

# Camada de transporte

Posicionada entre as camadas de aplicação e de rede, a camada de transporte é uma peça central da arquitetura de rede em camadas. Ela desempenha o papel fundamental de fornecer serviços de comunicação diretamente aos processos de aplicação que rodam em hospedeiros diferentes. A abordagem pedagógica que adotamos neste capítulo é alternar entre discussões de princípios de camada de transporte e o modo como tais princípios são colocados em prática em protocolos existentes; como de costume, daremos particular ênfase aos protocolos da Internet, em especial aos de camada de transporte: o Protocolo de Controle de Transmissão (TCP, do inglês *Transmission Control Protocol*) e o Protocolo de Datagrama de Usuário (UDP, do inglês *User Datagram Protocol*).

Começaremos discutindo a relação entre as camadas de transporte e de rede, preparando o cenário para o exame de sua primeira função importante – ampliar o serviço de entrega da camada de rede entre dois sistemas finais para um serviço de entrega entre dois processos da camada de aplicação que rodam nos sistemas finais. Ilustraremos essa função quando abordarmos o UDP, o protocolo de transporte não orientado para conexão da Internet.

Depois, retornaremos aos princípios e trataremos de um dos problemas mais fundamentais de redes de computadores – como duas entidades podem se comunicar de maneira confiável por um meio que pode perder e corromper dados. Mediante uma série de cenários cada vez mais complicados (e realistas!), construiremos um conjunto de técnicas que os protocolos de transporte utilizam para resolver esse problema. Então, mostraremos como esses princípios estão incorporados no TCP, o protocolo de transporte orientado para conexão da Internet.

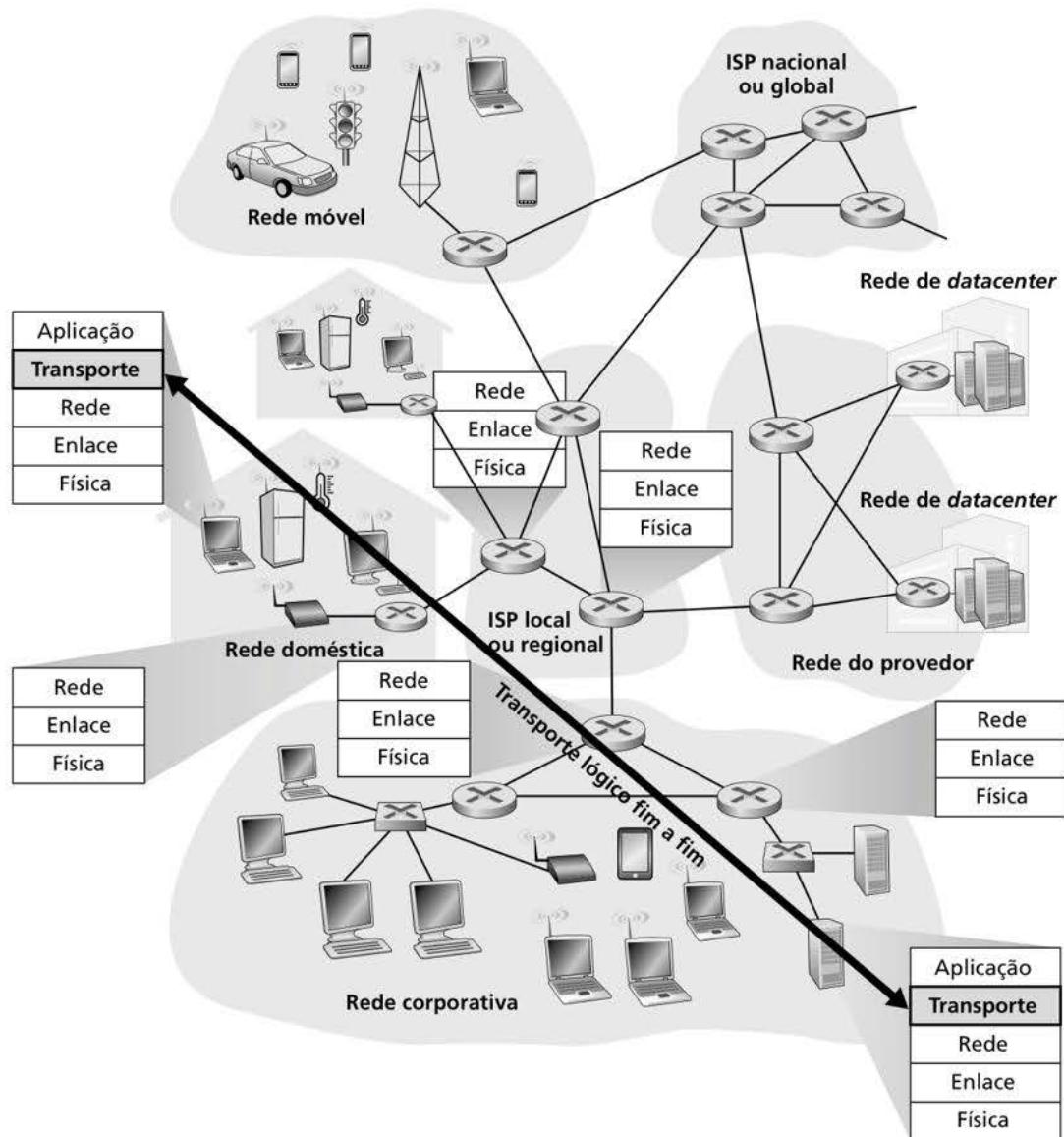
Em seguida, passaremos para um segundo problema fundamentalmente importante em redes – o controle da taxa de transmissão de entidades de camada de transporte para evitar ou se recuperar de congestionamentos dentro da rede. Consideraremos as causas e consequências do congestionamento, bem como técnicas de controle de congestionamento comumente usadas. Após adquirir um sólido conhecimento das questões que estão por trás do controle de congestionamento, estudaremos seu tratamento pelo TCP.

### 3.1 INTRODUÇÃO E SERVIÇOS DE CAMADA DE TRANSPORTE

Nos dois capítulos anteriores, citamos o papel da camada de transporte e os serviços que ela fornece. Vamos revisar rapidamente o que já aprendemos sobre a camada de transporte.

Um protocolo da camada de transporte fornece **comunicação lógica** entre processos de aplicação que rodam em hospedeiros diferentes. *Comunicação lógica* nesse contexto significa que, do ponto de vista de uma aplicação, tudo se passa como se os hospedeiros que rodam os processos estivessem conectados diretamente; na verdade, eles poderão estar em lados opostos do planeta, conectados por diversos roteadores e uma ampla variedade de tipos de enlace. Processos de aplicação usam a comunicação lógica fornecida pela camada de transporte para enviar mensagens entre si, livres da preocupação dos detalhes da infraestrutura física utilizada para transportá-las. A Figura 3.1 ilustra a noção de comunicação lógica.

Como vemos na Figura 3.1, protocolos da camada de transporte são implementados nos sistemas finais, mas não em roteadores de rede. No lado remetente, a camada de transporte



**Figura 3.1** A camada de transporte fornece comunicação lógica, e não física, entre processos de aplicações.

converte as mensagens que recebe de um processo de aplicação remetente em pacotes de camada de transporte, denominados **segmentos** de camada de transporte na terminologia da Internet. Isso é (possivelmente) feito fragmentando-se as mensagens da aplicação em pedaços menores e adicionando-se um cabeçalho de camada de transporte a cada pedaço para criar o segmento de camada de transporte. Essa camada, então, passa o segmento para a rede no sistema final remetente, onde ele é encapsulado em um pacote de camada de rede (um datagrama) e enviado ao destinatário. É importante notar que roteadores de rede agem somente nos campos de camada de rede do datagrama; isto é, não examinam os campos do segmento de camada de transporte encapsulado com o datagrama. No lado destinatário, a camada de rede extrai do datagrama o segmento de camada de transporte e passa-o para a camada de transporte, que, em seguida, processa o segmento recebido, disponibilizando os dados para a aplicação destinatária.

Vários protocolos de camada de transporte poderão estar disponíveis às aplicações de rede. Por exemplo, a Internet possui dois protocolos – TCP e UDP. Cada um oferece um conjunto diferente de serviços de camada de transporte à aplicação chamadora.

### 3.1.1 Relação entre as camadas de transporte e de rede

Lembre-se de que a camada de transporte se situa logo acima da camada de rede na pilha de protocolos. Enquanto um protocolo de camada de transporte fornece comunicação lógica entre *processos* que rodam em hospedeiros diferentes, um protocolo de camada de rede fornece comunicação lógica entre *hospedeiros*. A distinção é sutil, mas importante. Vamos examiná-la com o auxílio de uma analogia com moradias.

Considere duas casas, uma na Costa Leste e outra na Costa Oeste dos Estados Unidos, cada qual com uma dúzia de crianças. As crianças da Costa Leste são primas das crianças da Costa Oeste, e todas adoram escrever cartas umas para as outras – cada criança escreve a cada primo uma vez por semana, e cada carta é entregue pelo serviço de correio tradicional dentro de um envelope separado. Assim, uma casa envia 144 cartas por semana para a outra. (Essas crianças economizariam muito dinheiro se tivessem *e-mail!*) Em cada moradia há uma criança responsável pela coleta e distribuição da correspondência – Ann, na casa da Costa Oeste, e Bill, na da Costa Leste. Toda semana, Ann coleta a correspondência de seus irmãos e irmãs e a coloca no correio. Quando as cartas chegam à casa da Costa Oeste, também é Ann quem tem a tarefa de distribuir a correspondência, trazida pelo carteiro, a seus irmãos e irmãs. Bill realiza o mesmo trabalho na casa da Costa Leste.

Nesse exemplo, o serviço postal oferece uma comunicação lógica entre as duas casas – ele movimenta a correspondência de uma residência para outra, e não de uma pessoa para outra. Por outro lado, Ann e Bill oferecem comunicação lógica entre os primos – eles coletam e entregam a correspondência de seus irmãos e irmãs. Note que, do ponto de vista dos primos, Ann e Bill *são* o serviço postal, embora sejam apenas uma parte (a parte do sistema final) do processo de entrega fim a fim. Esse exemplo das moradias é uma analogia interessante para explicar como a camada de transporte se relaciona com a camada de rede:

mensagens de aplicação = cartas em envelopes

processos = primos

hospedeiros (também denominados sistemas finais) = casas

protocolo de camada de transporte = Ann e Bill

protocolo de camada de rede = serviço postal (incluindo os carteiros)

Continuando com essa analogia, observe que Ann e Bill fazem todo o trabalho dentro de suas respectivas casas; eles não estão envolvidos, por exemplo, com a classificação da correspondência em nenhuma central intermediária dos correios ou com o transporte de uma central a outra. De maneira semelhante, protocolos de camada de transporte moram nos sistemas finais, onde movimentam mensagens de processos de aplicação para a borda da rede (i.e., para a camada de rede) e vice-versa, mas não interferem no modo como as mensagens são movimentadas dentro do núcleo. Na verdade, como ilustrado na Figura 3.1, roteadores

intermediários não atuam sobre (nem reconhecem) qualquer informação que a camada de transporte possa ter anexado às mensagens da aplicação.

Prosseguindo com nossa saga familiar, suponha agora que, quando Ann e Bill saem de férias, outro par de primos – digamos, Susan e Harvey – substitua-os e encarregue-se da coleta interna da correspondência e de sua entrega. Infelizmente para as duas famílias, eles não desempenham essa tarefa do mesmo modo que Ann e Bill. Por serem crianças mais novas, Susan e Harvey recolhem e entregam a correspondência com menos frequência e, às vezes, perdem cartas (que acabam mastigadas pelo cão da família). Assim, o par de primos Susan e Harvey não oferece o mesmo conjunto de serviços (i.e., o mesmo modelo de serviço) proporcionado por Ann e Bill. De modo similar, uma rede de computadores pode oferecer vários protocolos de transporte, em que cada um oferece um modelo de serviço diferente às aplicações.

Os serviços que Ann e Bill podem fornecer são claramente limitados pelos possíveis serviços que os correios fornecem. Por exemplo, se o serviço postal não estipula um prazo máximo para entregar a correspondência entre as duas casas (p. ex., três dias), então não há nenhuma possibilidade de Ann e Bill definirem um atraso máximo para a entrega da correspondência entre qualquer par de primos. De maneira semelhante, os serviços que um protocolo de transporte pode fornecer são muitas vezes limitados pelo modelo de serviço do protocolo subjacente da camada de rede. Se o protocolo de camada de rede não puder dar garantias contra atraso ou garantias de largura de banda para segmentos de camada de transporte enviados entre hospedeiros, então o protocolo de camada de transporte não poderá dar essas mesmas garantias para mensagens de aplicação enviadas entre processos.

No entanto, certos serviços *podem* ser oferecidos por um protocolo de transporte mesmo quando o protocolo de rede subjacente não oferece o serviço correspondente na camada de rede. Por exemplo, como veremos neste capítulo, um protocolo de transporte pode oferecer serviço confiável de transferência de dados a uma aplicação mesmo quando o protocolo subjacente da rede não é confiável, isto é, mesmo quando o protocolo de rede perde, embaralha ou duplica pacotes. Como outro exemplo (que exploraremos no Capítulo 8, quando discutirmos segurança de rede), um protocolo de transporte pode usar criptografia para garantir que as mensagens da aplicação não sejam lidas por intrusos mesmo quando a camada de rede não puder garantir o sigilo de segmentos de camada de transporte.

### 3.1.2 Visão geral da camada de transporte na Internet

Lembre-se de que a Internet disponibiliza dois protocolos de transporte distintos para a camada de aplicação. Um deles é o **UDP**, que oferece à aplicação solicitante um serviço não confiável, não orientado para conexão. O segundo é o **TCP**, que oferece à aplicação solicitante um serviço confiável, orientado para conexão. Ao projetar uma aplicação de rede, o criador da aplicação deve especificar um desses dois protocolos de transporte. Como vimos na Seção 2.7, o desenvolvedor da aplicação escolhe entre o UDP e o TCP ao criar *sockets*.

Para simplificar a terminologia, chamaremos o pacote de camada de transporte de *segmento*. Devemos mencionar, contudo, que a literatura da Internet (p. ex., os RFCs) também se refere ao pacote de camada de transporte do TCP como um segmento, mas muitas vezes se refere ao pacote do UDP como um datagrama. Porém, a mesma literatura também usa o termo *datagrama* para o pacote de camada de rede! Como este é um livro de introdução a redes de computadores, acreditamos que será menos confuso se nos referirmos a ambos os pacotes TCP e UDP como segmentos; reservaremos o termo *datagrama* para o pacote de camada de rede.

Antes de continuarmos com nossa breve apresentação do UDP e do TCP, é útil dizer algumas palavras sobre a camada de rede da Internet. (A camada de rede é examinada detalhadamente nos Capítulos 4 e 5.) O protocolo de camada de rede da Internet tem um nome IP, que quer dizer Internet Protocol. O IP oferece comunicação lógica entre hospedeiros. O modelo de serviço do IP é um **serviço de entrega de melhor esforço**, o que significa que

o IP faz o “melhor esforço” para levar segmentos entre hospedeiros comunicantes, *mas não dá nenhuma garantia*. Em especial, o IP não garante a entrega de segmentos, a entrega ordenada de segmentos e tampouco a integridade dos dados nos segmentos. Por essas razões, ele é denominado um **serviço não confiável**. Mencionamos também neste livro que cada hospedeiro tem, no mínimo, um endereço de camada de rede, denominado endereço IP. Examinaremos endereçamento IP em detalhes no Capítulo 4. Para este capítulo, precisamos apenas ter em mente que *cada hospedeiro tem um endereço IP*.

Agora que abordamos de modo breve o modelo de serviço IP, vamos resumir os modelos de serviço providos por UDP e TCP. A responsabilidade fundamental do UDP e do TCP é ampliar o serviço de entrega IP entre dois sistemas finais para um serviço de entrega entre dois processos que rodam nos sistemas finais. A ampliação da entrega hospedeiro a hospedeiro para entrega processo a processo é denominada **multiplexação/demultiplexação de camada de transporte**. Discutiremos esse assunto na próxima seção. O UDP e o TCP também fornecem verificação de integridade ao incluir campos de detecção de erros nos cabeçalhos de seus segmentos. Esses dois serviços mínimos de camada de transporte – entrega de dados processo a processo e verificação de erros – são os únicos que o UDP fornece! Em especial, como o IP, o UDP é um serviço não confiável – ele não garante que os dados enviados por um processo cheguem (quando chegam!) intactos ao processo destinatário. O UDP será discutido detalhadamente na Seção 3.3.

O TCP, por outro lado, oferece vários serviços adicionais às aplicações. Primeiro, e mais importante, ele oferece **transferência confiável de dados**. Usando controle de fluxo, números de sequência, reconhecimentos e temporizadores (técnicas que exploraremos em pormenores neste capítulo), o protocolo assegura que os dados sejam entregues do processo remetente ao processo destinatário corretamente e em ordem. Assim, o TCP converte o serviço não confiável do IP entre sistemas finais em um serviço confiável de transporte de dados entre processos. Ele também oferece **controle de congestionamento**, que não é tanto um serviço fornecido à aplicação solicitante, e sim mais um serviço dirigido à Internet como um todo – para o bem geral. Em termos genéricos, o controle de congestionamento do TCP evita que qualquer outra conexão TCP abarrote os enlaces e roteadores entre hospedeiros comunicantes com uma quantidade excessiva de tráfego. Em princípio, o TCP permite que conexões TCP trafegando por um enlace de rede congestionado compartilhem em pé de igualdade a largura de banda daquele enlace. Isso é feito pela regulagem da taxa com a qual o lado remetente do TCP pode enviar tráfego para a rede. O tráfego UDP, por outro lado, não é regulado. Uma aplicação que usa transporte UDP pode enviar tráfego à taxa que quiser, pelo tempo que quiser.

Um protocolo que fornece transferência confiável de dados e controle de congestionamento é, necessariamente, complexo. Precisaremos de várias seções para detalhar os princípios da transferência confiável de dados e do controle de congestionamento, bem como de seções adicionais para explicar o protocolo TCP. Esses tópicos são analisados nas Seções 3.4 a 3.7. A abordagem escolhida neste capítulo é alternar entre princípios básicos e o protocolo TCP. Por exemplo, discutiremos primeiro a transferência confiável de dados em âmbito geral e, em seguida, como o TCP fornece especificamente a transferência confiável de dados. De maneira semelhante, iniciaremos discutindo o controle de congestionamento em âmbito geral e, em seguida, como o TCP realiza o controle de congestionamento. Porém, antes de chegarmos a essa parte, vamos examinar, primeiro, multiplexação/demultiplexação na camada de transporte.

## 3.2 MULTIPLEXAÇÃO E DEMULTIPLEXAÇÃO

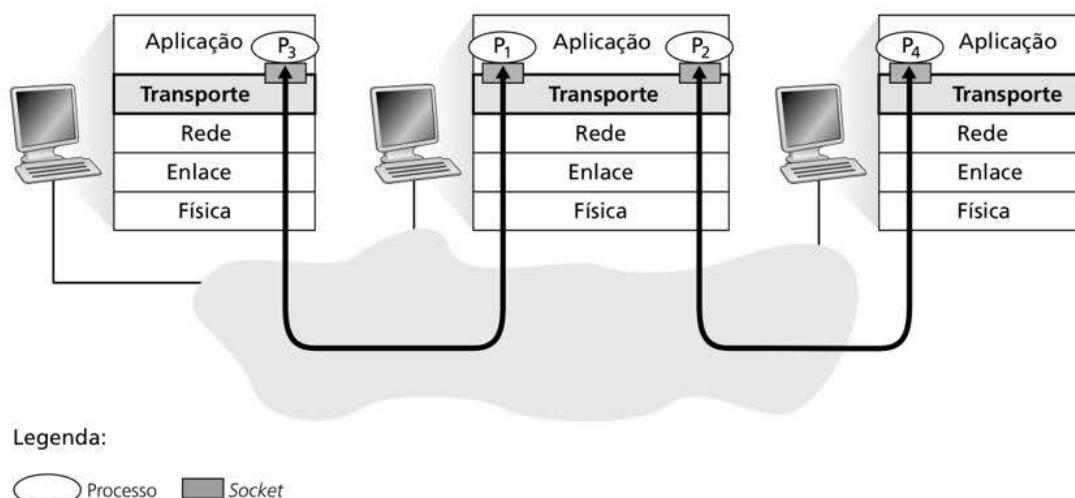
Nesta seção, discutiremos multiplexação e demultiplexação na camada de transporte, isto é, a ampliação do serviço de entrega hospedeiro a hospedeiro provido pela camada de rede para um serviço de entrega processo a processo para aplicações que rodam nesses hospedeiros.

Para manter a discussão em nível concreto, vamos examinar esse serviço básico da camada de transporte no contexto da Internet. Enfatizamos, contudo, que o serviço de multiplexação/demultiplexação é necessário para todas as redes de computadores.

No hospedeiro de destino, a camada de transporte recebe segmentos da camada de rede logo abaixo dela e tem a responsabilidade de entregar os dados desses segmentos ao processo de aplicação apropriado que roda no hospedeiro. Vamos examinar um exemplo. Suponha que você esteja sentado à frente de seu computador, baixando páginas Web enquanto roda uma sessão FTP e duas sessões Telnet. Por conseguinte, você tem quatro processos de aplicação de rede em execução – dois Telnet, um FTP e um HTTP. Quando a camada de transporte em seu computador receber dados da camada de rede abaixo dela, precisará direcionar os dados recebidos a um desses quatro processos. Vamos ver agora como isso é feito.

Em primeiro lugar, lembre-se de que dissemos, na Seção 2.7, que um processo (como parte de uma aplicação de rede) pode ter um ou mais *sockets*, portas pelas quais dados passam da rede para o processo e do processo para a rede. Assim, como mostra a Figura 3.2, a camada de transporte do hospedeiro destinatário na verdade não entrega dados diretamente a um processo, mas a um *socket* intermediário. Já que, a qualquer dado instante, pode haver mais de um *socket* no hospedeiro destinatário, cada um tem um identificador exclusivo. O formato do identificador depende de o *socket* ser UDP ou TCP, como discutiremos em breve.

Agora, vamos considerar como um hospedeiro destinatário direciona, ao *socket* apropriado, um segmento de camada de transporte que chega. Cada segmento de camada de transporte tem um conjunto de campos para tal finalidade. Na extremidade receptora, a camada de transporte examina esses campos para identificar a porta receptora e direcionar o segmento a esse *socket*. A tarefa de entregar os dados contidos em um segmento da camada de transporte ao *socket* correto é denominada **demultiplexação**. O trabalho de reunir, no hospedeiro de origem, porções de dados provenientes de diferentes *sockets*, encapsular cada parte de dados com informações de cabeçalho (que mais tarde serão usadas na demultiplexação) para criar segmentos, e passar esses segmentos para a camada de rede é denominada **multiplexação**. Note que a camada de transporte do hospedeiro que está no meio da Figura 3.2 tem de demultiplexar segmentos que chegam da camada de rede abaixo para os processos  $P_1$  ou  $P_2$  acima; isso é feito direcionando, ao *socket* do processo correspondente, os dados contidos no segmento que está chegando. A camada de transporte desse hospedeiro do meio também tem de juntar dados de saída desses *sockets*, formar segmentos de camada de transporte e passá-los à camada de rede. Embora tenhamos apresentado multiplexação e demultiplexação no contexto dos protocolos de transporte da Internet, é importante entender que essas operações estarão presentes sempre que um



**Figura 3.2** Multiplexação e demultiplexação na camada de transporte.

único protocolo em uma camada (na de transporte ou em qualquer outra) for usado por vários protocolos na camada mais alta seguinte.

Para ilustrar o serviço de demultiplexação, lembre-se da metáfora das moradias apresentada na seção anterior. Cada criança é identificada por seu nome próprio. Quando Bill recebe um lote de correspondência do carteiro, realiza uma operação de demultiplexação ao examinar a quem as cartas estão endereçadas e, em seguida, entregar a correspondência a seus irmãos e irmãs. Ann realiza uma operação de multiplexação quando coleta as cartas de seus irmãos e irmãs e entrega a correspondência na agência do correio.

Agora que entendemos os papéis da multiplexação e da demultiplexação na camada de transporte, vamos examinar como isso é feito em um hospedeiro. Sabemos, pela discussão anterior, que multiplexação na camada de rede requer (1) que as portas tenham identificadores exclusivos e (2) que cada segmento tenha campos especiais que indiquem a porta para a qual o segmento deve ser entregue. Esses campos especiais, ilustrados na Figura 3.3, são o **campo de número de porta de origem** e o **campo de número de porta de destino**. (Os segmentos UDP e TCP têm outros campos também, que serão examinados nas seções subsequentes deste capítulo.) Cada número de porta é um número de 16 bits na faixa de 0 a 65535. Os números de porta entre 0 e 1023 são denominados **números de porta bem conhecidos**; eles são restritos, o que significa que estão reservados para utilização por protocolos de aplicação bem conhecidos, como HTTP (que usa a porta número 80) e FTP (que usa a porta número 21). A lista dos números de porta bem conhecidos é apresentada no RFC 1700, e é atualizada em <<http://www.iana.org>> (RFC 3232). Quando desenvolvemos uma nova aplicação (como as desenvolvidas na Seção 2.7), devemos atribuir a ela um número de porta.

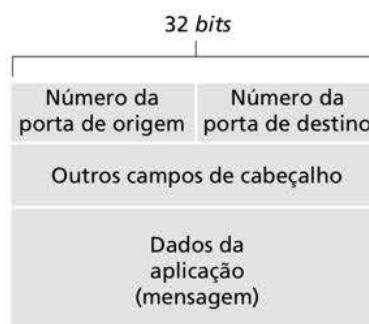
Agora já deve estar claro como a camada de transporte *poderia* realizar o serviço de demultiplexação: cada *socket* do hospedeiro pode receber um número designado; quando um segmento chega ao hospedeiro, a camada de transporte examina seu número de porta de destino e direciona o segmento ao *socket* correspondente. Então, os dados do segmento passam pela porta e entram no processo ligado a ela. Como veremos, é assim que o UDP faz demultiplexação. Todavia, veremos também que multiplexação/demultiplexação em TCP é ainda mais sutil.

### Multiplexação e demultiplexação não orientadas para conexão

Lembre-se, da Seção 2.7.1, de que um programa em Python que roda em um hospedeiro pode criar um *socket* UDP com a linha

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

Quando um *socket* UDP é criado dessa maneira, a camada de transporte automaticamente designa um número de porta ao *socket*. Em especial, a camada de transporte designa um número de porta na faixa de 1024 a 65535 que não esteja sendo usado naquele instante por



**Figura 3.3** Campos de número de porta de origem e de destino em um segmento de camada de transporte.

qualquer outro *socket* UDP no hospedeiro. Como alternativa, podemos incluir uma linha em nosso programa Python depois de criarmos o *socket* para associar um número de porta específico (digamos, 19157) a este *socket* UDP por meio do método **bind()** do *socket*:

```
clientSocket.bind(('', 19157))
```

Se o desenvolvedor responsável por escrever o código da aplicação estivesse executando o lado servidor de um “protocolo bem conhecido”, ele teria de designar o número de porta bem conhecido correspondente. O lado cliente da aplicação em geral permite que a camada de transporte designe o número de porta de modo automático (e transparente), ao passo que o lado servidor da aplicação designa um número de porta específico.

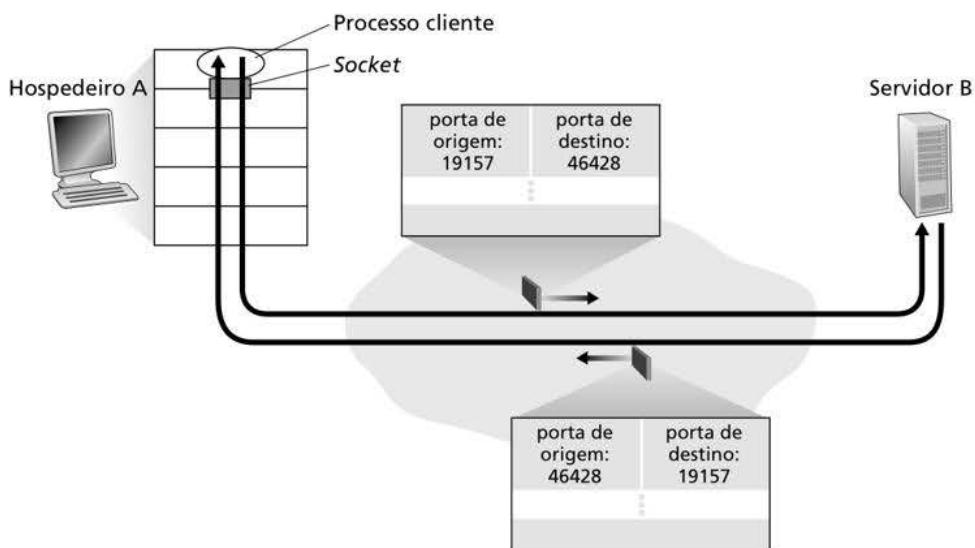
Agora que os *sockets* UDP já têm seus números de porta designados, podemos descrever multiplexação/demultiplexação UDP com precisão. Suponha que um processo no hospedeiro A, cujo número de porta UDP é 19157, queira enviar uma porção de dados de aplicação a um processo cujo número de porta UDP seja 46428 no hospedeiro B. A camada de transporte no hospedeiro A cria um segmento de camada de transporte que inclui os dados de aplicação, o número da porta de origem (19157), o número da porta de destino (46428) e mais outros dois valores (que serão discutidos mais adiante, mas que não são importantes para a discussão em curso). Então, a camada de transporte passa o segmento resultante para a camada de rede. Essa camada encapsula o segmento em um datagrama IP e faz uma tentativa de melhor esforço para entregar o segmento ao hospedeiro destinatário. Se o segmento chegar à máquina de destino B, a camada de destino no hospedeiro destinatário examinará o número da porta de destino no segmento (46428) e o entregará a seu *socket* identificado pelo número 46428. Note que a máquina B poderia estar rodando vários processos, cada um com sua própria porta UDP e número de porta associado. À medida que segmentos UDP chegassem da rede, a máquina B direcionaria (demultiplexaria) cada segmento à porta apropriada examinando o número de porta de destino do segmento.

É importante notar que um *socket* UDP é totalmente identificado por uma tupla com dois elementos, consistindo em um endereço IP de destino e um número de porta de destino. Por conseguinte, se dois segmentos UDP tiverem endereços IP de origem e/ou números de porta de origem diferentes, porém o mesmo endereço IP de *destino* e o mesmo número de porta de *destino*, eles serão direcionados ao mesmo processo de destino por meio do mesmo *socket* de destino.

É possível que agora você esteja imaginando qual é a finalidade do número da porta de origem. Como mostra a Figura 3.4, no segmento A-B, o número da porta de origem serve como parte de um “endereço de retorno” – quando B quer enviar um segmento de volta para A, a porta de destino no segmento B-A tomará seu valor do valor da porta de origem do segmento A-B. (O endereço de retorno completo é o endereço IP e o número de porta de origem de A.) Como exemplo, lembre-se do programa servidor UDP que estudamos na Seção 2.7. Em *UDPServer.py*, o servidor usa um método *recvfrom()* para extrair o número de porta cliente-servidor (de origem) do segmento que recebe do cliente; então, envia um novo segmento ao cliente, com o número de porta que extraiu servindo como o número de porta de destino desse novo segmento.

## Multiplexação e demultiplexação orientadas para conexão

Para entender demultiplexação TCP, temos de examinar de perto *sockets* TCP e estabelecimento de conexão TCP. Há uma diferença sutil entre um *socket* UDP e um TCP: o *socket* TCP é identificado por uma tupla de quatro elementos: (endereço IP de origem, número da porta de origem, endereço IP de destino, número da porta de destino). Assim, quando um segmento TCP que vem da rede chega a um hospedeiro, este usa todos os quatro valores para direcionar (demultiplexar) o segmento para o *socket* apropriado. Em especial, e ao contrário do UDP, dois segmentos TCP chegando com endereços IP de origem ou números de porta de origem diferentes serão direcionados para dois *sockets* diferentes (com exceção de um TCP



**Figura 3.4** Inversão dos números de porta de origem e destino.

que esteja carregando a requisição de estabelecimento de conexão original). Para perceber melhor, vamos considerar novamente o exemplo de programação cliente-servidor TCP apresentado na Seção 2.7.2:

- A aplicação servidor TCP tem um “*socket* de entrada” que espera requisições de estabelecimento de conexão vindas de clientes TCP (ver Figura 2.29) na porta número 12000.
- O cliente TCP cria um *socket* e envia um segmento de requisição de estabelecimento de conexão com as linhas:

```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, 12000))
```

- Uma requisição de estabelecimento de conexão nada mais é do que um segmento TCP com número de porta de destino 12000 e um bit especial de estabelecimento de conexão marcado no cabeçalho TCP (que será tratado na Seção 3.5). O segmento inclui também um número de porta de origem que foi escolhido pelo cliente.
- Quando o sistema operacional do computador que está rodando o processo servidor recebe o segmento de requisição de conexão que está chegando e cuja porta de destino é 12000, ele localiza o processo servidor que está à espera para aceitar uma conexão na porta número 12000. Então, o processo servidor cria um novo *socket*:

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

- A camada de transporte no servidor também nota os quatro valores seguintes no segmento de requisição de conexão: (1) o número da porta de origem no segmento, (2) o endereço IP do hospedeiro de origem, (3) o número da porta de destino no segmento e (4) seu próprio endereço IP. O *socket* de conexão recém-criado é identificado por esses quatro valores; todos os segmentos subsequentes que chegarem, cuja porta de origem, endereço IP de origem, porta de destino e endereço IP de destino combinarem com esses quatro valores, serão demultiplexados para esse *socket*. Com a conexão TCP agora ativa, o cliente e o servidor podem enviar dados um para o outro.

O hospedeiro servidor pode suportar vários *sockets* TCP simultâneos, sendo cada qual ligado a um processo e identificado por sua própria tupla de quatro elementos. Quando um segmento TCP chega ao hospedeiro, todos os quatro campos (endereço IP da origem, porta de origem, endereço IP de destino, porta de destino) são usados para direcionar (demultiplexar) o segmento para o *socket* apropriado.

## SEGURANÇA EM FOCO

### VARREDURA DE PORTA

Vimos que um processo servidor espera com paciência, em uma porta aberta, o contato de um cliente remoto. Algumas portas são reservadas para aplicações familiares (p. ex., servidores SMTP, Web, FTP e DNS); outras são utilizadas por convenção por aplicações populares (p. ex., o Microsoft Windows SQL Server ouve as solicitações na porta 1434 do UDP). Desta forma, se determinarmos que uma porta está aberta em um hospedeiro, talvez possamos mapeá-la para uma aplicação específica sendo executada no hospedeiro. Isso é muito útil para administradores de sistemas, que muitas vezes têm interesse em saber quais aplicações estão sendo executadas nos hospedeiros em suas redes. Porém, os atacantes, a fim de “examinarem o local”, também querem saber quais portas estão abertas nos hospedeiros direcionados. Se o hospedeiro estiver sendo executado em uma aplicação com uma falha de segurança conhecida (p. ex., um servidor SQL ouvindo em uma porta 1434 estava sujeito a esgotamento de

*buffer*, permitindo que um usuário remoto execute um código arbitrário no hospedeiro vulnerável, uma falha explorada pelo *worm Slammer* (CERT, 2003-04), então esse hospedeiro está pronto para o ataque.

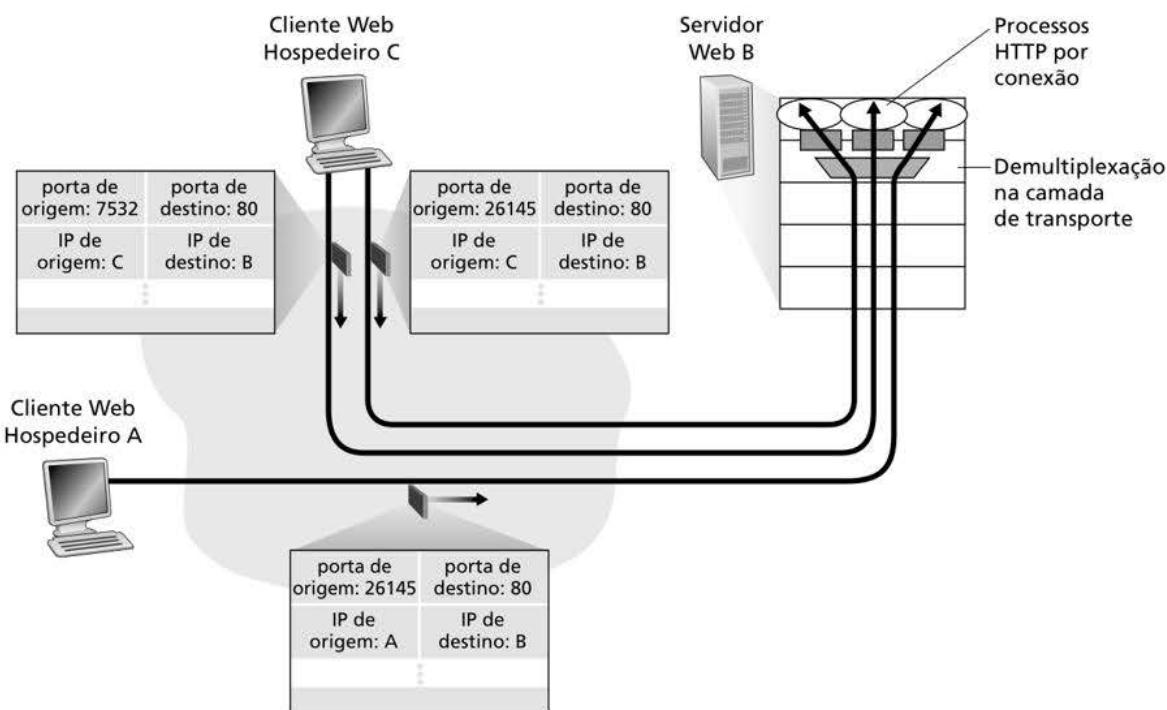
Determinar quais aplicações estão ouvindo em quais portas é uma tarefa de certo modo fácil. De fato, há inúmeros programas de domínio público, denominados varredores de porta, que fazem exatamente isso. Talvez o mais utilizado seja o nmap, disponível gratuitamente em <http://nmap.org> e incluído na maioria das distribuições Linux. Para o TCP, o nmap varre portas sequencialmente, procurando portas que aceitem conexões TCP. Para o UDP, o nmap de novo varre portas em sequência, procurando portas UDP que respondam aos segmentos UDP transmitidos. Em ambos os casos, o nmap retorna uma lista de portas abertas, fechadas ou inalcançáveis. Um hospedeiro executando o nmap pode tentar varrer qualquer hospedeiro direcionado em *qualquer* lugar da Internet. Voltaremos a falar sobre o nmap na Seção 3.5.6, ao discutirmos gerenciamento da conexão TCP.

