endereços IP de um servidor de conteúdo da KingCDN para o LDNS. Assim, é dentro do sistema de DNS da KingCDN que é especificado o servidor de CDN de onde o cliente receberá o seu conteúdo.
5. O LDNS encaminha o endereço IP do nó de conteúdo/serviço da CDN para o hospedeiro do usuário.
6. Uma vez que o cliente obtém o endereço IP de um servidor de conteúdo da KingCDN, ele estabelece uma conexão TCP direta com o servidor que se encontra nesse endereço IP e executa uma requisição HTTP GET para obter o vídeo. Se utilizar DASH, o servidor primeiro enviará ao cliente um arquivo de manifesto com uma lista de URLs, uma para cada versão do vídeo, e o cliente irá dinamicamente selecionar trechos das diferentes versões.

### Estratégias de seleção de *cluster*

No centro de qualquer distribuição de uma CDN está a **estratégia de seleção de *cluster***, isto é, um mecanismo para direcionamento dinâmico de clientes para um *cluster* de servidor ou um *datacenter* dentro da CDN. Como acabamos de ver, a CDN descobre qual o endereço IP do servidor LDNS do cliente através da consulta ao DNS do cliente. Após descobrir esse endereço IP, a CDN precisa selecionar um *cluster* apropriado baseado nesse endereço IP. As CDNs geralmente empregam estratégias próprias de seleção de *cluster*. Examinaremos agora algumas abordagens, cada uma com suas vantagens e desvantagens.

Uma estratégia simples é associar o cliente ao *cluster* que está **geograficamente mais próximo**. Usando bases de dados de geolocalização comerciais (p. ex., Quova [2020] e Max-Mind [2020]), cada endereço IP de LDNS é mapeado para uma localização geográfica. Quando uma requisição DNS é recebida de um determinado LDNS, a CDN escolhe o *cluster* mais próximo geograficamente, isto é, o *cluster* que está a uma distância menor, em quilômetros, do LDNS. Esta solução pode funcionar razoavelmente bem para uma boa parte dos clientes (Agarwal, 2009). No entanto, para alguns clientes, essa solução pode ter um desempenho ruim, uma vez que o *cluster* geograficamente mais próximo pode não ser o mais próximo se considerarmos o comprimento ou o número de saltos do caminho da rede. Além do mais, existe um problema inerente a todas as abordagens baseadas em DNS, que consiste no fato de que alguns usuários finais são configurados para utilizar LDNSs remotos (Shaikh, 2001; Mao, 2002). Nesses casos, a localização do LDNS pode ser muito longe da localização do cliente. Além disso, essa estratégia elementar ignora a variação no atraso e a largura de banda disponível dos caminhos da Internet, porque sempre associa o mesmo *cluster* a determinado cliente.

Para determinar o melhor *cluster* para um cliente baseado nas *atuais* condições de tráfego, as CDNs podem, como alternativa, realizar **medidas em tempo real** do atraso e problemas de baixo desempenho entre seus *clusters* e clientes. Por exemplo, uma CDN pode ter cada um de seus *clusters* periodicamente enviando mensagens de verificação (p. ex., mensagens *ping* ou consultas de DNS) para todos os LDNSs do mundo inteiro. Um obstáculo a essa técnica é o fato de que muitos LDNSs estão configurados para não responder a esse tipo de mensagem.

#### 2.6.4 Estudos de caso: Netflix e YouTube

Concluímos nossa discussão sobre *streaming* de vídeo armazenado observando duas implementações em larga escala de grande sucesso: Netflix e YouTube. Veremos que os dois sistemas utilizam métodos bastante diferentes, mas ambos empregam muitos dos princípios destacados e discutidos nesta seção.

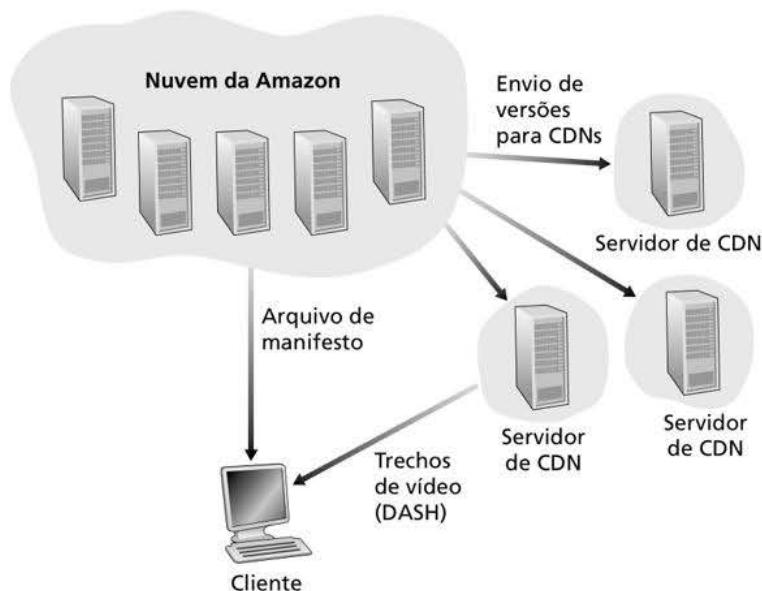
## Netflix

Desde 2020, a Netflix é o maior provedor de serviços de filmes e séries de TV on-line na América do Norte. Como discutiremos abaixo, a distribuição de vídeo da Netflix possui dois componentes principais: a nuvem da Amazon e a sua própria infraestrutura de CDN privada.

A Netflix possui um *site* que cuida de diversas funções, incluindo registro e *login* de usuários, faturas, catálogo de filmes para navegação e busca e um sistema de recomendações. Como mostra a Figura 2.26, esse *site* (e os bancos de dados associados no *back-end*) rodam totalmente nos servidores da Amazon, na nuvem da Amazon. Além disso, a nuvem da Amazon cuida das seguintes funções críticas:

- **Obtenção de conteúdo.** Antes de a Netflix poder distribuir um filme para seus clientes, é necessário obter e processar o filme. A Netflix recebe as versões principais de estúdio e as carrega para os hospedeiros na nuvem da Amazon.
- **Processamento de conteúdo.** As máquinas na nuvem da Amazon criam muitos formatos diferentes para cada filme, de acordo com uma série de tocadores de vídeo do cliente, rodando em computadores de mesa, *smartphones* e consoles de jogos conectados a aparelhos de televisão. Uma versão diferente é criada para cada formato e as múltiplas taxas de *bits*, permitindo assim que se utilize *streaming* de vídeo por HTTP usando DASH.
- **Descarregamento de versões para as CDNs.** Uma vez que todas as versões de um filme foram criadas, os hospedeiros na nuvem da Amazon descarregam as versões para as CDNs.

Quando lançou seu serviço de *streaming* de vídeo em 2007, a Netflix empregava CDNs de terceiros para distribuir o seu conteúdo de vídeo. Desde então, a empresa criou a própria CDN privada, da qual distribui todo o seu conteúdo. Para criar a sua própria CDN, a Netflix instalou armários de servidores em IXPs e nas próprias ISPs residenciais. Atualmente, a Netflix possui armários de servidores em mais de 200 locais de IXPs (para uma lista atual dos IXPs que hospedam estantes da Netflix, ver Bottger [2018] e Netflix Open Connect [2020]). Além disso, centenas de locais de ISPs hospedam armários de servidores da Netflix (ver também Netflix Open Connect [2020]), onde a empresa fornece aos ISPs parceiros em potencial instruções sobre como instalar um armário de servidores da Netflix (gratuita) para as suas redes. Cada servidor possui várias portas de Ethernet de 10 Gbits/s e mais



**Figura 2.26** Plataforma de streaming de vídeo da Netflix.

de 100 *terabytes* de armazenamento. O número de servidores em cada armário varia: as instalações nos IXPs muitas vezes têm dezenas de servidores e contêm toda a biblioteca de *streaming* de vídeo da Netflix, incluindo múltiplas versões dos vídeos para suportar o uso do DASH. A Netflix não usa *pull-caching* (Seção 2.2.5) para preencher os seus servidores de CDN nos IXPs e ISPs. Em vez disso, a Netflix distribui os vídeos para os servidores de CDN por um sistema de envio (*push*) fora do horário de pico. Para os locais que não podem armazenar todo o catálogo, a Netflix transmite apenas os vídeos mais populares, que são determinados diariamente. O projeto de CDN da Netflix está descrito em mais detalhes nos vídeos de YouTube (Netflix Video 1) e (Netflix Video 2); veja também (Bottger, 2018).

Tendo descrito os componentes da arquitetura da Netflix, vamos analisar mais de perto a interação entre o cliente e os vários servidores envolvidos na entrega do filme. Como indicado anteriormente, as páginas de Internet para se navegar pela biblioteca de vídeos da Netflix usam os servidores da nuvem da Amazon. Quando o usuário seleciona um filme para assistir, o *software* da Netflix, executado na nuvem da Amazon, primeiro determina quais dos seus servidores de CDN possuem cópias do filme. Entre os servidores que o possuem, o *software* determina então o “melhor” servidor para a solicitação daquele cliente. Se este utiliza um ISP residencial que possui um armário de servidores de CDN da Netflix instalado, e este armário possui uma cópia do filme solicitado, então normalmente um servidor nesse armário é selecionado. Se não, em geral, um servidor em um IXP próximo é selecionado.

Depois que determina o servidor de CDN que deve transmitir o conteúdo, a Netflix envia ao cliente o endereço IP do servidor específico junto com o arquivo de manifesto, que possui os URLs das diferentes versões do filme solicitado. O cliente e aquele servidor da CDN então interagem usando DASH. Mais especificamente, como foi descrito na Seção 2.6.2, o cliente utiliza o cabeçalho do intervalo de *bytes* nas mensagens de requisição HTTP GET para requisitar trechos das diferentes versões do filme. A Netflix usa trechos de cerca de 4 segundos (Adhikari, 2012). Enquanto os trechos vão sendo baixados, o cliente mede a vazão recebida e executa um algoritmo para determinação da taxa, a fim de definir a qualidade do próximo trecho da requisição seguinte.