A situação é ilustrada na Figura 3.5, na qual o hospedeiro C inicia duas sessões HTTP com o servidor B, e o hospedeiro A inicia uma sessão HTTP com o servidor B. Os hospedeiros A e C e o servidor B possuem, cada um, seu próprio endereço IP exclusivo: A, C e B, respectivamente. O hospedeiro C atribui dois números diferentes (26145 e 7532) da porta de origem às suas duas conexões HTTP. Como o hospedeiro A está escolhendo números de porta independentemente de C, ele poderia também atribuir um número da porta de origem 26145 à sua conexão HTTP. Apesar disso, o servidor B ainda será capaz de demultiplexar corretamente as duas conexões que têm o mesmo número de porta de origem, já que elas têm endereços IP de origem diferentes.

### Servidores Web e TCP

Antes de encerrar esta discussão, é instrutivo falar um pouco mais sobre servidores Web e como eles usam números de porta. Considere um hospedeiro rodando um servidor Web, tal como um Apache, na porta 80. Quando clientes (p. ex., navegadores) enviam segmentos ao servidor, *todos* os segmentos terão a porta de destino 80. Em especial, os segmentos que estabelecem a conexão inicial e os que carregam as mensagens de requisição HTTP, ambos terão a porta de destino 80. Como acabamos de descrever, o servidor distingue os segmentos dos diferentes clientes pelos endereços IP e pelos números da porta de origem.

A Figura 3.5 mostra um servidor Web que cria um novo processo para cada conexão. Como mostra a figura, cada um desses processos tem seu próprio *socket* de conexão pelo qual chegam requisições HTTP e são enviadas respostas HTTP. Mencionamos, contudo, que nem sempre existe uma correspondência unívoca entre *sockets* de conexão e processos. Na verdade, os servidores Web de alto desempenho atuais muitas vezes utilizam somente um processo, mas criam uma nova *thread* com um novo *socket* de conexão para cada nova conexão cliente. (Uma *thread* pode ser considerada um subprocesso leve.) Se você fez a primeira tarefa de programação do Capítulo 2, construiu um servidor Web que faz exatamente



**Figura 3.5** Dois clientes que usam o mesmo número de porta de destino (80) para se comunicar com a mesma aplicação de servidor Web.

isso. Para um servidor desses, a qualquer instante pode haver muitos *sockets* de conexão (com identificadores diferentes) ligados ao mesmo processo.

Se o cliente e o servidor estiverem usando HTTP persistente, então, durante toda a conexão persistente, trocarão mensagens HTTP pelo mesmo *socket* do servidor. Todavia, se usarem HTTP não persistente, então uma nova conexão TCP é criada e encerrada para cada requisição/resposta, e, portanto, um novo *socket* é criado e mais tarde encerrado para cada requisição/resposta. A criação e o encerramento frequentes de *sockets* podem causar sério impacto sobre o desempenho de um servidor Web movimentado (embora o sistema operacional consiga usar várias estratégias para atenuar o problema). Aconselhamos o leitor interessado em questões de sistema operacional referentes a HTTP persistente e não persistente a consultar (Nielsen, 1997; Nahum, 2002).

Agora que já discutimos multiplexação e demultiplexação na camada de transporte, passemos à discussão de um dos protocolos da Internet, o UDP. Na próxima seção, veremos que o UDP acrescenta pouco mais ao protocolo da camada de rede do que um serviço de multiplexação/demultiplexação.

### 3.3 TRANSPORTE NÃO ORIENTADO PARA CONEXÃO: UDP

Nesta seção, examinaremos o UDP mais de perto, como ele funciona e o que ele faz. Aconselhamos o leitor a rever o material apresentado na Seção 2.1, que inclui uma visão geral do modelo de serviço UDP, e o da Seção 2.7.1, que discute a programação de portas por UDP.

Para motivar nossa discussão sobre UDP, suponha que você esteja interessado em projetar um protocolo de transporte simples, bem básico. Como faria isso? De início, você deve considerar a utilização de um protocolo de transporte vazio. Em especial, do lado do

remetente, considere pegar as mensagens do processo da aplicação e passá-las diretamente para a camada de rede; do lado do destinatário, considere pegar as mensagens que chegam da camada de rede e passá-las diretamente ao processo da aplicação. Mas, como aprendemos na seção anterior, o que teremos de fazer é um pouco mais do que nada. No mínimo, a camada de transporte tem de fornecer um serviço de multiplexação/demultiplexação para passar os dados da camada de rede ao processo em nível de aplicação correto.

O UDP, definido no (RFC 768), faz apenas o mínimo que um protocolo de transporte pode fazer. À parte sua função de multiplexação/demultiplexação e de alguma verificação de erros simples, ele nada adiciona ao IP. Na verdade, se o desenvolvedor de aplicação escolher o UDP, em vez do TCP, a aplicação estará “falando” quase diretamente com o IP. O UDP pega as mensagens do processo da aplicação, anexa os campos de número da porta de origem e de destino para o serviço de multiplexação/demultiplexação, adiciona dois outros pequenos campos e passa o segmento resultante à camada de rede, que encapsula o segmento dentro de um datagrama IP e, em seguida, faz a melhor tentativa para entregar o segmento ao hospedeiro receptor. Se o segmento chegar ao hospedeiro receptor, o UDP usará o número de porta de destino para entregar os dados do segmento ao processo de aplicação correto. Note que, com o UDP, não há apresentação entre as entidades remetente e destinatária da camada de transporte antes de enviar um segmento. Por essa razão, dizemos que o UDP é *não orientado para conexão*.

O sistema de nome de domínio (DNS, do inglês *domain name system*) é um exemplo de protocolo de camada de aplicação que usa o UDP. Quando a aplicação DNS em um hospedeiro quer fazer uma consulta, constrói uma mensagem de consulta DNS e passa a mensagem para o UDP. Sem realizar nenhuma apresentação com a entidade UDP que está funcionando no sistema final de destino, o UDP do lado do hospedeiro adiciona campos de cabeçalho à mensagem e passa o segmento resultante à camada de rede, que encapsula o segmento UDP em um datagrama e o envia a um servidor de nomes. A aplicação DNS no hospedeiro requisitante então espera por uma resposta à sua consulta. Se não receber uma resposta (possivelmente porque a rede subjacente perdeu a consulta ou a resposta), ela poderia tentar reenviar a consulta, tentar enviar a consulta a outro servidor de nomes ou informar à aplicação consultante que não pôde obter uma resposta.

É possível que agora você esteja imaginando por que um desenvolvedor de aplicação escolheria construir uma aplicação sobre UDP, em vez de sobre TCP. O TCP não é sempre preferível ao UDP, já que fornece serviço confiável de transferência de dados e o UDP não? A resposta é “não”, pois algumas aplicações se adaptam melhor ao UDP pelas seguintes razões:

- *Melhor controle no nível da aplicação sobre quais dados são enviados e quando.* Com UDP, tão logo um processo de aplicação passe dados ao UDP, o protocolo os empacotará dentro de um segmento UDP e os passará imediatamente à camada de rede. O TCP, por outro lado, tem um mecanismo de controle de congestionamento que limita o remetente TCP da camada de transporte quando um ou mais enlaces entre os hospedeiros da origem e do destinatário ficam congestionados demais. O TCP também continuará a reenviar um segmento até que o hospedeiro destinatário reconheça a recepção desse segmento, pouco importando o tempo que a entrega confiável levar. Visto que aplicações de tempo real requerem uma taxa mínima de envio, não querem atrasar demais a transmissão de segmentos e podem tolerar alguma perda de dados, o modelo de serviço do TCP não é particularmente compatível com as necessidades dessas aplicações. Como discutiremos adiante, essas aplicações podem usar UDP e executar, como parte da aplicação, qualquer funcionalidade adicional necessária além do serviço de entrega de segmentos simples e básicos do UDP.
- *Não há estabelecimento de conexão.* Como discutiremos adiante, o TCP usa uma apresentação de três vias antes de começar a transferir dados. O UDP simplesmente envia mensagens sem nenhuma preliminar formal e, assim, não introduz atraso algum para estabelecer uma conexão. Talvez seja esta a principal razão pela qual o DNS roda sobre

UDP, e não sobre TCP – o DNS seria muito mais lento se rodasse em TCP. O HTTP usa o TCP, e não o UDP, porque a confiabilidade é fundamental para páginas Web com texto. Mas, como discutimos brevemente na Seção 2.2, o atraso de estabelecimento de uma conexão TCP é uma contribuição importante aos atrasos associados à recepção de documentos Web. Na verdade, o protocolo QUIC (do inglês Quick UDP Internet Connection, [IETF QUIC 2020]), usado no navegador Google Chrome, utiliza o UDP como protocolo de transporte subjacente e implementa a confiabilidade em um protocolo da camada de aplicação sobre o UDP. Analisaremos o QUIC em mais detalhes na Seção 3.8.

- *Não há estados de conexão.* O TCP mantém o estado de conexão nos sistemas finais. Esse estado inclui *buffers* de envio e recebimento, parâmetros de controle de congestionamento e parâmetros numéricos de sequência e de reconhecimento. Veremos na Seção 3.5 que essa informação de estado é necessária para implementar o serviço de transferência confiável de dados do TCP e para prover controle de congestionamento. O UDP, por sua vez, não mantém o estado de conexão e não monitora nenhum desses parâmetros. Por essa razão, um servidor dedicado a uma aplicação específica pode suportar um número muito maior de clientes ativos quando a aplicação roda sobre UDP e não sobre TCP.
- *Pequeno cabeçalho de pacote.* O segmento TCP tem 20 bytes (*overhead*) de cabeçalho, além dos dados para cada segmento, enquanto o UDP tem somente 8 bytes de cabeçalho.

A Figura 3.6 relaciona aplicações populares da Internet e os protocolos de transporte que usam. Como era esperado, o *e-mail*, o acesso a terminal remoto, a Web e a transferência de arquivos rodam sobre TCP – todas essas aplicações necessitam do serviço confiável de transferência de dados do TCP. No Capítulo 2, vimos que as primeiras versões do HTTP rodavam sobre TCP, mas que as versões mais recentes rodam sobre UDP, provendo seu próprio controle de erros e de congestionamento (entre outros serviços) na camada de aplicação. Não obstante, muitas aplicações importantes rodam sobre UDP, e não sobre TCP. Por exemplo, o UDP é usado para levar dados de gerenciamento de rede (SNMP; veja a Seção 5.7). Nesse caso, o UDP é preferível ao TCP, já que aplicações de gerenciamento de rede com frequência devem funcionar quando a rede está em estado sobrecarregado – exatamente quando é difícil conseguir transferência confiável de dados com congestionamento controlado. E também, como mencionamos, o DNS roda sobre UDP, evitando, desse modo, atrasos de estabelecimento de conexões TCP.

Como mostra a Figura 3.6, hoje o UDP e o TCP também são comumente usados para aplicações de multimídia, como telefone por Internet, videoconferência em tempo real e

Aplicação	Protocolo de camada de aplicação	Protocolo de transporte subjacente
Correio eletrônico	SMTP	TCP
Acesso a terminal remoto	Telnet	TCP
Acesso a terminal remoto seguro	SSH	TCP
Web	HTTP, HTTP/3	TCP (para HTTP), UDP (for HTTP/3)
Transferência de arquivo	FTP	TCP
Servidor de arquivo remoto	NFS	Tipicamente UDP
Streaming de multimídia	DASH	TCP
Telefonia por Internet	Tipicamente proprietário	UDP ou TCP
Gerenciamento de rede	SNMP	Tipicamente UDP
Tradução de nome	DNS	Tipicamente UDP

**Figura 3.6** Aplicações populares da Internet e seus protocolos de transporte subjacentes.

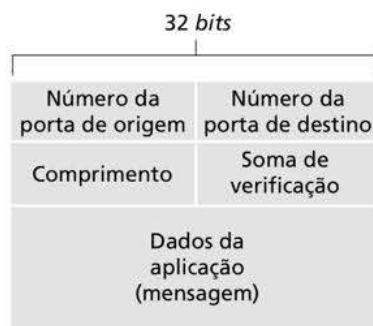
*streaming* de áudio e vídeo armazenados. No momento, mencionamos apenas que todas essas aplicações podem tolerar uma pequena quantidade de perda de pacotes, de modo que a transferência confiável de dados não é absolutamente crítica para o sucesso da aplicação. Além disso, aplicações em tempo real, como telefone por Internet e videoconferência, reagem muito mal ao controle de congestionamento do TCP. Por essas razões, os desenvolvedores de aplicações de multimídia muitas vezes optam por rodar suas aplicações sobre UDP em vez de sobre TCP. Quando as taxas de perda de pacote são baixas, junto com algumas empresas que bloqueiam o tráfego UDP por razões de segurança (veja Capítulo 8), o TCP se torna um protocolo cada vez mais atrativo para o transporte de mídia.

Embora hoje seja comum rodar aplicações de multimídia sobre UDP, isso exige cuidados. Como já mencionamos, o UDP não tem controle de congestionamento. Mas esse controle é necessário para evitar que a rede entre em um estado no qual pouquíssimo trabalho útil é realizado. Se todos começassem a assistir vídeos com alta taxa de *bits* sem usar nenhum controle de congestionamento, haveria tamanho transbordamento de pacotes nos roteadores que poucos pacotes UDP conseguiriam atravessar com sucesso o caminho da origem ao destino. Além do mais, as altas taxas de perda induzidas pelos remetentes UDP sem controle fariam com que os remetentes TCP (que, como veremos adiante, *reduzem* suas taxas de envio em face de congestionamento) reduzissem drasticamente suas taxas. Assim, a falta de controle de congestionamento no UDP pode resultar em altas taxas de perda entre um remetente e um destinatário UDP e no acúmulo de sessões TCP. Muitos pesquisadores propuseram novos mecanismos para forçar todas as origens, inclusive as origens UDP, a realizar um controle de congestionamento adaptativo (Mahdavi, 1997; Floyd, 2000; Kohler, 2006: RFC 4340).

Antes de discutirmos a estrutura do segmento UDP, mencionaremos que é possível uma aplicação ter transferência confiável de dados usando UDP. Isso pode ser feito se a confiabilidade for embutida na própria aplicação (p. ex., adicionando mecanismos de reconhecimento e de retransmissão, tais como os que estudaremos na próxima seção). Anteriormente, mencionamos que o protocolo QUIC implementa a confiabilidade em um protocolo da camada de aplicação sobre o UDP. Mas essa é uma tarefa não trivial, que manteria o desenvolvedor ocupado com a depuração por um longo tempo. Não obstante, embutir confiabilidade diretamente na aplicação permite que ela tire proveito de ambas as alternativas. Em outras palavras, os processos da aplicação podem se comunicar de maneira confiável sem ter de se sujeitar às limitações de taxa de transmissão impostas pelo mecanismo de controle de congestionamento do TCP.

### 3.3.1 Estrutura do segmento UDP

A estrutura do segmento UDP, mostrada na Figura 3.7, é definida no RFC 768. Os dados da aplicação ocupam o campo de dados do segmento UDP. Por exemplo, para o DNS, o campo de dados contém uma mensagem de consulta ou uma mensagem de resposta. Para uma aplicação de recepção de áudio, amostras de áudio preenchem o campo de dados. O cabeçalho UDP tem apenas quatro campos, cada um consistindo em 2 *bytes*. Como já discutido na seção anterior, os números de porta permitem que o hospedeiro destinatário passe os dados da aplicação ao processo correto que está funcionando no sistema final destinatário (i.e., realize a função de demultiplexação). O campo de comprimento especifica o número de *bytes* no segmento UDP (cabeçalho mais dados). Um valor de comprimento explícito é necessário, porque o tamanho do campo de dados pode ser diferente de um segmento UDP para o outro. A soma de verificação é usada pelo hospedeiro receptor para verificar se foram introduzidos erros no segmento. Na verdade, a soma de verificação também é calculada para alguns dos campos no cabeçalho IP, além do segmento UDP. Mas ignoramos esse detalhe para podermos enxergar a floresta por meio das árvores. Discutiremos o cálculo da soma de verificação adiante. Os princípios básicos da detecção de erros estão descritos na Seção 6.2. O campo de comprimento especifica o comprimento do segmento UDP, incluindo o cabeçalho, em *bytes*.



**Figura 3.7** Estrutura do segmento UDP.

### 3.3.2 Soma de verificação UDP

A soma de verificação UDP serve para detectar erros. Em outras palavras, é usada para determinar se *bits* dentro do segmento UDP foram alterados (p. ex., por ruído nos enlaces ou enquanto armazenados em um roteador) durante sua movimentação da origem até o destino. O UDP no lado remetente realiza o complemento de 1 da soma de todas as palavras de 16 *bits* do segmento, levando em conta o “vai um” em toda a soma. Esse resultado é colocado no campo de soma de verificação no segmento UDP. Damos aqui um exemplo simples do cálculo da soma de verificação. Se quiser saber detalhes sobre a implementação eficiente do algoritmo de cálculo, consulte o RFC 1071; sobre o desempenho com dados reais, consulte Stone (1998 e 2000). Como exemplo, suponha que tenhamos as seguintes três palavras de 16 *bits*:

0110011001100000  
0101010101010101  
1000111100001100

A soma das duas primeiras é:

0110011001100000  
0101010101010101  
1011101110110101

Adicionando a terceira palavra à soma anterior, temos:

1011101110110101  
1000111100001100  
0100101011000010

Note que a última adição teve “vai um” no *bit* mais significativo que foi somado ao *bit* menos significativo. O complemento de 1 é obtido pela conversão de todos os 0 em 1 e de todos os 1 em 0. Desse modo, o complemento de 1 da soma 0100101011000010 é 101101010011101, que passa a ser a soma de verificação. No destinatário, todas as quatro palavras de 16 *bits* são somadas, inclusive a soma de verificação. Se nenhum erro for introduzido no pacote, a soma no destinatário será, claro, 1111111111111111. Se um dos *bits* for um zero, saberemos então que um erro foi introduzido no pacote.

Talvez você esteja imaginando por que o UDP fornece uma soma de verificação primeiro, já que muitos protocolos de camada de enlace (inclusive, o popular Ethernet) também fornecem verificação de erros. A razão é que não há garantia de que todos os enlaces entre a origem e o destino fornecem tal verificação – um deles pode usar um protocolo de camada de enlace que não a forneça. Além disso, mesmo que os segmentos sejam corretamente transmitidos por um enlace, pode haver introdução de erros de *bits* quando um segmento é armazenado na memória de um roteador. Como não são garantidas a confiabilidade enlace a enlace e a detecção de erro na memória, o UDP deve prover detecção de erro  *fim a fim*

na camada de transporte se quisermos que o serviço de transferência de dados fim a fim forneça detecção de erro. Esse é um exemplo do famoso **princípio fim a fim** do projeto de sistemas (Saltzer, 1984). Tal princípio afirma que, visto ser dado como certo que funcionalidades (deteção de erro, neste caso) devem ser executadas fim a fim, “funções colocadas nos níveis mais baixos podem ser redundantes ou de pouco valor em comparação com o custo de fornecê-las no nível mais alto”.

Como se pretende que o IP rode sobre qualquer protocolo de camada 2, é útil que a camada de transporte forneça verificação de erros como medida de segurança. Embora o UDP forneça verificação de erros, ele nada faz para recuperar-se de um erro. Algumas implementações do UDP apenas descartam o segmento danificado; outras passam o segmento errado à aplicação acompanhado de um aviso.

Isso encerra nossa discussão sobre o UDP. Logo veremos que o TCP oferece transferência confiável de dados a suas aplicações, bem como outros serviços que o UDP não oferece. Naturalmente, o TCP também é mais complexo do que o UDP. Contudo, antes de discutirmos o TCP, primeiro devemos examinar os princípios subjacentes da transferência confiável de dados.

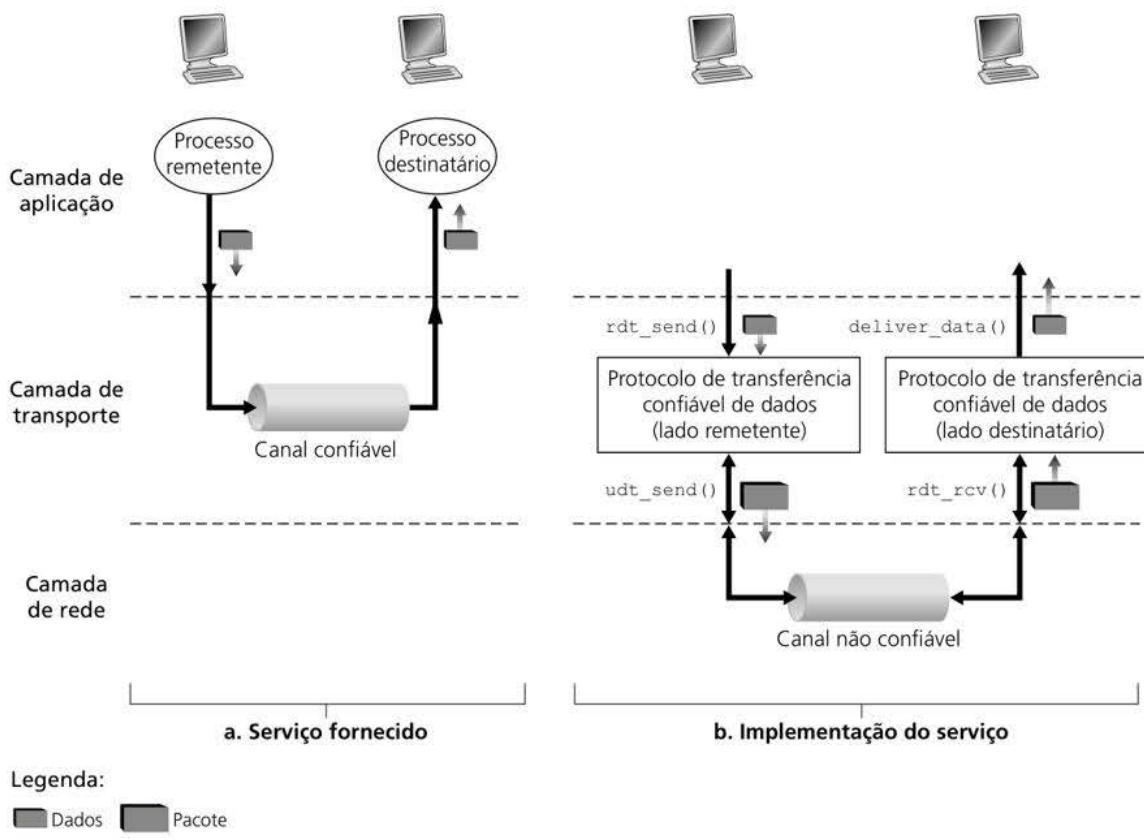
### 3.4 PRINCÍPIOS DA TRANSFERÊNCIA CONFIÁVEL DE DADOS

Nesta seção, consideraremos o problema conceitual da transferência confiável de dados. Isso é apropriado, já que o problema de realizar transferência confiável de dados ocorre não só na camada de transporte, mas também nas camadas de enlace e de aplicação. Assim, o problema geral é de importância central para o trabalho em rede. Na verdade, se tivéssemos de fazer uma lista dos dez problemas mais importantes para todo o trabalho em rede, o da transferência confiável de dados seria o candidato número um da lista. Na seção seguinte, examinaremos o TCP e mostraremos, em especial, que ele utiliza muitos dos princípios que descreveremos aqui.

A Figura 3.8 ilustra a estrutura para nosso estudo de transferência confiável de dados. A abstração do serviço fornecido às entidades das camadas superiores é a de um canal confiável através do qual dados podem ser transferidos. Com um canal confiável, nenhum dos dados transferidos é corrompido (trocado de 0 para 1 ou vice-versa) nem perdido, e todos são entregues na ordem em que foram enviados. Esse é exatamente o modelo de serviço oferecido pelo TCP às aplicações de Internet que recorrem a ele.

É responsabilidade de um **protocolo de transferência confiável de dados** implementar essa abstração de serviço. A tarefa é dificultada pelo fato de que a camada *abaixo* do protocolo de transferência confiável de dados talvez seja não confiável. Por exemplo, o TCP é um protocolo confiável de transferência de dados que é executado sobre uma camada de rede fim a fim não confiável (IP). De modo mais geral, a camada abaixo das duas extremidades que se comunicam de modo confiável pode consistir em um único enlace físico (como no caso de um protocolo de transferência de dados na camada de enlace) ou em uma rede global interligada (como em um protocolo de camada de transporte). Para nossa finalidade, contudo, podemos considerar essa camada mais baixa apenas como um canal ponto a ponto não confiável.

Nesta seção, desenvolveremos de modo gradual os lados remetente e destinatário de um protocolo confiável de transferência de dados, considerando modelos progressivamente mais complexos do canal subjacente. Por exemplo, vamos considerar que os mecanismos do protocolo são necessários quando o canal subjacente puder corromper *bits* ou perder pacotes inteiros. Uma suposição que adotaremos em toda essa discussão é que os pacotes serão entregues na ordem em que foram enviados, com alguns pacotes possivelmente sendo



**Figura 3.8** Transferência confiável de dados: modelo do serviço e implementação do serviço.

perdidos; ou seja, o canal subjacente não reordenará pacotes. A Figura 3.8(b) ilustra as interfaces de nosso protocolo de transferência de dados. O lado remetente do protocolo será invocado de cima, por uma chamada a `rdt_send()`. Ele passará os dados a serem entregues à camada superior no lado destinatário. (Aqui, `rdt` significa protocolo *reliable data transfer* – transferência confiável de dados – e `_send` indica que o lado remetente do `rdt` está sendo chamado. O primeiro passo no desenvolvimento de qualquer protocolo é dar-lhe um bom nome!) Do lado destinatário, `rdt_recv()` será chamado quando um pacote chegar pelo lado destinatário do canal. Quando o protocolo `rdt` quiser entregar dados à camada superior, ele o fará chamando `deliver_data()`. No que se segue, usamos a terminologia “pacote” em vez de “segmento” de camada de transporte. Como a teoria desenvolvida nesta seção se aplica a redes de computadores em geral, e não só à camada de transporte da Internet, o termo genérico “pacote” talvez seja mais apropriado aqui.

Nesta seção, consideraremos apenas o caso de **transferência unidirecional de dados**, isto é, transferência de dados do lado remetente ao lado destinatário. O caso de **transferência bidirecional confiável de dados** (i.e., *full-duplex*) não é conceitualmente mais difícil, mas é bem mais tedioso de explicar. Embora consideremos apenas a transferência unidirecional de dados, é importante notar que, apesar disso, os lados remetente e destinatário de nosso protocolo terão de transmitir pacotes em *ambas* as direções, como mostra a Figura 3.8. Logo veremos que, além de trocar pacotes contendo os dados a transferir, os lados remetente e destinatário do `rdt` também precisarão trocar pacotes de controle entre si. Ambos os lados de envio e destino do `rdt` enviam pacotes para o outro por meio de uma chamada a `udt_send()` (em que `udt` significa *unreliable data transfer* – transferência não confiável de dados).

### 3.4.1 Construindo um protocolo de transferência confiável de dados

Vamos percorrer agora uma série de protocolos que vão se tornando cada vez mais complexos, até chegar a um impecável protocolo de transferência confiável de dados.

#### Transferência confiável de dados sobre um canal perfeitamente confiável: `rdt1.0`

Consideremos primeiro o caso mais simples, em que o canal subjacente é completamente confiável. O protocolo em si, que denominaremos `rdt1.0`, é trivial. As definições de **máquina de estado finito (FSM, do inglês finite-state machine)** para o remetente e o destinatário `rdt1.0` são apresentadas na Figura 3.9. A FSM da Figura 3.9(a) define a operação do remetente, enquanto a FSM da Figura 3.9(b) define a operação do destinatário. É importante notar que há FSMs *separadas* para o remetente e o destinatário. Ambas as FSMs da Figura 3.9 têm apenas um estado. As setas na descrição da FSM indicam a transição do protocolo de um estado para outro. (Como cada FSM da Figura 3.9 tem apenas um estado, uma transição é, necessariamente, de um dado estado para ele mesmo; examinaremos diagramas de estados mais complicados em breve.) O evento que causou a transição é mostrado acima da linha horizontal que a rotula, e as ações realizadas quando ocorre o evento são mostradas abaixo dessa linha. Quando nenhuma ação é realizada em um evento, ou quando não ocorre nenhum evento e uma ação é realizada, usaremos o símbolo  $\Lambda$ , acima ou abaixo da linha horizontal, para indicar a falta de uma ação ou de um evento, respectivamente. O estado inicial da FSM é indicado pela seta tracejada. Embora as FSMs da Figura 3.9 tenham apenas um estado, as outras que veremos em breve têm vários, portanto, será importante identificar o estado inicial de cada FSM.

O lado remetente do `rdt` apenas aceita dados da camada superior pelo evento `rdt_send(data)`, cria um pacote que contém os dados (pela ação `make_pkt(data)`) e o envia para dentro do canal. Na prática, o evento `rdt_send(data)` resultaria de uma chamada de procedimento (p. ex., para `rdt_send()`) pela aplicação da camada superior.

Do lado destinatário, `rdt` recebe um pacote do canal subjacente pelo evento `rdt_recv(packet)`, extrai os dados do pacote (pela ação `extract(packet, data)`) e os



a. `rdt1.0: lado remetente`



b. `rdt1.0: lado destinatário`

**Figura 3.9** `rdt1.0` – Um protocolo para um canal completamente confiável.

passa para a camada superior (pela ação `deliver_data(data)`). Na prática, o evento `rdt_rcv(packet)` resultaria de uma chamada de procedimento (p. ex., para `rdt_rcv()`) do protocolo da camada inferior.

Nesse protocolo simples, não há diferença entre a unidade de dados e um pacote. E, também, todo o fluxo de pacotes flui do remetente para o destinatário; com um canal perfeitamente confiável, não há necessidade de o lado destinatário fornecer qualquer informação ao remetente, já que nada pode dar errado! Note que também admitimos que o destinatário está capacitado a receber dados seja qual for a velocidade em que o remetente os envie. Assim, não há necessidade de pedir para o remetente desacelerar!

### Transferência confiável de dados por um canal com erros de *bits*: `rdt2.0`

Um modelo mais realista de canal subjacente é um canal em que os *bits* de um pacote podem ser corrompidos. Esses erros de *bits* ocorrem em geral nos componentes físicos de uma rede enquanto o pacote é transmitido, propagado ou armazenado. Continuaremos a admitir, por enquanto, que todos os pacotes transmitidos sejam recebidos (embora seus *bits* possam estar corrompidos) na ordem em que foram enviados.

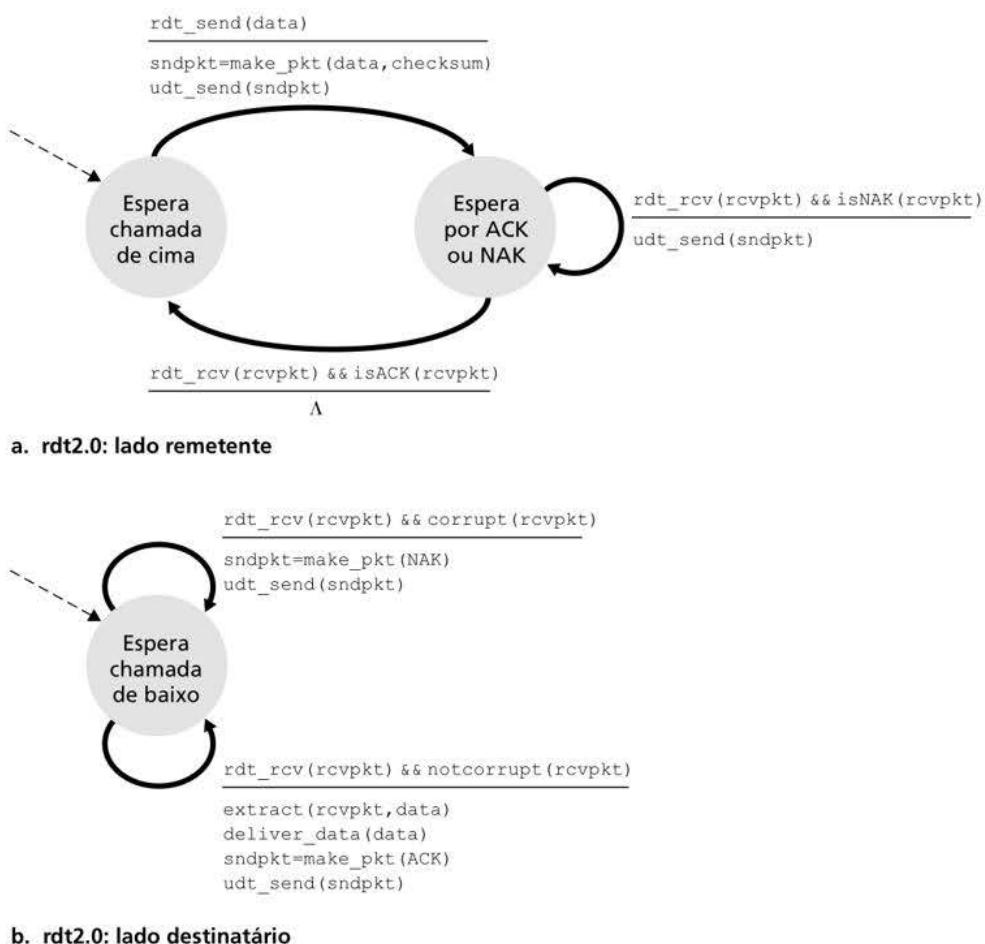
Antes de desenvolver um protocolo para se comunicar de maneira confiável com esse canal, considere primeiro como as pessoas enfrentariam uma situação como essa. Imagine como você ditaria uma mensagem longa pelo telefone. Em um cenário típico, quem estivesse anotando a mensagem diria “OK” após cada sentença que ouvisse, entendesse e anotasse. Se a pessoa ouvisse uma mensagem truncada, pediria que você a repetisse. Esse protocolo de ditado de mensagem usa **reconhecimentos positivos** (“OK”) e **reconhecimentos negativos** (“Repita, por favor”). Tais mensagens de controle permitem que o destinatário faça o remetente saber o que foi recebido corretamente e o que foi recebido com erro e, portanto, exige repetição. Em um arranjo de rede de computadores, protocolos de transferência confiável de dados baseados nesse tipo de retransmissão são conhecidos como **protocolos ARQ** (do inglês *Automatic Repeat reQuest – solicitação automática de repetição*).

Basicamente, são exigidas três capacitações adicionais dos protocolos ARQ para manipular a presença de erros de *bits*:

- *Detecção de erros.* Primeiro, é preciso um mecanismo que permita ao destinatário detectar quando ocorrem erros. Lembre-se de que dissemos na seção anterior que o UDP usa o campo de soma de verificação da Internet exatamente para essa finalidade. No Capítulo 6, examinaremos, com mais detalhes, técnicas de detecção e de correção de erros. Elas permitem que o destinatário detecte e talvez corrija erros de *bits* de pacotes. Por enquanto, basta saber que essas técnicas exigem que *bits* extras (além dos *bits* dos dados originais a serem transferidos) sejam enviados do remetente ao destinatário. Esses *bits* são colocados no campo de soma de verificação do pacote de dados do protocolo `rdt2.0`.
- *Realimentação do destinatário.* Como remetente e destinatário em geral estão rodando em sistemas finais diferentes, possivelmente separados por milhares de quilômetros, o único modo de o remetente saber qual é a visão de mundo do destinatário (neste caso, se um pacote foi recebido corretamente ou não) é o destinatário fornecer realimentação explícita ao remetente. As respostas de reconhecimento positivo (ACK, do inglês *acknowledgement*) ou negativo (NAK, do inglês *negative acknowledgement*) no cenário do ditado da mensagem são exemplos dessa realimentação. Nossa protocolo `rdt2.0` devolverá, dessa mesma maneira, pacotes ACK e NAK do destinatário ao remetente. Em princípio, esses pacotes precisam apenas ter o comprimento de um *bit*; por exemplo, um valor 0 poderia indicar um NAK, e um valor 1 poderia indicar um ACK.
- *Retransmissão.* Um pacote que é recebido com erro no destinatário será retransmitido pelo remetente.

A Figura 3.10 mostra a representação por FSM do rdt2.0, um protocolo de transferência de dados que emprega detecção de erros, reconhecimentos positivos e reconhecimentos negativos.

O lado remetente do rdt2.0 tem dois estados. No estado mais à esquerda, o protocolo do lado remetente está esperando que os dados sejam passados pela camada superior. Quando o evento `rdt_send(data)` ocorrer, o remetente criará um pacote (`sndpkt`) contendo os dados a serem enviados, junto com uma soma de verificação do pacote (p. ex., como discutimos na Seção 3.3.2 para o caso de um segmento UDP) e, então, enviará o pacote pela operação `udt_send(sndpkt)`. No estado mais à direita, o protocolo remetente está esperando por um pacote ACK ou NAK da parte do destinatário. Se um pacote ACK for recebido (a notação `rdt_rcv(rcvpkt) && isACK(rcvpkt)` na Figura 3.10 corresponde a esse evento), o remetente saberá que o pacote transmitido mais recentemente foi recebido corretamente. Assim, o protocolo volta ao estado de espera por dados vindos da camada superior. Se for recebido um NAK, o protocolo retransmitirá o último pacote e esperará por um ACK ou NAK a ser devolvido pelo destinatário em resposta ao pacote de dados retransmitido. É importante notar que, quando o destinatário está no estado de espera por ACK ou NAK, *não* pode receber mais dados da camada superior; isto é, o evento `rdt_send()` não pode ocorrer; isso somente acontecerá após o remetente receber um ACK e sair desse estado. Assim, o remetente não enviará novos dados até ter certeza de que o destinatário recebeu corretamente o pacote em questão. Devido a esse comportamento, protocolos como o rdt2.0 são conhecidos como **protocolos pare e espere** (do inglês *stop-and-wait*).



**Figura 3.10** rdt2.0 – Um protocolo para um canal com erros de bits.

A FSM do lado destinatário para o rdt2.0 tem um único estado. Quando o pacote chega, o destinatário responde com um ACK ou um NAK, dependendo de o pacote recebido estar ou não corrompido. Na Figura 3.10, a notação `rdt _ recv(rcvpkt) && corrupt(rcvpkt)` corresponde ao evento em que um pacote é recebido e existe um erro.

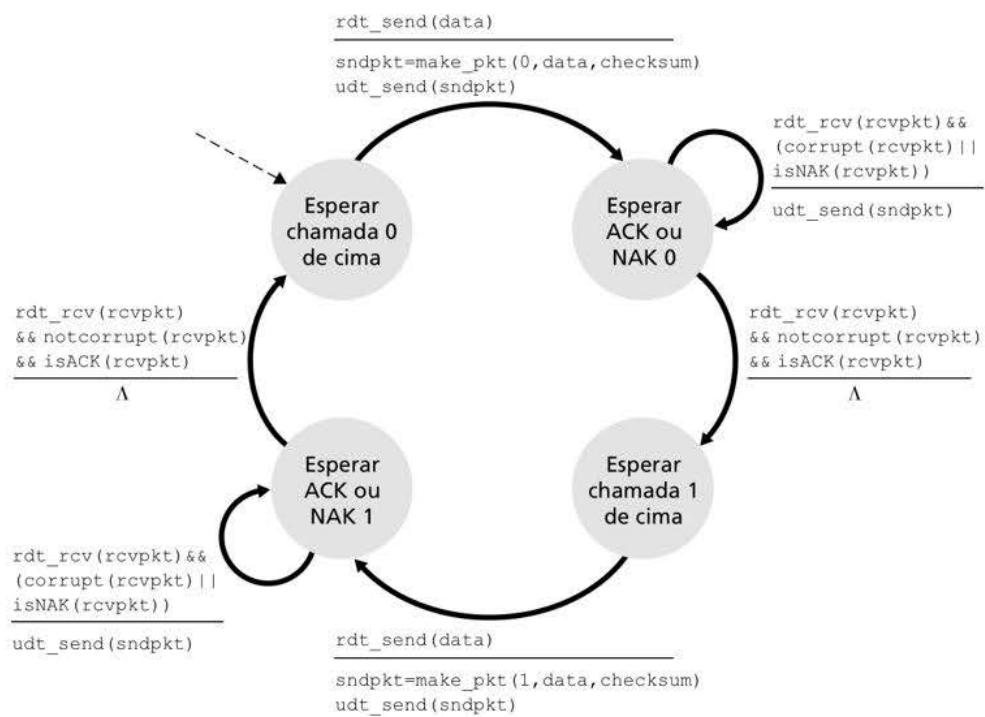
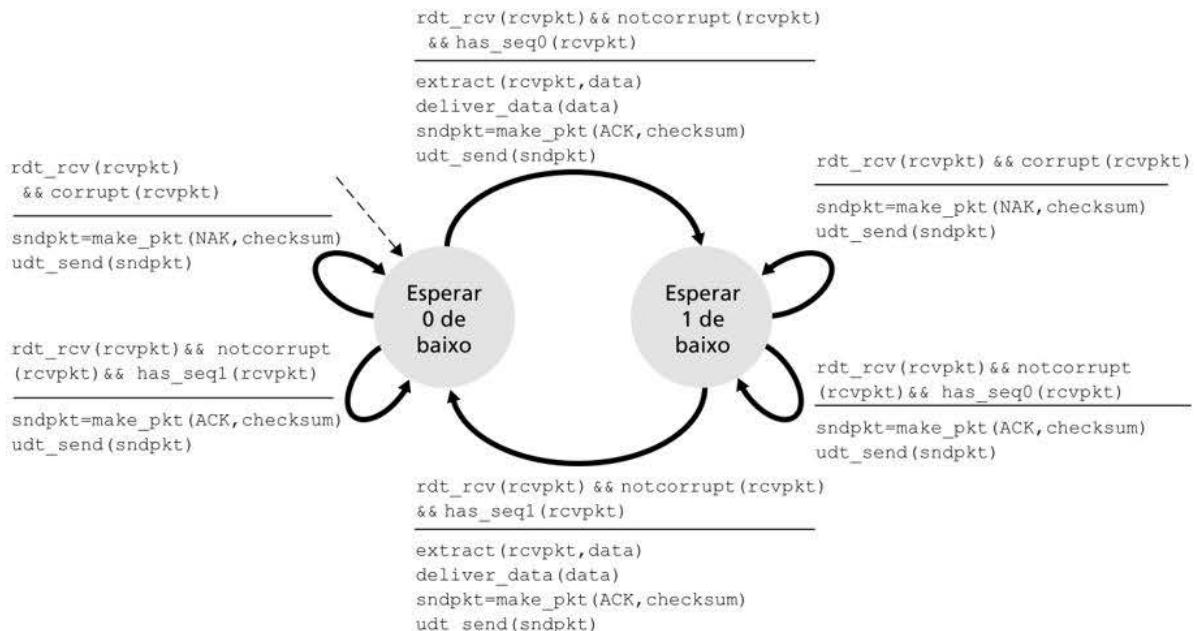
Pode parecer que o protocolo rdt2.0 funciona, mas infelizmente ele tem um defeito fatal. Em especial, ainda não tratamos da possibilidade de o pacote ACK ou NAK estar corrompido! (Antes de continuar, é bom você começar a pensar em como esse problema pode ser resolvido.) Lamentavelmente, nossa pequena omissão não é tão inofensiva quanto possa parecer. No mínimo, precisaremos adicionar aos pacotes ACK/NAK *bits* de soma de verificação para detectar esses erros. A questão mais difícil é como o protocolo deve se recuperar de erros em pacotes ACK ou NAK. Nesse caso, a dificuldade é que, se um ACK ou um NAK estiver corrompido, o remetente não terá como saber se o destinatário recebeu ou não corretamente a última parte de dados transmitidos.

Considere três possibilidades para manipular ACKs ou NAKs corrompidos:

- Para a primeira possibilidade, imagine o que um ser humano faria no cenário do ditado da mensagem. Se quem estiver ditando não entender o “OK” ou o “Repita, por favor” do destinatário, provavelmente perguntará: “O que foi que você disse?” (introduzindo assim um novo tipo de pacote remetente-destinatário em nosso protocolo). O destinatário então repetiria a resposta. Mas e se a frase “O que foi que você disse?” estivesse corrompida? O destinatário, sem ter nenhuma noção se a sentença corrompida era parte do ditado ou um pedido para repetir a última resposta, provavelmente responderia: “O que foi que você disse?”. E então, é claro, essa resposta também poderia estar truncada. É óbvio que estamos entrando em um caminho difícil.
- Uma segunda alternativa é adicionar um número suficiente de *bits* de soma de verificação para permitir que o remetente não só detecte, mas também se recupere de erros de *bits*. Isso resolve o problema imediato para um canal que pode corromper pacotes, mas não perdê-los.
- Uma terceira técnica é o remetente reenviar o pacote de dados corrente quando receber um pacote ACK ou NAK truncado. Esse método, no entanto, introduz **pacotes duplicados** no canal remetente-destinatário. A dificuldade fundamental com pacotes duplicados é que o destinatário não sabe se o último ACK ou NAK que ele próprio enviou foi bem recebido no remetente. Assim, ele não pode saber *a priori* se um pacote que chega contém novos dados ou se é uma retransmissão!

Uma solução simples para esse novo problema (e que é adotada em quase todos os protocolos de transferência de dados existentes, inclusive o TCP) é adicionar um novo campo ao pacote de dados e fazer o remetente numerar seus pacotes de dados colocando um **número de sequência** nesse campo. O destinatário então teria apenas de verificar esse número de sequência para determinar se o pacote recebido é ou não uma retransmissão. Para esse caso simples de protocolo pare e espere, um número de sequência de um *bit* é suficiente, já que permitirá que o destinatário saiba se o remetente está reenviando o pacote previamente transmitido (isso ocorre quando o número de sequência do pacote recebido é o mesmo do último pacote anteriormente recebido) ou um novo pacote (isso ocorre quando o número de sequência muda, indo “para a frente” em progressão aritmética de módulo 2). Como estamos admitindo que esse é um canal que não perde pacotes, os pacotes ACK e NAK em si não precisam indicar o número de sequência do pacote que estão reconhecendo. O remetente sabe que um pacote ACK ou NAK recebido (truncado ou não) foi gerado em resposta ao seu pacote de dados transmitidos mais recentemente.

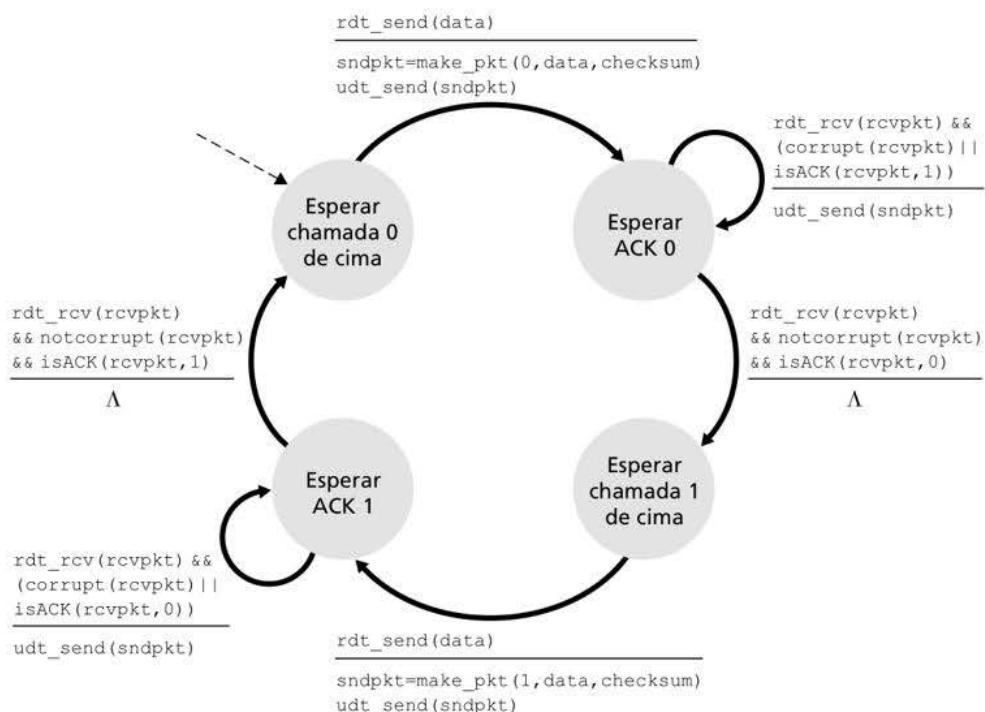
As Figuras 3.11 e 3.12 mostram a descrição da FSM para o rdt2.1, nossa versão corrigida do rdt2.0. Cada rdt2.1 remetente e destinatário da FSM agora tem um número duas vezes maior de estados do que antes. Isso acontece porque o estado do protocolo deve agora refletir se o pacote que está sendo correntemente enviado (pelo remetente) ou aguardado (no destinatário) deveria ter um número de sequência 0 ou 1. Note que as ações nos estados em que um pacote numerado com 0 está sendo enviado ou aguardado são imagens especulares

**Figura 3.11** rdt2.1 remetente.**Figura 3.12** rdt2.1 destinatário.

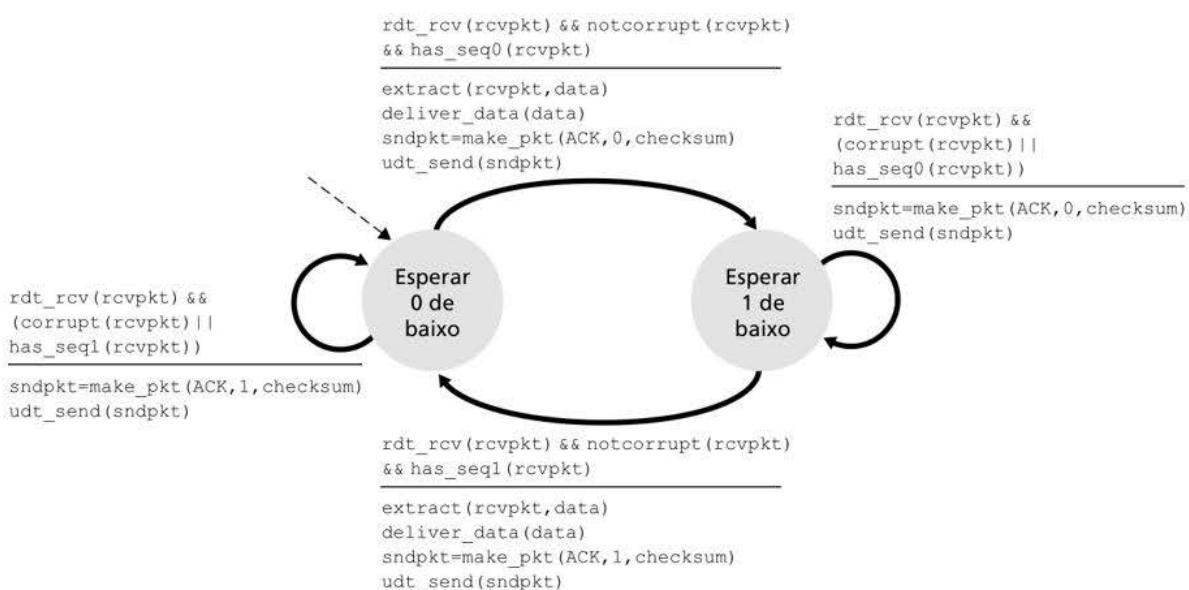
daquelas que devem funcionar quando estiver sendo enviado ou aguardado um pacote numerado com 1; as únicas diferenças têm a ver com a manipulação do número de sequência.

O protocolo rdt2.1 usa tanto o reconhecimento positivo como o negativo do remetente ao destinatário. Quando um pacote fora de ordem é recebido, o destinatário envia um reconhecimento positivo para o pacote que recebeu; quando um pacote corrompido é recebido, ele envia um reconhecimento negativo. Podemos conseguir o mesmo efeito de um pacote

NAK se, em vez de enviarmos um NAK, enviarmos um ACK em seu lugar para o último pacote corretamente recebido. Um remetente que recebe dois ACKs para o mesmo pacote (i.e., **ACKs duplicados**) sabe que o destinatário não recebeu corretamente o pacote seguinte àquele para o qual estão sendo dados dois ACKs. Nossa protocolo de transferência confiável de dados sem NAK para um canal com erros de *bits* é o rdt2.2, mostrado nas Figuras 3.13 e 3.14. Uma modificação sutil entre rdt2.1 e rdt2.2 é que o destinatário agora deve incluir o número de sequência do pacote que está sendo reconhecido por uma mensagem



**Figura 3.13** rdt2.2 remetente.



**Figura 3.14** rdt2.2 destinatário.

ACK (o que é feito incluindo o argumento ACK, 0 ou ACK, 1 em `make_pkt()` na FSM destinatária) e o remetente agora deve verificar o número de sequência do pacote que está sendo reconhecido por uma mensagem ACK recebida (o que é feito incluindo o argumento 0 ou 1 em `isACK()` na FSM remetente).

### Transferência confiável de dados por um canal com perda e com erros de bits: `rdt3.0`

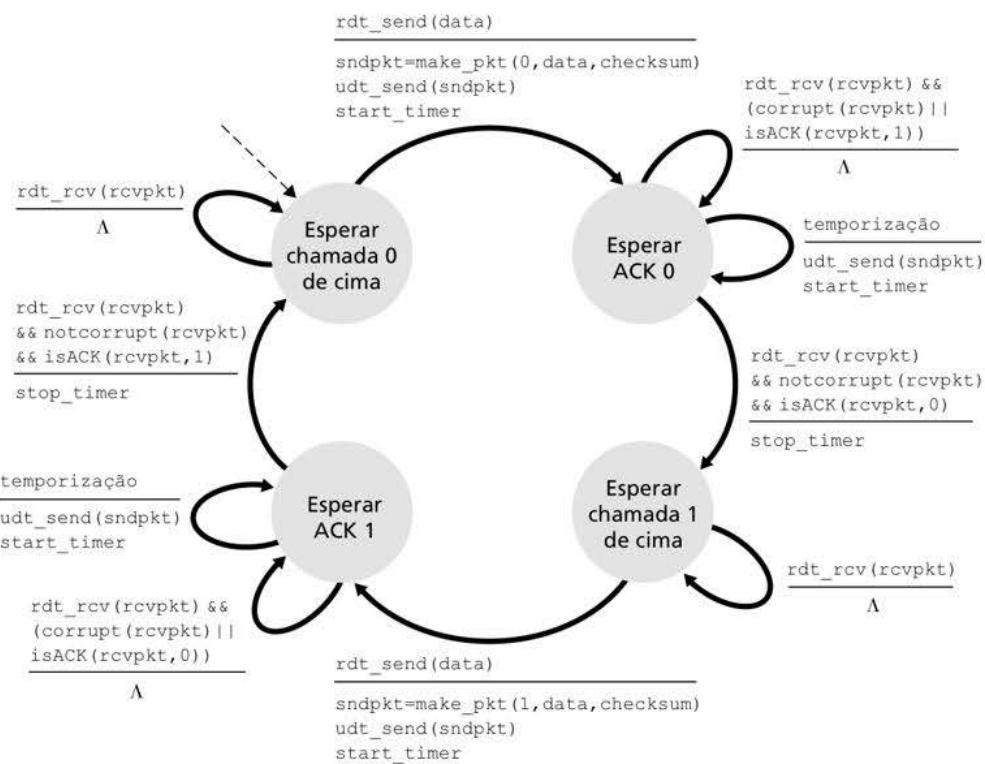
Suponha agora que, além de corromper *bits*, o canal subjacente possa *perder* pacotes, um acontecimento que não é incomum nas redes de computadores de hoje (incluindo a Internet). Duas preocupações adicionais devem agora ser tratadas pelo protocolo: como detectar perda de pacote e o que fazer quando isso ocorre. A utilização de soma de verificação, números de sequência, pacotes ACK e retransmissões – as técnicas já desenvolvidas em `rdt2.2` – nos permitirá atender à última preocupação. Lidar com a primeira preocupação, por sua vez, exigirá a adição de um novo mecanismo de protocolo.

Há muitas abordagens possíveis para lidar com a perda de pacote (e diversas delas serão estudadas nos exercícios ao final deste capítulo). Aqui, atribuiremos ao remetente o encargo de detectar e se recuperar das perdas de pacote. Suponha que o remetente transmita um pacote de dados e que esse pacote, ou o ACK do seu destinatário, seja perdido. Em qualquer um dos casos, nenhuma resposta chegará ao remetente vinda do destinatário. Se o remetente estiver disposto a esperar o tempo suficiente para ter *certeza* de que o pacote foi perdido, ele poderá apenas retransmitir o pacote de dados. É preciso que você se convença de que esse protocolo funciona mesmo.

Mas quanto o remetente precisa esperar para ter certeza de que algo foi perdido? É claro que deve aguardar no mínimo o tempo de um atraso de ida e volta entre ele e o destinatário (o que pode incluir *buffers* em roteadores ou equipamentos intermediários) e mais o tempo que for necessário para processar um pacote no destinatário. Em muitas redes, o atraso máximo para esses piores casos é muito difícil até de estimar, quanto mais saber com certeza. Além disso, o ideal seria que o protocolo se recuperasse da perda de pacotes logo que possível; esperar pelo atraso do pior dos casos pode significar um longo tempo até que a recuperação do erro seja iniciada. Assim, a técnica adotada na prática é a seguinte: o remetente faz uma escolha ponderada de um valor de tempo dentro do qual seria provável, mas não garantido, que a perda tivesse acontecido. Se não for recebido um ACK nesse período, o pacote é retransmitido. Note que, se um pacote sofrer um atraso particularmente longo, o remetente poderá retransmiti-lo mesmo que nem o pacote de dados nem o seu ACK tenham sido perdidos. Isso introduz a possibilidade de **pacotes de dados duplicados** no canal remetente-destinatário. Felizmente, o protocolo `rdt2.2` já dispõe de funcionalidade suficiente (i.e., números de sequência) para tratar dos casos de pacotes duplicados.

Do ponto de vista do remetente, a retransmissão é uma panaceia. O remetente não sabe se um pacote de dados foi perdido, se um ACK foi perdido ou se o pacote ou o ACK apenas estavam muito atrasados. Em todos os casos, a ação é a mesma: retransmitir. Para implementar um mecanismo de retransmissão com base no tempo, é necessário um **temporizador de contagem regressiva** que interrompa o processo remetente após ter decorrido um dado tempo. Assim, será preciso que o remetente possa (1) acionar o temporizador todas as vezes que um pacote for enviado (quer seja a primeira vez, quer seja uma retransmissão), (2) responder a uma interrupção feita pelo temporizador (realizando as ações necessárias) e (3) parar o temporizador.

A Figura 3.15 mostra a FSM remetente para o `rdt3.0`, um protocolo que transfere dados de modo confiável por um canal que pode corromper ou perder pacotes; nos “Exercícios de fixação”, pediremos a você que projete a FSM destinatária para `rdt3.0`. A Figura 3.16 mostra como o protocolo funciona sem pacotes perdidos ou atrasados e como manipula pacotes de dados perdidos. Nessa figura, a passagem do tempo ocorre do topo do diagrama para baixo. Note que o instante de recebimento de um pacote tem de ser posterior ao instante



**Figura 3.15** rdt3.0 remetente.

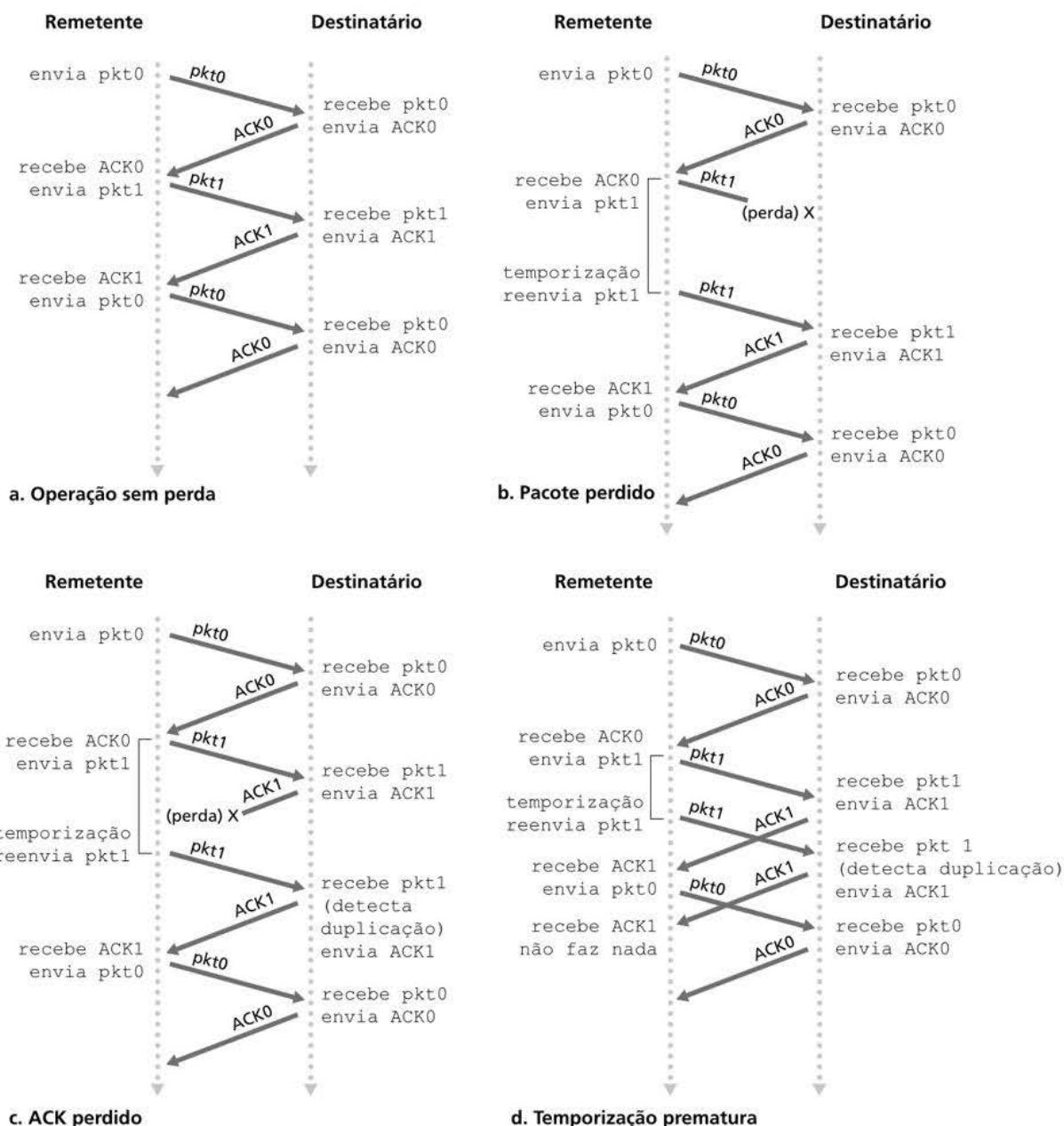
de envio de um pacote, como resultado de atrasos de transmissão e de propagação. Nas Figuras 3.16(b-d), os colchetes do lado remetente indicam os instantes em que o temporizador foi acionado e, mais tarde, os instantes em que ele parou. Vários dos aspectos mais sutis desse protocolo são examinados nos exercícios ao final deste capítulo. Como os números de sequência se alternam entre 0 e 1, o protocolo rdt3.0 às vezes é conhecido como **protocolo bit alternante**.

Agora já reunimos os elementos fundamentais de um protocolo de transferência de dados. Somas de verificação, números de sequência, temporizadores e pacotes de reconhecimento negativo e positivo – cada um desempenha um papel crucial e necessário na operação do protocolo. Temos agora em funcionamento um protocolo de transferência confiável de dados!

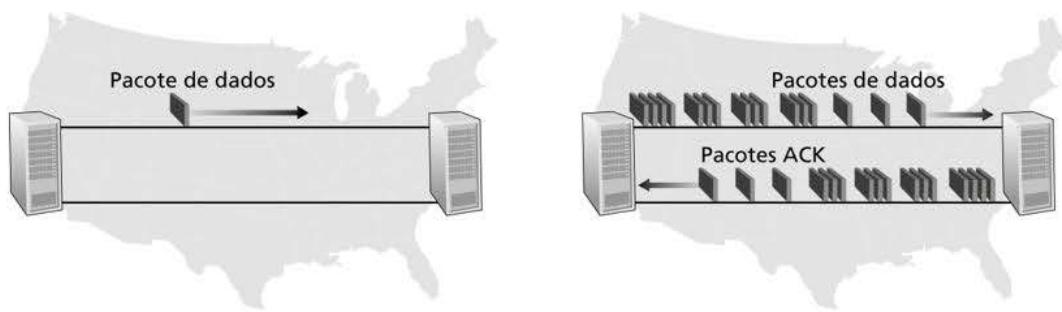
### 3.4.2 Protocolos de transferência confiável de dados com paralelismo

O protocolo rdt3.0 é correto em termos funcionais, mas é pouco provável que alguém fique contente com o desempenho dele, ainda mais nas redes de alta velocidade de hoje. No coração do problema do desempenho do rdt3.0 está o fato de ele ser um protocolo do tipo pare e espere.

Para avaliar o impacto sobre o desempenho causado pelo comportamento “pare e espere”, considere um caso ideal de dois hospedeiros, um localizado na Costa Oeste dos Estados Unidos e outro na Costa Leste, como mostra a Figura 3.17. O atraso de propagação de ida e volta à velocidade da luz (RTT, do inglês *round-trip time*) entre esses dois sistemas finais é de cerca de 30 milissegundos. Suponha que eles estejam conectados por um canal com capacidade de transmissão,  $R$ , de 1 Gbit/s ( $10^9$  bits por segundo). Para um tamanho de pacote,  $L$ ,



**Figura 3.16** Operação do rdt3.0, o protocolo bit alternante.

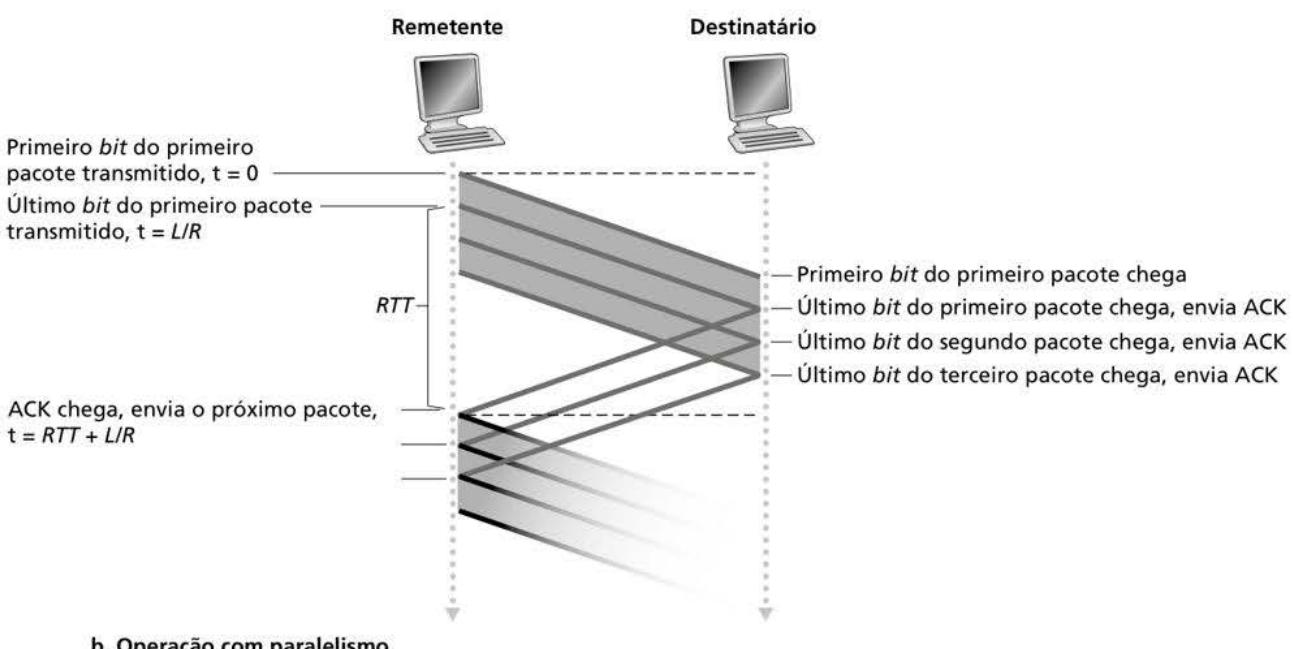
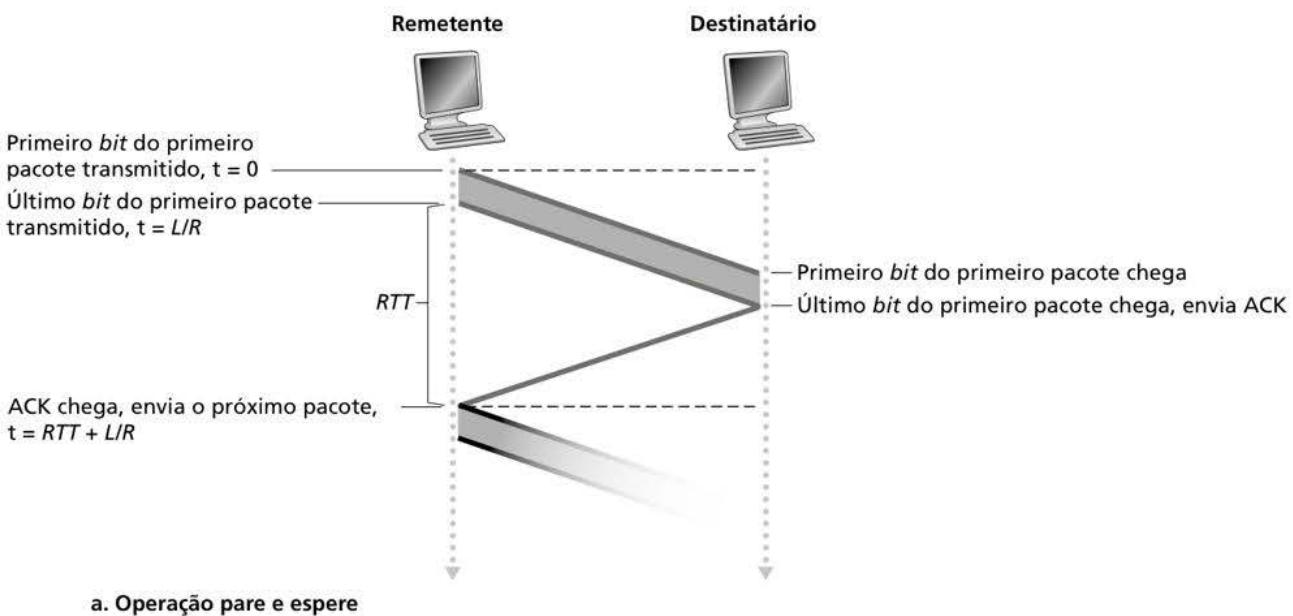


**Figura 3.17** Protocolo pare e espere versus protocolo com paralelismo.

de mil *bytes* (8 mil *bits*), incluindo o campo de cabeçalho e também o de dados, o tempo necessário para realmente transmitir o pacote para o enlace de 1 Gbit/s é:

$$d_{\text{trans}} = \frac{L}{R} = \frac{8.000 \text{ bits}}{10^9 \text{ bits/s}} = 8 \text{ microsegundos}$$

A Figura 3.18(a) mostra que, com nosso protocolo pare e espere, se o remetente começar a enviar o pacote em  $t = 0$ , então em  $t = L/R = 8 \mu\text{s}$ , o último *bit* entrará no canal do lado remetente. O pacote então faz sua jornada de 15 ms atravessando o país, com o último *bit* do pacote emergindo no destinatário em  $t = \text{RTT}/2 + L/R = 15,008 \text{ ms}$ . Supondo, para



**Figura 3.18** Envio com pare e espere e com paralelismo.

simplificar, que pacotes ACK sejam extremamente pequenos (para podermos ignorar seu tempo de transmissão) e que o destinatário pode enviar um ACK logo que receber o último *bit* de um pacote de dados, o ACK emergirá de volta no remetente em  $t = \text{RTT} + L/R = 30,008 \text{ ms}$ . Nesse ponto, o remetente agora poderá transmitir a próxima mensagem. Assim, em 30,008 ms, o remetente esteve enviando por apenas 0,008 ms. Se definirmos a **utilização** do remetente (ou do canal) como a fração de tempo em que o remetente está realmente ocupado enviando *bits* para dentro do canal, a análise da Figura 3.18(a) mostra que o protocolo pare e espere tem uma utilização do remetente  $U_{\text{remet}}$  bastante desanimadora, de:

$$U_{\text{remet}} = \frac{L/R}{\text{RTT} + L/R} = \frac{0,008}{30,008} = 0,00027$$

Portanto, o remetente ficou ocupado apenas 2,7 centésimos de 1% do tempo! Visto de outra maneira, ele só foi capaz de enviar 1.000 *bytes* em 30,008 milissegundos, uma vazão efetiva de apenas 267 kbit/s – mesmo estando disponível um enlace de 1 *gigabit* por segundo! Imagine o infeliz administrador de redes que acabou de pagar uma fortuna para ter capacidade de enlace da ordem de *gigabits*, mas consegue uma vazão de apenas 267 Kb por segundo! Este é um exemplo radical de como protocolos de rede podem limitar as capacidades oferecidas pelo *hardware* subjacente de rede. Além disso, desprezamos também os tempos de processamento de protocolo das camadas inferiores no remetente e no destinatário, bem como os atrasos de processamento e de fila que ocorreriam em quaisquer roteadores intermediários existentes entre o remetente e o destinatário. Incluir esses efeitos serviria apenas para aumentar ainda mais o atraso e piorar ainda mais o fraco desempenho.

A solução para esse problema de desempenho em especial é simples: em vez de operar em modo pare e espere, o remetente é autorizado a enviar vários pacotes sem esperar por reconhecimentos, como mostra a Figura 3.17(b). A Figura 3.18(b) mostra que, se um remetente for autorizado a transmitir três pacotes antes de ter de esperar por reconhecimentos, sua utilização será triplicada. Uma vez que os muitos pacotes em trânsito entre remetente e destinatário podem ser visualizados como se estivessem enchendo uma tubulação, essa técnica é conhecida, em inglês, como **pipelining\*** (tubulação). O paralelismo gera as seguintes consequências para protocolos de transferência confiável de dados:

- A faixa de números de sequência tem de ser ampliada, pois cada pacote em trânsito (sem contar as retransmissões) precisa ter um número de sequência exclusivo, e pode haver vários pacotes não reconhecidos em trânsito.
- Os lados remetente e destinatário dos protocolos podem ter de reservar *buffers* para mais de um pacote. No mínimo, o remetente terá de providenciar *buffers* para pacotes que foram transmitidos, mas que ainda não foram reconhecidos. O *buffer* de pacotes recebidos sem erro pode também ser necessário no destinatário, como discutiremos a seguir.
- A faixa de números de sequência necessária e as necessidades de *buffer* dependerão da maneira como um protocolo de transferência de dados responde a pacotes perdidos, corrompidos e demasiadamente atrasados. Duas abordagens básicas em relação à recuperação de erros com paralelismo podem ser identificadas: **Go-Back-N** e **repetição seletiva**.

### 3.4.3 Go-Back-N (GBN)

Em um **protocolo Go-Back-N (GBN)**, o remetente é autorizado a transmitir múltiplos pacotes (se disponíveis) sem esperar por um reconhecimento, mas fica limitado a ter não mais do que algum número máximo permitido,  $N$ , de pacotes não reconhecidos na “tubulação”. Nesta seção, descreveremos o protocolo GBN com detalhes. Mas antes de continuar a leitura, convidamos você para se divertir com a animação GBN (que é incrível!) no *site* de apoio do livro.

---

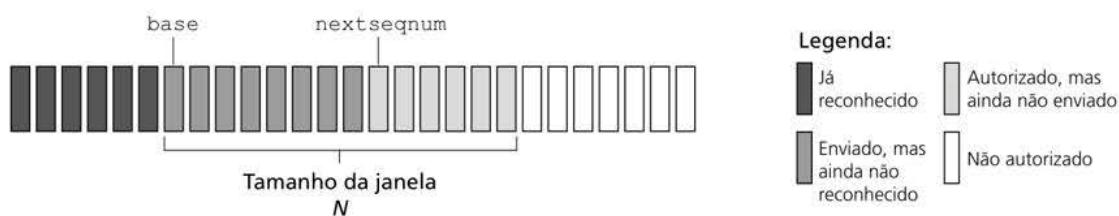
\*N. de T.: Porém, como essa expressão é difícil de traduzir para o português, preferimos usar “paralelismo”, embora a transmissão de dados seja de fato sequencial.