A Netflix representa muitos dos princípios-chave discutidos antes nesta seção, incluindo *streaming* adaptativo e distribuição por CDN. Contudo, como usa a sua própria CDN privada, que distribui apenas vídeo (e não páginas Web), a Netflix pode simplificar e adaptar o seu projeto de CDN. Em especial, a Netflix não precisa empregar redirecionamento de DNS, como discutido na Seção 2.6.3, para conectar um determinado cliente a um servidor de CDN; em vez disso, o *software* da empresa (executado na nuvem da Amazon) informa o cliente diretamente que deve usar um determinado servidor de CDN. Além disso, a CDN na Netflix usa *push-caching* (armazenagem de informações não solicitadas), não *pull-caching* (armazenagem de informações solicitadas) (Seção 2.2.5): o conteúdo é mandado para os servidores em horários agendados, fora dos horários de pico, em vez de dinamicamente, em resposta a ausências no *cache*.

## YouTube

Com centenas de horas de vídeo enviadas para o *site* a cada minuto e vários bilhões de visualizações por dia, o YouTube é indiscutivelmente o maior *site* da Internet no mundo para compartilhamento de vídeos. O YouTube começou seus serviços em abril de 2005 e foi adquirido pela Google em novembro de 2006. Embora o projeto e os protocolos da Google/YouTube sejam proprietários, através de muitos esforços de medição independentes, podemos ter um entendimento básico sobre como o YouTube opera (Zink, 2009; Torres, 2011; Adhikari, 2011a). Assim como a Netflix, o YouTube faz uso extensivo da tecnologia CDN para distribuir seus vídeos (Torres, 2011). Também como a Netflix, a Google utiliza sua própria CDN privada para distribuir os vídeos do YouTube e tem instalado *clusters* de servidores em centenas de locais de IXPs e ISPs diferentes. Desses locais e diretamente dos seus enormes *datacenters*, a Google distribui os vídeos do YouTube (Adhikari 2011a). Ao contrário da Netflix, no entanto, a Google usa *pull-caching* (armazenagem de informações

solicitadas) como descrito na Seção 2.2.5, e redirecionamentos de DNS, como descrito na Seção 2.6.3. Na maior parte do tempo, a estratégia de seleção de *cluster* da Google direciona o cliente ao *cluster* para o qual o RTT entre o cliente e o *cluster* seja o menor; porém, a fim de balancear a carga através dos *clusters*, algumas vezes o cliente é direcionado (via DNS) a um *cluster* mais distante (Torres, 2011).

O YouTube emprega *streaming* por HTTP, e muitas vezes disponibiliza uma pequena quantidade de versões diferentes do vídeo, cada uma com uma taxa de *bits* diferente e correspondente ao nível de qualidade. O YouTube não utiliza *streaming* adaptativo (como o DASH), preferindo exigir que o usuário selecione manualmente uma versão. No intuito de economizar tanto largura de banda quanto recursos do servidor, que poderiam ser desperdiçados por repositionamento ou por término precoce, o YouTube usa a requisição HTTP de intervalo de *bytes*, de modo a limitar o fluxo de dados transmitidos após uma quantidade do vídeo ter sido obtida pela pré-busca.

Vários milhões de vídeos são enviados ao YouTube diariamente. Não apenas vídeos do YouTube são enviados do servidor ao cliente por HTTP, mas os produtores também enviam seus vídeos do cliente ao servidor por HTTP. O YouTube processa cada vídeo que recebe, convertendo-o para um formato de vídeo próprio e criando várias versões em diferentes taxas de *bits*. Esse processamento ocorre todo dentro dos *datacenters* da Google. (Veja o estudo de caso sobre a infraestrutura de rede da Google, na Seção 2.6.3.)

## 2.7 PROGRAMAÇÃO DE SOCKETS: CRIANDO APLICAÇÕES DE REDE

Agora que já examinamos várias importantes aplicações de rede, vamos explorar como são escritos programas de aplicação de rede. Lembre-se de que na Seção 2.1 dissemos que muitas aplicações de rede consistem em um par de programas – um programa cliente e um programa servidor – que residem em dois sistemas finais diferentes. Quando são executados, criam-se um processo cliente e um processo servidor, que se comunicam entre si lendo de seus *sockets* e escrevendo através deles. Ao criar uma aplicação de rede, a tarefa principal do programador é escrever o código tanto para o programa cliente como para o programa servidor.

Há dois tipos de aplicações de rede. Um deles é uma execução cuja operação é especificada em um padrão de protocolo, por exemplo, em um RFC ou algum outro documento de padrões; essa aplicação às vezes é denominada “aberta”, pois as regras que especificam sua operação são conhecidas de todos. Para essa implementação, os programas cliente e servidor devem obedecer às regras ditadas pelo RFC. Por exemplo, o programa cliente poderia ser uma execução do lado do cliente do protocolo HTTP descrito na Seção 2.2 e definido precisamente no RFC 2616; e o programa servidor, uma implementação do protocolo de servidor HTTP também descrito de modo preciso no RFC 2616. Se um programador escrever um código para o programa cliente e outro programador independente escrever um código para o programa servidor e ambos seguirem com atenção as regras do RFC, então os dois programas poderão interagir. De fato, muitas das aplicações de rede de hoje envolvem comunicação entre programas cliente e servidor que foram criados por programadores diferentes – por exemplo, um navegador Google Chrome que se comunica com um servidor Web Apache, ou um cliente BitTorrent que se comunica com um rastreador BitTorrent.

O outro tipo de aplicação de rede é uma aplicação de rede proprietária. Nesse caso, os programas cliente e servidor empregam um protocolo de camada de aplicação que *não* foi publicado abertamente em um RFC ou em outro lugar. Um único desenvolvedor (ou equipe de desenvolvimento) cria ambos os programas cliente e servidor e tem completo controle sobre o que entra no código. Mas como o código não implementa um protocolo de domínio público, outros programadores independentes não poderão desenvolver código que interage com a aplicação.

Nesta seção e na próxima, examinaremos as questões fundamentais do desenvolvimento de uma aplicação cliente-servidor, e “sujaremos nossas mãos” examinando o código que executa uma aplicação cliente-servidor muito simples. Durante a fase de desenvolvimento, uma das primeiras decisões que o programador deve tomar é se a aplicação rodará em TCP ou UDP. Lembre-se de que o TCP é orientado para conexão e provê um canal confiável de cadeia de *bytes*, pelo qual fluem dados entre dois sistemas finais. O UDP não é orientado para conexão e envia pacotes de dados independentes de um sistema final ao outro, sem nenhuma garantia de entrega. Lembre-se também que, quando um programa, cliente ou servidor, executa um protocolo definido em um RFC, deve usar o número de porta conhecido associado com o protocolo; por outro lado, ao desenvolver uma aplicação proprietária, o programador deve ter o cuidado de evitar esses números de porta conhecidos. (Números de portas foram discutidos brevemente na Seção 2.1. Eles serão examinados mais detalhadamente no Capítulo 3.)

Apresentamos a programação de *sockets* UDP e TCP por meio de aplicações UDP e TCP simples em Python. Poderíamos escrevê-las em linguagem Java, C ou C++, mas optamos por Python por diversas razões, principalmente porque Python expõe com clareza os conceitos principais de *sockets*. Com Python, há menos linhas de codificação, e cada uma delas pode ser explicada a programadores iniciantes sem muita dificuldade. Mas não precisa ficar assustado se não estiver familiarizado com a linguagem. Você conseguirá acompanhar o código com facilidade se tiver experiência de programação em Java, C ou C++.

Se estiver interessado em programação cliente-servidor em linguagem Java, veja o *site* de apoio que acompanha este livro; lá você poderá achar todos os exemplos desta seção (e laboratórios associados) em Java. Para os interessados em programação cliente-servidor em C, há várias boas referências à disposição (Donahoo, 2001; Stevens, 1997; Frost, 1994); nossos exemplos em Python a seguir possuem um estilo semelhante a C.

### 2.7.1 Programação de *sockets* com UDP

Nesta subseção, vamos escrever programas cliente-servidor simples que usam UDP; na próxima, escreveremos programas semelhantes que usam TCP.

Comentamos na Seção 2.1 que processos que rodam em máquinas diferentes se comunicam entre si enviando mensagens para *sockets*. Dissemos que cada processo é semelhante a uma casa, e que o *socket* do processo é semelhante a uma porta. A aplicação reside em um lado da porta na casa; o protocolo da camada de transporte reside no outro lado da porta, no mundo exterior. O programador da aplicação controla tudo que está no lado da camada de aplicação da porta; contudo, tem pouco controle do lado da camada de transporte.

Agora, vejamos mais de perto a interação entre dois processos que se comunicam, que utilizam *sockets* UDP. Antes que o processo emissor consiga empurrar um pacote de dados pela porta do *socket*, ao usar UDP, ele deve primeiro incluir um endereço de destino no pacote. Depois que o pacote passa pelo *socket* do emissor, a Internet usará esse endereço de destino para rotear o pacote pela Internet até o *socket* no processo receptor. Quando o pacote chega no *socket* receptor, o processo receptor apanha o pacote através do *socket* e depois inspeciona o conteúdo do pacote e toma a ação apropriada.

Assim, você pode agora querer saber: o que há no endereço de destino que é acrescentado ao pacote? Como é de se esperar, o endereço IP do hospedeiro de destino faz parte do endereço de destino. Ao incluir o endereço IP do destino no pacote, os roteadores na Internet poderão rotear o pacote pela Internet até o hospedeiro de destino. Mas como o hospedeiro pode estar rodando muitos processos de aplicação de rede, cada um com um ou mais *sockets*, também é preciso identificar o *socket* em particular no hospedeiro de destino. Quando um *socket* é criado, um identificador, chamado **número de porta**, é designado para ele. Assim, como é de se esperar, o endereço de destino do pacote também inclui o número de porta do *socket*. Resumindo, o processo emissor inclui no pacote um endereço de destino que consiste no endereço IP do hospedeiro de destino e o número de porta do *socket* de destino.

Além do mais, como veremos em breve, o endereço de origem do emissor – consistindo no endereço IP do hospedeiro de origem e no número de porta do *socket* de origem – também é acrescentado ao pacote. Porém, a inclusão do endereço de origem ao pacote normalmente *não* é feita pelo código da aplicação UDP; em vez disso, ela é feita automaticamente pelo sistema operacional.