A Figura 3.19 mostra a visão que o remetente tem da faixa de números de sequência em um protocolo GBN. Se definirmos `base` como o número de sequência do mais antigo pacote não reconhecido e `nextseqnum` como o menor número de sequência não utilizado (i.e., o número de sequência do próximo pacote a ser enviado), então quatro intervalos na faixa de números de sequência poderão ser identificados. Os números de sequência no intervalo  $[0, \text{base}-1]$  correspondem aos pacotes que já foram transmitidos e reconhecidos. O intervalo  $[\text{base}, \text{nextseqnum}-1]$  corresponde aos pacotes enviados, mas que ainda não foram reconhecidos. Os números de sequência no intervalo  $[\text{nextseqnum}, \text{base}+N-1]$  podem ser usados para pacotes que podem ser enviados imediatamente, caso cheguem dados vindos da camada superior. Por fim, números de sequência maiores ou iguais a  $\text{base}+N$  não podem ser usados até que um pacote não reconhecido que esteja pendente seja reconhecido (especificamente, o pacote cujo número de sequência é `base`).

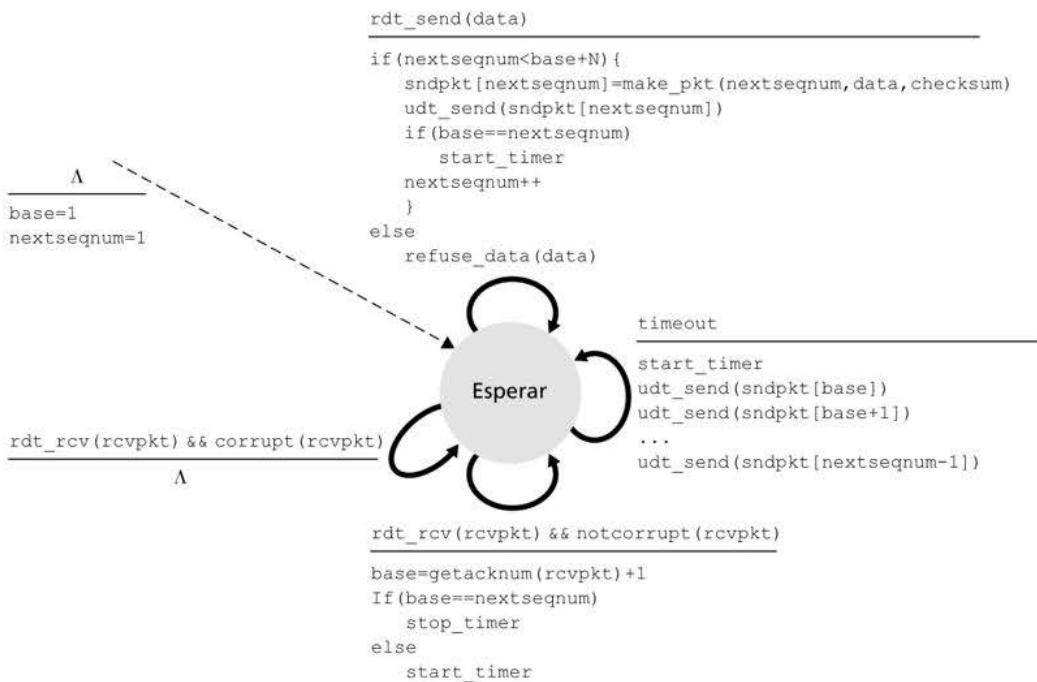
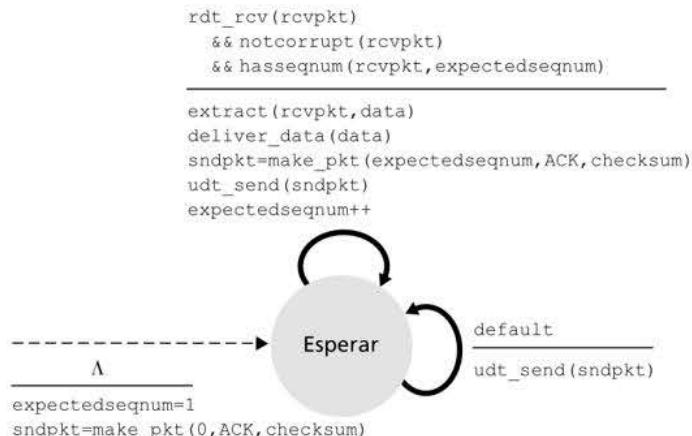
Como sugere a Figura 3.19, a faixa de números de sequência permitidos para pacotes transmitidos, porém ainda não reconhecidos, pode ser vista como uma janela de tamanho  $N$  sobre a faixa de números de sequência. À medida que o protocolo opera, a janela se desloca para a frente sobre o espaço de números de sequência. Por essa razão,  $N$  é muitas vezes denominado **tamanho de janela**, e o protocolo GBN em si, **protocolo de janela deslizante (do inglês sliding-window protocol)**. É possível que você esteja pensando que razão teríamos, primeiro, para limitar o número de pacotes pendentes não reconhecidos a um valor  $N$ . Por que não permitir um número ilimitado deles? Veremos na Seção 3.5 que o controle de fluxo é uma das razões para impor um limite ao remetente. Examinaremos outra razão para isso na Seção 3.7, quando estudarmos o controle de congestionamento do TCP.

Na prática, o número de sequência de um pacote é carregado em um campo de comprimento fixo no cabeçalho do pacote. Se  $k$  for o número de bits no campo de número de sequência do pacote, a faixa de números de sequência será então  $[0, 2^k - 1]$ . Com uma faixa finita de números de sequência, toda a aritmética que envolver números de sequência deverá ser feita usando aritmética de módulo  $2^k$ . (Em outras palavras, o espaço do número de sequência pode ser imaginado como um anel de tamanho  $2^k$ , em que o número de sequência  $2^k - 1$  é seguido de imediato pelo número de sequência 0.) Lembre-se de que rdt3.0 tem um número de sequência de 1 bit e uma faixa de números de sequência de  $[0,1]$ . Vários problemas ao final deste capítulo tratam das consequências de uma faixa finita de números de sequência. Veremos na Seção 3.5 que o TCP tem um campo de número de sequência de 32 bits, em que os números de sequência do TCP contam bytes na cadeia de bytes na sequência de transmissão, em vez de pacotes.

As Figuras 3.20 e 3.21 descrevem uma FSM estendida dos lados remetente e destinatário de um protocolo GBN baseado em ACK, mas sem NAK. Referimo-nos a essa descrição de FSM como *FSM estendida* porque adicionamos variáveis (semelhantes às variáveis de linguagem de programação) para `base` e `nextseqnum`; também adicionamos operações sobre essas variáveis e ações condicionais que as envolvem. Note que a especificação da FSM estendida agora está começando a parecer um pouco com uma especificação de linguagem de programação. Bochman (1984) fornece um excelente levantamento sobre extensões adicionais às técnicas FSM, bem como sobre outras técnicas para especificação de protocolos baseadas em linguagens.



**Figura 3.19** Visão do remetente para os números de sequência no protocolo Go-Back-N.

**Figura 3.20** Descrição da FSM estendida do remetente GBN.**Figura 3.21** Descrição da FSM estendida do destinatário GBN.

O remetente GBN deve responder a três tipos de eventos:

- *Chamada vinda de cima.* Quando `rdt_send()` é chamado de cima, o remetente primeiro verifica se a janela está cheia, isto é, se há  $N$  pacotes pendentes não reconhecidos. Se a janela não estiver cheia, um pacote é criado e enviado, e as variáveis são adequadamente atualizadas. Se a janela estiver cheia, o remetente apenas devolve os dados à camada superior – uma indicação implícita de que a janela está cheia. Presume-se que a camada superior então teria de tentar outra vez mais tarde. Em uma execução real, o remetente muito provavelmente teria colocado esses dados em um *buffer* (mas não os teria enviado imediatamente) ou teria um mecanismo de sincronização (p. ex., um semáforo ou uma *flag*) que permitiria que a camada superior chamassem `rdt_send()` apenas quando as janelas não estivessem cheias.
- *Recebimento de um ACK.* Em nosso protocolo GBN, um reconhecimento de pacote com número de sequência  $n$  seria tomado como um **reconhecimento cumulativo**, indicando

que todos os pacotes com número de sequência até e inclusive  $n$  tinham sido corretamente recebidos no destinatário. Voltaremos a esse assunto em breve, quando examinarmos o lado destinatário do GBN.

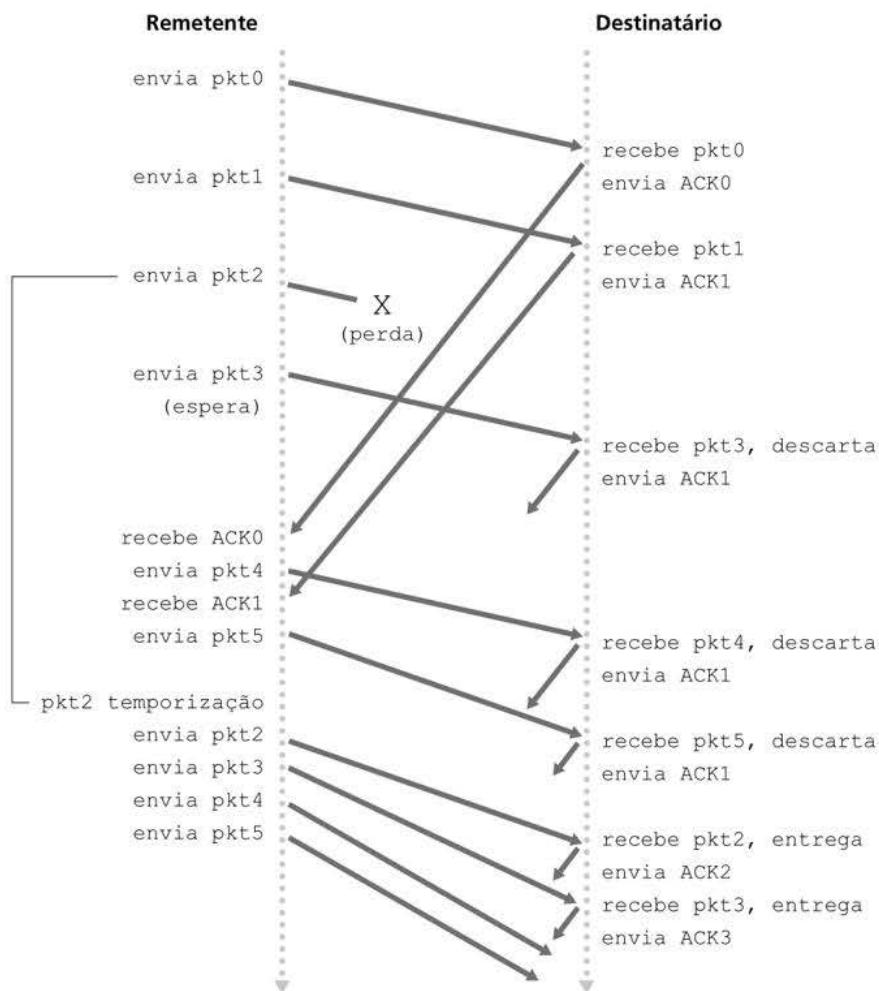
- Um evento de esgotamento de temporização (*timeout*). O nome “Go-Back-N” deriva do comportamento do remetente em relação a pacotes perdidos ou demasiadamente atrasados. Como no protocolo pare e espere, um temporizador é usado para recuperar a perda de dados ou reconhecer pacotes. Se ocorrer o esgotamento da temporização, o remetente reenvia *todos* os pacotes que tinham sido previamente enviados, mas que ainda não tinham sido reconhecidos. Nossa remetente da Figura 3.20 usa apenas um único temporizador, que pode ser imaginado como um temporizador para o mais antigo pacote já transmitido, porém ainda não reconhecido. Se for recebido um ACK e ainda houver pacotes adicionais transmitidos mas ainda não reconhecidos, o temporizador será reiniciado. Se não houver nenhum pacote pendente não reconhecido, o temporizador será desligado.

As ações do destinatário no GBN também são simples. Se um pacote com número de sequência  $n$  for recebido corretamente e estiver na ordem (i.e., os últimos dados entregues à camada superior vieram de um pacote com número de sequência  $n - 1$ ), o destinatário enviará um ACK para o pacote  $n$  e entregará a parte de dados do pacote à camada superior. Em todos os outros casos, o destinatário descarta o pacote e reenvia um ACK para o mais recente que foi recebido na ordem correta. Dado que são entregues à camada superior um por vez, se o pacote  $k$  tiver sido recebido e entregue, então todos os pacotes com número de sequência menores do que  $k$  também terão sido entregues. Assim, o uso de reconhecimentos cumulativos é uma escolha natural para o GBN.

Em nosso protocolo GBN, o destinatário descarta os pacotes que chegam fora de ordem. Embora pareça bobagem e perda de tempo descartar um pacote corretamente recebido (mas fora de ordem), existem justificativas para isso. Lembre-se de que o destinatário deve entregar dados na ordem certa à camada superior. Suponha agora que o pacote  $n$  esteja sendo esperado, mas quem chega é o pacote  $n + 1$ . Como os dados devem ser entregues na ordem certa, o destinatário *poderia* conservar o pacote  $n + 1$  no *buffer* (salvá-lo) e entregá-lo à camada superior mais tarde, após ter recebido o pacote  $n$ . Contudo, se o pacote  $n$  for perdido,  $n$  e  $n + 1$  serão ambos por fim retransmitidos como resultado da regra de retransmissão do GBN no remetente. Assim, o destinatário pode apenas descartar o pacote  $n + 1$ . A vantagem dessa abordagem é a simplicidade da manipulação de *buffers* no destinatário – ele não precisa *colocar* no *buffer* *nenhum* pacote que esteja fora de ordem. Desse modo, enquanto o remetente deve manter os limites superior e inferior de sua janela e a posição de *nextseqnum* dentro dela, a única informação que o destinatário precisa manter é o número de sequência do próximo pacote esperado conforme a ordem. Esse valor é retido na variável *expecte-dseqnum* mostrada na FSM destinatária da Figura 3.21. Claro, a desvantagem de jogar fora um pacote recebido corretamente é que a retransmissão subsequente desse pacote pode ser perdida ou ficar truncada, caso em que ainda mais retransmissões seriam necessárias.

A Figura 3.22 mostra a operação do protocolo GBN para o caso de um tamanho de janela de quatro pacotes. Em razão da limitação do tamanho dessa janela, o remetente envia os pacotes de 0 a 3, mas, em seguida, tem de esperar que um ou mais desses pacotes sejam reconhecidos antes de prosseguir. À medida que cada ACK sucessivo (p. ex., ACK0 e ACK1) é recebido, a janela se desloca para a frente e o remetente pode transmitir um novo pacote (pkt4 e pkt5, respectivamente). Do lado destinatário, o pacote 2 é perdido. Desse modo, verifica-se que os pacotes 3, 4 e 5 estão fora de ordem e, portanto, são descartados.

Antes de encerrarmos nossa discussão sobre o GBN, devemos ressaltar que uma implementação desse protocolo em uma pilha de protocolo provavelmente seria estruturada de modo semelhante à da FSM estendida da Figura 3.20. A implementação também seria estruturada sob a forma de vários procedimentos que implementam as ações a serem executadas em resposta aos vários eventos que podem ocorrer. Nessa **programação baseada em eventos**, os vários procedimentos são chamados (invocados) por outros procedimentos presentes



**Figura 3.22** Go-Back-N em operação.

na pilha de protocolo ou como resultado de uma interrupção. No remetente, seriam: (1) uma chamada pela entidade da camada superior invocando `rdt_send()`, (2) uma interrupção pelo temporizador e (3) uma chamada pela camada inferior invocando `rdt_rcv()` quando chega um pacote. Os exercícios de programação ao final deste capítulo lhe darão a chance de executar de verdade essas rotinas em um ambiente de rede simulado, mas realista.

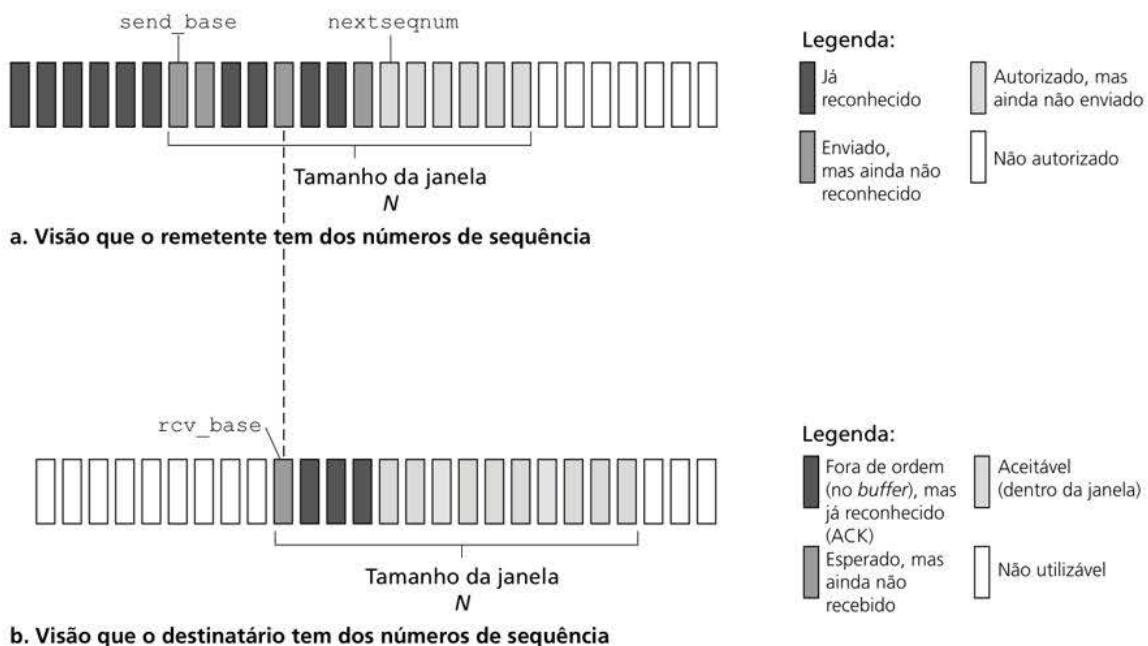
Salientamos que o protocolo GBN incorpora quase todas as técnicas que encontraremos quando estudarmos, na Seção 3.5, os componentes de transferência confiável de dados do TCP. Essas técnicas incluem a utilização de números de sequência, reconhecimentos cumulativos, somas de verificação e uma operação de esgotamento de temporização/retransmissão.

#### 3.4.4 Repetição seletiva (SR)

O protocolo GBN permite que o remetente potencialmente “encha a rede” com pacotes na Figura 3.17, evitando, assim, os problemas de utilização de canal observados em protocolos do tipo pare e espere. Há, contudo, casos em que o próprio GBN sofre com problemas de desempenho. Em especial, quando o tamanho da janela e o produto entre o atraso e a largura de banda são grandes, pode haver muitos pacotes pendentes na rede. Assim, um único erro de pacote pode fazer o GBN retransmitir um grande número de pacotes – muitos deles sem necessidade. À medida que aumenta a probabilidade de erros no canal, a rede pode ficar

lotada com essas retransmissões desnecessárias. Imagine se, em nosso cenário de conversa, toda vez que uma palavra fosse pronunciada de maneira truncada as outras mil que a circundam (p. ex., um tamanho de janela de mil palavras) tivessem de ser repetidas. A conversa sofreria atrasos em virtude de todas essas palavras reiteradas.

Como o próprio nome sugere, protocolos de repetição seletiva (SR, do inglês *selective-repeat*) evitam retransmissões desnecessárias porque fazem o remetente retransmitir apenas os pacotes suspeitos de terem sido recebidos com erro (i.e., que foram perdidos ou corrompidos) no destinatário. Essa retransmissão individual, só quando necessária, exige que o destinatário reconheça *individualmente* os pacotes recebidos de modo correto. Uma janela de tamanho  $N$  será usada novamente para limitar o número de pacotes pendentes não reconhecidos dentro da rede. Contudo, ao contrário do GBN, o remetente já terá recebido ACKs para alguns dos pacotes na janela. A Figura 3.23 mostra a visão que o protocolo de SR remetente tem do espaço do número de sequência; a Figura 3.24 detalha as várias ações executadas pelo protocolo SR remetente.



**Figura 3.23** Visões que os protocolos SR remetente e destinatário têm do espaço de número de sequência.

1. *Dados recebidos de cima*. Quando são recebidos dados de cima, o protocolo SR remetente verifica o próximo número de sequência disponível para o pacote. Se o número de sequência está dentro da janela do remetente, os dados são empacotados e enviados; do contrário, eles são armazenados ou devolvidos à camada superior para transmissão posterior, como acontece no GBN.
2. *Esgotamento de temporização*. Novamente são usados temporizadores para proteção contra perda de pacotes. Contudo, cada pacote agora deve ter seu próprio temporizador lógico, já que apenas um pacote será transmitido quando a temporização se esgotar. Um único hardware de temporizador pode ser usado para emular a operação de múltiplos temporizadores lógicos (Varghese, 1997).
3. *ACK recebido*. Se for recebido um ACK, o SR remetente marcará aquele pacote como recebido, contanto que esteja na janela. Se o número de sequência do pacote for igual a `send_base`, a base da janela se deslocará para a frente até o pacote não reconhecido que tiver o menor número de sequência. Se a janela se deslocar e houver pacotes não transmitidos com números de sequência que agora caem dentro da janela, esses pacotes serão transmitidos.

**Figura 3.24** Eventos e ações do protocolo SR remetente.

O protocolo SR destinatário reconhecerá um pacote corretamente recebido esteja ele ou não na ordem certa. Pacotes fora de ordem ficam no *buffer* até que todos os faltantes (i.e., os que têm números de sequência menores) sejam recebidos, quando então um conjunto de pacotes poderá ser entregue à camada superior na ordem correta. A Figura 3.25 apresenta as várias ações realizadas pelo protocolo SR destinatário. A Figura 3.26 mostra um exemplo de operação do protocolo SR quando ocorre perda de pacotes. Note que, nessa figura, o destinatário de início armazena os pacotes 3, 4 e 5 e os entrega junto com o pacote 2 à camada superior, quando o pacote 2 é enfim recebido.

É importante notar que na etapa 2 da Figura 3.25 o destinatário reconhece novamente (em vez de ignorar) pacotes já recebidos com certos números de sequência que estão *abaixo* da atual base da janela. É bom que você se convença de que esse reconhecimento duplo é de fato necessário. Dados os espaços dos números de sequência do remetente e do destinatário na Figura 3.23, por exemplo, se não houver ACK para pacote com número `send_base` propagando-se do destinatário ao remetente, este acabará retransmitindo o pacote `send_base`, embora esteja claro (para nós, e não para o remetente!) que o destinatário já o recebeu. Caso o destinatário não o reconhecesse, a janela do remetente jamais se deslocaria para a frente! Esse exemplo ilustra um importante aspecto dos protocolos SR (e também de muitos outros): o remetente e o destinatário nem sempre têm uma visão idêntica do que foi recebido corretamente e do que não foi. Para protocolos SR, isso significa que as janelas do remetente e do destinatário nem sempre coincidirão.

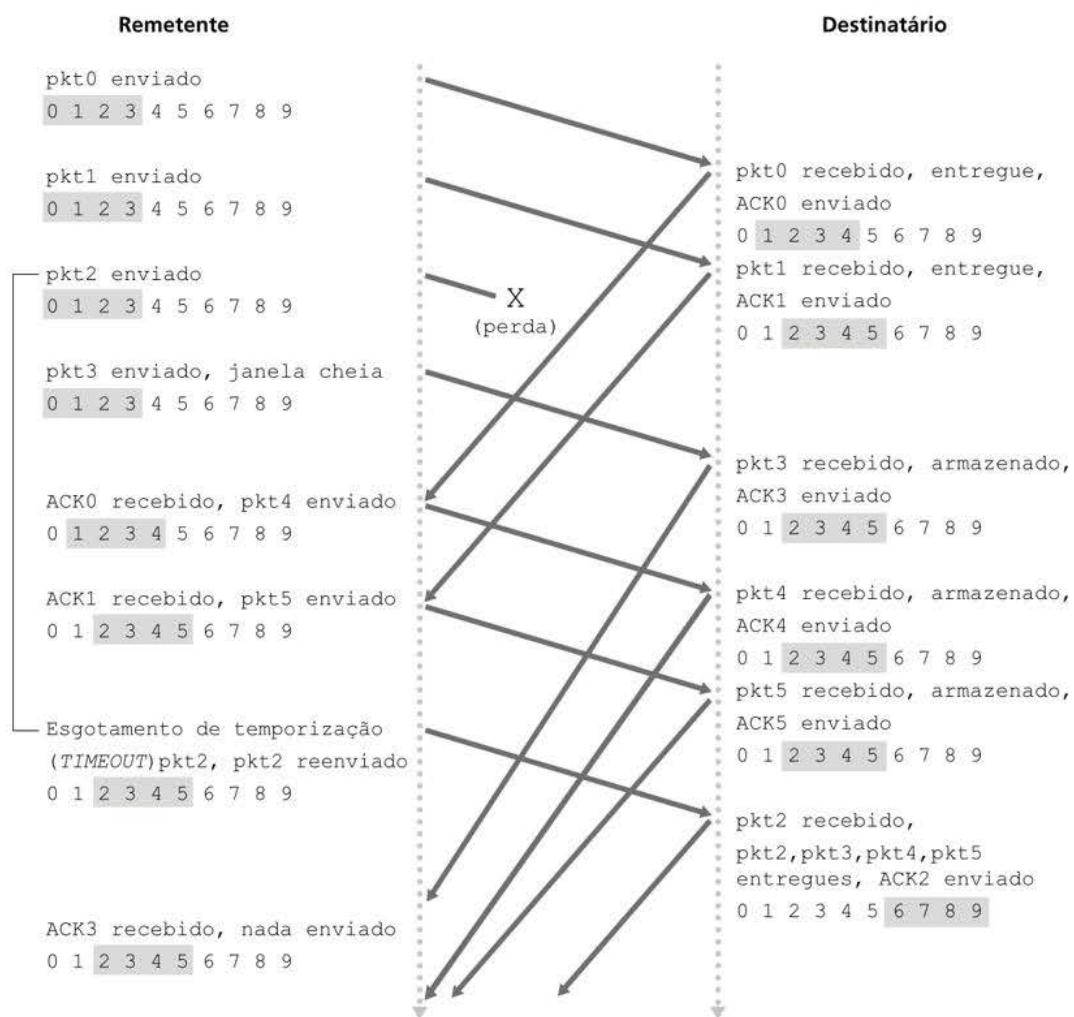
A falta de sincronização entre as janelas do remetente e do destinatário tem importantes consequências quando nos defrontamos com a realidade de uma faixa finita de números de sequência. Considere o que poderia acontecer, por exemplo, com uma faixa finita de quatro números de sequência de pacotes (0, 1, 2, 3) e um tamanho de janela de três. Suponha que os pacotes de 0 a 2 sejam transmitidos, recebidos e reconhecidos corretamente no destinatário. Nesse ponto, a janela do destinatário está sobre o quarto, o quinto e o sexto pacotes, que têm os números de sequência 3, 0 e 1, respectivamente. Agora, considere dois cenários. No primeiro, mostrado na Figura 3.27(a), os ACKs para os três primeiros pacotes foram perdidos e o remetente os retransmite. Assim, o que o destinatário recebe em seguida é um pacote com o número de sequência 0 – uma cópia do primeiro pacote enviado.

No segundo cenário, mostrado na Figura 3.27(b), os ACKs para os três primeiros pacotes foram entregues de modo correto. Assim, o remetente desloca sua janela para a frente e envia o quarto, o quinto e o sexto pacotes com os números de sequência 3, 0 e 1, respectivamente. O pacote com o número de sequência 3 é perdido, mas o pacote com o número de sequência 0 chega – um pacote que contém dados *novos*.

Agora, na Figura 3.27, considere o ponto de vista do destinatário, que tem uma cortina imaginária entre o remetente e ele, já que o destinatário não pode “ver” as ações executadas

1. Pacote com número de sequência no intervalo  $[rcv\_base, rcv\_base+N-1]$  foi corretamente recebido. Nesse caso, o pacote recebido cai dentro da janela do destinatário e um pacote ACK seletivo é devolvido ao remetente. Se o pacote não tiver sido recebido anteriormente, irá para o *buffer*. Se esse pacote tiver um número de sequência igual à base da janela do destinatário (`rcv_base` na Figura 3.22), então ele e quaisquer outros pacotes armazenados no *buffer* e numerados consecutivamente (começando com `rcv_base`) serão entregues à camada superior. A janela do destinatário é então deslocada para a frente de acordo com o número de pacotes entregues à camada superior. Como exemplo, considere a Figura 3.26. Quando um pacote com número de sequência `rcv_base=2` é recebido, ele e os pacotes 3, 4 e 5 podem ser entregues à camada superior.
2. Pacote com número de sequência no intervalo  $[rcv\_base-N, rcv\_base-1]$  foi corretamente recebido. Nesse caso, um ACK deve ser gerado mesmo que esse pacote já tenha sido reconhecido pelo destinatário.
3. Qualquer outro. Ignore o pacote.

**Figura 3.25** Eventos e ações do protocolo SR destinatário.



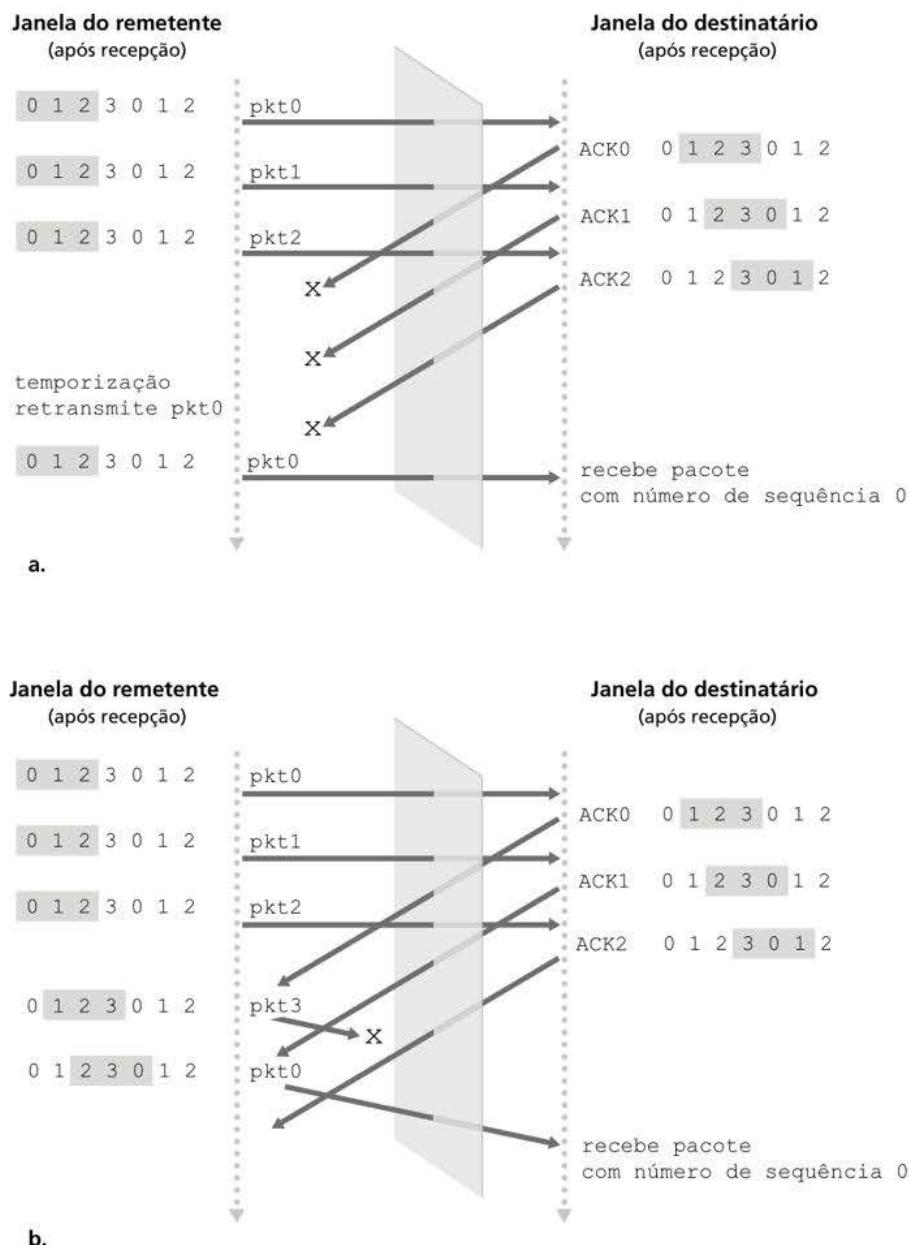
**Figura 3.26** Operação SR.

pelo remetente. Tudo o que o destinatário observa é a sequência de mensagens que ele recebe do canal e envia para o canal. No que lhe concerne, os dois cenários da Figura 3.27 são *idênticos*. Não há um modo de distinguir a retransmissão do primeiro pacote da transmissão original do quinto pacote. Fica claro que um tamanho de janela que seja igual ao tamanho do espaço de numeração sequencial menos 1 não vai funcionar. Mas qual deve ser o tamanho da janela? Um problema ao final deste capítulo pede que você demonstre que o tamanho pode ser menor ou igual à metade do tamanho do espaço de numeração sequencial para os protocolos SR.

No site de apoio do livro, você encontrará uma animação que ilustra a operação do protocolo SR. Tente realizar os mesmos experimentos feitos com a animação GBN. Os resultados combinam com o que você espera?

Isso encerra nossa discussão sobre protocolos de transferência confiável de dados. Percorremos um *longo* caminho e apresentamos numerosos mecanismos que, juntos, proveem transferência confiável de dados. A Tabela 3.1 resume esses mecanismos. Agora que já vimos todos eles em operação e podemos enxergar “o quadro geral”, aconselhamos que você leia novamente esta seção para perceber como esses mecanismos foram adicionados pouco a pouco, de modo a abordar modelos (realistas) de complexidade crescente do canal que conecta o remetente ao destinatário ou para melhorar o desempenho dos protocolos.

Encerraremos nossa explanação considerando uma premissa remanescente em nosso modelo de canal subjacente. Lembre-se de que admitimos que pacotes não podem ser



**Figura 3.27** Dilema do remetente SR com janelas muito grandes: um novo pacote ou uma retransmissão?

reordenados dentro do canal entre o remetente e o destinatário. Esta é uma premissa em geral razoável quando o remetente e o destinatário estão conectados por um único fio físico. Contudo, quando o “canal” que conecta os dois é uma rede, pode ocorrer reordenação de pacotes. Uma manifestação da reordenação de pacotes é que podem aparecer cópias antigas de um pacote com número de sequência ou de reconhecimento  $x$ , mesmo que nem a janela do remetente nem a do destinatário contenham  $x$ . Com a reordenação de pacotes, podemos considerar que o canal usa armazenamento de pacotes e emite-os espontaneamente em algum momento *qualquer* do futuro. Como números de sequência podem ser reutilizados, devemos tomar algum cuidado para nos prevenir contra esses pacotes duplicados. A abordagem adotada na prática é garantir que um número de sequência não seja reutilizado até que o remetente esteja “certo” de que nenhum pacote enviado antes com número de sequência  $x$  está na rede. Isso é feito admitindo que um pacote não pode “viver” na rede mais do que um

**TABELA 3.1** Resumo de mecanismos de transferência confiável de dados e sua utilização

Mecanismo	Uso, comentários
Soma de verificação	Soma de verificação usada para detectar erros de <i>bits</i> em um pacote transmitido.
Temporizador	Usado para controlar a temporização/retransmissão de um pacote, possivelmente porque o pacote (ou seu ACK) foi perdido dentro do canal. Já que pode ocorrer esgotamento de temporização quando um pacote está atrasado, mas não perdido (esgotamento de temporização prematuro), ou quando um pacote foi recebido pelo destinatário, mas o ACK remetente-destinatário foi perdido, um destinatário pode receber cópias duplicadas de um pacote.
Número de sequência	Usado para numeração sequencial de pacotes de dados que transitam do remetente ao destinatário. Lacunas nos números de sequência de pacotes recebidos permitem que o destinatário detecte um pacote perdido. Pacotes com números de sequência duplicados permitem que o destinatário detecte cópias duplicadas de um pacote.
Reconhecimento	Usado pelo destinatário para avisar o remetente que um pacote ou conjunto de pacotes foi recebido corretamente. Reconhecimentos normalmente portam o número de sequência do pacote, ou pacotes, que estão sendo reconhecidos. Reconhecimentos podem ser individuais ou cumulativos, dependendo do protocolo.
Reconhecimento negativo	Usado pelo destinatário para avisar o remetente que um pacote não foi recebido corretamente. Reconhecimentos negativos normalmente portam o número de sequência do pacote que não foi recebido corretamente.
Janela, paralelismo	O remetente pode ficar restrito a enviar somente pacotes com números de sequência que caiam dentro de uma determinada faixa. Ao permitir que vários pacotes sejam transmitidos, ainda que não reconhecidos, a utilização do remetente pode ser aumentada em relação ao modo de operação pare e espere. Em breve veremos que o tamanho da janela pode ser estabelecido com base na capacidade de o destinatário receber e armazenar mensagens ou com base no nível de congestionamento na rede, ou em ambos.

tempo máximo fixado. As extensões do TCP para redes de alta velocidade (RFC 7323) usam um tempo de vida máximo de pacote de cerca de três minutos. Sunshine (1978) descreve um método para usar números de sequência tais que os problemas de reordenação podem ser completamente evitados.

## 3.5 TRANSPORTE ORIENTADO PARA CONEXÃO: TCP

Agora que já vimos os princípios subjacentes à transferência confiável de dados, vamos voltar para o TCP – o protocolo de transporte confiável da camada de transporte da Internet, orientado para conexão. Nesta seção, veremos que, para poder fornecer transferência confiável de dados, o TCP conta com muitos dos princípios subjacentes discutidos na seção anterior, incluindo detecção de erro, retransmissões, reconhecimentos cumulativos, temporizadores e campos de cabeçalho para números de sequência e de reconhecimento. O TCP está definido nos RFCs 793, 1122, 2018, 5681 e 7323.

### 3.5.1 A conexão TCP

Dizemos que o TCP é **orientado para conexão** porque, antes que um processo de aplicação possa começar a enviar dados a outro, os dois processos precisam primeiro se “apresentar” (*handshake*) – isto é, devem enviar alguns segmentos preliminares um ao outro para estabelecer os parâmetros da transferência de dados. Como parte do estabelecimento da conexão

TCP, ambos os lados da conexão iniciarão muitas variáveis de estado (muitas das quais serão discutidas nesta seção e na Seção 3.7) associadas com a conexão TCP.

A “conexão” TCP não é um circuito TDM ou de multiplexação por divisão de frequência (FDM, do inglês *frequency-division multiplexing*) fim a fim, como acontece em uma rede de comutação de circuitos. Em vez disso, trata-se de uma “conexão” lógica, com o estado comum residindo apenas nos TCPs nos dois sistemas finais em comunicação. Lembre-se de que, como o protocolo TCP roda apenas nos sistemas finais e não nos elementos intermediários da rede (roteadores e *switches*), os elementos intermediários não mantêm o estado de conexão TCP. Na verdade, os roteadores intermediários ignoram totalmente as conexões TCP; eles veem datagramas, não conexões.

Uma conexão TCP provê um **serviço full-duplex**: se houver uma conexão TCP entre o processo A em um hospedeiro e o processo B em outro hospedeiro, os dados da camada de aplicação poderão fluir de A para B ao mesmo tempo em que os dados da camada de aplicação fluem de B para A. A conexão TCP é sempre **ponto a ponto**, isto é, entre um único remetente e um único destinatário. O chamado “*multicast*” (consulte o material complementar *online*) – a transferência de dados de um remetente para vários destinatários em uma única operação de envio – não é possível com o TCP. Com o TCP, ter dois hospedeiros é bom; três é demais!

Vamos agora examinar como uma conexão TCP é estabelecida. Suponha que um processo que roda em um hospedeiro queira iniciar a conexão com outro processo em outro hospedeiro. Lembre-se de que o processo que está iniciando a conexão é denominado *processo cliente*, e o outro é denominado *processo servidor*. O processo de aplicação cliente primeiro informa à camada de transporte no cliente que ele quer estabelecer uma conexão com um processo no servidor. Lembre-se (Seção 2.7.2) de que um programa cliente em Python faz isso executando o comando

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

em que *serverName* é o nome do servidor e *serverPort* identifica o processo no servidor. O TCP no cliente então passa a estabelecer uma conexão TCP com o TCP no servidor. Discutiremos com algum detalhe o procedimento de estabelecimento de conexão ao final desta seção. Por enquanto, basta saber que o cliente primeiro envia um segmento TCP especial; o servidor responde com um segundo segmento TCP especial e, por fim, o cliente

## HISTÓRICO DO CASO

### VINTON CERF, ROBERT KAHN E TCP/IP

No início da década de 1970, as redes de comutação de pacotes começaram a proliferar. A ARPAnet – precursora da Internet – era apenas mais uma entre tantas que tinham, cada uma, seu próprio protocolo. Dois pesquisadores, Vinton Cerf e Robert Kahn, reconheceram a importância de interconectar essas redes e inventaram um protocolo inter-redes denominado TCP/IP, que quer dizer Transmission Control Protocol/Internet Protocol (protocolo de controle de transmissão/protocolo da Internet). Embora no começo Cerf e Kahn considerassem o protocolo uma entidade única, mais tarde ele foi dividido em duas partes, TCP e IP, que operavam separadamente. Cerf e Kahn publicaram um artigo sobre o TCP/IP em maio de 1974 na *IEEE Transactions on Communications Technology* (Cerf, 1974).

O protocolo TCP/IP, que é o “feijão com arroz” da Internet de hoje, foi elaborado antes dos PCs, estações de trabalho, *smartphones* e *tablets*, antes da proliferação da Ethernet, cabo, DSL, WiFi e outras tecnologias de redes locais, antes da Web, redes sociais e *streaming* de vídeo. Cerf e Kahn perceberam a necessidade de um protocolo de rede que, de um lado, fornecesse amplo suporte para aplicações ainda a serem definidas e que, de outro, permitisse a interoperação de hospedeiros arbitrários e protocolos de camada de enlace.

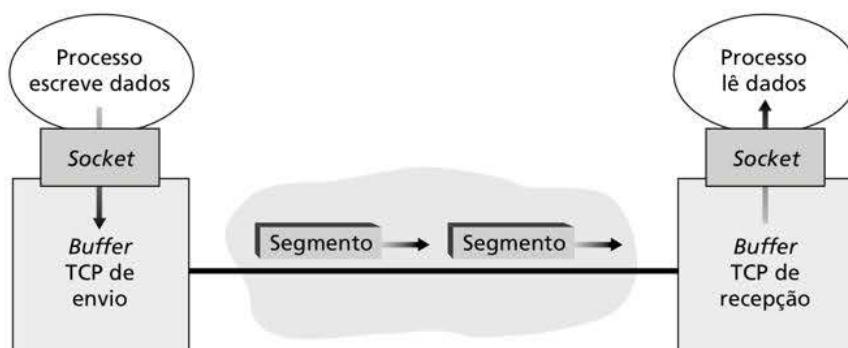
Em 2004, Cerf e Kahn receberam o prêmio ACM Turing Award, considerado o Prêmio Nobel da Computação pelo “trabalho pioneiro sobre interligação em rede, incluindo o projeto e a implementação dos protocolos de comunicação da Internet, TCP/IP e por inspirarem liderança na área de redes”.

responde novamente com um terceiro segmento especial. Os primeiros dois segmentos não contêm nenhuma “carga útil”, isto é, nenhum dado da camada de aplicação; o terceiro pode carregar uma carga útil. Como três segmentos são enviados entre dois hospedeiros, esse procedimento de estabelecimento de conexão é muitas vezes denominado **apresentação de três vias (three-way handshake)**.

Uma vez estabelecida uma conexão TCP, os dois processos de aplicação podem enviar dados um para o outro. Vamos considerar o envio de dados do processo cliente para o processo servidor. O processo cliente passa uma cadeia de dados pelo *socket* (a porta do processo), como descrito na Seção 2.7. Tão logo passem pelo *socket*, os dados estão nas mãos do TCP que está rodando no cliente. Como mostra a Figura 3.28, o TCP direciona seus dados para o **buffer de envio** da conexão, que é um dos *buffers* reservados durante a apresentação de três vias inicial. Periodicamente, o TCP arranca pedaços de dados do *buffer* de envio e passa os dados à camada de rede. O interessante é que a especificação do TCP (RFC 793) é muito vaga ao indicar quando o TCP deve de fato enviar dados que estão nos *buffers*, determinando apenas que o TCP “deve enviar aqueles dados em segmentos segundo sua própria conveniência”. A quantidade máxima de dados que pode ser retirada e colocada em um segmento é limitada pelo **tamanho máximo do segmento (MSS, do inglês maximum segment size)**. O MSS normalmente é estabelecido determinando primeiro o tamanho do maior quadro de camada de enlace que pode ser enviado pelo hospedeiro remetente local, denominado **unidade máxima de transmissão (MTU, do inglês maximum transmission unit)** e, em seguida, estabelecendo um MSS que garanta que um segmento TCP (quando encapsulado em um datagrama IP) mais o comprimento do cabeçalho TCP/IP (em geral, 40 bytes) caberão em um único quadro de camada de enlace. Os protocolos da camada de enlace Ethernet e PPP possuem um MTU de 1.500 bytes. Assim, um valor típico do MSS é 1460 bytes. Também foram propostas técnicas para descobrir a MTU do caminho – o maior quadro de camada de enlace que pode ser enviado por todos os enlaces desde a origem até o destino (RFC 1191) – e definir o MSS com base no valor da MTU do caminho. Note que o MSS é a quantidade máxima de dados de camada de aplicação no segmento, e não o tamanho máximo do segmento TCP incluindo cabeçalhos. (Essa terminologia é confusa, mas temos de conviver com ela, pois já está consolidada.)

O TCP combina cada porção de dados do cliente com um cabeçalho TCP, formando, assim, **segmentos TCP**. Os segmentos são passados para baixo, para a camada de rede, onde são encapsulados separadamente dentro dos datagramas IP da camada de rede. Os datagramas IP são então enviados para dentro da rede. Quando o TCP recebe um segmento na outra extremidade, os dados do segmento são colocados no *buffer* de recepção da conexão, como ilustra a Figura 3.28. A aplicação lê a cadeia de dados desse *buffer*. Cada lado da conexão tem seus próprios *buffers* de envio e seu próprio *buffer* de recepção. (Você pode ver a animação interativa sobre o controle de fluxo *online* em <<http://www.awl.com/kurose-ross>>, que oferece uma animação dos *buffers* de envio e de recepção.)

Entendemos, dessa discussão, que uma conexão TCP consiste em *buffers*, variáveis e um *socket* de conexão de um processo em um hospedeiro e outro conjunto de *buffers*, variáveis



**Figura 3.28** Buffers TCP de envio e de recepção.

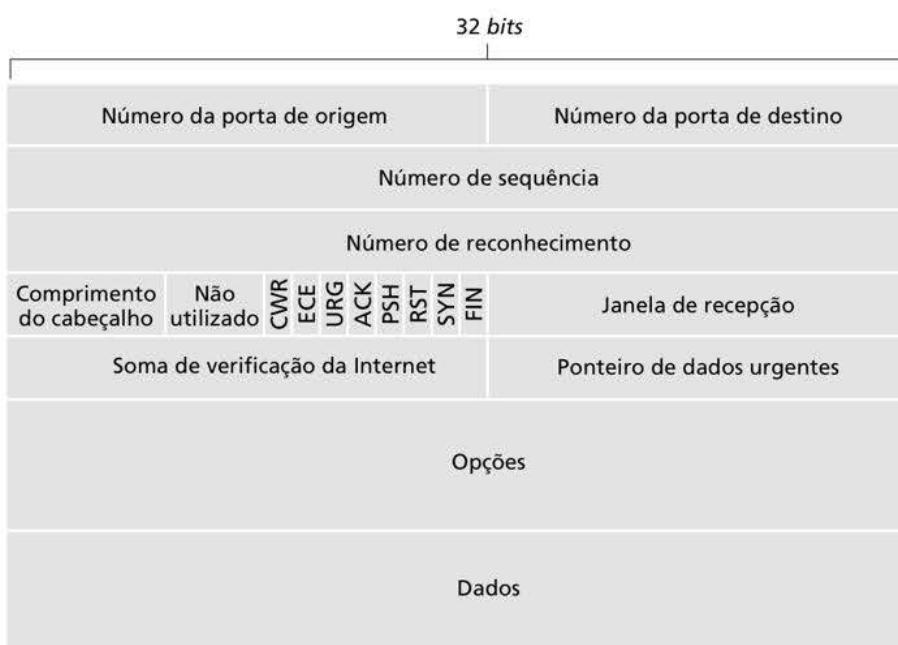
e um *socket* de conexão de um processo em outro hospedeiro. Como mencionamos, nenhum *buffer* nem variáveis são alocados à conexão nos elementos da rede (roteadores, *switches* e repetidores) existentes entre os hospedeiros.

### 3.5.2 Estrutura do segmento TCP

Agora que examinamos de modo breve a conexão TCP, vamos verificar a estrutura do segmento TCP, que consiste em campos de cabeçalho e um campo de dados. O campo de dados contém uma quantidade de dados de aplicação. Como já dissemos, o MSS limita o tamanho máximo do campo de dados de um segmento. Quando o TCP envia um arquivo grande, tal como uma imagem de uma página Web, ele costuma fragmentar o segmento em pedaços de tamanho MSS (exceto o último pedaço, que muitas vezes é menor do que o MSS). Aplicações interativas, contudo, geralmente transmitem quantidades de dados menores do que o MSS. Por exemplo, com aplicações de *login* remoto como Telnet e ssh, o campo de dados do segmento TCP é, muitas vezes, de apenas 1 *byte*. Como o cabeçalho TCP tem tipicamente 20 *bytes* (12 *bytes* mais do que o cabeçalho UDP), o comprimento dos segmentos enviados por Telnet pode ser de apenas 21 *bytes*.

A Figura 3.29 mostra a estrutura do segmento TCP. Como acontece com o UDP, o cabeçalho inclui **números de porta de origem e de destino**, que são usados para multiplexação e demultiplexação de dados para aplicações de camadas superiores, e, assim como no UDP, inclui um **campo de soma de verificação**. Um cabeçalho de segmento TCP também contém os seguintes campos:

- O **campo de número de sequência** de 32 bits e o **campo de número de reconhecimento** de 32 bits são usados pelos TCPs remetente e destinatário na execução de um serviço confiável de transferência de dados, como discutido a seguir.
- O campo de **janela de recepção** de 16 bits é usado para controle de fluxo. Veremos em breve que esse campo é usado para indicar o número de *bytes* que um destinatário está disposto a aceitar.
- O **campo de comprimento de cabeçalho** de 4 bits especifica o comprimento do cabeçalho TCP em palavras de 32 bits. O cabeçalho TCP pode ter comprimento variável em



**Figura 3.29** Estrutura do segmento TCP.

razão do campo de opções TCP. (O campo de opções TCP em geral está vazio, de modo que o comprimento do cabeçalho TCP típico é 20 bytes.)

- O **campo de opções**, opcional e de comprimento variável, é usado quando um remetente e um destinatário negociam o MSS, ou como um fator de aumento de escala da janela para utilização em redes de alta velocidade. Uma opção de marca de tempo é também definida. Consulte o RFC 854 e o RFC 1323 para detalhes adicionais.
- O **campo de flag** contém 6 bits. O bit ACK é usado para indicar se o valor carregado no campo de reconhecimento é válido, isto é, se o segmento contém um reconhecimento para um segmento que foi recebido com sucesso. Os bits RST, SYN e FIN são usados para estabelecer e encerrar a conexão, como discutiremos ao final desta seção. Os bits CWR e ECE são usados na notificação explícita de congestionamento, como discutido na Seção 3.7.2. Marcar o bit PSH indica que o destinatário deve passar os dados para a camada superior imediatamente. Por fim, o bit URG é usado para mostrar que há dados nesse segmento que a entidade da camada superior do lado remetente marcou como “urgentes”. A localização do último byte desses dados urgentes é indicada pelo **campo de ponteiro de urgência** de 16 bits. O TCP deve informar à entidade da camada superior do lado destinatário quando existem dados urgentes e passar a ela um ponteiro para o final desses dados. (Na prática, o PSH, o URG e o ponteiro de dados urgentes não são usados. Contudo, mencionamos esses campos para descrever todos.)

Na nossa experiência enquanto professores, alguns alunos acham a discussão sobre formatos de pacotes um tanto árida e maçante. Para uma maneira divertida e fantasiosa de pensar sobre campos de cabeçalho TCP, particularmente se for tão fã de Legos<sup>TM</sup> quanto nós, consulte Pomeranz (2010).

### Números de sequência e números de reconhecimento

Dois dos mais importantes campos do cabeçalho do segmento TCP são o de número de sequência e o de número de reconhecimento. Esses campos são parte fundamental do serviço de transferência confiável de dados do TCP. Mas antes de discutirmos como são utilizados, vamos explicar exatamente o que o TCP coloca nesses campos.

O TCP vê os dados como uma cadeia de bytes não estruturada, mas ordenada. O uso que o TCP faz dos números de sequência reflete essa visão, pois esses números são aplicados sobre a cadeia de bytes transmitidos, e *não* sobre a série de segmentos transmitidos. O **número de sequência para um segmento** é o número do primeiro byte do segmento. Vamos ver um exemplo. Suponha que um processo no hospedeiro A queira enviar uma cadeia de dados para um processo no hospedeiro B por uma conexão TCP. O TCP do hospedeiro A vai implicitamente numerar cada byte da cadeia de dados. Suponha que a cadeia de dados consista em um arquivo composto por 500 mil bytes, que o MSS seja de 1.000 bytes e que seja atribuído o número 0 ao primeiro byte da cadeia de dados. Como mostra a Figura 3.30, o TCP constrói 500 segmentos a partir da cadeia de dados. O primeiro recebe o número de sequência 0; o segundo, o número de sequência 1.000; o terceiro, o número de sequência 2.000, e assim por diante. Cada número de sequência é inserido no campo de número de sequência no cabeçalho do segmento TCP apropriado.



**Figura 3.30** Dividindo os dados do arquivo em segmentos TCP.

Vamos agora considerar os números de reconhecimento. Eles são um pouco mais complicados do que os números de sequência. Lembre-se de que o TCP é *full-duplex*, portanto o hospedeiro A pode estar recebendo dados do hospedeiro B enquanto envia dados ao hospedeiro B (como parte da mesma conexão TCP). Cada segmento que chega do hospedeiro B tem um número de sequência para os dados que estão fluindo de B para A. *O número de reconhecimento que o hospedeiro A atribui a seu segmento é o número de sequência do próximo byte que ele estiver aguardando do hospedeiro B.* É bom examinarmos alguns exemplos para entendermos o que está acontecendo aqui. Suponha que o hospedeiro A tenha recebido do hospedeiro B todos os bytes numerados de 0 a 535 e que esteja prestes a enviar um segmento ao hospedeiro B. O hospedeiro A está esperando pelo byte 536 e por todos os bytes subsequentes da cadeia de dados do hospedeiro B. Assim, ele coloca o número 536 no campo de número de reconhecimento do segmento que envia para o hospedeiro B.

Como outro exemplo, suponha que o hospedeiro A tenha recebido um segmento do hospedeiro B contendo os bytes de 0 a 535 e outro segmento contendo os bytes de 900 a 1.000. Por alguma razão, o hospedeiro A ainda não recebeu os bytes de 536 a 899. Nesse exemplo, ele ainda está esperando pelo byte 536 (e os superiores) para poder recriar a cadeia de dados de B. Assim, o segmento seguinte que A envia a B conterá 536 no campo de número de reconhecimento. Como o TCP somente reconhece bytes até o primeiro byte que estiver faltando na cadeia, dizemos que o TCP provê **reconhecimentos cumulativos**.

Este último exemplo também revela uma questão importante, mas sutil. O hospedeiro A recebeu o terceiro segmento (bytes de 900 a 1.000) antes do segundo (bytes de 536 a 899). Portanto, o terceiro segmento chegou fora de ordem. E o que um hospedeiro faz quando recebe segmentos fora de ordem em uma conexão TCP? Eis a questão. O interessante é que os RFCs do TCP não impõem nenhuma regra para isso e deixam a decisão para os programadores que estiverem implementando a execução TCP. Há basicamente duas opções: (1) o destinatário descarta imediatamente os segmentos fora de ordem (o que, como discutimos antes, pode simplificar o projeto do destinatário) ou (2) o destinatário conserva os bytes fora de ordem e espera pelos bytes faltantes para preencher as lacunas. Claro que a segunda alternativa é mais eficiente em termos de largura de banda de rede e é a abordagem adotada na prática.

Na Figura 3.30, admitimos que o número de sequência inicial era 0. Na verdade, ambos os lados de uma conexão TCP escolhem ao acaso um número de sequência inicial. Isso é feito para minimizar a possibilidade de um segmento de uma conexão já encerrada entre dois hospedeiros e ainda presente na rede ser tomado por um segmento válido em uma conexão posterior entre esses dois mesmos hospedeiros (que também podem estar usando os mesmos números de porta da conexão antiga) (Sunshine, 1978).

### Telnet: um estudo de caso para números de sequência e números de reconhecimento

O Telnet, definido no RFC 854, é um protocolo popular de camada de aplicação utilizado para fazer *login* remoto. Ele roda sobre TCP e é projetado para trabalhar entre qualquer par de hospedeiros. Diferentemente das aplicações de transferência de dados em grandes blocos, que foram discutidas no Capítulo 2, o Telnet é uma aplicação interativa. Discutiremos, agora, um exemplo de Telnet, pois ilustra muito bem números de sequência e de reconhecimento do TCP. Observamos que muitos usuários agora preferem usar o protocolo SSH, visto que dados enviados por uma conexão Telnet (incluindo senhas!) não são criptografados, o que torna essa aplicação vulnerável a ataques de bisbilhoteiros (como discutiremos na Seção 8.7).

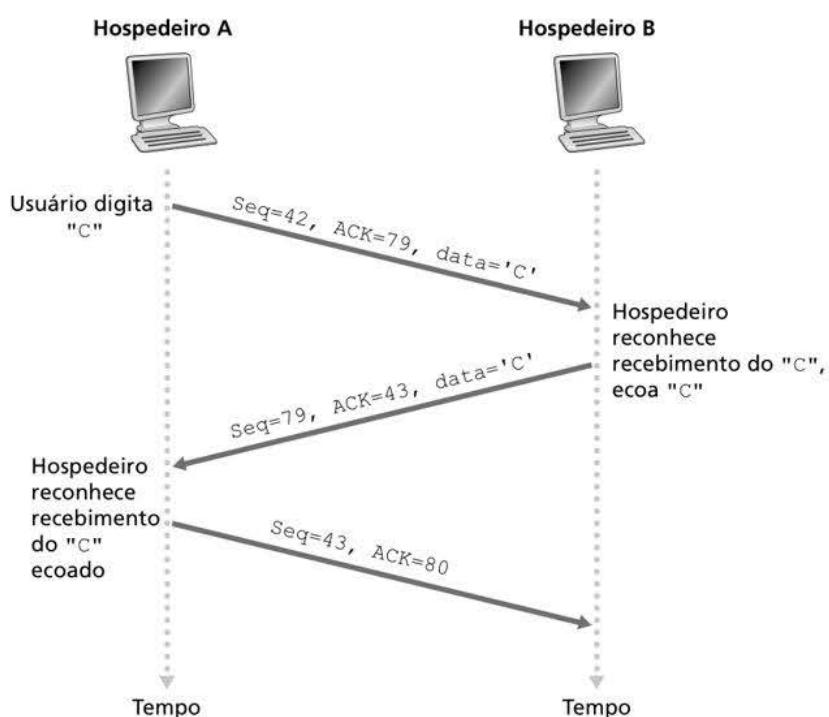
Suponha que o hospedeiro A inicie uma sessão Telnet com o hospedeiro B. Como o hospedeiro A inicia a sessão, ele é rotulado de cliente, enquanto B é rotulado de servidor. Cada caractere digitado pelo usuário (no cliente) será enviado ao hospedeiro remoto; este devolverá uma cópia (“eco”) de cada caractere, que será apresentada na tela Telnet do usuário. Esse

eco é usado para garantir que os caracteres vistos pelo usuário do Telnet já foram recebidos e processados no local remoto. Assim, cada caractere atravessa a rede duas vezes entre o momento em que o usuário aperta o teclado e o momento em que o caractere é apresentado em seu monitor.

Suponha agora que o usuário digite a letra “C” e saia para tomar um café. Vamos examinar os segmentos TCP que são enviados entre o cliente e o servidor. Como mostra a Figura 3.31, admitamos que os números de sequência iniciais sejam 42 e 79 para cliente e servidor, respectivamente. Lembre-se de que o número de sequência de um segmento será o número de sequência do primeiro *byte* do seu campo de dados. Assim, o primeiro segmento enviado do cliente terá número de sequência 42; o primeiro segmento enviado do servidor terá número de sequência 79. Note que o número de reconhecimento será o número de sequência do próximo *byte* de dados que o hospedeiro estará aguardando. Após o estabelecimento da conexão TCP, mas antes de quaisquer dados serem enviados, o cliente ficará esperando pelo byte 79 e o servidor, pelo byte 42.

Como ilustra a Figura 3.31, são enviados três segmentos. O primeiro é enviado do cliente ao servidor, contendo, em seu campo de dados, um *byte* com a representação ASCII para a letra “C”. O primeiro segmento também tem 42 em seu campo de número de sequência, como acabamos de descrever. E mais, como o cliente ainda não recebeu nenhum dado do servidor, esse segmento terá o número 79 em seu campo de número de reconhecimento.

O segundo segmento é enviado do servidor ao cliente. Esse segmento tem dupla finalidade. A primeira é fornecer um reconhecimento para os dados que o servidor recebeu. Ao colocar 43 no campo de reconhecimento, o servidor está dizendo ao cliente que recebeu com sucesso tudo até o byte 42 e agora está aguardando os bytes de 43 em diante. A segunda finalidade desse segmento é ecoar a letra “C”. Assim, o segundo segmento tem a representação ASCII de “C” em seu campo de dados. Ele tem o número de sequência 79, que é o número de sequência inicial do fluxo de dados de servidor para cliente dessa conexão TCP, pois este é o primeiríssimo *byte* de dados que o servidor está enviando. Note que o reconhecimento para dados do cliente para o servidor é levado em um segmento que carrega



**Figura 3.31** Números de sequência e de reconhecimento para uma aplicação Telnet simples sobre TCP.

dados do servidor para o cliente. Dizemos que esse reconhecimento **pegou uma carona (piggybacked)** no segmento de dados do servidor ao cliente.

O terceiro segmento é enviado do cliente ao servidor. Seu único propósito é reconhecer os dados que recebeu do servidor. (Lembre-se de que o segundo segmento continha dados – a letra “C” – do servidor para o cliente.) Esse terceiro segmento tem um campo de dados vazio (i.e., o reconhecimento não está pegando carona com nenhum dado do cliente para o servidor). O segmento tem o número 80 no campo do número de reconhecimento porque o cliente recebeu a cadeia de dados até o byte com número de sequência 79 e agora está aguardando os bytes de 80 em diante. É possível que você esteja pensando que é estranho que esse segmento também tenha um número de sequência, já que não contém dados. Mas como o TCP tem um campo de número de sequência, o segmento precisa apresentar algum número para preenchê-lo.

### 3.5.3 Estimativa do tempo de viagem de ida e volta e de esgotamento de temporização

O TCP, assim como o nosso protocolo `rdt` da Seção 3.4, utiliza um mecanismo de controle de temporização/retransmissão para recuperar segmentos perdidos. Embora conceitualmente simples, surgem muitas questões sutis quando executamos um mecanismo de controle de temporização/retransmissão em um protocolo real como o TCP. Talvez a pergunta mais óbvia seja a duração dos intervalos de controle. Claro, esse intervalo deve ser maior do que o RTT, isto é, o tempo decorrido entre o envio de um segmento e seu reconhecimento. Se não fosse assim, seriam enviadas retransmissões desnecessárias. Mas quão maior deve ser o intervalo e, antes de tudo, como o RTT deve ser estimado? Deve-se associar um temporizador a cada segmento não reconhecido? São tantas perguntas! Nesta seção, nossa discussão se baseia no trabalho de Jacobson (1988) sobre TCP e nas recomendações da Força de Trabalho de Engenharia da Internet (IETF, do inglês *Internet Engineering Task Force*) vigentes para o gerenciamento de temporizadores TCP (RFC 6298).

#### Estimativa do tempo de viagem de ida e volta

Vamos iniciar nosso estudo do gerenciamento do temporizador TCP considerando como esse protocolo estima o tempo de viagem de ida e volta entre remetente e destinatário, o que apresentaremos a seguir. O RTT para um segmento, denominado `SampleRTT` no exemplo, é o tempo transcorrido entre o momento em que o segmento é enviado (i.e., passado ao IP) e o momento em que é recebido um reconhecimento para ele. Em vez de medir um `SampleRTT` para cada segmento transmitido, a maioria das implementações de TCP executa apenas uma medição de `SampleRTT` por vez. Isto é, em qualquer instante, o `SampleRTT` estará sendo estimado para apenas um dos segmentos transmitidos mas ainda não reconhecidos, o que resulta em um novo valor de `SampleRTT`, para mais ou menos para cada RTT. E mais, o TCP nunca computa um `SampleRTT` para um segmento que foi retransmitido; apenas mede-o para segmentos que foram transmitidos uma vez (Karn, 1987). (Um dos problemas ao final do capítulo perguntará por quê.)

Claro, os valores de `SampleRTT` sofrerão variação de segmento para segmento em decorrência do congestionamento nos roteadores e das variações de carga nos sistemas finais. Em virtude dessa variação, qualquer dado valor de `SampleRTT` pode ser atípico. Portanto, para estimar um RTT comum, é natural tomar alguma espécie de média dos valores de `SampleRTT`. O TCP mantém uma média, denominada `EstimatedRTT`, dos valores de `SampleRTT`. Ao obter um novo `SampleRTT`, o TCP atualiza `EstimatedRTT` de acordo com a seguinte fórmula:

$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$$

Essa fórmula está escrita na forma de um comando de linguagem de programação – o novo valor de **EstimatedRTT** é uma combinação ponderada entre o valor anterior de **EstimatedRTT** e o novo valor para **SampleRTT**. O valor recomendado de  $\alpha$  é  $\alpha = 0,125$  (i.e., 1/8) (RFC 6298), caso em que essa fórmula se torna:

$$\text{EstimatedRTT} = 0,875 \cdot \text{EstimatedRTT} + 0,125 \cdot \text{SampleRTT}$$

Note que **EstimatedRTT** é uma média ponderada dos valores de **SampleRTT**. Como veremos em um exercício ao final deste capítulo, essa média ponderada atribui um peso maior às amostras recentes do que às antigas. Isso é natural, pois as amostras mais recentes refletem melhor o estado atual de congestionamento da rede. Em estatística, esse tipo de média é denominado **média móvel exponencial ponderada (MMEP)**. A palavra “exponencial” aparece na MMEP porque o peso atribuído a um dado **SampleRTT** diminui exponencialmente à medida que as atualizações são realizadas. Os exercícios pedirão que você derive o termo exponencial em **EstimatedRTT**.

A Figura 3.32 mostra os valores de **SampleRTT** e **EstimatedRTT** para um valor de  $\alpha = 1/8$ , para uma conexão TCP entre `gaia.cs.umass.edu` (em Amherst, Massachusetts) e `fantasia.eurecom.fr` (no sul da França). Fica claro que as variações em **SampleRTT** são atenuadas no cálculo de **EstimatedRTT**.

Além de ter uma estimativa do RTT, também é valioso ter uma medida de sua variabilidade. O (RFC 6298) define a variação do RTT, **DevRTT**, como uma estimativa do desvio típico entre **SampleRTT** e **EstimatedRTT**:

$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot | \text{SampleRTT} - \text{EstimatedRTT} |$$

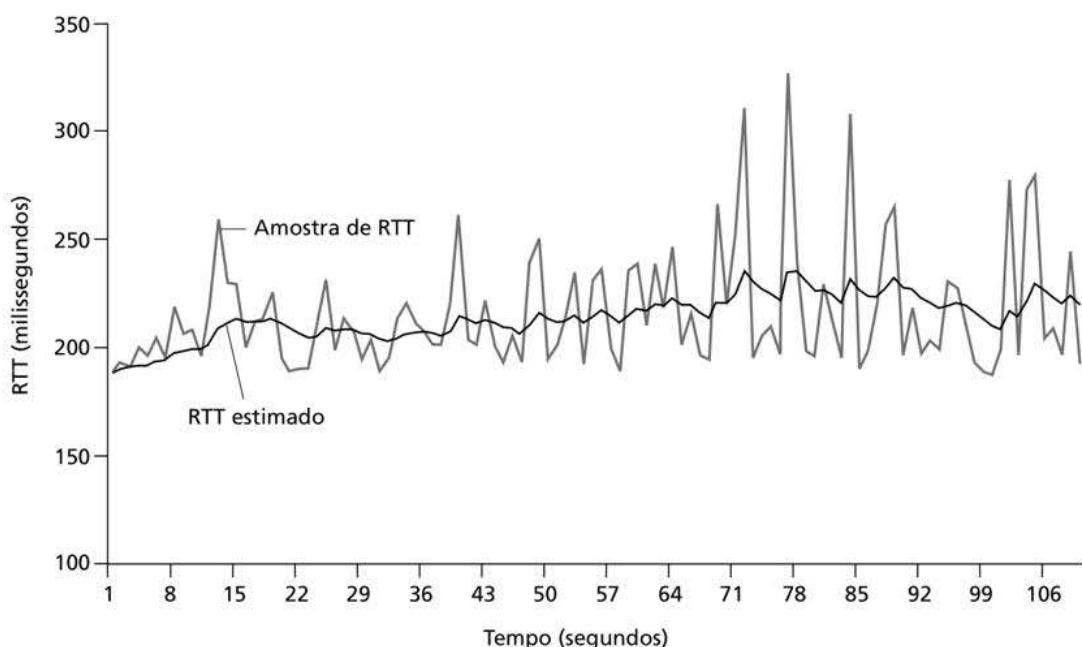
Note que **DevRTT** é uma MMEP da diferença entre **SampleRTT** e **EstimatedRTT**. Se os valores de **SampleRTT** apresentarem pouca variação, então **DevRTT** será pequeno; por outro lado, se houver muita variação, **DevRTT** será grande. O valor recomendado para  $\beta$  é 0,25.

## PRINCÍPIOS NA PRÁTICA

O TCP fornece transferência confiável de dados usando reconhecimentos positivos e temporizadores, de modo muito parecido com o que estudamos na Seção 3.4. O protocolo reconhece dados que foram recebidos corretamente e retransmite segmentos quando entende que eles ou seus reconhecimentos correspondentes foram perdidos ou corrompidos. Certas versões do TCP também têm um mecanismo NAK implícito – com o mecanismo de retransmissão rápida do TCP. O recebimento de três ACKs duplicados para um dado segmento serve como um NAK implícito para o seguinte, acionando a retransmissão daquele segmento antes que o tempo se esgote. O TCP usa sequência de números para permitir que o destinatário identifique segmentos perdidos ou duplicados. Exatamente como no caso de nosso protocolo de transferência confiável de dados rdt3.0, o TCP em si não pode determinar com certeza

se um segmento, ou seu ACK, está perdido, corrompido ou atrasado demais. No remetente, a resposta do TCP será a mesma: retransmitir o segmento.

O TCP também utiliza paralelismo, permitindo que o remetente tenha, a qualquer tempo, múltiplos segmentos transmitidos mas ainda não reconhecidos. Vimos antes que o paralelismo pode melhorar muito a vazão de uma sessão quando a razão entre o tempo de transmissão do segmento e o atraso de viagem de ida e volta é pequena. O número específico de segmentos não reconhecidos que um remetente pode ter é determinado pelos mecanismos de controle de fluxo e controle de congestionamento do TCP. O controle de fluxo do TCP é discutido no final desta seção; o controle de congestionamento do TCP é discutido na Seção 3.7. Por enquanto, devemos apenas ficar cientes de que o TCP remetente usa paralelismo.



**Figura 3.32** Amostras e estimativas de RTT.

### Estabelecimento e gerenciamento da temporização de retransmissão

De acordo com os valores de `EstimatedRTT` e `DevRTT`, qual valor deve ser utilizado para a temporização de retransmissão do TCP? É óbvio que o intervalo deve ser maior ou igual a `EstimatedRTT`, caso contrário seriam enviadas retransmissões desnecessárias. Mas a temporização de retransmissão não deve ser muito maior do que `EstimatedRTT`, senão, quando um segmento fosse perdido, o TCP não o retransmitiria rápido, o que resultaria em grandes atrasos de transferência de dados. Portanto, é desejável que o valor estabelecido para a temporização seja igual a `EstimatedRTT` mais certa margem, que deverá ser grande quando houver muita variação nos valores de `SampleRTT` e pequena quando houver pouca variação. Assim, o valor de `DevRTT` deverá entrar em cena. Todas essas considerações são levadas em conta no método do TCP para determinar a temporização de retransmissão:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}$$

Recomenda-se um valor inicial de 1 segundo para `TimeoutInterval` (RFC 6298). Além disso, quando há temporização, o valor de `TimeoutInterval` é dobrado para evitar que haja uma temporização para um segmento subsequente, que logo será reconhecido. Porém, assim que o segmento é recebido e `EstimatedRTT` é atualizado, o `TimeoutInterval` novamente é calculado usando a fórmula anterior.

### 3.5.4 Transferência confiável de dados

Lembre-se de que o serviço da camada de rede da Internet (serviço IP) não é confiável. O IP não garante a entrega de datagramas na ordem correta nem a integridade de seus dados nos datagramas. Com o serviço IP, os datagramas podem transbordar dos *buffers* dos roteadores e jamais alcançar seu destino; podem também chegar fora de ordem. Além disso, os *bits* dos datagramas podem ser corrompidos (passar de 0 para 1 e vice-versa). Como os segmentos da camada de transporte são carregados pela rede por datagramas IPs, eles também podem sofrer esses mesmos problemas.

O TCP cria um **serviço de transferência confiável de dados** sobre o serviço de melhor esforço do IP. Esse serviço de transferência garante que a cadeia de dados que um processo lê a partir de seu *buffer* de recebimento TCP não está corrompida, não tem lacunas, não tem duplicações e está em sequência, isto é, a cadeia de *bytes* é idêntica à cadeia de *bytes* enviada pelo sistema final que está do outro lado da conexão. O modo como o TCP oferece transferência confiável de dados envolve muitos dos princípios estudados na Seção 3.4.

Quando desenvolvíamos técnicas de transferência confiável de dados, era conceitualmente mais fácil admitir que existia um temporizador individual associado com cada segmento transmitido mas ainda não reconhecido. Embora, em teoria, isso seja ótimo, o gerenciamento de temporizadores pode exigir considerável sobrecarga. Assim, os procedimentos recomendados no (RFC 6298) para gerenciamento de temporizadores TCP utilizam apenas um *único* temporizador de retransmissão, mesmo que haja vários segmentos transmitidos ainda não reconhecidos. O protocolo TCP apresentado nesta seção segue essa recomendação.

Discutiremos como o TCP provê transferência confiável de dados em duas etapas incrementais. Primeiro, apresentaremos uma descrição muito simplificada de um remetente TCP que utiliza apenas controle de temporizadores para se recuperar da perda de segmentos; em seguida, apresentaremos uma descrição mais complexa, que utiliza reconhecimentos duplicados além de temporizadores de retransmissão. Na discussão que se segue, admitimos que os dados estão sendo enviados em uma direção somente, do hospedeiro A ao hospedeiro B, e que o hospedeiro A está enviando um arquivo grande.