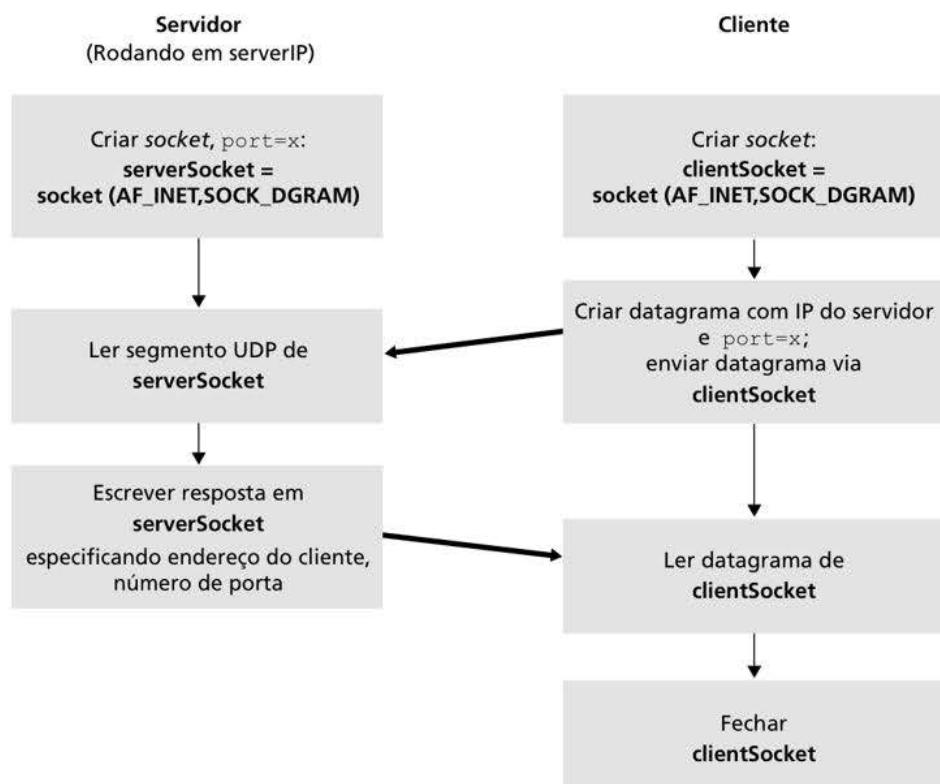
Usaremos a aplicação cliente-servidor simples a seguir para demonstrar a programação de *socket* para UDP e TCP:

1. Um cliente lê uma linha de caracteres (dados) do teclado e a envia para o servidor.
2. O servidor recebe os dados e converte os caracteres para maiúsculas.
3. O servidor envia os dados modificados ao cliente.
4. O cliente recebe os dados modificados e apresenta a linha em sua tela.

A Figura 2.27 destaca a principal atividade relacionada ao *socket* realizada pelo cliente e pelo servidor, que se comunicam por meio de um serviço de transporte.

Agora vamos pôr as mãos na massa e dar uma olhada no par de programas cliente-servidor para uma implementação UDP dessa aplicação de exemplo. Também oferecemos uma análise detalhada, linha a linha, após cada programa. Começamos com um cliente UDP, que enviará uma mensagem simples, em nível de aplicação, ao servidor. Para que o servidor possa receber e responder à mensagem do cliente, ele precisa estar pronto e rodando – ou seja, precisa estar rodando como um processo antes que o cliente envie sua mensagem.

O programa cliente é denominado `UDPClient.py`, e o programa servidor é denominado `UDPServer.py`. Para enfatizar os principais pontos, oferecemos intencionalmente um código que seja mínimo. Um “código bom” certamente teria muito mais linhas auxiliares, particularmente para tratar de casos de erro. Para esta aplicação, escolhemos arbitrariamente 12000 para o número de porta do servidor.



**Figura 2.27** A aplicação cliente-servidor usando UDP.

## UDPClient.py

Aqui está o código para o lado cliente da aplicação:

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),(serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print(modifiedMessage.decode())
clientSocket.close()
```

Agora, vamos examinar as linhas de código em UDPClient.py.

```
from socket import *
```

O módulo `socket` forma a base de todas as comunicações de rede em Python. Incluindo esta linha, poderemos criar *sockets* dentro do nosso programa.

```
serverName = 'hostname'
serverPort = 12000
```

A primeira linha define a variável `serverName` como a cadeia “`hostname`”. Aqui, oferecemos uma cadeia contendo ou o endereço IP do servidor (p. ex., “128.138.32.126”) ou o nome de hospedeiro do servidor (p. ex., “`cis.poly.edu`”). Se usarmos o nome do hospedeiro, então uma pesquisa DNS será automaticamente realizada para obter o endereço IP. A segunda linha define a variável inteira `serverPort` como 12000.

```
clientSocket = socket(AF_INET, SOCK_DGRAM)
```

Esta linha cria o *socket* do cliente, denominado `clientSocket`. O primeiro parâmetro indica a família do endereço; em particular, `AF_INET` indica que a rede subjacente está usando IPv4. (Não se preocupe com isso agora – vamos discutir sobre o IPv4 no Capítulo 4.) O segundo parâmetro indica que o *socket* é do tipo `SOCK_DGRAM`, o que significa que é um *socket* UDP (em vez de um *socket* TCP). Observe que não estamos especificando o número de porta do *socket* cliente quando o criamos; em vez disso, deixamos que o sistema operacional o faça por nós. Agora que a porta do processo cliente já foi criada, queremos criar uma mensagem para enviar pela porta.

```
message = input('Input lowercase sentence:')
```

`input()` é uma função interna da linguagem Python. Quando esse comando é executado, o usuário no cliente recebe o texto “Input lowercase sentence:”. Então, o usuário usa seu teclado para digitar uma linha, que é colocada na variável `message`. Agora que temos um *socket* e uma mensagem, queremos enviar a mensagem pelo *socket* ao hospedeiro de destino.

```
clientSocket.sendto(message.encode(), (serverName, serverPort))
```

Nesta linha, primeiro convertemos a mensagem do tipo cadeia para o tipo `byte`, pois precisamos enviar `bytes` para um *socket*; para tanto, usamos o método `encode()`. O método `sendto()` acrescenta o endereço de destino (`serverName, serverPort`) à mensagem

e envia o pacote resultante pelo *socket* do processo, `clientSocket`. (Como já dissemos, o endereço de origem também é conectado ao pacote, embora isso seja feito automaticamente, e não pelo código.) O envio de uma mensagem do cliente ao servidor por meio de um *socket* UDP é simples assim! Depois de enviar o pacote, o cliente espera receber dados do servidor.

```
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
```

Com esta linha, quando um pacote chega da Internet no *socket* do cliente, os dados são colocados na variável `modifiedMessage`, e o endereço de origem do pacote é colocado na variável `serverAddress`. A variável `serverAddress` contém tanto o endereço IP do servidor quanto o número de porta do servidor. O programa `UDPClient` não precisa realmente dessa informação de endereço do servidor, pois já sabe o endereço do servidor desde o início; mas esta linha de Python oferece o endereço do servidor, apesar disso. O método `recvfrom` também toma o tamanho do *buffer*, 2048, como entrada. (Esse tamanho de *buffer* funciona para quase todos os fins.)

```
print(modifiedMessage.decode())
```

Esta linha imprime `modifiedMessage` na tela do usuário, após converter a mensagem de *bytes* para uma cadeia. Essa deverá ser a linha original que o usuário digitou, mas agora em letras maiúsculas.

```
clientSocket.close()
```

Esta linha fecha o *socket*. O processo, então, é concluído.

### UDPServer.py

Vamos agora dar uma olhada no lado servidor da aplicação:

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(("", serverPort))
print("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

Observe que o início de `UDPServer` é semelhante ao de `UDPClient`. Ele também importa o módulo *socket*, também define a variável inteira `serverPort` como 12000 e cria um *socket* do tipo `SOCK_DGRAM` (um *socket* UDP). A primeira linha de código que é significativamente diferente de `UDPClient` é:

```
serverSocket.bind(("", serverPort))
```

Esta linha vincula (ou seja, designa) o número de porta 12000 ao *socket* do servidor. Assim, em `UDPServer`, o código (escrito pelo programador de aplicação) está designando um número de porta ao *socket*. Dessa forma, quando alguém enviar um pacote à porta 12000 no endereço IP do servidor, ele será direcionado a este *socket*. `UDPServer`, então, entra em um laço `while`; o laço `while` permitirá que `UDPServer` receba e processe pacotes dos clientes indefinidamente. No laço `while`, `UDPServer` espera um pacote chegar.

```
message, clientAddress = serverSocket.recvfrom(2048)
```

Esta linha de código é semelhante à que vimos em UDPClient. Quando um pacote chega no *socket* do servidor, os dados são colocados na variável `message`, e o endereço de origem é colocado na variável `clientAddress`. A variável `clientAddress` contém o endereço IP e o número de porta do cliente. Aqui, UDPServer *usará* essa informação de endereço, pois oferece um endereço de retorno, semelhante ao do remetente no serviço postal comum. Com essa informação, o servidor agora sabe para onde deve direcionar sua resposta.

```
modifiedMessage = message.decode().upper()
```

Esta linha é o núcleo da nossa aplicação simples. Ela apanha a linha enviada pelo cliente e, após converter a mensagem em uma cadeia, usa o método `upper()` para passá-la para letras maiúsculas.

```
serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

Esta última linha anexa o endereço do cliente (endereço IP e número de porta) à mensagem em letras maiúsculas (após converter a cadeia em *bytes*), enviando o pacote resultante ao *socket* do servidor. (Como já dissemos, o endereço do servidor também é anexado ao pacote, embora isso seja feito automaticamente, e não pelo código.) A Internet, então, entregará o pacote ao endereço do cliente. Depois que o servidor envia o pacote, ele permanece no laço `while`, esperando até que outro pacote UDP chegue (de qualquer cliente rodando em qualquer hospedeiro).

Para testar o par de programas, você instala e compila UDPClient.py em um hospedeiro e UDPServer.py em outro. Não se esqueça de incluir o nome de hospedeiro ou endereço IP do servidor em UDPClient.py. Em seguida, você executa UDPServer.py, o programa servidor compilado, no hospedeiro servidor. Isso cria um processo no servidor que fica ocioso até que seja chamado por algum cliente. Depois, você executa UDPClient.py, o programa cliente compilado, no cliente. Isso cria um processo no cliente. Por fim, para usar a aplicação no cliente, você digita uma sentença seguida por um Enter.

Para desenvolver sua própria aplicação cliente-servidor UDP, você pode começar modificando um pouco os programas cliente e servidor. Por exemplo, em vez de converter todas as letras para maiúsculas, o servidor poderia contar o número de vezes que a letra *s* aparece e retornar esse número. Ou então o cliente pode ser modificado para que, depois de receber uma sentença em maiúsculas, o usuário possa continuar a enviar mais sentenças ao servidor.

## 2.7.2 Programação de *sockets* com TCP

Ao contrário do UDP, o TCP é um protocolo orientado à conexão. Isso significa que, antes que cliente e servidor possam começar a enviar dados um para o outro, eles precisam primeiro se apresentar e estabelecer uma conexão TCP. Uma ponta dessa conexão está ligada ao *socket* cliente e a outra está ligada a um *socket* servidor. Ao criar a conexão TCP, associamos a ela o endereço de *socket* (endereço IP e número de porta) do cliente e do servidor. Com a conexão estabelecida, quando um lado quer enviar dados para o outro, basta enviá-los na conexão TCP por meio de seu *socket*. Isso é diferente do UDP, para o qual o servidor precisa anexar um endereço de destino ao pacote, antes de enviá-lo ao *socket*.

Agora, vamos examinar mais de perto a interação dos programas cliente e servidor em TCP. O cliente tem a tarefa de iniciar contato com o servidor. Para que o servidor possa reagir ao contato inicial do cliente, ele tem de estar pronto, o que implica duas coisas. Primeiro, como acontece no UDP, o programa servidor TCP precisa estar rodando como um processo antes de o cliente tentar iniciar contato. Segundo, o programa servidor tem de ter alguma porta especial – mais precisamente, um *socket* especial – que acolha algum contato

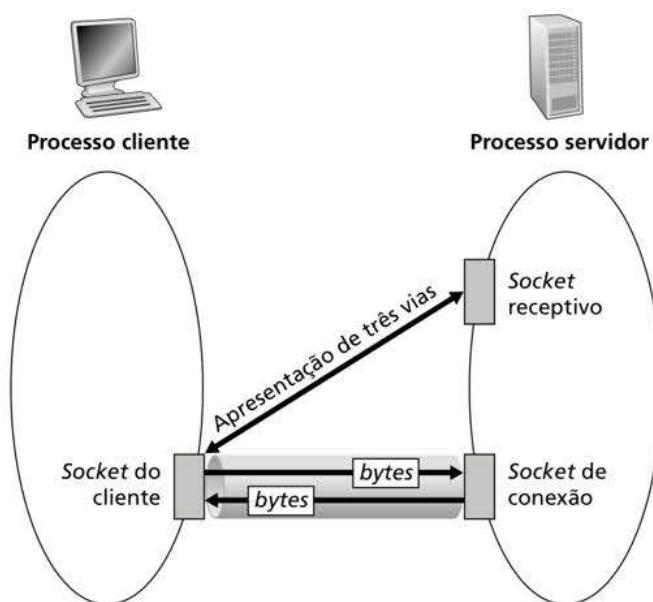
inicial de um processo cliente que esteja rodando em um hospedeiro qualquer. Recorrendo à analogia casa/porta para processo/socket, às vezes nos referiremos ao contato inicial do cliente como “bater à porta”.

Com o processo servidor em execução, o processo cliente pode iniciar uma conexão TCP com o servidor, o que é feito no programa cliente pela criação de um *socket* TCP. Quando cria seu *socket* TCP, o cliente especifica o endereço do *socket* receptivo do servidor, a saber, o endereço IP do hospedeiro servidor e o número de porta do *socket*. Após a criação de seu *socket*, o cliente inicia uma apresentação de três vias e estabelece uma conexão TCP com o servidor. Essa apresentação, que ocorre dentro da camada de transporte, é completamente invisível para os programas cliente e servidor.

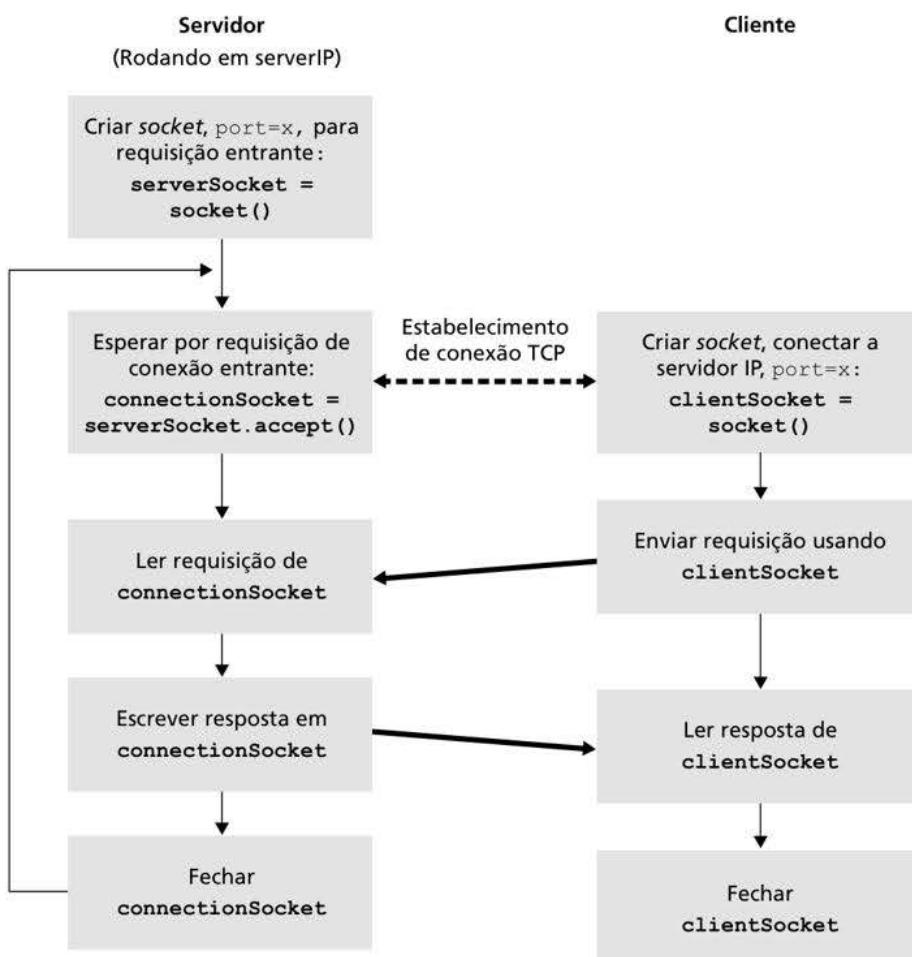
Durante a apresentação de três vias, o processo cliente bate na porta de entrada do processo servidor. Quando o servidor “ouve” a batida, cria uma nova porta (mais precisamente, um *novo socket*) dedicada àquele cliente. No exemplo a seguir, a porta de entrada é um objeto *socket* do TCP que denominamos *serverSocket*; o *socket* recém-criado, dedicado ao cliente que faz a conexão, é denominado *connectionSocket*. Os estudantes que encontram *sockets* TCP pela primeira vez às vezes confundem o *socket* de entrada (que é o ponto de contato inicial para todos os clientes que querem se comunicar com o servidor) com cada *socket* de conexão no lado do servidor, que é criado em seguida para a comunicação com cada cliente.

Do ponto de vista da aplicação, o *socket* do cliente e o de conexão do servidor estão conectados diretamente, como se houvesse uma tubulação entre eles. Como vemos na Figura 2.28, o processo cliente pode enviar *bytes* para seu *socket* de modo arbitrário; o TCP garante que o processo servidor receberá (pelo *socket* de conexão) cada *byte* na ordem em que foram enviados. Assim, o TCP provê um serviço confiável entre os processos cliente e servidor. Além disso, assim como pessoas podem entrar e sair pela mesma porta, o processo cliente não somente envia *bytes* a seu *socket*, mas também os recebe dele; de modo semelhante, o processo servidor não só recebe *bytes* de seu *socket* de conexão, mas também os envia por ele.

Usamos a mesma aplicação cliente-servidor simples para demonstrar programação de *sockets* para TCP: o cliente envia uma linha de dados ao servidor, este converte a linha para letras maiúsculas e a envia de volta ao cliente. A Figura 2.29 destaca a principal atividade relacionada a *socket* do cliente e servidor que se comunicam pelo serviço de transporte TCP.



**Figura 2.28** O processo TCPServer tem dois *sockets*.



**Figura 2.29** A aplicação cliente-servidor usando TCP.

### TCPClient.py

Aqui está o código para o lado cliente da aplicação:

```

from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print('From Server:', modifiedSentence.decode())
clientSocket.close()

```

Vamos agora examinar as várias linhas do código, que diferem ligeiramente da implementação UDP. A primeira linha é a criação do *socket* do cliente.