A Figura 3.33 apresenta uma descrição muito simplificada de um remetente TCP. Vemos que há três eventos importantes relacionados com a transmissão e a retransmissão de dados no TCP remetente: dados recebidos da aplicação; esgotamento do temporizador e recebimento de ACK. Quando ocorre o primeiro evento importante, o TCP recebe dados

```
/* Suponha que o remetente não seja compelido pelo fluxo de TCP ou controle de congestionamento, que o tamanho dos dados vindos de cima seja menor do que o MSS, e que a transferência de dados ocorra apenas em uma direção. */

NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

        event: dados recebidos da aplicação de cima
            criar segmento TCP com número de sequência NextSeqNum
            if (temporizador atualmente parado)
                iniciar temporizador
            passar segmento ao IP
            NextSeqNum=NextSeqNum+length(dados)
            break;

        event: esgotamento do temporizador
            retransmitir segmento ainda não reconhecido com o
            menor número de sequência
            iniciar temporizador
            break;

        evento: ACK recebido, com valor do campo ACK y
            if (y > SendBase) {
                SendBase=y
                if (há atualmente segmentos ainda não reconhecidos)
                    iniciar temporizador
            }
            break;
    } /* fim do loop forever */
}
```

**Figura 3.33** Remetente TCP simplificado.

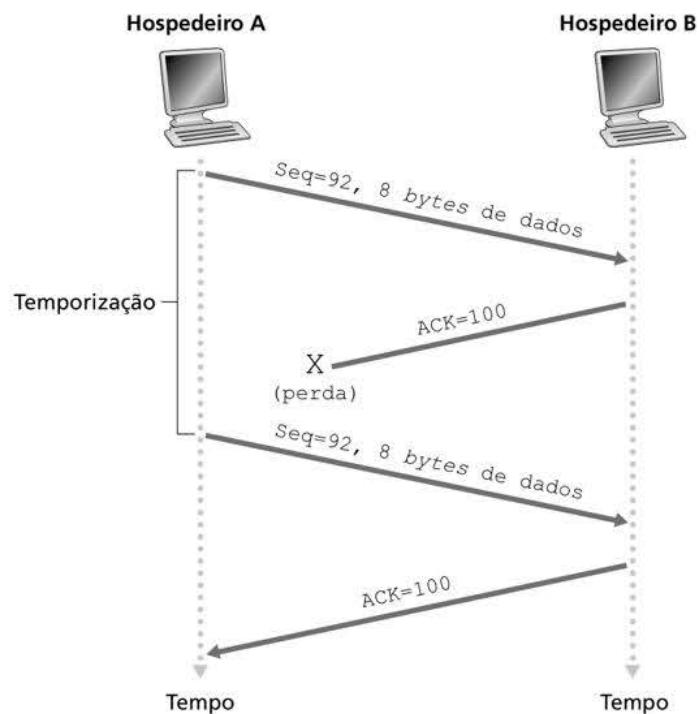
da camada de aplicação, encapsula-os em um segmento e passa-o ao IP. Note que cada segmento inclui um número de sequência que é o número da corrente de *bytes* do primeiro *byte* de dados no segmento, como descrito na Seção 3.5.2. Note também que, se o temporizador não estiver funcionando naquele instante para algum outro segmento, o TCP aciona o temporizador quando o segmento é passado para o IP. (Fica mais fácil se você imaginar que o temporizador está associado com o mais antigo segmento não reconhecido.) O intervalo de expiração para esse temporizador é o *Timeout Interval*, calculado a partir de *EstimatedRTT* e *DevRTT*, como descrito na Seção 3.5.3.

O segundo evento importante é o esgotamento do temporizador. O TCP responde a esse evento retransmitindo o segmento que causou o esgotamento da temporização, e então reinicia o temporizador.

O terceiro evento importante que deve ser manipulado pelo TCP remetente é a chegada de um segmento de reconhecimento (ACK) do destinatário (mais especificamente, um segmento contendo um valor de campo de ACK válido). Quando da ocorrência, o TCP compara o valor do ACK,  $y$ , com sua variável *SendBase*. A variável de estado *SendBase* do TCP é o número de sequência do mais antigo *byte* não reconhecido. (Assim, *SendBase*-1 é o número de sequência do último *byte* que se sabe ter sido recebido pelo destinatário de modo correto e na ordem certa.) Como comentamos, o TCP usa reconhecimentos cumulativos, de maneira que  $y$  reconhece o recebimento de todos os *bytes* antes do *byte* número  $y$ . Se  $y > \text{SendBase}$ , então o ACK está reconhecendo um ou mais *bytes* não reconhecidos antes. Desse modo, o remetente atualiza sua variável *SendBase* e reinicia o temporizador se houver quaisquer segmentos ainda não reconhecidos.

### Alguns cenários interessantes

Acabamos de descrever uma versão muito simplificada do modo como o TCP provê transferência confiável de dados, mas mesmo essa descrição tão simplificada tem muitas sutilezas. Para ter uma boa ideia de como esse protocolo funciona, vamos agora examinar alguns cenários simples. A Figura 3.34 ilustra o primeiro cenário, em que um hospedeiro A envia



**Figura 3.34** Retransmissão devido a um reconhecimento perdido.

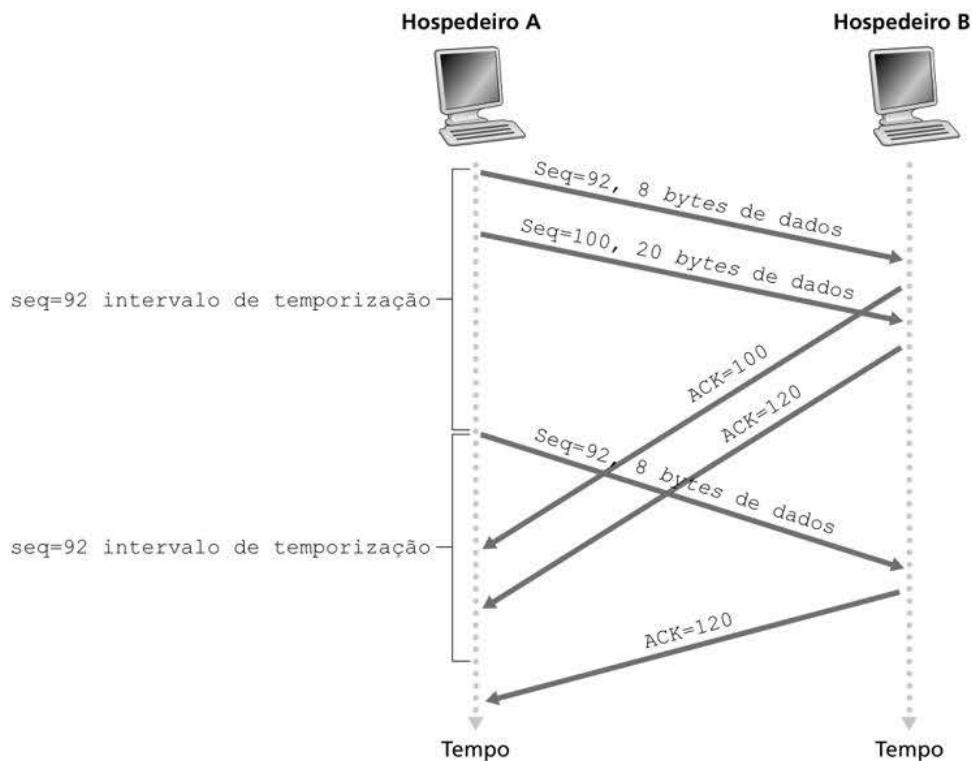
um segmento ao hospedeiro B. Suponha que esse segmento tenha número de sequência 92 e contenha 8 bytes de dados. Após enviá-lo, o hospedeiro A espera por um segmento de B com número de reconhecimento 100. Embora o segmento de A seja recebido em B, o reconhecimento de B para A se perde. Nesse caso, ocorre o evento de expiração do temporizador, e o hospedeiro A retransmite o mesmo segmento. É claro que, quando recebe a retransmissão, o hospedeiro B observa, pelo número de sequência, que o segmento contém dados que já foram recebidos. Assim, o TCP no hospedeiro B descarta os bytes do segmento retransmitido.

Em um segundo cenário, mostrado na Figura 3.35, o hospedeiro A envia dois segmentos seguidos. O primeiro tem número de sequência 92 e 8 bytes de dados. O segundo tem número de sequência 100 e 20 bytes de dados. Suponha que ambos cheguem intactos em B, e que B envie dois reconhecimentos separados para cada um desses segmentos. O primeiro deles tem número de reconhecimento 100; o segundo, número 120. Suponha agora que nenhum dos reconhecimentos chegue ao hospedeiro A antes do esgotamento do temporizador. Quando ocorre o evento de expiração do temporizador, o hospedeiro A reenvia o primeiro segmento com número de sequência 92 e reinicia o temporizador. Contanto que o ACK do segundo segmento chegue antes que o temporizador expire novamente, o segundo segmento não será retransmitido.

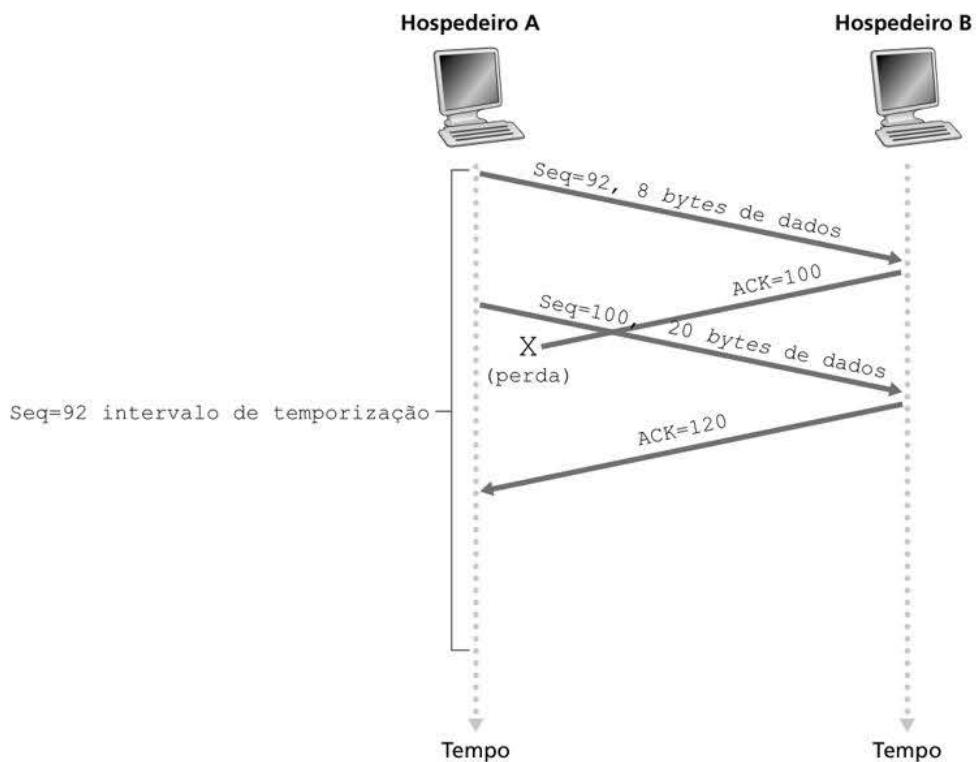
Em um terceiro e último cenário, suponha que o hospedeiro A envie dois segmentos, assim como no segundo exemplo. O reconhecimento do primeiro segmento é perdido na rede, mas, um pouco antes do evento de expiração, A recebe um reconhecimento com número 120. O hospedeiro A, portanto, sabe que B recebeu *tudo* até o byte 119; logo, ele não reenvia nenhum dos dois segmentos. Tal cenário está ilustrado na Figura 3.36.

### Duplicação do tempo de expiração

Discutiremos agora algumas modificações empregadas por grande parte das implementações do TCP. A primeira refere-se à duração do tempo de expiração após a expiração de



**Figura 3.35** Segmento 100 não retransmitido.



**Figura 3.36** Um reconhecimento cumulativo evita retransmissão do primeiro segmento.

um temporizador. Nessa modificação, sempre que ocorre tal evento, o TCP retransmite o segmento ainda não reconhecido que tenha o menor número de sequência, como descrevemos anteriormente. Mas a cada retransmissão, o TCP ajusta o próximo tempo de expiração para o dobro do valor anterior em vez de derivá-lo dos últimos EstimatedRTT e DevRTT (como descrito na Seção 3.5.3). Por exemplo, suponha que o TimeoutInterval associado com o mais antigo segmento ainda não reconhecido seja 0,75 segundo quando o temporizador expirar pela primeira vez. O TCP então retransmite esse segmento e ajusta o novo tempo de expiração para 1,5 segundo. Se o temporizador expirar novamente 1,5 segundo mais tarde, o TCP retransmitirá de novo esse segmento, agora ajustando o tempo de expiração para 3,0 segundos. Assim, o tempo aumenta exponencialmente após cada retransmissão. Todavia, sempre que o temporizador é iniciado após qualquer um dos outros dois eventos (i.e., dados recebidos da aplicação anterior e ACK recebido), o TimeoutInterval será derivado dos valores mais recentes de EstimatedRTT e DevRTT.

Essa modificação provê uma forma limitada de controle de congestionamento. (Maneiras mais abrangentes de controle de congestionamento no TCP serão estudadas na Seção 3.7.) A causa mais provável da expiração do temporizador é o congestionamento na rede, isto é, um número muito grande de pacotes chegando a uma (ou mais) fila de roteadores no caminho entre a origem e o destino, o que provoca descarte de pacotes e/ou longos atrasos de fila. Se as origens continuarem a retransmitir pacotes de modo persistente durante um congestionamento, este pode piorar. Em vez disso, o TCP age com mais educação: cada remetente retransmite após intervalos cada vez mais longos. Veremos que uma ideia semelhante a essa é utilizada pela Ethernet, quando estudarmos CSMA/CD no Capítulo 6.

### Retransmissão rápida

Um dos problemas de retransmissões acionadas por expiração de temporizador é que o período de expiração pode ser um tanto longo. Quando um segmento é perdido, esse longo

**TABELA 3.2** Recomendações para geração de ACKs pelo TCP (RFC 5681)

Evento	Ação do TCP destinatário
Chegada de segmento na ordem com número de sequência esperado. Todos os dados até o número de sequência esperado já reconhecidos.	ACK retardado. Espera de até 500 ms pela chegada de outro segmento na ordem. Se o segmento seguinte na ordem não chegar nesse intervalo, envia um ACK.
Chegada de segmento na ordem com número de sequência esperado. Outro segmento na ordem esperando por transmissão de ACK.	Envio imediato de um único ACK cumulativo, reconhecendo ambos os segmentos.
Chegada de um segmento fora da ordem com número de sequência mais alto do que o esperado. Lacuna detectada.	Envio imediato de um ACK duplicado, indicando número de sequência do byte seguinte esperado (que é a extremidade mais baixa da lacuna).
Chegada de um segmento que preenche, parcial ou completamente, a lacuna nos dados recebidos.	Envio imediato de um ACK, contanto que o segmento comece na extremidade mais baixa da lacuna.

período força o remetente a atrasar o reenvio do pacote perdido, aumentando o atraso fim a fim. Felizmente, o remetente pode, com frequência, detectar perda de pacote bem antes de ocorrer o evento de expiração, observando os denominados ACKs duplicados. Um **ACK duplicado** é um ACK que reconhece novamente um segmento para o qual o remetente já recebeu um reconhecimento anterior. Para entender a resposta do remetente a um ACK duplicado, devemos examinar por que o destinatário envia um ACK duplicado em primeiro lugar. A Tabela 3.2 resume a política de geração de ACKs do TCP destinatário (RFC 5681). Quando um TCP destinatário recebe um segmento com número de sequência maior do que o seguinte, esperado, na ordem, ele detecta uma lacuna no fluxo de dados – ou seja, um segmento faltando. Essa lacuna poderia ser o resultado de segmentos perdidos ou reordenados dentro da rede. Como o TCP não usa reconhecimentos negativos, o destinatário não pode enviar um reconhecimento negativo explícito de volta ao destinatário. Em vez disso, ele apenas reconhece mais uma vez (i.e., gera um ACK duplicado para) o último byte de dados na ordem que foi recebido. (Observe que a Tabela 3.2 tem provisão para o caso em que o destinatário não descarta segmentos fora de ordem.)

Como um remetente quase sempre envia um grande número de segmentos, um atrás do outro, se um segmento for perdido, provavelmente existirão muitos ACKs duplicados, também um após o outro. Se o TCP remetente receber três ACKs duplicados para os mesmos dados, ele tomará isso como indicação de que o segmento que se seguiu ao segmento reconhecido três vezes foi perdido. (Nos exercícios de fixação, consideraremos por que o remetente espera três ACKs duplicados e não apenas um.) No caso de receber três ACKs duplicados, o TCP remetente realiza uma **retransmissão rápida** (RFC 5681), retransmitindo o segmento que falta *antes* da expiração do temporizador do segmento. Isso é mostrado na Figura 3.37, em que o segundo segmento é perdido, e então retransmitido antes da expiração do temporizador. Para o TCP com retransmissão rápida, o seguinte trecho de codificação substitui o evento ACK recebido na Figura 3.33:

```

evento: ACK recebido, com valor do campo ACK y
    if (y > SendBase) {
        SendBase=y
        if (atualmente ainda não há segmentos
            reconhecidos)
            iniciar temporizador
    }

```

```

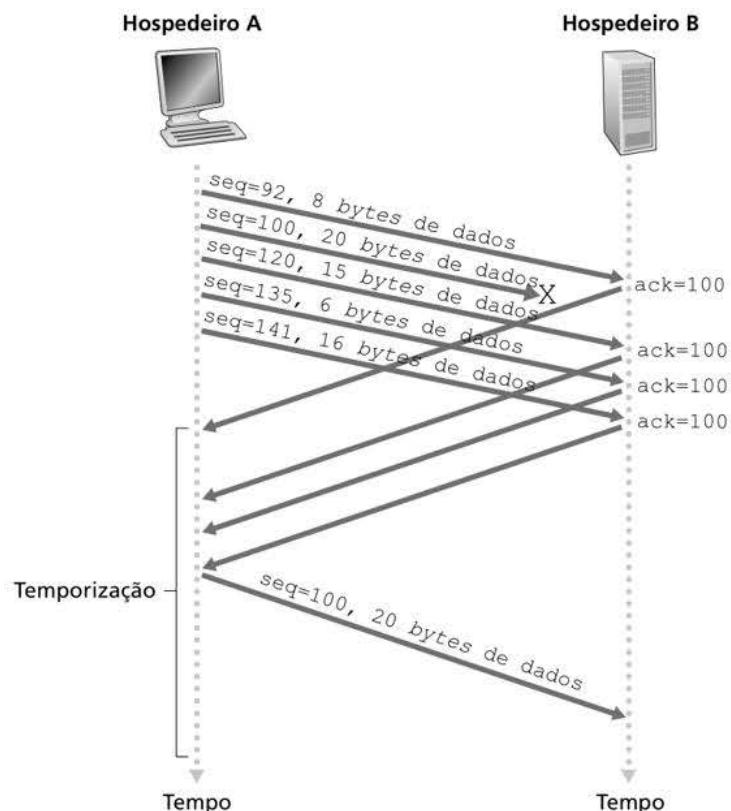
else { /* um ACK duplicado para segmento
       já reconhecido */
    incrementar número de ACKs duplicados
    recebidos para y
    if (número de ACKS duplicados recebidos
        para y é igual a 3)
        /* retransmissão rápida do TCP */
        reenviar segmento com número
        de sequência y
    }
break;

```

Observamos anteriormente que muitas questões sutis vêm à tona quando um mecanismo de controle de temporização/retransmissão é executado em um protocolo real como o TCP. Os procedimentos anteriores, cuja evolução é resultado de mais de 30 anos de experiência com temporizadores TCP, devem convencê-lo de que, na realidade, é isso que acontece!

### Go-Back-N ou repetição seletiva?

Vamos encerrar nosso estudo do mecanismo de recuperação de erros do TCP considerando a seguinte pergunta: o TCP é um protocolo GBN ou SR? Lembre-se de que, no TCP, os reconhecimentos são cumulativos, e segmentos recebidos de modo correto, mas fora da ordem, não são reconhecidos (ACK) individualmente pelo destinatário. Em consequência, como mostrou a Figura 3.33 (veja também a Figura 3.19), o TCP remetente precisa tão somente lembrar o menor número de sequência de um *byte* transmitido, porém não reconhecido



**Figura 3.37** Retransmissão rápida: retransmitir o segmento que falta antes da expiração do temporizador do segmento.

(SendBase) e o número de sequência do *byte* seguinte a ser enviado (NextSeqNum). Nesse sentido, o TCP se parece muito com um protocolo ao estilo do GBN. Porém, há algumas diferenças surpreendentes entre o TCP e o GBN. Muitas execuções do TCP armazenarão segmentos recebidos corretamente, mas fora da ordem (Stevens, 1994). Considere também o que acontece quando o remetente envia uma sequência de segmentos  $1, 2, \dots, N$  e todos os segmentos chegam ao destinatário na ordem e sem erro. Além disso, suponha que o reconhecimento para o pacote  $n < N$  se perca, mas que os  $N - 1$  reconhecimentos restantes cheguem ao remetente antes do esgotamento de suas respectivas temporizações. Nesse exemplo, o GBN retransmitiria não só o pacote  $n$ , mas também todos os subsequentes  $n + 1, n + 2, \dots, N$ . O TCP, por outro lado, retransmitiria no máximo um segmento, a saber,  $n$ . E mais, o TCP nem ao menos retransmitiria o segmento  $n$  se o reconhecimento para  $n + 1$  chegasse antes do final da temporização para o segmento  $n$ .

Uma modificação proposta para o TCP, denominada **reconhecimento seletivo** (RFC 2018), permite que um destinatário TCP reconheça seletivamente segmentos fora de ordem, em vez de apenas reconhecer de modo cumulativo o último segmento recebido corretamente e na ordem. Quando combinado com retransmissão seletiva – isto é, saltar a retransmissão de segmentos que já foram reconhecidos de modo seletivo pelo destinatário –, o TCP se parece muito com nosso protocolo SR genérico. Assim, o mecanismo de recuperação de erros do TCP talvez seja mais bem caracterizado como um híbrido dos protocolos GBN e SR.

### 3.5.5 Controle de fluxo

Lembre-se de que os hospedeiros de cada lado de uma conexão TCP reservam um *buffer* de recepção para a conexão. Quando a conexão TCP recebe *bytes* que estão corretos e em sequência, ele coloca os dados no *buffer* de recepção. O processo de aplicação associado lerá os dados a partir desse *buffer*, mas não necessariamente no momento em que são recebidos. Na verdade, a aplicação receptora pode estar ocupada com alguma outra tarefa e nem ao menos tentar ler os dados até muito depois da chegada deles. Se a aplicação for relativamente lenta na leitura dos dados, o remetente pode muito facilmente saturar o *buffer* de recepção da conexão por enviar demasiados dados muito rapidamente.

O TCP provê um **serviço de controle de fluxo** às suas aplicações, para eliminar a possibilidade de o remetente esgotar o *buffer* do destinatário. Assim, controle de fluxo é um serviço de compatibilização de velocidades – compatibiliza a taxa à qual o remetente está enviando com aquela à qual a aplicação receptora está lendo. Como já notamos, um TCP remetente também pode ser estrangulado em razão do congestionamento dentro da rede IP. Esse modo de controle do remetente é denominado **controle de congestionamento**, um tópico que será examinado em detalhes nas Seções 3.6 e 3.7. Mesmo que as ações executadas pelo controle de fluxo e pelo controle de congestionamento sejam semelhantes (a regulagem [*throttling*] do remetente), fica evidente que elas são executadas por razões muito diferentes. Infelizmente, muitos autores usam os termos de modo intercambiável, e o leitor atento tem de tomar muito cuidado para distinguir os dois casos. Vamos agora discutir como o TCP provê seu serviço de controle de fluxo. Para podermos enxergar o quadro geral, sem nos fixarmos nos detalhes, nesta seção admitiremos que essa implementação do TCP é tal que o receptor TCP descarta segmentos fora da ordem.

O TCP oferece serviço de controle de fluxo ao fazer o *remetente* manter uma variável denominada **janela de recepção**. De modo informal, a janela de recepção é usada para dar ao remetente uma ideia do espaço de *buffer* livre disponível no destinatário. Como o TCP é *full-duplex*, o remetente de cada lado da conexão mantém uma janela de recepção distinta. Vamos examinar a janela de recepção no contexto de uma transferência de arquivo. Suponha que o hospedeiro A esteja enviando um arquivo grande ao hospedeiro B por uma conexão TCP. O hospedeiro B aloca um *buffer* de recepção a essa conexão; denominemos seu tamanho *RcvBuffer*. De tempos em tempos, o processo de aplicação no hospedeiro B faz a leitura do *buffer*. São definidas as seguintes variáveis:

- *LastByteRead*: número do último *byte* na cadeia de dados lido do *buffer* pelo processo de aplicação em B.
- *LastByteRcvd*: número do último *byte* na cadeia de dados que chegou da rede e foi colocado no *buffer* de recepção de B.

Como o TCP não tem permissão para saturar o *buffer* alocado, devemos ter:

$$\text{LastByteRcvd} - \text{LastByteRead} \leq \text{RcvBuffer}$$

A janela de recepção, denominada *rwnd*, é ajustada para a quantidade de espaço disponível no *buffer*:

$$\text{rwnd} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$

Como o espaço disponível muda com o tempo, *rwnd* é dinâmica. Esta variável está ilustrada na Figura 3.38.

Como a conexão usa a variável *rwnd* para prover o serviço de controle de fluxo? O hospedeiro B diz ao hospedeiro A quanto espaço disponível ele tem no *buffer* da conexão colocando o valor corrente de *rwnd* no campo de janela de recepção de cada segmento que envia a A. No começo, o hospedeiro B estabelece *rwnd* = *RcvBuffer*. Note que, para conseguir isso, o hospedeiro B deve monitorar diversas variáveis específicas da conexão.

O hospedeiro A, por sua vez, monitora duas variáveis, *LastByteSent* e *LastByteAcked*, cujos significados são óbvios. Note que a diferença entre essas duas variáveis, *LastByteSent* - *LastByteAcked*, é a quantidade de dados não reconhecidos que A enviou para a conexão. Mantendo a quantidade de dados não reconhecidos menor que o valor de *rwnd*, o hospedeiro A tem certeza de que não está fazendo transbordar o *buffer* de recepção no hospedeiro B. Assim, A tem de certificar-se, durante toda a duração da conexão, de que:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{rwnd}$$

Há um pequeno problema técnico com esse esquema. Para percebê-lo, suponha que o *buffer* de recepção do hospedeiro B fique tão cheio que *rwnd* = 0. Após anunciar ao hospedeiro A que *rwnd* = 0, imagine que B não tenha *nada* para enviar ao hospedeiro A. Agora considere o que acontece. Enquanto o processo de aplicação em B esvazia o *buffer*, o TCP não envia novos segmentos com novos valores *rwnd* para o hospedeiro A. Na verdade, o TCP lhe enviará um segmento somente se tiver dados ou um reconhecimento para enviar. Por conseguinte, o hospedeiro A nunca será informado de que foi aberto algum espaço no *buffer* de recepção do hospedeiro B: ele ficará bloqueado e não poderá transmitir mais dados! Para resolver esse problema, a especificação do TCP requer que o hospedeiro



**Figura 3.38** A janela de recepção (*rwnd*) e o *buffer* de recepção (*RcvBuffer*).

A continue a enviar segmentos com um *byte* de dados quando a janela de recepção de B for zero. Esses segmentos serão reconhecidos pelo receptor. Por fim, o *buffer* começará a esvaziar, e os reconhecimentos conterão um valor diferente de zero em *rwnd*.

O *site* de apoio deste livro fornece uma animação interativa que ilustra a operação da janela de recepção do TCP.

Agora que descrevemos o serviço de controle de fluxo do TCP, mencionaremos de maneira breve que o UDP não provê controle de fluxo e, por consequência, segmentos podem ser perdidos no destinatário devido ao esgotamento do *buffer*. Por exemplo, considere o envio de uma série de segmentos UDP de um processo no hospedeiro A para um processo no hospedeiro B. Para uma execução UDP típica, o UDP anexará os segmentos a um *buffer* de tamanho finito que “precede” o *socket* correspondente (i.e., o *socket* para o processo). O processo lê um segmento inteiro do *buffer* por vez. Se o processo não ler os segmentos com rapidez suficiente, o *buffer* transbordará e os segmentos serão descartados.

### 3.5.6 Gerenciamento da conexão TCP

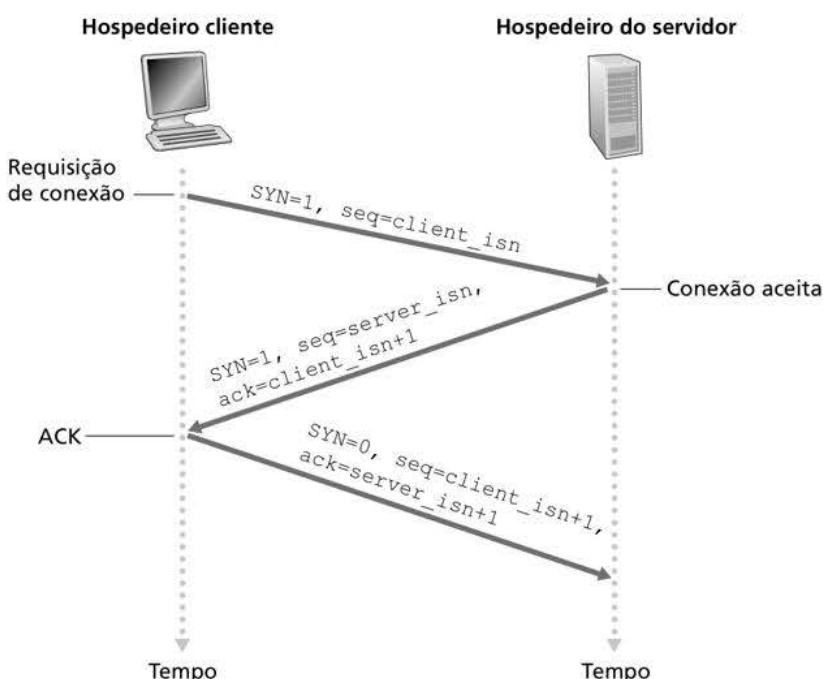
Nesta subseção, examinamos mais de perto como uma conexão TCP é estabelecida e encerrada. Embora esse tópico talvez não pareça de particular interesse, é importante, porque o estabelecimento da conexão TCP tem um peso significativo nos atrasos percebidos (p. ex., ao navegar pela Web). Além disso, muitos dos ataques mais comuns a redes – entre eles, o incrivelmente popular ataque de inundação SYN (ver nota em destaque) – exploram vulnerabilidades no gerenciamento da conexão TCP. Em primeiro lugar, vamos ver como essa conexão é estabelecida. Suponha que um processo que roda em um hospedeiro (cliente) queira iniciar uma conexão com outro processo em outro hospedeiro (servidor). O processo de aplicação cliente primeiro informa ao TCP cliente que quer estabelecer uma conexão com um processo no servidor. O TCP no cliente então estabelece uma conexão TCP com o TCP no servidor da seguinte maneira:

- *Etapa 1.* O lado cliente do TCP primeiro envia um segmento TCP especial ao lado servidor do TCP. Esse segmento não contém nenhum dado de camada de aplicação, mas um dos *bits de flag* no seu cabeçalho (veja a Figura 3.29), o *bit SYN*, é ajustado para 1. Por essa razão, o segmento é denominado um segmento SYN. Além disso, o cliente escolhe ao acaso um número de sequência inicial (*client\_isn*) e o coloca no campo de número de sequência do segmento TCP SYN inicial. Esse segmento é encapsulado em um datagrama IP e enviado ao servidor. A aleatoriedade adequada da escolha de *client\_isn* de modo a evitar certos ataques à segurança tem despertado considerável interesse (CERT, 2001–09; RFC 4987).
- *Etapa 2.* Assim que o datagrama IP contendo o segmento TCP SYN chega ao hospedeiro servidor (admitindo-se que ele chegue mesmo!), o servidor extrai o segmento TCP SYN do datagrama, aloca *buffers* e variáveis TCP à conexão e envia um segmento de aceitação de conexão ao TCP cliente. (Veremos, no Capítulo 8, que a alocação desses *buffers* e variáveis, antes da conclusão da terceira etapa da apresentação de três vias, torna o TCP vulnerável a um ataque de recusa de serviço conhecido como inundação SYN ou *SYN flood*.) Esse segmento de aceitação de conexão também não contém nenhum dado de camada de aplicação. Contudo, contém três informações importantes no cabeçalho do segmento: o *bit SYN* está com valor 1; o campo de reconhecimento do cabeçalho do segmento TCP está ajustado para *client\_isn+1*; e, por fim, o servidor escolhe seu próprio número de sequência inicial (*server\_isn*) e coloca esse valor no campo de número de sequência do cabeçalho do segmento TCP. Esse segmento de aceitação de conexão está dizendo, na prática, “Recebi seu pacote SYN para começar uma conexão com seu número de sequência inicial *client\_isn*. Concordo em estabelecer essa conexão. Meu número de sequência inicial é *server\_isn*”. O segmento de aceitação da conexão às vezes é denominado **segmento SYNACK**.

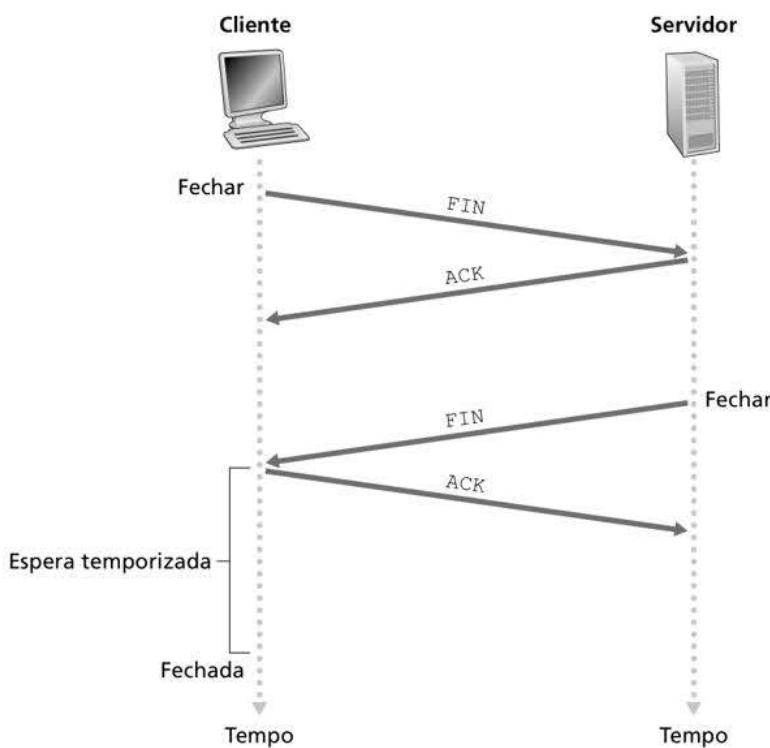
- *Etapa 3.* Ao receber o segmento SYNACK, o cliente também reserva buffers e variáveis para a conexão. O hospedeiro cliente então envia ao servidor mais um segmento. Este último reconhece o segmento de confirmação da conexão do servidor (o cliente o faz colocando o valor `server_isn+1` no campo de reconhecimento do cabeçalho do segmento TCP). O bit SYN é ajustado para 0, já que a conexão está estabelecida. A terceira etapa da apresentação de três vias pode conduzir os dados cliente/servidor na carga útil do segmento.

Completadas as três etapas, os hospedeiros cliente e servidor podem enviar segmentos contendo dados um ao outro. Em cada um desses futuros segmentos, o bit SYN estará ajustado para 0. Note que, para estabelecer a conexão, três pacotes são enviados entre dois hospedeiros, como ilustra a Figura 3.39. Por tal razão, esse procedimento de estabelecimento de conexão é com frequência denominado **apresentação de três vias (three-way handshake)**. Vários aspectos da apresentação de três vias do TCP são tratados nos exercícios ao final deste capítulo (Por que são necessários os números de sequência iniciais? Por que é preciso uma apresentação de três vias, e não apenas de duas vias?). É interessante notar que um alpinista e seu amarrador (que fica mais abaixo e cuja tarefa é passar a corda de segurança ao alpinista) usam um protocolo de comunicação de apresentação de três vias idêntico ao do TCP para garantir que ambos os lados estejam prontos antes de o alpinista iniciar a escalada.

Tudo o que é bom dura pouco, e o mesmo é válido para uma conexão TCP. Qualquer um dos dois processos que participam de uma conexão TCP pode encerrar a conexão. Quando esta termina, os “recursos” (i.e., os buffers e as variáveis) nos hospedeiros são liberados. Como exemplo, suponha que o cliente decida encerrar a conexão, como mostra a Figura 3.40. O processo de aplicação cliente emite um comando para fechar. Isso faz o TCP cliente enviar um segmento TCP especial ao processo servidor, cujo bit de flag no cabeçalho do segmento, denominado bit FIN (veja a Figura 3.29), tem valor ajustado em 1. Quando o servidor recebe esse segmento, envia de volta ao cliente um segmento de reconhecimento. O servidor então envia seu próprio segmento de encerramento, que tem o bit FIN ajustado em 1. Por fim, o cliente reconhece o segmento de encerramento do servidor. Nesse ponto, todos os recursos dos dois hospedeiros estão liberados.