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

Essa linha cria o *socket* do cliente, denominado *clientSocket*. O primeiro parâmetro novamente indica que a rede subjacente está usando IPv4. O segundo parâmetro indica que

o *socket* é do tipo `SOCK_STREAM`, ou seja, é um *socket* TCP (em vez de um UDP). Observe que, de novo, não estamos especificando o número de porta do *socket* cliente quando o criamos; em vez disso, deixamos que o sistema operacional o faça por nós. Agora, a próxima linha de código é muito diferente do que vimos em `UDPCClient`:

```
clientSocket.connect((serverName, serverPort))
```

Lembre-se de que, antes de um cliente poder enviar dados ao servidor (e vice-versa) usando um *socket* TCP, primeiro deve ser estabelecida uma conexão TCP entre eles, o que é feito por meio dessa linha. O parâmetro do método `connect()` é o endereço do lado servidor da conexão. Depois que essa linha de código é executada, é feita uma apresentação de três vias e uma conexão TCP é estabelecida.

```
sentence = input('Input lowercase sentence:')
```

Assim como em `UDPCClient`, essa linha obtém uma sentença do usuário. A cadeia `sentence` continua a reunir caracteres até que o usuário termine a linha digitando um Enter. A linha de código seguinte também é muito diferente do `UDPCClient`:

```
clientSocket.send(sentence.encode())
```

Essa linha envia a cadeia `sentence` pelo *socket* do cliente e para a conexão TCP. Observe que o programa *não* cria um pacote explicitamente, anexando o endereço de destino ao pacote, como foi feito com os *sockets* UDP. Em vez disso, apenas deixa os *bytes* da cadeia `sentence` na conexão TCP. O cliente, então, espera para receber *bytes* do servidor.

```
modifiedSentence = clientSocket.recv(2048)
```

Quando os caracteres chegam do servidor, eles são colocados na cadeia `modifiedSentence`. Os caracteres continuam a ser acumulados em `modifiedSentence` até que a linha termine com um caractere de Enter. Depois de exibir a sentença em maiúsculas, fechamos o *socket* do cliente:

```
clientSocket.close()
```

Essa última linha fecha o *socket* e, portanto, fecha a conexão TCP entre cliente e servidor. Ela faz o TCP no cliente enviar uma mensagem TCP ao servidor (ver Seção 3.5).

## TCPServer.py

Agora vamos examinar o programa servidor.

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print('The server is ready to receive')
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
    connectionSocket.close()
```

Vejamos agora as linhas que diferem significativamente de UDPServer e TCPClient.

Assim como em TCPClient, o servidor cria um *socket* TCP com:

```
serverSocket=socket(AF_INET,SOCK_STREAM)
```

De modo semelhante a UDPServer, associamos o número de porta do servidor, `serverPort`, ao *socket*:

```
serverSocket.bind(('',serverPort))
```

Porém, com TCP, `serverSocket` será nosso *socket* de entrada. Depois de estabelecer essa porta de entrada, vamos esperar e ficar escutando até que algum cliente bata à porta:

```
serverSocket.listen(1)
```

Essa linha faz com que o servidor escute as requisições de conexão TCP do cliente. O parâmetro especifica o número máximo de conexões em fila (pelo menos 1).

```
connectionSocket, addr = serverSocket.accept()
```

Quando o cliente bate a essa porta, o programa chama o método `accept()` para `serverSocket`, que cria um novo *socket* no servidor, chamado `connectionSocket`, dedicado a esse cliente específico. Cliente e servidor, então, completam a apresentação, criando uma conexão TCP entre o `clientSocket` do cliente e o `connectionSocket` do servidor. Após estabelecer a conexão TCP, cliente e servidor podem enviar *bytes* um para o outro por ela. Com TCP, todos os *bytes* enviados de um lado têm garantias não apenas de que chegarão ao outro lado, mas também na ordem.

```
connectionSocket.close()
```

Nesse programa, depois de enviar a sentença modificada ao cliente, fechamos o *socket* da conexão. Mas como `serverSocket` permanece aberto, outro cliente agora pode bater à porta e enviar uma sentença ao servidor, para que seja modificada.

Isso conclui nossa discussão sobre programação de *sockets* em TCP. Encorajamos o leitor a executar os dois programas em dois hospedeiros separados, e também a modificá-los para realizar objetivos ligeiramente diferentes. Compare o par de programas UDP com o par de programas TCP e repare suas diferenças. Você também deverá realizar as várias tarefas de programação de *sockets* descritas ao final dos Capítulos 2, 4 e 9. Por fim, esperamos que, um dia, depois de dominar estes e outros programas de *sockets* mais avançados, você escreva sua própria aplicação popular para redes, fique rico, famoso e lembre-se dos autores deste livro!

## 2.8 RESUMO

Neste capítulo, estudamos os aspectos conceituais e os aspectos de implementação de aplicações de rede. Conhecemos a onipresente arquitetura cliente-servidor adotada por aplicações da Internet e examinamos sua utilização nos protocolos HTTP, SMTP e DNS. Analisamos esses importantes protocolos de camada de aplicação e suas aplicações associadas (Web, transferência de arquivos, e-mail e DNS) com algum detalhe. Conhecemos também a arquitetura P2P e a contrastamos com a arquitetura cliente-servidor. Também aprendemos sobre o *streaming* de vídeo e como os sistemas modernos de distribuição de vídeo utilizam as CDNs. Vimos como o API *socket* pode ser usado para construir aplicações de

rede. Examinamos a utilização de *sockets* para serviços de transporte fim a fim orientados à conexão (TCP) e não orientados à conexão (UDP). A primeira etapa de nossa jornada de descida pela arquitetura das camadas da rede está concluída!

Logo no começo deste livro, na Seção 1.1, demos uma definição um tanto vaga e despojada de um protocolo. Dissemos que um protocolo é “o formato e a ordem das mensagens trocadas entre duas ou mais entidades comunicantes, bem como as ações realizadas na transmissão e/ou no recebimento de uma mensagem ou outro evento”. O material deste capítulo – em particular, o estudo detalhado dos protocolos HTTP, SMTP e DNS – agregou considerável substância a essa definição. Protocolos são o conceito fundamental de redes. Nossa estudo sobre protocolos de aplicação nos deu agora a oportunidade de desenvolver uma noção mais intuitiva do que eles realmente são.

Na Seção 2.1, descrevemos os modelos de serviço que o TCP e o UDP oferecem às aplicações que os chamam. Nós os examinamos ainda mais de perto quando desenvolvemos, na Seção 2.7, aplicações simples que executam em TCP e UDP. Contudo, pouco dissemos sobre como o TCP e o UDP fornecem esses modelos de serviços. Por exemplo, sabemos que o TCP provê um serviço de dados confiável, mas ainda não mencionamos como ele o faz. No próximo capítulo, examinaremos cuidadosamente não apenas o que são protocolos de transporte, mas também o como e o porquê deles.

Agora que conhecemos a estrutura da aplicação da Internet e os protocolos de camada de aplicação, estamos prontos para continuar a descer a pilha de protocolos e examinar a camada de transporte no Capítulo 3.

---

## Exercícios de fixação e perguntas

---

### Questões de revisão do Capítulo 2

#### SEÇÃO 2.1

- R1. Relacione cinco aplicações da Internet não proprietárias e os protocolos de camada de aplicação que elas usam.
- R2. Qual é a diferença entre arquitetura de rede e arquitetura de aplicação?
- R3. Para uma sessão de comunicação entre um par de processos, qual processo é o cliente e qual é o servidor?
- R4. Em uma aplicação de compartilhamento de arquivos P2P, você concorda com a afirmação: “não existe nenhuma noção de lados cliente e servidor de uma sessão de comunicação”? Justifique sua resposta.
- R5. Qual informação é usada por um processo que está rodando em um hospedeiro para identificar um processo que está rodando em outro hospedeiro?
- R6. Suponha que você queria fazer uma transação de um cliente remoto para um servidor da maneira mais rápida possível. Você usaria o UDP ou o TCP? Por quê?
- R7. Com referência à Figura 2.4, vemos que nenhuma das aplicações relacionadas nela requer “sem perda de dados” e “temporização”. Você consegue imaginar uma aplicação que requeira “sem perda de dados” e seja também altamente sensível ao atraso?
- R8. Relacione quatro classes de serviços que um protocolo de transporte pode prover. Para cada uma, indique se o UDP ou o TCP (ou ambos) fornece tal serviço.
- R9. Lembre-se de que o TCP pode ser aprimorado com o TLS para fornecer serviços de segurança processo a processo, incluindo a codificação. O TLS opera na camada de transporte ou na camada de aplicação? Se o desenvolvedor da aplicação quer que o TCP seja aprimorado com o TLS, o que ele deve fazer?

## SEÇÕES 2.2 a 2.5

- R10. O que significa protocolo de apresentação (*handshaking protocol*)?
- R11. Por que HTTP, SMTP e IMAP rodam sobre TCP e não sobre UDP?
- R12. Considere um *site* de comércio eletrônico que quer manter um registro de compras para cada um de seus clientes. Descreva como isso pode ser feito com *cookies*.
- R13. Descreva como o *cache* Web pode reduzir o atraso na recepção de um objeto requisitado. O *cache* Web reduzirá o atraso para todos os objetos requisitados por um usuário ou somente para alguns objetos? Por quê?
- R14. Digite um comando Telnet em um servidor Web e envie uma mensagem de requisição com várias linhas. Inclua nessa mensagem a linha de cabeçalho `If-modified-since:` para forçar uma mensagem de resposta com a codificação de estado 304 `Not Modified`.
- R15. Liste vários aplicativos populares de mensagens instantâneas. Eles usam os mesmos protocolos que o SMS?
- R16. Suponha que Alice envie uma mensagem a Bob por meio de uma conta de e-mail da Web (como o Hotmail ou Gmail), e que Bob acesse seu e-mail por seu servidor de correio usando IMAP. Descreva como a mensagem vai do hospedeiro de Alice até o hospedeiro de Bob. Não se esqueça de relacionar a série de protocolos de camada de aplicação usados para movimentar a mensagem entre os dois hospedeiros.
- R17. Imprima o cabeçalho de uma mensagem de e-mail que tenha recebido recentemente. Quantas linhas de cabeçalho `Received`: há nela? Analise cada uma.
- R18. O que é o problema de bloqueio HOL no HTTP/1.1? Como o HTTP/2 tenta resolvê-lo?
- R19. É possível que o servidor Web e o servidor de correio de uma organização tenham exatamente o mesmo apelido para um nome de hospedeiro (p. ex., `foo.com`)? Qual seria o tipo de RR que contém o nome de hospedeiro do servidor de correio?
- R20. Examine seus e-mails recebidos e veja o cabeçalho de uma mensagem enviada de um usuário com um endereço de correio eletrônico `.edu`. É possível determinar, pelo cabeçalho, o endereço IP do hospedeiro do qual a mensagem foi enviada? Faça o mesmo para uma mensagem enviada de uma conta do Gmail.

## SEÇÃO 2.5

- R21. No BitTorrent, suponha que Alice forneça blocos para Bob durante um intervalo de 30 segundos. Bob retornará, necessariamente, o favor e fornecerá blocos para Alice no mesmo intervalo? Por quê?
- R22. Considere um novo par, Alice, que entra no BitTorrent sem possuir nenhum bloco. Sem qualquer bloco, ela não pode se tornar uma das quatro melhores exportadoras de dados para qualquer dos outros pares, visto que ela não possui nada para enviar. Então, como Alice obterá seu primeiro bloco?
- R23. O que é uma rede de sobreposição? Ela inclui roteadores? O que são as arestas da rede de sobreposição?

## SEÇÃO 2.6

- R24. CDNs geralmente adotam uma de duas filosofias de posicionamento de servidor diferentes. Relacione-as e descreva-as.
- R25. Além das considerações relacionadas à rede, como atraso, perda e desempenho da largura de banda, existem outros fatores importantes que afetam o projeto de uma estratégia de seleção de CDN. Quais são eles?

## SEÇÃO 2.7

- R26. O servidor UDP descrito na Seção 2.7 precisava de um *socket* apenas, ao passo que o servidor TCP precisava de dois. Por quê? Se um servidor TCP tivesse de suportar  $n$  conexões simultâneas, cada uma de um hospedeiro cliente diferente, de quantos *sockets* precisaria?
- R27. Para a aplicação cliente-servidor por TCP descrita na Seção 2.7, por que o programa servidor deve ser executado antes do programa cliente? Para a aplicação cliente-servidor por UDP, por que o programa cliente pode ser executado antes do programa servidor?

---

## Problemas

---

- P1. Falso ou verdadeiro?
- Um usuário requisita uma página Web que consiste em algum texto e três imagens. Para essa página, o cliente enviará uma mensagem de requisição e receberá quatro mensagens de resposta.
  - Duas páginas Web distintas (p. ex., [www.mit.edu/research.html](http://www.mit.edu/research.html) e [www.mit.edu/students.html](http://www.mit.edu/students.html)) podem ser enviadas pela mesma conexão persistente.
  - Com conexões não persistentes entre navegador e servidor de origem, é possível que um único segmento TCP transporte duas mensagens distintas de requisição HTTP.
  - O cabeçalho Date: na mensagem de resposta HTTP indica a última vez que o objeto da resposta foi modificado.
  - As mensagens de resposta HTTP nunca possuem um corpo de mensagem vazio.
- P2. SMS, iMessage, Wechat e WhatsApp são todos sistemas de mensagem em tempo real para *smartphones*. Após pesquisar um pouco na Internet, escreva, para cada um desses sistemas, um parágrafo sobre os protocolos que utilizam. A seguir, escreva um parágrafo explicando as diferenças entre eles.
- P3. Considere um cliente HTTP que queira obter um documento Web em um dado URL. Inicialmente, o endereço IP do servidor HTTP é desconhecido. Nesse cenário, quais protocolos de transporte e de camada de aplicação são necessários, além do HTTP?
- P4. Considere a seguinte cadeia de caracteres ASCII capturada pelo Wireshark quando o navegador enviou uma mensagem HTTP GET (ou seja, o conteúdo real de uma mensagem HTTP GET). Os caracteres *<cr><lf>* são *carriage return* e *line feed* (ou seja, a cadeia de caracteres em itálico *<cr>* no texto abaixo representa o caractere único *carriage return* que estava contido, naquele momento, no cabeçalho HTTP). Responda às seguintes questões, indicando onde estão as respostas na mensagem HTTP GET a seguir.

```
GET /cs453/index.html HTTP/1.1<cr><lf>Host: gai.a.cs.  
umass.edu<cr><lf>User-Agent: Mozilla/5.0 (Windows; U;  
Windows NT 5.1; en-US; rv:1.7.2) Gec ko/20040804  
Netscape/7.2 (ax) <cr><lf>Accept:ex t/xml, application/  
xml, application/xhtml+xml, text /html;q=0.9, text/  
plain;q=0.8,image/png,*/*;q=0.5 <cr><lf>Accept-Language: en-  
us,en;q=0.5<cr><lf>Accept-Encoding: zip,deflate<cr><lf>Accept-  
Charset: ISO -8859-1,utf-8;q=0.7,*;q=0.7<cr><lf>Keep-Alive:  
300<cr> <lf>Connection:keep-alive<cr><lf><cr><lf>
```

- a. Qual é o URL do documento requisitado pelo navegador?
  - b. Qual versão do HTTP o navegador está rodando?
  - c. O navegador requisita uma conexão não persistente ou persistente?
  - d. Qual é o endereço IP do hospedeiro no qual o navegador está rodando?
  - e. Qual tipo de navegador inicia essa mensagem? Por que é necessário o tipo de navegador em uma mensagem de requisição HTTP?
- P5. O texto a seguir mostra a resposta enviada do servidor em reação à mensagem HTTP GET na questão anterior. Responda às seguintes questões, indicando onde estão as respostas na mensagem.

```
HTTP/1.1 200 OK<cr><lf>Date: Tue, 07 Mar 2008  
12:39:45GMT<cr><lf>Server: Apache/2.0.52 (Fedora) <cr><lf>Last-  
Modified: Sat, 10 Dec2005 18:27:46 GMT<cr><lf>ETag: "526c3-  
f22-a88a4c80"<cr><lf>Accept-Ranges: bytes<cr><lf>Content-  
Length: 3874<cr><lf>Keep-Alive: timeout=max=100<cr><lf>Conn-  
ection: Keep-Alive<cr><lf>Content-Type: text/html; charset=  
ISO-8859-1<cr><lf><cr><lf><!doctype html public "-//w3c//  
dtd html 4.0transitional//en"><lf><html><lf> <head><lf> <meta  
http-equiv="Content-Type" content="text/html; charset=iso-  
8859-1"><lf> <meta name="GENERATOR" content="Mozilla/4.79 [en]  
(Windows NT 5.0; U) Netscape]"><lf> <title>CMPSCI 453 / 591 /  
NTU-ST550ASpring 2005 homepage</title><lf></head><lf>< muito  
mais texto do documento em seguida (não mostrado)>
```

- a. O servidor foi capaz de encontrar o documento com sucesso ou não? A que horas foi apresentada a resposta do documento?
  - b. Quando o documento foi modificado pela última vez?
  - c. Quantos *bytes* existem no documento que está retornando?
  - d. Quais são os 5 primeiros *bytes* do documento que está retornando? O servidor aceitou uma conexão persistente?
- P6. Obtenha a especificação HTTP/1.1 (RFC 2616). Responda às seguintes perguntas:
- a. Explique o mecanismo de sinalização que cliente e servidor utilizam para indicar que uma conexão persistente está sendo fechada. O cliente, o servidor, ou ambos, podem sinalizar o encerramento de uma conexão?
  - b. Quais serviços de criptografia são providos pelo HTTP?
  - c. O cliente é capaz de abrir três ou mais conexões simultâneas com um determinado servidor?
  - d. Um servidor ou um cliente pode abrir uma conexão de transporte entre eles se um dos dois descobrir que a conexão ficou lenta por um tempo. É possível que um lado comece a encerrar a conexão enquanto o outro está transmitindo dados por meio dessa conexão? Explique.
- P7. Suponha que você clique com seu navegador Web sobre um *hyperlink* para obter uma página e que o endereço IP para o URL associado não esteja no *cache* de seu hospedeiro local. Portanto, será necessária uma consulta ao DNS para obter o endereço IP. Considere que  $n$  servidores DNS sejam visitados antes que seu hospedeiro receba o endereço IP do DNS; as visitas sucessivas incorrem em um RTT igual a  $RTT_1, \dots, RTT_n$ . Suponha ainda que a página associada ao *hyperlink* contenha exatamente um objeto que consiste em uma pequena quantidade de texto HTML. Seja  $RTT_0$  o RTT entre o hospedeiro local e o servidor que contém o objeto. Admitindo que o tempo de

- transmissão seja zero, quanto tempo passará desde que o cliente clica no *hyperlink* até que receba o objeto?
- P8. Com referência ao Problema P7, suponha que o arquivo HTML referencie oito objetos muito pequenos no mesmo servidor. Desprezando tempos de transmissão, quanto tempo passa usando-se:
- HTTP não persistente sem conexões TCP paralelas?
  - HTTP não persistente com o navegador configurado para 6 conexões paralelas?
  - HTTP persistente?
- P9. Considere a Figura 2.12, que mostra uma rede institucional conectada à Internet. Suponha que o tamanho médio do objeto seja 1 milhão de *bits* e que a taxa média de requisição dos navegadores da instituição aos servidores de origem seja 16 requisições por segundo. Suponha também que a quantidade de tempo que leva desde o instante em que o roteador do lado da Internet do enlace de acesso transmite uma requisição HTTP até que receba a resposta seja 3 segundos em média (veja Seção 2.2.5). Modele o tempo total médio de resposta como a soma do atraso de acesso médio (i.e., o atraso entre o roteador da Internet e o roteador da instituição) e o tempo médio de atraso da Internet. Para a média de atraso de acesso, use  $\Delta(1 - \Delta\beta)$ , sendo  $\Delta$  o tempo médio requerido para enviar um objeto pelo enlace de acesso e  $\beta$  a taxa de chegada de objetos ao enlace de acesso.
- Determine o tempo total médio de resposta.
  - Agora, considere que um *cache* é instalado na LAN institucional e que a taxa de resposta local seja 0,4. Determine o tempo total de resposta.
- P10. Considere um enlace curto de 10 metros através do qual um remetente pode transmitir a uma taxa de 150 *bits/s* em ambas as direções. Suponha que os pacotes com dados tenham 100 mil *bits* de comprimento, e os pacotes que contêm controle (p. ex., ACK ou apresentação) tenham 200 *bits* de comprimento. Admita que  $N$  conexões paralelas recebam cada  $1/N$  da largura de banda do enlace. Agora, considere o protocolo HTTP e suponha que cada objeto baixado tenha 100 *Kbits* de comprimento e que o objeto inicial baixado contenha 10 objetos referenciados do mesmo remetente. Os *downloads* paralelos por meio de instâncias paralelas de HTTP não persistente fazem sentido nesse caso? Agora considere o HTTP persistente. Você espera ganhos significativos sobre o caso não persistente? Justifique sua resposta.
- P11. Considere o cenário apresentado na questão anterior. Agora suponha que o enlace é compartilhado por Bob e mais quatro usuários. Bob usa instâncias paralelas de HTTP não persistente, e os outros quatro usam HTTP não persistente sem *downloads* paralelos.
- As conexões paralelas de Bob o ajudam a acessar páginas Web mais rapidamente? Por quê? Por que não?
  - Se cinco usuários abrirem cinco instâncias paralelas de HTTP não persistente, então as conexões paralelas de Bob ainda seriam úteis? Por quê? Por que não?
- P12. Escreva um programa TCP simples para um servidor que aceite linhas de entrada de um cliente e envie as linhas para a saída-padrão do servidor. (Você pode fazer isso modificando o programa `TCPServer.py` no texto.) Compile e execute seu programa. Em qualquer outra máquina que contenha um navegador Web, defina o servidor *proxy* no navegador para a máquina que está executando seu programa servidor e também configure o número de porta adequadamente. Seu navegador deverá agora enviar suas mensagens de requisição GET a seu servidor, e este deverá apresentar as mensagens em sua saída-padrão. Use essa plataforma para determinar se seu navegador gera mensagens GET condicionais para objetos que estão em *caches* locais.

- P13. Considere enviar por HTTP/2 uma página Web composta por um clipe de vídeo e cinco imagens. Imagine que o clipe de vídeo é transportado como 2.000 quadros, e que cada imagem tem três quadros.
- Se todos os quadros de vídeo são enviados sem intercalação, quantos “tempos de quadro” são necessários até todas as cinco imagens serem enviadas?
  - Se os quadros são intercalados, quantos tempos de quadro são necessários até todas as cinco imagens serem enviadas?
- P14. Considere a página Web do Problema P13. Agora, emprega-se priorização HTTP/2. Suponha que todas as imagens recebem prioridade em relação ao clipe de vídeo e que a primeira imagem recebe prioridade em relação à segunda, a segunda em relação à terceira e assim por diante. Quantos tempos de quadro serão necessários até a segunda imagem ser enviada?
- P15. Qual é a diferença entre `MAIL FROM:` em SMTP e `From:` na própria mensagem de correio?
- P16. Como o SMTP marca o final de um corpo de mensagem? E o HTTP? O HTTP pode usar o mesmo método que o SMTP para marcar o fim de um corpo de mensagem? Explique.
- P17. Leia o RFC 5321 para SMTP. O que significa MTA? Considere a seguinte mensagem *spam* recebida (modificada de um *spam* verdadeiro). Admitindo que o criador desse *spam* seja malicioso e que todos os outros hospedeiros sejam honestos, identifique o hospedeiro malicioso que criou essa mensagem *spam*.