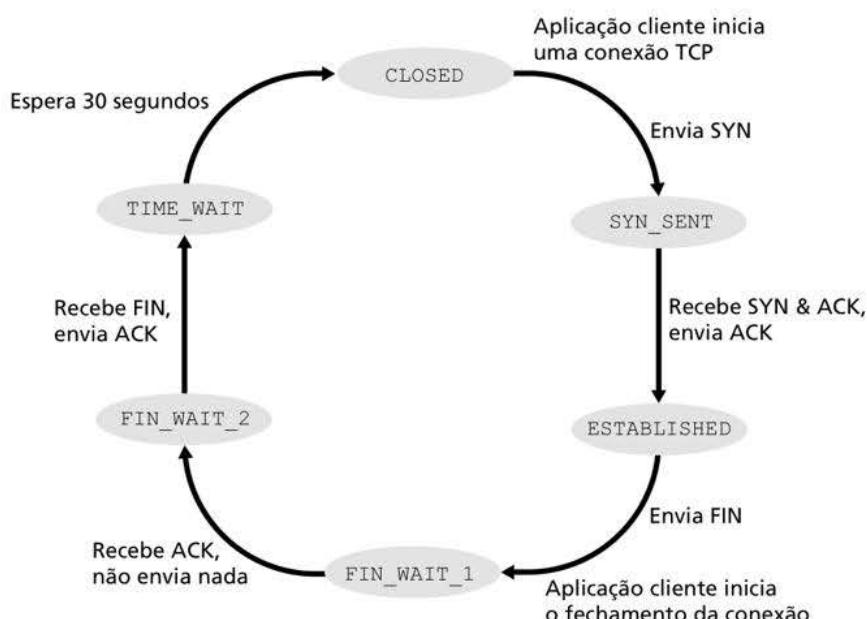


**Figura 3.39** Apresentação de três vias do TCP: troca de segmentos.



**Figura 3.40** Encerramento de uma conexão TCP.

Durante a vida de uma conexão TCP, o protocolo TCP que roda em cada hospedeiro faz transições pelos vários **estados do TCP**. A Figura 3.41 ilustra uma sequência típica de estados do TCP visitados pelo TCP *cliente*. O TCP cliente começa no estado CLOSED. A aplicação no lado cliente inicia uma nova conexão TCP (criando um objeto *Socket* como nos exemplos em Python do Capítulo 2). Isso faz o TCP no cliente enviar um segmento SYN ao



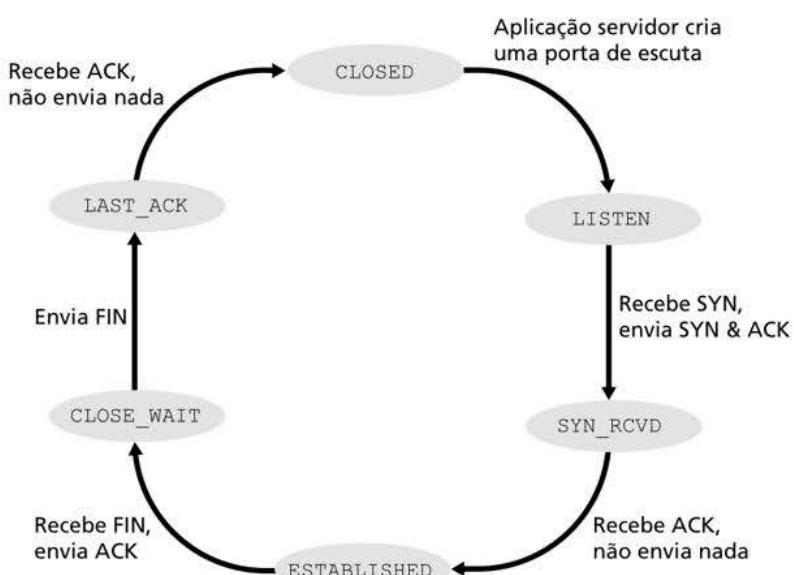
**Figura 3.41** Uma sequência típica de estados do TCP visitados por um TCP cliente.

TCP no servidor. Após o envio, o TCP cliente entra no estado SYN\_SENT e, enquanto isso, o TCP cliente espera por um segmento do TCP servidor que inclui um reconhecimento para o segmento anterior do cliente, e tem o bit SYN ajustado para o valor 1. Assim que recebe esse segmento, o TCP cliente entra no estado ESTABLISHED, quando pode enviar e receber segmentos TCP que contêm carga útil de dados (i.e., gerados pela aplicação).

Suponha que a aplicação cliente decida que quer fechar a conexão. (Note que o servidor também tem a alternativa de fechá-la.) Isso faz o TCP cliente enviar um segmento TCP com o bit FIN ajustado em 1 e entrar no estado FIN\_WAIT\_1. No estado FIN\_WAIT\_1, o TCP cliente espera por um segmento TCP do servidor com um reconhecimento. Quando recebe esse segmento, o TCP cliente entra no estado FIN\_WAIT\_2. No estado FIN\_WAIT\_2, ele espera por outro segmento do servidor com o bit FIN ajustado para 1. Após recebê-lo, o TCP cliente reconhece o segmento do servidor e entra no estado TIME\_WAIT. Esse estado permite que o TCP cliente reenvie o reconhecimento final, caso o ACK seja perdido. O tempo passado no estado TIME\_WAIT depende da implementação, mas os valores típicos são 30 segundos, 1 minuto e 2 minutos. Após a espera, a conexão se encerra formalmente e todos os recursos do lado cliente (inclusive os números de porta) são liberados.

A Figura 3.42 ilustra a série de estados normalmente visitados pelo TCP do lado servidor, admitindo-se que é o cliente quem inicia o encerramento da conexão. As transições são autoexplicativas. Nesses dois diagramas de transição de estados, mostramos apenas como uma conexão TCP é em geral estabelecida e fechada. Não descrevemos o que acontece em certos cenários patológicos, por exemplo, quando ambos os lados de uma conexão querem iniciar ou fechar ao mesmo tempo. Se estiver interessado em aprender mais sobre esse assunto e sobre outros mais avançados referentes ao TCP, consulte o abrangente livro de Stevens (1994).

Nossa discussão anterior concluiu que o cliente e o servidor estão preparados para se comunicar, isto é, que o servidor está ouvindo na porta pela qual o cliente envia seu segmento SYN. Vamos considerar o que acontece quando um hospedeiro recebe um segmento TCP cujos números de porta ou endereço IP não são compatíveis com nenhum dos *sockets* existentes no hospedeiro. Por exemplo, imagine que um hospedeiro receba um pacote TCP SYN com porta de destino 80, mas não está aceitando conexões nessa porta (i.e., não está rodando um servidor Web na porta 80). Então, ele enviará à origem um segmento especial de reinicialização. Esse segmento TCP tem o bit de flag RST ajustado para 1 (veja Seção 3.5.2).



**Figura 3.42** Uma sequência típica de estados do TCP visitados por um TCP do lado do servidor.

Assim, quando um hospedeiro envia um segmento de reinicialização, ele está dizendo à origem: “Eu não tenho um *socket* para esse segmento. Favor não enviá-lo novamente”. Quando um hospedeiro recebe um pacote UDP cujo número de porta de destino não é compatível com as portas de um UDP em curso, ele envia um datagrama ICMP especial, como será discutido no Capítulo 5.

Agora que obtivemos uma boa compreensão sobre gerenciamento da conexão TCP, vamos voltar à ferramenta de varredura de porta nmap e analisar mais precisamente como ela funciona. Para explorar uma porta TCP, digamos que a porta 6789, o nmap enviará ao computador-alvo um segmento TCP SYN com a porta de destino. Os três possíveis resultados são:

- *O computador de origem recebe um segmento TCP SYNACK de um computador-alvo.* Já que isso significa que uma aplicação está sendo executada com a porta TCP 6789 no computador-alvo, o nmap retorna “aberto”.
- *O computador de origem recebe um segmento TCP RST de um computador-alvo.* Isto significa que o segmento SYN atingiu o computador-alvo, mas este não está executando

## SEGURANÇA EM FOCO

### O ATAQUE SYN FLOOD

Vimos em nossa discussão sobre a apresentação de três vias do TCP que um servidor aloca e inicializa as variáveis da conexão e os *buffers* em resposta ao SYN recebido. O servidor, então, envia um SYNACK em resposta e aguarda um segmento ACK do cliente. Se o cliente não enviar um ACK para completar o terceiro passo da apresentação de três vias, com o tempo (em geral, após um minuto ou mais), o servidor finalizará a conexão semiaberta e recuperará os recursos alocados.

Esse protocolo de gerenciamento da conexão TCP abre caminho para um ataque DoS clássico, ou seja, o **ataque SYN flood**. Neste ataque, o vilão envia um grande número de segmentos SYN TCP, sem concluir a terceira etapa de apresentação. Com esse acúmulo de segmentos SYN, os recursos de conexão do servidor podem se esgotar depressa já que são alocados (mas nunca usados) para conexões semiabertas; clientes legítimos, então, não são atendidos. Esses ataques SYN flood estavam entre os primeiros ataques DoS documentados pelo CERT (CERT SYN, 1996). Felizmente, uma defesa eficaz, conhecida como **SYN cookies** (RFC 4987), agora é empregada na maioria dos principais sistemas operacionais. SYN cookies funcionam da seguinte forma:

- Quando o servidor recebe um segmento SYN, não se sabe se ele vem de um usuário verdadeiro ou se é parte desse ataque. Então, em vez de criar uma conexão TCP semiaberta para esse SYN, o servidor cria um número de sequência TCP inicial, que é uma função *hash* de endereços de origem e endereços de destino IP e números de porta do

segmento SYN, assim como de um número secreto conhecido apenas pelo usuário. Esse número de sequência inicial criado cuidadosamente é o assim chamado “cookie”. O servidor, então, envia ao cliente um pacote SYNACK com esse número de sequência especial. É importante mencionar que o servidor não se lembra do cookie ou de qualquer outra informação de estado correspondente ao SYN.

- Se o cliente for verdadeiro, então um segmento ACK retornará. O servidor, ao receber esse ACK, precisa verificar se ele corresponde a algum SYN enviado antes. Como isto é feito se ele não guarda nenhuma memória sobre os segmentos SYN? Como você deve ter imaginado, o processo é realizado com o cookie. Para um ACK legítimo, em especial, o valor no campo de reconhecimento é igual ao número de sequência no SYNACK (o valor do cookie, neste caso) mais um (veja Figura 3.39). O servidor, então, executará a mesma função utilizando os mesmos campos no segmento ACK (que são os mesmos que no SYN original) e o número secreto. Se o resultado da função mais um for o mesmo que o número de reconhecimento (cookie) no SYNACK do cliente, o servidor conclui que o ACK corresponde a um segmento SYN anterior e, portanto, é válido. O servidor, então, cria uma conexão totalmente aberta com um *socket*.
- Por outro lado, se o cliente não retorna um segmento ACK, então o SYN original não causou nenhum dano ao servidor, uma vez que este não alocou nenhum recurso em resposta ao SYN falso original.

uma aplicação com a porta TCP 6789. Mas o atacante, pelo menos, sabe que os segmentos destinados ao computador na porta 6789 não estão bloqueados pelo *firewall* no percurso entre o computador de origem e o alvo. (*Firewalls* são abordados no Capítulo 8.)

- *A origem não recebe nada.* Isto, provavelmente, significa que o segmento SYN foi bloqueado por um *firewall* no caminho e nunca atingiu o computador-alvo.

O nmap é uma ferramenta potente, que pode “sondar o local” não só em busca de portas TCP abertas, mas também de portas UDP abertas, *firewalls* e suas configurações, e até mesmo as versões de aplicações e sistemas operacionais. A maior parte é feita pela manipulação dos segmentos de gerenciamento da conexão TCP. É possível fazer *download* do nmap pelo site <[www.nmap.org](http://www.nmap.org)>.

Com isso, concluímos nossa introdução ao controle de erro e controle de fluxo em TCP. Voltaremos ao TCP na Seção 3.7, e então examinaremos em mais detalhes o controle de congestionamento do TCP. Antes, contudo, vamos analisar a questão do controle de congestionamento em um contexto mais amplo.

## 3.6 PRINCÍPIOS DE CONTROLE DE CONGESTIONAMENTO

Nas seções anteriores, examinamos os princípios gerais e os mecanismos específicos do TCP usados para prover um serviço de transferência confiável de dados em relação à perda de pacotes. Mencionamos antes que, na prática, essa perda resulta, de modo característico, de uma saturação de *buffers* de roteadores à medida que a rede fica congestionada. Assim, a retransmissão de pacotes trata de um sintoma de congestionamento de rede (a perda de um segmento específico de camada de transporte), mas não trata da causa do congestionamento da rede: demasiadas fontes tentando enviar dados a uma taxa muito alta. Para tratar da causa do congestionamento de rede, são necessários mecanismos para regular os remetentes quando ele ocorre.

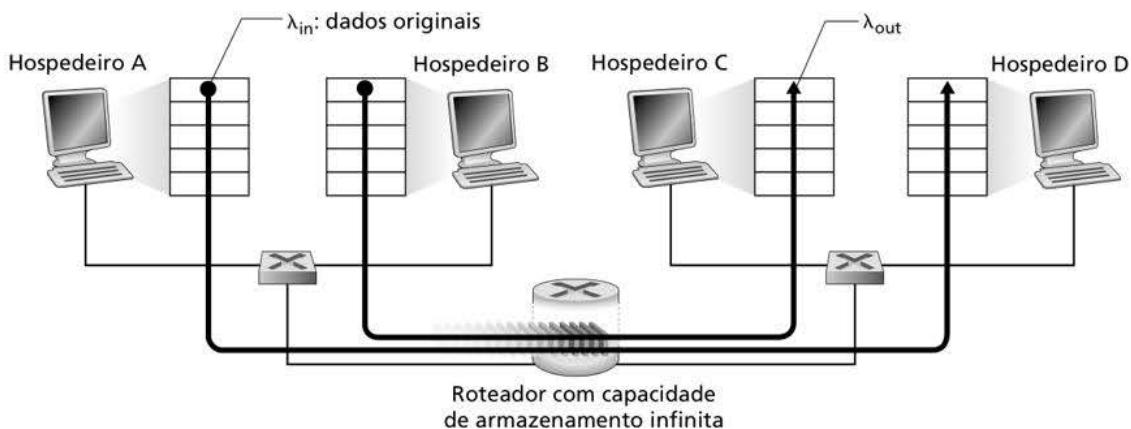
Nesta seção, consideramos o problema do controle de congestionamento em um contexto geral, buscando entender por que a congestão é algo ruim, como o congestionamento de rede se manifesta no desempenho recebido por aplicações da camada superior e várias medidas que podem ser adotadas para evitá-lo ou reagir a ele. Esse estudo mais geral do controle de congestionamento é apropriado, já que, como acontece com a transferência confiável de dados, o congestionamento é um dos “dez mais” da lista de problemas fundamentalmente importantes no trabalho em rede. A seção seguinte contém um estudo detalhado do algoritmo de controle de congestionamento do TCP.

### 3.6.1 As causas e os custos do congestionamento

Vamos começar nosso estudo geral do controle de congestionamento examinando três cenários de complexidade crescente nos quais ele ocorre. Em cada caso, examinaremos, primeiro, por que ele ocorre e, depois, seu custo (no que se refere aos recursos não utilizados integralmente e ao baixo desempenho recebido pelos sistemas finais). Não focalizaremos (ainda) como reagir a ele, ou evitá-lo; preferimos estudar uma questão mais simples, que é entender o que acontece quando hospedeiros aumentam sua taxa de transmissão e a rede fica congestionada.

#### Cenário 1: dois remetentes, um roteador com *buffers* infinitos

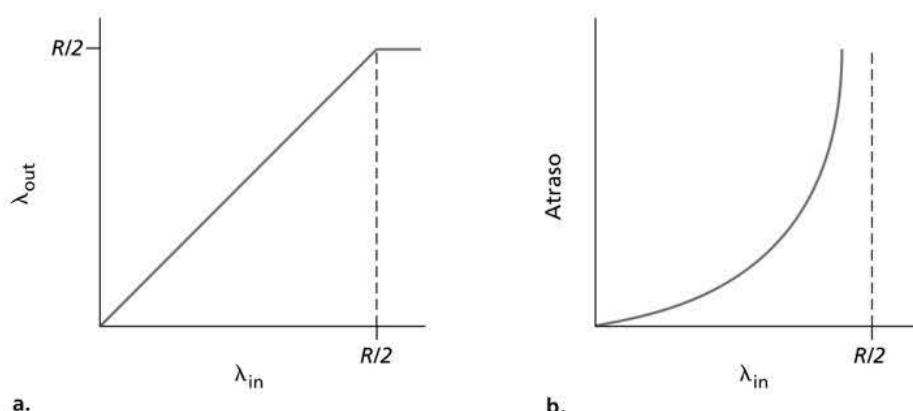
Começamos considerando o que talvez seja o cenário de congestionamento mais simples possível: dois hospedeiros (A e B), cada um usando uma conexão que compartilha um único trecho de rede entre a origem e o destino, conforme mostra a Figura 3.43.



**Figura 3.43** Cenário de congestionamento 1: duas conexões compartilhando um único roteador com número infinito de buffers.

Vamos admitir que a aplicação no hospedeiro A esteja enviando dados para a conexão (p. ex., passando dados para o protocolo de camada de transporte por um *socket*) a uma taxa média de  $\lambda_{in}$  bytes/s. Esses dados são originais no sentido de que cada unidade de dados é enviada para dentro do *socket* apenas uma vez. O protocolo de camada de transporte subjacente é simples. Os dados são encapsulados e enviados; não há recuperação de erros (p. ex., retransmissão), controle de fluxo, nem controle de congestionamento. Desprezando a sobrecarga adicional causada pela adição de informações de cabeçalhos de camada de transporte e de camadas mais baixas, a taxa à qual o hospedeiro A oferece tráfego ao roteador nesse primeiro cenário é  $\lambda_{in}$  bytes/s. O hospedeiro B funciona de maneira semelhante, e admitimos, por simplicidade, que ele também esteja enviando dados a uma taxa de  $\lambda_{in}$  bytes/s. Os pacotes dos hospedeiros A e B passam por um roteador e por um enlace de saída compartilhado de capacidade  $R$ . O roteador tem *buffers* que lhe permitem armazenar os pacotes que chegam quando a taxa de chegada excede a capacidade do enlace de saída. No primeiro cenário, admitimos que o roteador tenha capacidade de armazenamento infinita.

A Figura 3.44 apresenta o desempenho da conexão do hospedeiro A nesse primeiro cenário. O gráfico da esquerda mostra a **vazão por conexão** (número de bytes por segundo no destinatário) como uma função da taxa de envio da conexão. Para uma taxa de transmissão entre 0 e  $R/2$ , a vazão no destinatário é igual à velocidade de envio do remetente – tudo o que este envia é recebido no destinatário com um atraso finito. Quando a velocidade de envio estiver acima de  $R/2$ , contudo, a vazão será somente  $R/2$ . Esse limite superior da vazão é consequência do compartilhamento da capacidade do enlace entre duas conexões. O enlace



**Figura 3.44** Cenário de congestionamento 1: vazão e atraso em função da taxa de envio do hospedeiro.

simplesmente não consegue entregar os pacotes a um destinatário com uma taxa em estado constante que excede  $R/2$ . Não importa quão altas sejam as taxas de envio ajustadas nos hospedeiros A e B, eles jamais alcançarão uma vazão maior do que  $R/2$ .

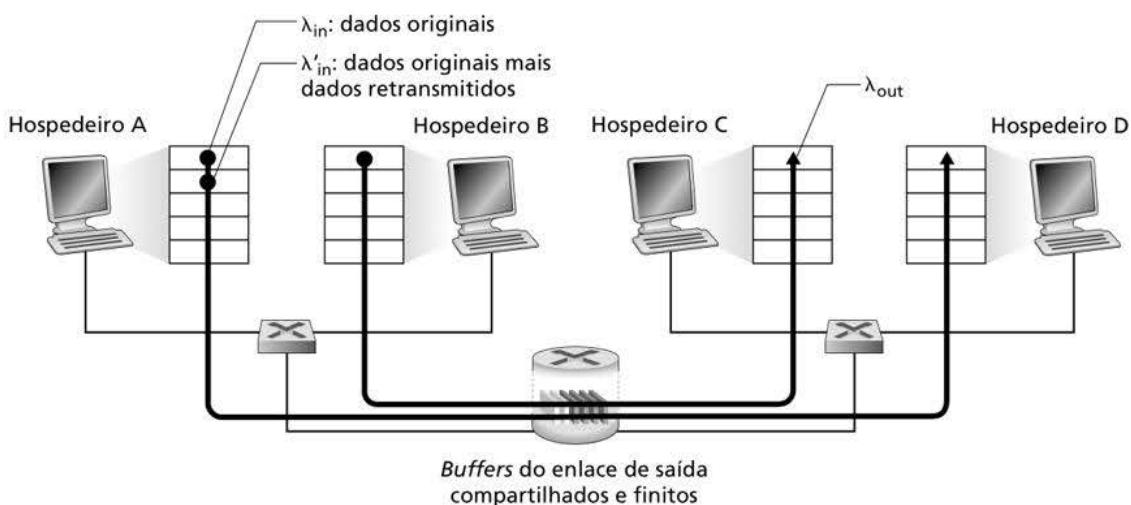
Alcançar uma vazão de  $R/2$  por conexão pode até parecer uma coisa boa, pois o enlace está sendo integralmente utilizado para entregar pacotes no destinatário. No entanto, o gráfico do lado direito da Figura 3.44 mostra as consequências de operar próximo à capacidade máxima do enlace. À medida que a taxa de envio se aproxima de  $R/2$  (partindo da esquerda), o atraso médio fica cada vez maior. Quando a taxa de envio ultrapassa  $R/2$ , o número médio de pacotes na fila no roteador é ilimitado, e o atraso médio entre a fonte e o destino se torna infinito (admitindo que as conexões operem a essas velocidades de transmissão durante um período infinito e que a capacidade de armazenamento também seja infinita). Assim, embora operar a uma vazão agregada próxima a  $R$  possa ser ideal do ponto de vista da vazão, está bem longe de ser ideal do ponto de vista do atraso. *Mesmo nesse cenário (extremamente) idealizado, já descobrimos um custo da rede congestionada – há grandes atrasos de fila quando a taxa de chegada de pacotes se aproxima da capacidade do enlace.*

### Cenário 2: dois remetentes, um roteador com buffers finitos

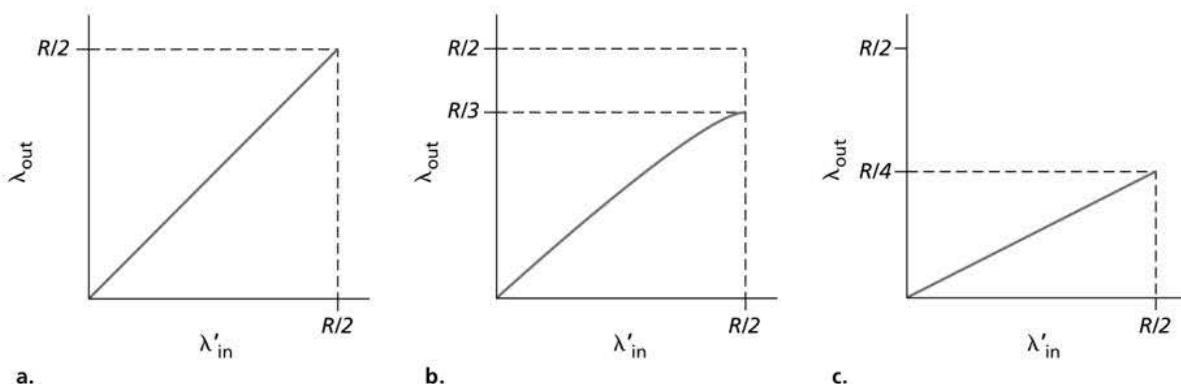
Vamos agora modificar um pouco o cenário 1 dos dois modos seguintes (veja a Figura 3.45).

Primeiro, admitamos que a capacidade de armazenamento do roteador seja finita. Em uma situação real, essa suposição teria como consequência o descarte de pacotes que chegam a um buffer que já está cheio. Segundo, admitamos que cada conexão seja confiável. Se um pacote contendo um segmento de camada de transporte for descartado no roteador, o remetente por fim o retransmitirá. Como os pacotes podem ser retransmitidos, agora temos de ser mais cuidadosos com o uso da expressão “*taxa de envio*”. Especificamente, vamos de novo designar a taxa com que a aplicação envia dados originais para dentro do *socket* como  $\lambda_{in}$  bytes/s. A taxa com que a camada de transporte envia segmentos (contendo dados originais e dados retransmitidos) para dentro da rede será denominada  $\lambda'_{in}$  bytes/s. Essa taxa ( $\lambda'_{in}$ ) às vezes é denominada **carga oferecida** à rede.

O desempenho obtido no cenário 2 agora dependerá muito de como a retransmissão é realizada. Primeiro, considere o caso não realista em que o hospedeiro A consiga, de algum modo (fazendo mágica!), determinar se um buffer do roteador está livre no roteador e, portanto, envia um pacote apenas quando um buffer estiver livre. Nesse caso, não ocorreria nenhuma perda,  $\lambda_{in}$  seria igual a  $\lambda'_{in}$  e a vazão da conexão seria igual a  $\lambda_{in}$ . Esse caso é mostrado pela curva superior da Figura 3.46(a). Do ponto de vista da vazão, o desempenho



**Figura 3.45** Cenário 2: dois hospedeiros (com retransmissões) e um roteador com buffers finitos.



**Figura 3.46** Desempenho no cenário 2 com buffers finitos.

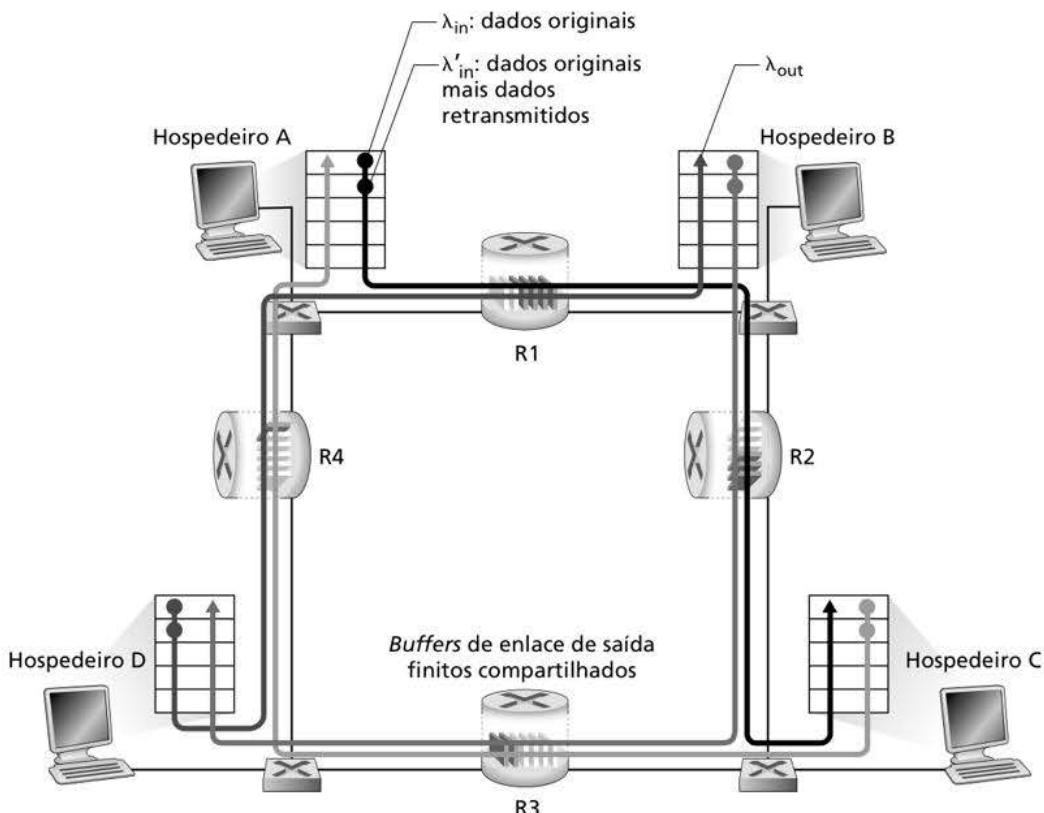
é ideal – tudo o que é enviado é recebido. Note que, nesse cenário, a taxa média de envio do hospedeiro não pode ultrapassar  $R/2$ , já que admitimos que nunca ocorre perda de pacote.

Considere, em seguida, o caso um pouco mais realista em que o remetente retransmite apenas quando sabe, com certeza, que o pacote foi perdido. (De novo, essa suposição é um pouco forçada. Contudo, é possível ao hospedeiro remetente ajustar seu temporizador de retransmissão para uma duração longa o suficiente para ter razoável certeza de que um pacote que não foi reconhecido foi perdido.) Nesse caso, o desempenho pode ser parecido com o mostrado na Figura 3.46(b). Para avaliar o que está acontecendo aqui, considere o caso em que a carga oferecida,  $\lambda'_{\text{in}}$  (a taxa de transmissão dos dados originais mais as retransmissões), é igual a  $R/2$ . De acordo com a Figura 3.46(b), nesse valor de carga oferecida, a velocidade com a qual os dados são entregues à aplicação do destinatário é  $R/3$ . Assim, de  $0,5R$  unidade de dados transmitida,  $0,333R$  byte/s (em média) são dados originais e  $0,166R$  byte/s (em média) são dados retransmitidos. *Observamos aqui outro custo de uma rede congestionada – o remetente deve realizar retransmissões para compensar os pacotes descartados (perdidos) pelo esgotamento do buffer.*

Finalmente, vamos considerar o caso em que a temporização do remetente se esgota prematuramente e ele retransmita um pacote que ficou atrasado na fila, mas que ainda não está perdido. Aqui, tanto o pacote de dados original quanto a retransmissão podem alcançar o destinatário. É claro que o destinatário precisa apenas de uma cópia desse pacote e descartará a retransmissão. Nesse caso, o trabalho realizado pelo roteador ao repassar a cópia retransmitida do pacote original é desperdiçado, pois o destinatário já terá recebido a cópia original do pacote. Em vez disso, seria melhor o roteador usar a capacidade de transmissão do enlace para enviar um pacote diferente. *Eis aqui mais um custo da rede congestionada – retransmissões desnecessárias feitas pelo remetente em face de grandes atrasos podem fazer um roteador usar sua largura de banda de enlace para repassar cópias desnecessárias de um pacote.* A Figura 3.46(c) mostra a vazão versus a carga oferecida admitindo-se que cada pacote seja enviado (em média) duas vezes pelo roteador. Visto que cada pacote é enviado duas vezes, a vazão terá um valor assintótico de  $R/4$  à medida que a carga oferecida se aproximar de  $R/2$ .

### Cenário 3: quatro remetentes, roteadores com buffers finitos e trajetos com múltiplos roteadores

Em nosso cenário final de congestionamento, quatro hospedeiros transmitem pacotes sobre trajetos sobrepostos que apresentam dois saltos, como ilustrado na Figura 3.47. Novamente admitimos que cada hospedeiro use um mecanismo de temporização/retransmissão para executar um serviço de transferência confiável de dados, que todos os hospedeiros tenham o mesmo valor de  $\lambda_{\text{in}}$  e que todos os enlaces dos roteadores tenham capacidade de  $R$  bytes/s.

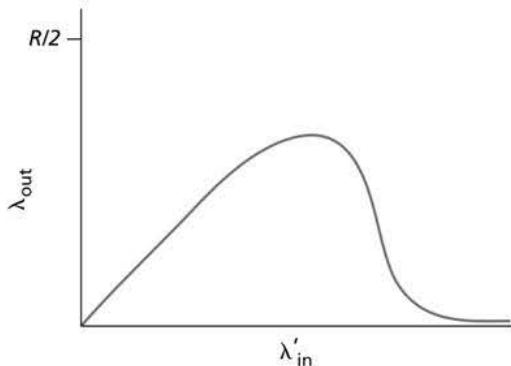


**Figura 3.47** Quatro remetentes, roteadores com *buffers* finitos e trajetos com vários saltos.

Vamos considerar a conexão do hospedeiro A ao hospedeiro C que passa pelos roteadores R1 e R2. A conexão A–C compartilha o roteador R1 com a conexão D–B e o roteador R2 com a conexão B–D. Para valores extremamente pequenos de  $\lambda_{in}$ , esgotamentos de *buffers* são raros (como acontecia nos cenários de congestionamento 1 e 2), e a vazão é quase igual à carga oferecida. Para valores de  $\lambda_{in}$  um pouco maiores, a vazão correspondente é também maior, pois mais dados originais estão sendo transmitidos para a rede e entregues no destino, e os esgotamentos ainda são raros. Assim, para valores pequenos de  $\lambda_{in}$ , um aumento em  $\lambda_{in}$  resulta em um aumento em  $\lambda_{out}$ .

Como já analisamos o caso de tráfego extremamente baixo, vamos examinar aquele em que  $\lambda_{in}$  (e, portanto,  $\lambda'_{in}$ ) é extremamente alto. Considere o roteador R2. O tráfego A–C que chega ao roteador R2 (após ter sido repassado de R1) pode ter uma taxa de chegada em R2 de, no máximo,  $R$ , que é a capacidade do enlace de R1 a R2, não importando qual seja o valor de  $\lambda_{in}$ . Se  $\lambda'_{in}$  for extremamente alto para todas as conexões (incluindo a conexão B–D), então a taxa de chegada do tráfego B–D em R2 poderá ser muito maior do que a taxa do tráfego A–C. Como os tráfegos A–C e B–D têm de competir no roteador R2 pelo espaço limitado de *buffer*, a quantidade de tráfego A–C que consegue passar por R2 (i.e., que não se perde pelo congestionamento de *buffer*) diminui cada vez mais à medida que a carga oferecida de B–D vai ficando maior. No limite, quando a carga oferecida se aproxima do infinito, um *buffer* vazio em R2 é logo preenchido por um pacote B–D, e a vazão da conexão A–C em R2 cai a zero. Isso, por sua vez, implica que a vazão fim a fim de A–C vai a zero no limite de tráfego pesado. Essas considerações dão origem ao comportamento da carga oferecida versus a vazão mostrada na Figura 3.48.

A razão para o decréscimo final da vazão com o crescimento da carga oferecida é evidente quando considerarmos a quantidade de trabalho desperdiçado realizado pela rede. No cenário de alto tráfego que acabamos de descrever, sempre que um pacote é descartado em um segundo roteador, o trabalho realizado pelo primeiro para enviar o pacote ao



**Figura 3.48** Desempenho obtido no cenário 3, com buffers finitos e trajetos com múltiplos roteadores.

segundo acaba sendo “desperdiçado”. A rede teria se saído igualmente bem (melhor dizendo, igualmente mal) se o primeiro roteador tivesse apenas descartado aquele pacote e ficado inativo. Sendo mais objetivos, a capacidade de transmissão utilizada no primeiro roteador para enviar o pacote ao segundo teria sido maximizada para transmitir um pacote diferente. (P. ex., ao selecionar um pacote para transmissão, seria melhor que um roteador desse prioridade a pacotes que já atravessaram alguns roteadores anteriores.) *Portanto, vemos aqui mais um custo, o do descarte de pacotes devido ao congestionamento – quando um pacote é descartado ao longo de um caminho, a capacidade de transmissão que foi usada em cada um dos enlaces anteriores para repassar o pacote até o ponto em que foi descartado acaba sendo desperdiçada.*

### 3.6.2 Mecanismos de controle de congestionamento

Na Seção 3.7, examinaremos em detalhes os mecanismos específicos do TCP para o controle de congestionamento. Aqui, identificaremos os dois procedimentos mais comuns adotados, na prática, para esse controle. Além disso, examinaremos arquiteturas específicas de rede e protocolos de controle que incorporam tais procedimentos.

No nível mais alto, podemos distinguir mecanismos de controle de congestionamento conforme a camada de rede ofereça ou não assistência explícita à camada de transporte com a finalidade de controle de congestionamento.

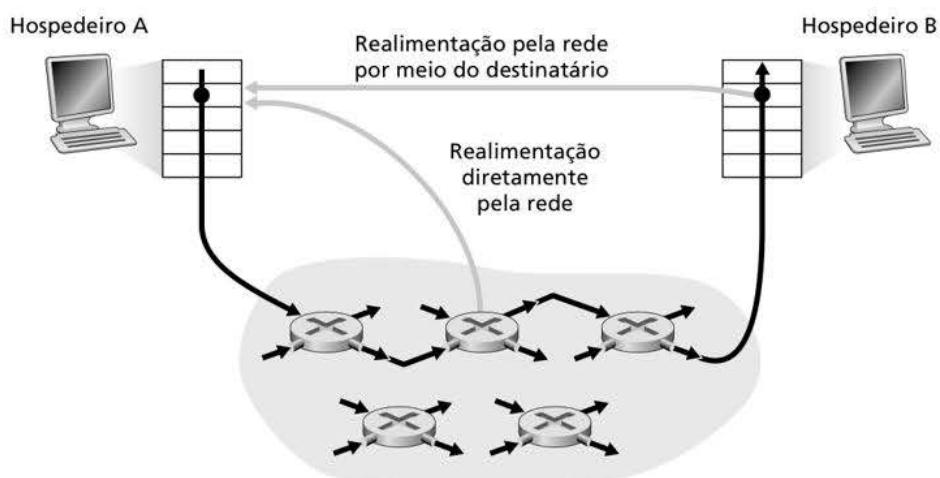
- *Controle de congestionamento sim a fim.* Nesse método, a camada de rede não fornece nenhum suporte explícito à camada de transporte com a finalidade de controle de congestionamento. Até mesmo a presença de congestionamento na rede deve ser intuída pelos sistemas finais com base apenas na observação do comportamento da rede (p. ex., perda de pacotes e atraso). Veremos na Seção 3.7.1 que o TCP adota esse método fim a fim para o controle de congestionamento, uma vez que a camada IP não fornece realimentação de informações aos hospedeiros quanto ao congestionamento da rede. A perda de segmentos TCP (apontada por uma temporização ou por três reconhecimentos duplicados) é tomada como indicação de congestionamento de rede, e o TCP reduz o tamanho da janela de acordo com isso. Veremos também que as novas propostas para o TCP usam valores de atraso de viagem de ida e volta crescentes como indicadores de aumento do congestionamento da rede.
- *Controle de congestionamento assistido pela rede.* Com esse método, os roteadores fornecem retroalimentação específica de informações ao remetente a respeito do estado de congestionamento na rede. Essa retroalimentação pode ser tão simples como um único bit indicando o congestionamento em um enlace, abordagem adotada nas primeiras arquiteturas de rede IBM SNA (Schwartz, 1982), DECnet (Jain, 1989;

Ramakrishnan, 1990) e ATM (Black, 1995). A retroalimentação mais sofisticada de rede também é possível. Por exemplo, no controle de congestionamento **ATM ABR (Available Bit Rate)** que estudaremos mais adiante, um roteador informa explicitamente ao remetente a taxa de envio máxima que ele (o roteador) pode suportar em um enlace de saída. Como mencionado acima, as versões padrões para a Internet do IP e do TCP adotam uma abordagem fim a fim em relação ao controle de congestionamento. Na Seção 3.7.2, no entanto, veremos que, mais recentemente, o IP e o TCP passaram a ter também a opção de implementar o controle de congestionamento assistido pela rede.

Para controle de congestionamento assistido pela rede, a informação sobre congestionamento é em geral realimentada da rede para o remetente por um de dois modos, como ilustra a Figura 3.49. Retroalimentação direta pode ser enviada de um roteador de rede ao remetente. Esse modo de notificação em geral toma a forma de um pacote de congestionamento (*choke packet*) (que, em essência, diz: “Estou congestionado!”). O segundo modo de notificação ocorre quando um roteador marca/atualiza um campo em um pacote que está fluindo do remetente ao destinatário para indicar congestionamento. Ao receber um pacote marcado, o destinatário informa ao remetente a indicação de congestionamento. Esse último modo de notificação leva o tempo total de uma viagem de ida e volta.