```
From - Fri Nov 07 13:41:30 2008
Return-Path: <tennis5@pp33head.com>
Received: from barmail.cs.umass.edu (barmail.cs.umass.edu
[128.119.240.3]) by cs.umass.edu (8.13.1/8.12.6) for <hg@cs.umass.
edu>; Fri, 7 Nov 2008 13:27:10 -0500
Received: from asusus-4b96 (localhost [127.0.0.1]) by barmail.
cs.umass.edu (Spam Firewall) for <hg@cs.umass.edu>; Fri, 7
Nov 2008 13:27:07 -0500 (EST)
Received: from asusus-4b96 ([58.88.21.177]) by barmail.cs.umass.
edu for <hg@cs.umass.edu>; Fri, 07 Nov 2008 13:27:07 -0500 (EST)
Received: from [58.88.21.177] by inbnd55.exchangeddd.com; Sat, 8
Nov 2008 01:27:07 +0700
From: "Jonny" <tennis5@pp33head.com>
To: <hg@cs.umass.edu>

Subject: How to secure your savings.
```

- P18. a. O que é um banco de dados *whois*?
- Use vários bancos de dados *whois* da Internet para obter os nomes de dois servidores DNS. Cite quais bancos de dados *whois* você utilizou.
  - Use nslookup em seu hospedeiro local para enviar consultas DNS a três servidores DNS: seu servidor DNS local e os dois servidores DNS que encontrou na alternativa (b). Tente consultar registros dos tipos A, NS e MX. Faça um resumo do que encontrou.
  - Use nslookup para encontrar um servidor Web que tenha vários endereços IP. O servidor de sua instituição (escola ou empresa) tem vários endereços IP?
  - Use o banco de dados *whois* ARIN para determinar a faixa de endereços IP usados por sua universidade.
  - Descreva como um invasor pode usar bancos de dados *whois* e a ferramenta nslookup para fazer o reconhecimento de uma instituição antes de lançar um ataque.
  - Discuta por que bancos de dados *whois* devem estar disponíveis publicamente.

- P19. Neste problema, utilizamos a ferramenta funcional *dig* disponível em hospedeiros Unix e Linux para explorar a hierarquia dos servidores DNS. Lembre-se de que, na Figura 2.19, um servidor DNS de nível superior na hierarquia do DNS delega uma consulta DNS para um servidor DNS de nível inferior na hierarquia, enviando de volta ao cliente DNS o nome daquele servidor DNS de nível inferior. Primeiro, leia a *man page* sobre a ferramenta *dig* e responda às seguintes questões:
- Iniciando com o servidor DNS raiz (de um dos servidores raiz [a-m].root-servers.net), construa uma sequência de consultas para o endereço IP para seu servidor de departamento utilizando o *dig*. Mostre a relação de nomes de servidores DNS na cadeia de delegação ao responder à sua consulta.
  - Repita a alternativa (a) com vários *sites* da Internet populares, como google.com, yahoo.com ou amazon.com.
- P20. Suponha que você consiga acessar os *caches* nos servidores DNS locais do seu departamento. Você é capaz de propor uma maneira de determinar, em linhas gerais, os servidores (fora de seu departamento) que são mais populares entre os usuários do seu departamento? Explique.
- P21. Suponha que seu departamento possua um servidor DNS local para todos os computadores do departamento. Você é um usuário comum (ou seja, não é um administrador de rede/sistema). Você consegue encontrar um modo de determinar se um *site* da Internet externo foi muito provavelmente acessado de um computador do seu departamento alguns segundos atrás? Explique.
- P22. Considere um arquivo de distribuição de  $F = 20 \text{ Gbits}$  para  $N$  pares. O servidor possui uma taxa de *upload* de  $u_s = 30 \text{ Mbits/s}$  e cada par possui uma taxa de *download* de  $d_i = 2 \text{ Mbits/s}$  e uma taxa de *upload* de  $u$ . Para  $N = 10, 100$  e  $1.000$  e  $u = 300 \text{ Kbits/s}, 700 \text{ Kbits/s}$  e  $2 \text{ Mbits/s}$ , prepare um gráfico apresentando o tempo mínimo de distribuição para cada uma das combinações de  $N$  e  $u$  para o modo cliente-servidor e para o modo distribuição P2P.
- P23. Considere distribuir um arquivo de  $F$  bits para  $N$  pares utilizando uma arquitetura cliente-servidor. Admita um modelo fluido no qual o servidor pode transmitir de modo simultâneo para diversos pares, a diferentes taxas, desde que a taxa combinada não ultrapasse  $u_s$ .
  - Suponha que  $u_s/N \leq d_{\min}$ . Especifique um esquema de distribuição que possua o tempo de distribuição de  $NF/u_s$ .
  - Suponha que  $u_s/N \geq d_{\min}$ . Especifique um esquema de distribuição que possua o tempo de distribuição de  $F/d_{\min}$ .
  - Conclua que o tempo mínimo de distribuição é, geralmente, dado por máx  $\{NF/u_s, F/d_{\min}\}$ .
- P24. Considere distribuir um arquivo de  $F$  bits para  $N$  pares utilizando uma arquitetura P2P. Admita um modelo fluido e presuma que  $d_{\min}$  é muito grande, de modo que a largura de banda do *download* do par nunca é um gargalo.
  - Suponha que  $u_s \dots (u_s + u_1 + \dots + u_N)/N$ . Especifique um esquema de distribuição que possua o tempo de distribuição de  $F/u_s$ .
  - Suponha que  $u_s \geq (u_s + u_1 + \dots + u_N)/N$ . Especifique um esquema de distribuição que possua o tempo de distribuição de  $NF/(u_s + u_1 + \dots + u_N)$ .
  - Conclua que o tempo mínimo de distribuição é, em geral, dado por máx  $\{F/u_s, NF/(u_s + u_1 + \dots + u_N)\}$ .
- P25. Considere uma rede de sobreposição com  $N$  pares ativos, em que cada dupla de pares possua uma conexão TCP. Além disso, suponha que as conexões TCP passem por um total de  $M$  roteadores. Quantos nós e arestas há na rede de sobreposição correspondente?

- P26. Suponha que Bob tenha entrado no BitTorrent, mas ele não quer fazer o *upload* de nenhum dado para qualquer outro par (denominado carona).
- Bob alega que consegue receber uma cópia completa do arquivo compartilhado pelo grupo. A alegação de Bob é possível? Por quê?
  - Bob alega ainda que ele pode “pegar carona” de um modo mais eficiente usando um conjunto de diversos computadores (com endereços IP distintos) no laboratório de informática de seu departamento. Como ele pode fazer isso?
- P27. Considere um sistema DASH para o qual existem  $N$  versões de vídeo (em  $N$  diferentes taxas e qualidades) e  $N$  versões de áudio (em  $N$  taxas e versões diferentes). Suponha que queiramos permitir que o dispositivo de reprodução escolha, a qualquer momento, qualquer uma das  $N$  versões de vídeo e qualquer uma das  $N$  versões de áudio.
- Se criarmos arquivos de modo que o áudio seja misturado com o vídeo, de modo que o servidor envie somente um fluxo de mídia em determinado momento, quantos arquivos o servidor precisará armazenar (cada um com um URL diferente)?
  - Se o servidor, em vez disso, envia os fluxos de áudio e vídeo separadamente e o cliente sincroniza os fluxos, quantos arquivos o servidor precisa armazenar?
- P28. Instale e compile os programas Python TCPClient e UDPClient em um hospedeiro e TCPServer e UDPServer em outro.
- Suponha que você execute TCPClient\* antes de executar TCPServer. O que acontece? Por quê?
  - Imagine que você execute UDPClient antes de UDPServer. O que acontece? Por quê?
  - O que acontece se você usar números de porta diferentes para os lados cliente e servidor?
- P29. Suponha que, em UDPClient.py, depois de criarmos o *socket*, acrescentemos a linha:
- ```
clientSocket.bind(('', 5432))
```
- Será necessário mudar UDPServer.py? Quais são os números de porta para os *sockets* em UDPClient e UDPServer? Quais eram esses números antes dessa mudança?
- P30. Você consegue configurar seu navegador para abrir várias conexões simultâneas com um *site*? Quais são as vantagens e desvantagens de ter um grande número de conexões TCP simultâneas?
- P31. Vimos que os *sockets* TCP da Internet tratam os dados sendo enviados como um fluxo de *bytes*, mas *sockets* UDP reconhecem limites de mensagem. Quais são uma vantagem e uma desvantagem da API orientada a *byte* em relação a fazer com que a API reconheça e preserve explicitamente os limites de mensagem definidos pela aplicação?
- P32. O que é o servidor Web Apache? Quanto ele custa? Qual funcionalidade ele possui atualmente? Você pode consultar a Wikipédia para responder a essa pergunta.

## Tarefas de programação de *sockets*

O *site* de apoio deste livro inclui seis tarefas de programação de *sockets*. As quatro primeiras são resumidas a seguir. A quinta utiliza o protocolo ICMP e está resumida ao final do Capítulo 5. Recomenda-se que os alunos completem várias, se não todas, dessas tarefas. Detalhes completos dessas tarefas, bem como trechos importantes do código Python, estão disponíveis para os alunos no *site* <http://www.pearsonhighered.com/cs-resources>.

\*N. de T.: Usar máquinas virtuais é uma boa dica para realizar este exercício com um único computador.

## Tarefa 1: Servidor Web

Nesta tarefa, você desenvolverá um servidor Web simples em Python, capaz de processar apenas uma requisição. Seu servidor Web (i) criará um *socket* de conexão quando contatado por um cliente (navegador); (ii) receberá a requisição HTTP dessa conexão; (iii) analisará a requisição para determinar o arquivo específico sendo requisitado; (iv) obterá o arquivo requisitado do sistema de arquivo do servidor; (v) criará uma mensagem de resposta HTTP consistindo no arquivo requisitado precedido por linhas de cabeçalho; e (vi) enviará a resposta pela conexão TCP ao navegador requisitante. Se um navegador requisitar um arquivo que não está presente no seu servidor, seu servidor deverá retornar uma mensagem de erro “404 Not Found”.

No *site* de apoio, oferecemos o código estrutural para o seu servidor. Sua tarefa é concluir o código, rodar seu servidor e depois testá-lo enviando requisições de navegadores rodando em hospedeiros diferentes. Se você rodar seu servidor em um hospedeiro que já tem um servidor Web rodando nele, então deverá usar uma porta diferente da porta 80 para o seu servidor.

## Tarefa 2: UDP Pinger

Nesta tarefa de programação, você escreverá um programa *ping* do cliente em Python. Seu cliente enviará uma mensagem *ping* simples a um servidor, receberá uma mensagem *pong* correspondente de volta do servidor e determinará o atraso entre o momento em que o cliente enviou a mensagem *ping* e recebeu a mensagem *pong*. Esse atraso é denominado tempo de viagem de ida e volta (RTT). A funcionalidade oferecida pelo cliente e servidor é semelhante à fornecida pelo programa *ping* padrão, disponível nos sistemas operacionais modernos. Porém, os programas *ping* padrão usam o Internet Control Message Protocol (ICMP) (que veremos no Capítulo 5). Aqui, criaremos um programa *ping* baseado em UDP, fora do padrão (porém simples!).

Seu programa *ping* deverá enviar 10 mensagens *ping* ao servidor de destino por meio de UDP. Para cada mensagem, seu cliente deverá determinar e imprimir o RTT quando a mensagem *pong* correspondente for retornada. Como o UDP é um protocolo não confiável, um pacote enviado pelo cliente ou servidor poderá ser perdido. Por esse motivo, o cliente não poderá esperar indefinidamente por uma resposta a uma mensagem *ping*. Você deverá fazer o cliente esperar até 1 segundo por uma resposta do servidor; se nenhuma resposta for recebida, o cliente deverá considerar que o pacote foi perdido e imprimir uma mensagem de acordo.

Nesta tarefa, você receberá o código completo para o servidor (disponível no *site* de apoio). Sua tarefa é escrever o código cliente, que será semelhante ao código do servidor. Recomendamos que, primeiro, você estude cuidadosamente o código do servidor. Depois, poderá escrever seu código cliente, cortando e colando à vontade as linhas do código do servidor.

## Tarefa 3: Cliente de correio

O objetivo desta tarefa de programação é criar um cliente de correio simples, que envia e-mail a qualquer destinatário. Seu cliente precisará estabelecer uma conexão TCP com um servidor de correio (p. ex., um servidor de correio da Google), dialogar com esse servidor usando o protocolo SMTP, enviar uma mensagem de correio a um destinatário (p. ex., seu amigo) pelo servidor de correio e, por fim, fechar a conexão TCP com o servidor de correio.

Para esta tarefa, o *site* de apoio oferece o código estrutural para o seu cliente. Sua tarefa é completar o código e testar seu cliente, enviando e-mail para contas de usuário diferentes. Você também pode tentar enviar por diferentes servidores (p. ex., por um servidor de correio da Google e pelo servidor de correio da sua universidade).

## Tarefa 4: Servidor proxy Web

Nesta tarefa, você desenvolverá um *proxy* da Web. Quando seu *proxy* receber de um navegador uma requisição HTTP para um objeto, ele gerará uma nova requisição HTTP para o mesmo objeto e a enviará para o servidor de origem. Quando o *proxy* receber a resposta HTTP correspondente com o objeto do servidor de origem, ele criará uma nova resposta HTTP, incluindo o objeto, e a enviará ao cliente.

Para esta tarefa, o *site* de apoio oferece o código estrutural para o servidor *proxy*. Seu trabalho é completar o código e depois testá-lo fazendo diferentes navegadores requisitarem objetos Web por meio do seu *proxy*.

---

## Wireshark Lab: HTTP

Depois de ter experimentado com o analisador de pacotes Wireshark no Laboratório 1, agora estamos prontos para usar o Wireshark para investigar protocolos em operação. Neste laboratório, vamos explorar diversos aspectos do protocolo HTTP: a interação básica GET/resposta, formatos de mensagem HTTP, recuperação de grandes arquivos HTML, recuperação de arquivos HTML com URLs embutidos, conexões persistentes e não persistentes, e autenticação e segurança HTTP.

Como acontece com todos os laboratórios Wireshark, o desenvolvimento completo está disponível no *site* de apoio do livro.

---

## Wireshark Lab: DNS

Neste laboratório, examinamos mais de perto o lado cliente do DNS, o protocolo que traduz nomes de hospedeiro da Internet em endereços IP. Lembre-se de que, na Seção 2.5, vimos que o papel do cliente no DNS é bastante simples – um cliente envia uma consulta ao seu servidor DNS local e recebe uma resposta de volta. Muita coisa pode acontecer, invisível aos clientes DNS, à medida que servidores DNS hierárquicos se comunicam entre si para resolver, de forma recursiva ou iterativa, a consulta DNS do cliente. Porém, do ponto de vista do cliente DNS, o protocolo é bem simples – uma consulta é formulada ao servidor DNS local e uma resposta é recebida desse servidor. Observamos o DNS em ação neste laboratório.

Como acontece com todos os laboratórios Wireshark, o desenvolvimento completo está disponível no *site* de apoio do livro.

## ENTREVISTA

### Tim Berners-Lee

Sir Tim Berners-Lee é conhecido como o inventor da World Wide Web. Em 1989, quando trabalhava como *fellow* no CERN, propôs um sistema de distribuição de gestão da informação baseado na Internet, incluindo a versão original do protocolo HTTP. No mesmo ano, conseguiu implementar o seu projeto em um cliente e servidor. Berners-Lee recebeu o Prêmio Turing de 2016 por “inventar a World Wide Web, o primeiro navegador Web e os protocolos e algoritmos fundamentais que permitem que a Web cresça em escala”. Ele é cofundador da World Wide Web Foundation e atualmente ocupa o cargo de *Fellow Professor de Ciência da Computação* na Universidade de Oxford e de professor na CSAIL do MIT.



Imagem cortesia de Tim Berners-Lee

#### Originalmente, você estudou física. No que as redes se assemelham à física?

Quando estuda física, você imagina que regras de comportamento, em uma escala muito pequena, poderiam dar origem ao mundo em larga escala que vemos. Quando projeta um sistema global com a Web, você tenta inventar regras de comportamento de páginas Web e *links* e coisas que poderiam, no total, criar o mundo em larga escala como gostaríamos de vê-lo. Um é análise, o outro é síntese, mas os dois são bastante parecidos.

#### O que o influenciou a se especializar em redes?

Após me formar em física, as empresas de pesquisa em telecomunicação pareciam ser os lugares mais interessantes. O microprocessador acabara de ser lançado e as telecomunicações estavam passando muito rapidamente da lógica por circuitos fixos para os sistemas baseados em microprocessadores. Era emocionante.

#### Qual parte de seu trabalho lhe apresenta mais desafios?

Quando dois grupos discordam fortemente sobre alguma coisa, mas querem atingir um objetivo em comum, descobrir exatamente o que cada um quer dizer e quais são os mal-entendidos pode ser bastante difícil, como sabe o diretor de qualquer grupo de trabalho. Contudo, é isso que se exige para progredirmos em direção ao consenso em larga escala.

#### Quais pessoas o inspiraram profissionalmente?

Meus pais, que participaram dos primeiros momentos da computação, me deixaram fascinados pelo assunto como um todo. Mike Sendall e Peggy Rimmer, para quem trabalhei diversas vezes no CERN, estão entre aqueles que me ensinaram e me incentivaram. Mais tarde, aprendi a admirar aqueles que, como Vanevar Bush, Doug Englebart e Ted Nelson, haviam tido sonhos semelhantes na sua época, mas não tiveram a vantagem da existência dos PCs e da Internet para poder transformá-los em realidade.