### 3.7 CONTROLE DE CONGESTIONAMENTO NO TCP

Nesta seção, voltamos ao estudo do TCP. Como aprendemos na Seção 3.5, o TCP provê um serviço de transferência confiável entre dois processos que rodam em hospedeiros diferentes. Outro componente de extrema importância do TCP é seu mecanismo de controle de congestionamento. Como indicamos na seção anterior, o que poderíamos chamar de TCP “clássico” – a versão do TCP padronizada no (RFC 2581) e mais recentemente no (RFC 5681) – usa controle de congestionamento fim a fim em vez de controle assistido pela rede, já que a camada IP não fornece aos sistemas finais realimentação explícita relativa ao congestionamento da rede. Abordaremos essa versão “clássica” do TCP mais profundamente na Seção 7.3.1. Na Seção 7.3.2, analisaremos novas variantes do TCP, que usam uma indicação de congestionamento explícita fornecida pela camada de rede, ou diferem um pouco do TCP “clássico” em diversos aspectos diferentes. A seguir, trabalharemos o desafio de criar equidade entre os fluxos da camada de transporte que precisam compartilhar um enlace congestionado.



**Figura 3.49** Dois caminhos de realimentação para informação sobre congestionamento indicado pela rede.

### 3.7.1 Controle de congestionamento no TCP clássico

A abordagem adotada pelo TCP é obrigar cada remetente a limitar a taxa pela qual enviam tráfego para sua conexão como uma função do congestionamento de rede percebido. Se um remetente TCP perceber que há pouco congestionamento no caminho entre ele e o destinatário, aumentará sua taxa de envio; se perceber que há congestionamento, reduzirá sua taxa de envio. Mas essa abordagem levanta três questões. Primeiro, como um remetente TCP limita a taxa pela qual envia tráfego para sua conexão? Segundo, como um remetente TCP percebe que há congestionamento entre ele e o destinatário? E terceiro, que algoritmo o remetente deve utilizar para modificar sua taxa de envio como uma função do congestionamento fim a fim percebido?

Para começar, vamos examinar como um remetente TCP limita a taxa de envio pela qual envia tráfego para sua conexão. Na Seção 3.5, vimos que cada lado de uma conexão TCP consiste em um *buffer* de recepção, um *buffer* de envio e diversas variáveis (`LastByteRead`, `rwnd` e assim por diante). O mecanismo de controle de congestionamento que opera no remetente monitora uma variável adicional, a **janela de congestionamento**. Esta, denominada `cwnd`, impõe uma limitação à taxa pela qual um remetente TCP pode enviar tráfego para dentro da rede. Especificamente, a quantidade de dados não reconhecidos em um hospedeiro não pode exceder o mínimo de `cwnd` e `rwnd`, ou seja:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \min\{\text{cwnd}, \text{rwnd}\}$$

Para concentrar a discussão no controle de congestionamento (ao contrário do controle de fluxo), daqui em diante vamos admitir que o *buffer* de recepção TCP seja tão grande que a limitação da janela de recepção pode ser desprezada; assim, a quantidade de dados não reconhecidos no remetente estará limitada apenas por `cwnd`. Vamos admitir também que o remetente sempre tenha dados para enviar, isto é, que todos os segmentos dentro da janela de congestionamento sejam enviados.

A restrição citada limita a quantidade de dados não reconhecidos no remetente e, por conseguinte, limita indiretamente a taxa de envio do remetente. Para entender melhor, considere uma conexão na qual perdas e atrasos de transmissão de pacotes sejam desprezíveis. Então, em linhas gerais, no início de cada RTT, a limitação permite que o remetente envie `cwnd bytes` de dados para a conexão; ao final do RTT, o remetente recebe reconhecimentos para os dados. Assim, a taxa de envio do remetente é aproximadamente `cwnd/RTT bytes por segundo`. Portanto, ajustando o valor de `cwnd`, o remetente pode ajustar a taxa pela qual envia dados para sua conexão.

Em seguida, vamos considerar como um remetente TCP percebe que há congestionamento no caminho entre ele e o destino. Definimos “evento de perda” em um remetente TCP como a ocorrência de um esgotamento de temporização (*timeout*) ou do recebimento de três ACKs duplicados do destinatário (lembre-se da nossa discussão, na Seção 3.5.4, do evento de temporização apresentado na Figura 3.33 e da subsequente modificação para incluir transmissão rápida quando do recebimento de três ACKs duplicados). Quando há congestionamento excessivo, então um (ou mais) *buffer* de roteadores ao longo do caminho transborda, fazendo um datagrama (contendo um segmento TCP) ser descartado. O datagrama descartado, por sua vez, resulta em um evento de perda no remetente – ou um esgotamento de temporização ou o recebimento de três ACKs duplicados –, que é tomado por ele como uma indicação de congestionamento no caminho remetente-destinatário.

Já consideramos como é detectado o congestionamento; agora vamos analisar o caso mais otimista de uma rede livre de congestionamento, isto é, quando não ocorre um evento de perda. Nesse caso, o TCP remetente receberá reconhecimentos para segmentos não reconhecidos antes. Como veremos, o TCP considerará a chegada desses reconhecimentos como uma indicação de que tudo está bem – os segmentos que estão sendo transmitidos para a rede estão sendo entregues com sucesso no destinatário – e usará os reconhecimentos para aumentar o tamanho de sua janela de congestionamento (e, por conseguinte, sua taxa de transmissão). Note que, se os reconhecimentos chegarem ao remetente a uma taxa

relativamente baixa (p. ex., se o atraso no caminho fim a fim for alto ou se nele houver um enlace de baixa largura de banda), então a janela de congestionamento será aumentada a uma taxa um tanto baixa. Por outro lado, se os reconhecimentos chegarem a uma taxa alta, então a janela de congestionamento será aumentada mais depressa. Como o TCP utiliza reconhecimentos para acionar (ou regular) o aumento de tamanho de sua janela de congestionamento, diz-se que o TCP é **autorregulado** (*self-clocking*).

Dado o *mecanismo* de ajustar o valor de *cwnd* para controlar a taxa de envio, permanece a importante pergunta: *Como* um remetente TCP deve determinar a taxa pela qual deve enviar? Se os remetentes TCP enviam coletivamente muito rápido, eles podem congestionar a rede, levando ao tipo de congestionamento que vimos na Figura 3.48. De fato, a versão de TCP que vamos estudar de modo breve foi desenvolvida em resposta aos congestionamentos da Internet observados em versões anteriores do TCP (Jacobson, 1988). Entretanto, se os remetentes forem muito cautelosos e enviarem lentamente, eles podem subutilizar a largura de banda na rede; ou seja, os remetentes TCP podem enviar a uma taxa mais alta sem congestionar a rede. Então, como os remetentes TCP determinam suas taxas de envio de um modo que não congestionem, mas, ao mesmo tempo, façam uso de toda a largura de banda? Os remetentes TCP são claramente coordenados, ou existe uma abordagem distribuída na qual eles podem ajustar suas taxas de envio baseando-se apenas nas informações locais? O TCP responde a essas perguntas utilizando os seguintes princípios.

- *Um segmento perdido implica congestionamento, portanto, a taxa do remetente TCP deve diminuir quando um segmento é perdido.* Lembre-se da nossa discussão na Seção 3.5.4, de que um evento de esgotamento do temporizador ou o recebimento de quatro reconhecimentos para dado segmento (um ACK original e, depois, três ACKs duplicados) é interpretado como uma indicação de “evento de perda” absoluto do segmento subsequente ao ACK quadruplicado, acionando uma retransmissão do segmento perdido. De um ponto de vista do controle, a pergunta é como o remetente TCP deve diminuir sua janela de congestionamento e, portanto, sua taxa de envio, em resposta ao suposto evento de perda.
- *Um segmento reconhecido indica que a rede está enviando os segmentos do remetente ao destinatário e, por isso, a taxa do remetente pode aumentar quando um ACK chegar para um segmento não reconhecido antes.* A chegada de reconhecimentos é considerada uma indicação absoluta de que tudo está bem – os segmentos estão sendo enviados com sucesso do remetente ao destinatário; assim, a rede não fica congestionada. Dessa forma, o tamanho da janela de congestionamento pode ser elevado.
- *Busca por largura de banda.* Dados os ACKs que indicam um percurso de origem a destino sem congestionamento, e eventos de perda que indicam um percurso congestionado, a estratégia do TCP de ajustar sua taxa de transmissão é aumentar a taxa em resposta aos ACKs que chegam até que ocorra um evento de perda, momento em que a taxa de transmissão diminui. Desse modo, o remetente TCP aumenta sua taxa de transmissão para buscar a taxa pela qual o congestionamento se inicia, recua dela e de novo faz a busca para ver se a taxa de início do congestionamento foi alterada. O comportamento do remetente TCP é análogo a uma criança que pede (e ganha) cada vez mais doces até por fim receber um “Não！”, recuar, mas começar a pedir novamente pouco tempo depois. Observe que não há nenhuma sinalização explícita de congestionamento pela rede – os ACKs e eventos de perda servem como sinais implícitos – e que cada remetente TCP atua em informações locais em momentos diferentes de outros.

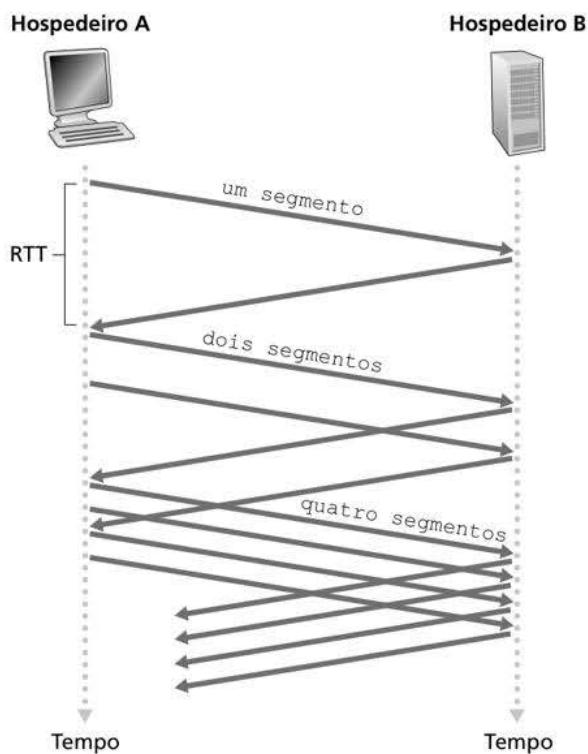
Após essa visão geral sobre controle de congestionamento no TCP, agora podemos considerar os detalhes do renomado **algoritmo de controle de congestionamento TCP**, sendo primeiro descrito em Jacobson (1988) e padronizado em (RFC 5681). O algoritmo possui três componentes principais: (1) partida lenta, (2) contenção de congestionamento e (3) recuperação rápida. A partida lenta e a contenção de congestionamento são componentes obrigatórios do TCP, diferenciando em como eles aumentam o tamanho do *cwnd* em resposta a ACKs recebidos. Abordaremos em poucas palavras que a partida lenta aumenta o tamanho

do  $cwnd$  de forma mais rápida (apesar do nome!) do que a contenção de congestionamento. A recuperação rápida é recomendada, mas não exigida, para remetentes TCP.

### Partida lenta

Quando uma conexão TCP começa, o valor de  $cwnd$  costuma ser inicializado em 1 MSS (RFC 3390), resultando em uma taxa inicial de envio de aproximadamente  $MSS/RTT$ . Como exemplo, se  $MSS = 500\ bytes$  e  $RTT = 200\ ms$ , então a taxa de envio inicial resultante é cerca de  $20\ kbytes/s$  apenas. Como a largura de banda disponível para a conexão pode ser muito maior do que  $MSS/RTT$ , o remetente TCP gostaria de aumentar a quantidade de largura de banda rapidamente. Dessa forma, no estado de **partida lenta**, o valor de  $cwnd$  começa em 1 MSS e aumenta 1 MSS toda vez que um segmento transmitido é reconhecido. No exemplo da Figura 3.50, o TCP envia o primeiro segmento para a rede e aguarda um reconhecimento. Quando este chega, o remetente TCP aumenta a janela de congestionamento em 1 MSS e envia dois segmentos de tamanho máximo. Esses segmentos são reconhecidos, e o remetente aumenta a janela de congestionamento em 1 MSS para cada reconhecimento de segmento, fornecendo uma janela de congestionamento de 4 MSS e assim por diante. Esse processo resulta em uma multiplicação da taxa de envio a cada RTT. Assim, a taxa de envio TCP se inicia lenta, mas cresce exponencialmente durante a fase de partida lenta.

Mas em que momento esse crescimento exponencial termina? A partida lenta apresenta diversas respostas para essa pergunta. Primeiro, se houver um evento de perda (i.e., um congestionamento) indicado por um esgotamento de temporização, o remetente TCP estabelece o valor de  $cwnd$  em 1 e inicia o processo de partida lenta novamente. Ele também estabelece o valor de uma segunda variável de estado,  $ssthresh$  (abreviação de *slow start threshold* [limiar de partida lenta]), em  $cwnd/2$  – metade do valor da janela de congestionamento quando este foi detectado. O segundo modo pelo qual a partida lenta pode terminar é ligado diretamente ao valor de  $ssthresh$ . Visto que  $ssthresh$  é metade do valor de  $cwnd$  quando o congestionamento foi detectado pela última vez, pode ser uma atitude precipitada



**Figura 3.50** Partida lenta TCP.

continuar duplicando cwnd ao atingir ou ultrapassar o valor de ssthresh. Assim, quando o valor de cwnd se igualar ao de ssthresh, a partida lenta termina e o TCP é alterado para o modo de prevenção de congestionamento. Como veremos, o TCP aumenta cwnd com mais cautela quando está no modo de prevenção de congestionamento. O último modo pelo qual a partida lenta pode terminar é se três ACKs duplicados forem detectados, caso no qual o TCP apresenta uma retransmissão rápida (veja Seção 3.5.4) e entra no estado de recuperação rápida, como discutido a seguir. O comportamento do TCP na partida lenta está resumido na descrição FSM do controle de congestionamento no TCP na Figura 3.51. O algoritmo de partida lenta foi de início proposto por Jacobson (1998); uma abordagem semelhante à partida lenta também foi proposta de maneira independente em Jain (1986).

### Prevenção de congestionamento

Ao entrar no estado de prevenção de congestionamento, o valor de cwnd é cerca de metade de seu valor quando o congestionamento foi encontrado pela última vez – o congestionamento poderia estar por perto! Desta forma, em vez de duplicar o valor de cwnd a cada RTT, o TCP adota um método mais conservador e aumenta o valor de cwnd por meio de um único MSS a cada RTT (RFC 5681). Isso pode ser realizado de diversas formas. Uma abordagem comum é o remetente aumentar cwnd por MSS bytes ( $MSS/cwnd$ ) no momento em que um novo reconhecimento chegar. Por exemplo, se o MSS possui 1.460 bytes e cwnd, 14.600 bytes, então 10 segmentos estão sendo enviados dentro de um RTT. Cada ACK que chega (considerando um ACK por segmento) aumenta o tamanho da janela de congestionamento em 1/10 MSS e, assim, o valor da janela de congestionamento terá aumentado em 1 MSS após os ACKs quando todos os segmentos tiverem sido recebidos.

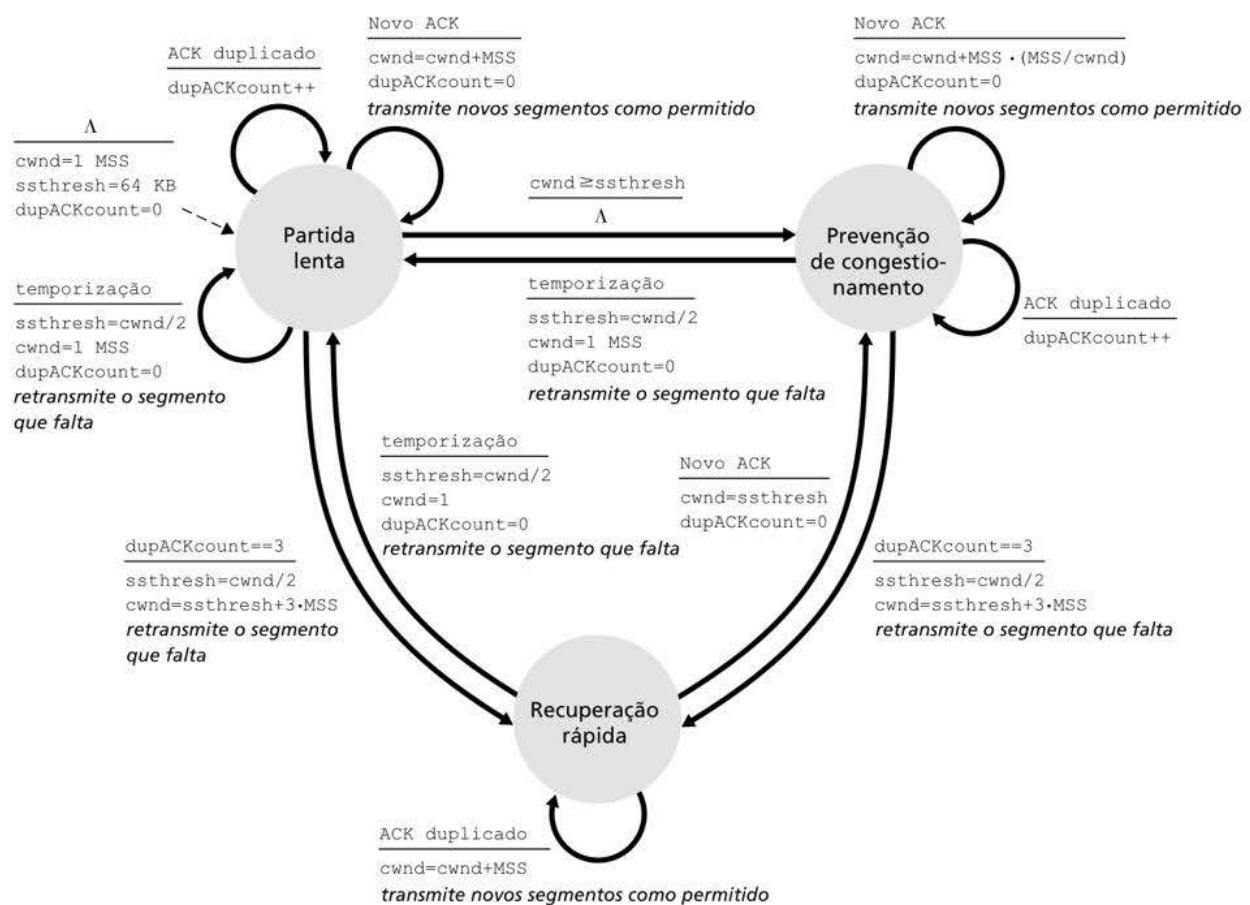


Figura 3.51 Descrição FSM do controle de congestionamento no TCP.

Mas em que momento o aumento linear da prevenção de congestionamento (de 1 MSS por RTT) deve terminar? O algoritmo de prevenção de congestionamento TCP se comporta da mesma forma quando ocorre um esgotamento de temporização. Como no caso da partida lenta: o valor de *cwnd* é ajustado para 1 MSS, e o valor de *ssthresh* é atualizado para metade do valor de *cwnd* quando ocorreu o evento de perda. Lembre-se, entretanto, de que um evento de perda também pode ser acionado por um evento ACK duplicado triplo. Neste caso, a rede continua a enviar segmentos do remetente ao destinatário (como indicado pelo recebimento de ACKs duplicados). Portanto, o comportamento do TCP para esse tipo de evento de perda deve ser menos drástico do que com uma perda de esgotamento de temporização: o TCP reduz o valor de *cwnd* para metade (adicionando em 3 MSS a mais para contabilizar os ACKs duplicados triplos recebidos) e registra o valor de *ssthresh* como metade do de *cwnd* quando os ACKs duplicados triplos foram recebidos. Então, entra-se no estado de recuperação rápida.

## Recuperação rápida

Na recuperação rápida, o valor de *cwnd* é aumentado em 1 MSS para cada ACK duplicado recebido no segmento perdido que fez o TCP entrar no modo de recuperação rápida. Mais

## PRINCÍPIOS NA PRÁTICA

### DIVISÃO DO TCP: OTIMIZANDO O DESEMPENHO DE SERVIÇOS DA NUVEM

Para serviços da nuvem como busca, *e-mail* e redes sociais, deseja-se prover uma alta capacidade de resposta, de preferência dando aos usuários a ilusão de que os serviços estão rodando dentro de seus próprios sistemas finais (inclusive seus *smartphones*). Isso pode ser um grande desafio, pois os usuários em geral estão localizados distantes dos *datacenters* que são responsáveis por servir o conteúdo dinâmico associado aos serviços da nuvem. Na verdade, se o sistema final estiver longe de um *datacenter*, então o RTT será grande, potencialmente levando a um tempo de resposta maior, devido à partida lenta do TCP.

Como um estudo de caso, considere o atraso no recebimento de uma resposta para uma consulta. Em geral, o servidor requer três janelas TCP durante a partida lenta para entregar a resposta (Pathak, 2010). Assim, o tempo desde que um sistema final inicia uma conexão TCP até o momento em que ele recebe o último pacote da resposta é cerca de  $4 \cdot \text{RTT}$  (um RTT para estabelecer a conexão TCP mais três RTTs para as três janelas de dados) mais o tempo de processamento no *datacenter*. Esses atrasos de RTT podem levar a um atraso observável no retorno de resultados de busca para uma fração significativa de consultas. Além do mais, pode haver uma significativa perda de pacotes nas redes de acesso, ocasionando retransmissões do TCP e até mesmo atrasos maiores.

Um modo de aliviar esse problema e melhorar o desempenho percebido pelo usuário é (1) instalar servidores de *front-end* mais perto dos usuários e (2) utilizar a **divisão do TCP**, quebrando a conexão TCP no servidor de *front-end*. Com a divisão do TCP, o cliente estabelece uma conexão TCP com o *front-end* nas proximidades, e o *front-end* mantém uma conexão TCP persistente com o *datacenter*, com uma janela de congestionamento TCP muito grande (Tariq, 2008; Pathak, 2010; Chen, 2011). Com essa técnica, o tempo de resposta torna-se cerca de  $4 \cdot \text{RTT}_{\text{FE}} + \text{RTT}_{\text{BE}} + \text{tempo de processamento}$ , em que  $\text{RTT}_{\text{FE}}$  é o tempo de viagem de ida e volta entre cliente e servidor de *front-end*, e  $\text{RTT}_{\text{BE}}$  é o tempo de viagem de ida e volta entre o servidor de *front-end* e o *datacenter* (servidor de *back-end*). Se o servidor de *front-end* estiver perto do cliente, o tempo de resposta torna-se mais ou menos RTT mais tempo de processamento, pois  $\text{RTT}_{\text{FE}}$  é insignificativamente pequeno, e  $\text{RTT}_{\text{BE}}$  é mais ou menos RTT. Resumindo, a divisão do TCP pode reduzir o atraso da rede de cerca de  $4 \cdot \text{RTT}$  para RTT, melhorando significativamente o desempenho percebido pelo usuário, em particular para usuários que estão longe do *datacenter* mais próximo. A divisão do TCP também ajuda a reduzir os atrasos de retransmissão do TCP causados por perdas nas redes de acesso. A Google e a Akamai utilizam bastante seus servidores CDN em redes de acesso (lembre-se da nossa discussão na Seção 2.6) para realizar a divisão do TCP para os serviços de nuvem que elas suportam (Chen, 2011).

cedo ou mais tarde, quando um ACK chega ao segmento perdido, o TCP entra no modo de prevenção de congestionamento após reduzir  $cwnd$ . Se houver um evento de esgotamento de temporização, a recuperação rápida é alterada para o modo de partida lenta após desempenhar as mesmas ações que a partida lenta e a prevenção de congestionamento: o valor de  $cwnd$  é ajustado para 1 MSS, e o valor de  $ssthresh$ , para metade do valor de  $cwnd$  no momento em que o evento de perda ocorreu.

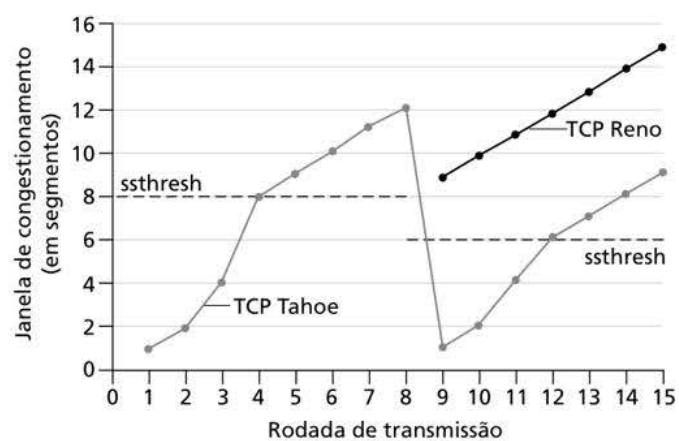
A recuperação rápida é recomendada, mas não exigida, para o protocolo TCP (RFC 5681). É interessante o fato de que uma antiga versão do TCP, conhecida como **TCP Tahoe**, reduzia incondicionalmente sua janela de congestionamento para 1 MSS e entrava na fase de partida lenta após um evento de perda de esgotamento do temporizador ou de ACK duplicado triplo. A versão atual do TCP, a **TCP Reno**, incluiu a recuperação rápida.

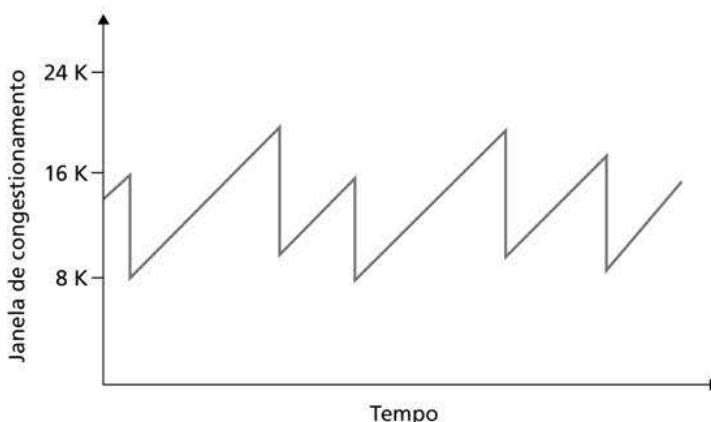
A Figura 3.52 ilustra a evolução da janela de congestionamento do TCP para as versões Reno e Tahoe. Nessa figura, o limiar é, no início, igual a 8 MSS. Nas primeiras oito sessões de transmissão, as duas versões possuem ações idênticas. A janela de congestionamento se eleva exponencialmente rápido durante a partida lenta e atinge o limiar na quarta sessão de transmissão. A janela de congestionamento, então, se eleva de modo linear até que ocorra um evento ACK duplicado triplo, logo após a oitava sessão de transmissão. Observe que a janela de congestionamento é  $12 \cdot MSS$  quando ocorre o evento de perda. O valor de  $ssthresh$  é, então, ajustado para  $0,5 \cdot cwnd = 6 \cdot MSS$ . No TCP Reno, a janela é ajustada para  $cwnd = 9 \cdot MSS$ , e depois cresce linearmente. No TCP Tahoe, é ajustada para 1 MSS e cresce de modo exponencial até que alcance o valor de  $ssthresh$ , quando começa a crescer linearmente.

A Figura 3.51 apresentou a descrição FSM completa dos algoritmos de controle de congestionamento – partida lenta, prevenção de congestionamento e recuperação rápida. A figura também indica onde pode ocorrer transmissão de novos segmentos ou segmentos retransmitidos. Embora seja importante diferenciar controle/retransmissão de erro no TCP de controle de congestionamento no TCP, também é importante avaliar como esses dois aspectos do TCP estão inseparavelmente ligados.

### Controle de congestionamento do TCP: Retrospectiva

Após nos aprofundarmos em detalhes sobre partida lenta, prevenção de congestionamento e recuperação rápida, vale a pena agora voltar e ver a floresta através das árvores. Desconsiderando o período inicial de partida lenta, quando uma conexão se inicia, e supondo que as perdas são indicadas por ACKs duplicados triplos e não por esgotamentos de temporização, o controle de congestionamento no TCP consiste em um aumento linear (aditivo) em  $cwnd$  de 1 MSS por RTT e, então, uma redução à metade (diminuição multiplicativa) de  $cwnd$





**Figura 3.53** Controle de congestionamento por aumento auditivo, diminuição multiplicativa.

em um evento ACK duplicado triplo. Por esta razão, o controle de congestionamento no TCP é quase sempre denominado **aumento aditivo, diminuição multiplicativa (AIMD)**, do inglês *Additive-Increase, Multiplicative-Decrease*). O controle de congestionamento AIMD faz surgir o comportamento semelhante a “dentes de serra”, mostrado na Figura 3.53, a qual também ilustra de forma interessante nossa intuição anterior sobre a “sondagem” do TCP por largura de banda – o TCP aumenta linearmente o tamanho de sua janela de congestionamento (e, portanto, sua taxa de transmissão) até que ocorra um evento ACK duplicado triplo. Então, ele reduz o tamanho de sua janela por um fator de dois, mas começa de novo a aumentá-la linearmente, buscando saber se há uma largura de banda adicional disponível.

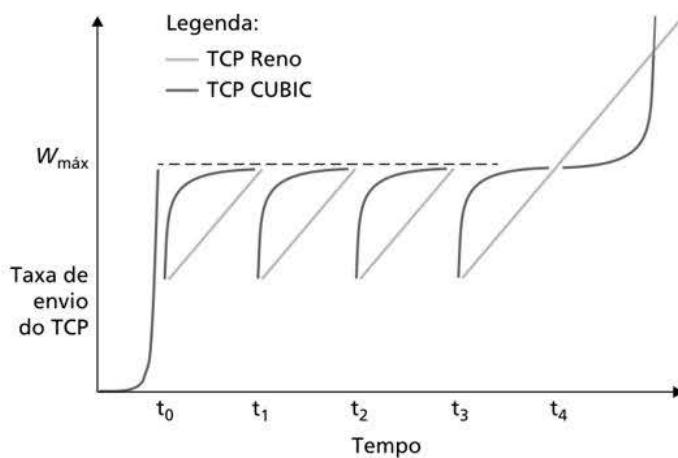
O algoritmo AIMD do TCP foi desenvolvido com base em um grande trabalho de engenharia e experiências com controle de congestionamento em redes operacionais. Dez anos após o desenvolvimento do TCP, a análise teórica mostrou que o algoritmo de controle de congestionamento do TCP serve como um algoritmo de otimização assíncrona distribuída, que resulta em vários aspectos importantes do desempenho do usuário e da rede sendo otimizados simultaneamente (Kelly, 1998). Uma rica teoria de controle de congestionamento foi desenvolvida desde então (Srikant, 2012).

### TCP Cubic

Dada a abordagem de aumento aditivo, diminuição multiplicativa ao controle de congestionamento, poderíamos, naturalmente, nos perguntar se essa é mesmo a melhor maneira de “sondar” uma taxa de envio de pacotes que fica logo abaixo do limiar para acionar a perda de pacotes. Na verdade, reduzir a taxa de envio à metade (ou pior, reduzi-la a um pacote por RTT em uma versão anterior do TCP, chamada TCP Tahoe) e então aumentá-la lentamente pode ser excesso de cautela. Se o estado do enlace congestionado no qual ocorreu a perda de pacotes não se alterou muito, então talvez seja melhor acelerar rapidamente a taxa de envio para se aproximar da taxa pré-perda e só então, cautelosamente, buscar a largura de banda. Essa ideia está no cerne do tipo de TCP conhecido pelo nome de TCP CUBIC (Ha, 2008; RFC 8312).

O TCP CUBIC difere pouco do TCP Reno. Mais uma vez, a janela de congestionamento é ampliada apenas após a recepção do ACK, e as fases de partida lenta e recuperação rápida permanecem iguais. O CUBIC muda apenas a fase de prevenção de congestionamento.

- Seja  $W_{\max}$  o tamanho da janela de controle de congestionamento do TCP na última vez em que a perda foi detectada e  $K$  o ponto futuro em que o tamanho da janela do TCP CUBIC será novamente  $W_{\max}$ , pressupondo não ocorrer perdas. Diversos parâmetros ajustáveis do CUBIC determinam o valor  $K$ , ou seja, a velocidade com a qual o tamanho da janela de congestionamento do protocolo atingiria  $W_{\max}$ .



**Figura 3.54** Taxas de envio de prevenção de congestionamento do TCP: TCP Reno e TCP CUBIC.

- O CUBIC aumenta a janela de congestionamento como função do *cubo* da distância entre o horário corrente,  $t$ , e  $K$ . Assim, quando  $t$  está mais distante de  $K$ , os aumentos do tamanho da janela de congestionamento são muito maiores do que quando  $t$  está próximo de  $K$ . Em outras palavras, CUBIC aumenta rapidamente a taxa de envio do TCP para se aproximar da taxa pré-perda,  $W_{\text{máx}}$ , e só então busca cuidadosamente a largura de banda à medida que se aproxima de  $W_{\text{máx}}$ .
- Quando  $t$  é maior do que  $K$ , a regra cúbica sugere que os aumentos da janela de congestionamento do CUBIC são pequenos quando  $t$  ainda está próximo de  $K$  (o que é bom se o nível de congestionamento do enlace que causa a perda não se alterou muito), mas então aumenta rapidamente quando  $t$  supera  $K$  (o que permite que o CUBIC encontre mais rapidamente um novo ponto de operação se o nível de congestionamento do enlace que causou a perda se alterou significativamente).

Sob essas regras, o desempenho idealizado do TCP Reno e do TCP CUBIC são comparados na Figura 3.54, adaptada de Huston (2017). Vemos que a fase de partida lenta termina em  $t_0$ . Depois, quando a perda por congestionamento ocorre em  $t_1$ ,  $t_2$  e  $t_3$ , o CUBIC acelera mais rapidamente até se aproximar de  $W_{\text{máx}}$  (e, portanto, produz uma vazão geral maior do que o TCP Reno). Na Figura, enxergamos graficamente como o TCP CUBIC tenta manter o fluxo tanto quanto possível logo abaixo do limiar de congestionamento (que o remetente desconhece). Observe que, em  $t_3$ , o nível de congestionamento supostamente diminuiu de forma significativa, permitindo que ambos o TCP Reno e o TCP CUBIC obtenham taxas de envio maiores do que  $W_{\text{máx}}$ .

A adoção do TCP CUBIC se ampliou bastante nos últimos anos. Medições realizadas em torno do ano 2000 em servidores Web populares mostravam que praticamente todos rodavam alguma versão do TCP Reno (Padhye, 2001), mas medições mais recentes dos 5.000 servidores Web mais populares mostram que quase 50% rodam uma versão do TCP CUBIC (Yang, 2014), que é também a versão padrão do TCP usada no sistema operacional Linux.

### Descrição macroscópica da vazão do TCP Reno

Dado o comportamento de dentes de serra do TCP Reno, é natural considerar qual seria a vazão média (i.e., a taxa média) de uma conexão TCP Reno de longa duração. Nessa análise, vamos ignorar as fases de partida lenta que ocorrem após eventos de esgotamento de temporização. (Elas em geral são muito curtas, visto que o remetente sai com rapidez exponencial.) Durante determinado intervalo de viagem de ida e volta, a taxa pela qual o TCP

envia dados é uma função da janela de congestionamento e do *RTT* corrente. Quando o tamanho da janela for  $w$  bytes e o tempo de viagem de ida e volta for *RTT* segundos, a taxa de transmissão do TCP será mais ou menos  $w/RTT$ . Então, o TCP faz uma sondagem em busca de alguma largura de banda adicional aumentando  $w$  em 1 MSS a cada *RTT* até ocorrer um evento de perda. Seja  $W$  o valor de  $w$  quando ocorre um evento de perda. Admitindo que *RTT* e  $W$  são mais ou menos constantes no período da conexão, a taxa de transmissão fica na faixa de  $W/(2 \cdot RTT)$  a  $W/RTT$ .

Essas suposições levam a um modelo macroscópico muito simplificado para o comportamento do TCP em estado constante. A rede descarta um pacote da conexão quando a taxa aumenta para  $W/RTT$ ; então a taxa é reduzida à metade e, em seguida, aumenta em MSS/*RTT* a cada *RTT* até alcançar  $W/RTT$  novamente. Esse processo se repete de modo contínuo. Como a vazão do TCP (i.e., sua taxa) aumenta linearmente entre os dois valores extremos, temos:

$$\text{vazão média de uma conexão} = \frac{0,75 \cdot W}{RTT}$$

Usando esse modelo muito idealizado para a dinâmica de regime permanente do TCP, podemos também derivar uma interessante expressão que relaciona a taxa de perda de uma conexão com sua largura de banda disponível (Mathis, 1997). Essa derivação está delineada nos exercícios de fixação. Um modelo mais sofisticado que demonstrou empiricamente estar de acordo com dados medidos é apresentado em Padhye (2000).

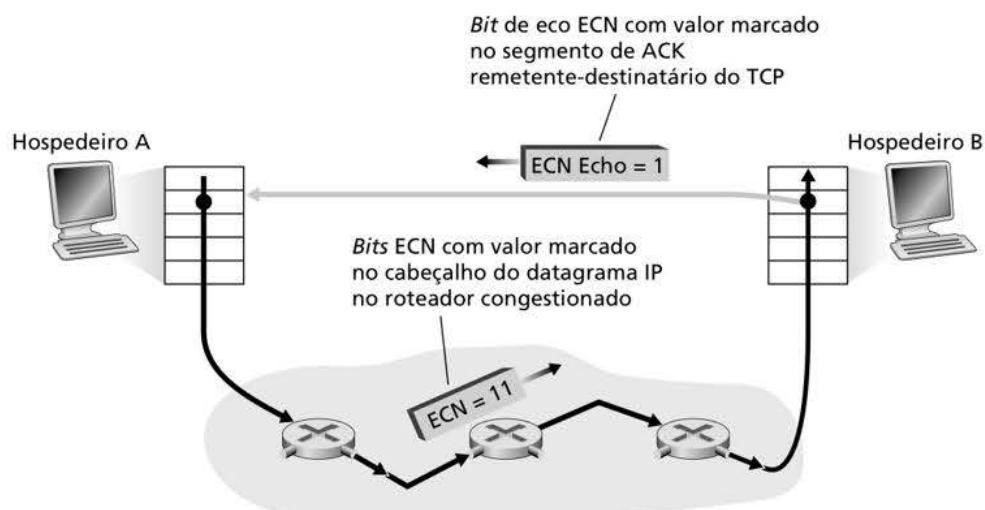
### 3.7.2 Notificação explícita de congestionamento assistido pela rede e controle de congestionamento baseado em atrasos

Desde a padronização inicial da partida lenta e da prevenção do congestionamento no final da década de 1980 (RFC 1122), o TCP implementou a forma de controle de congestionamento fim a fim que estudamos na Seção 3.7.1: um remetente TCP não recebe indicações explícitas de congestionamento da camada de rede; em vez disso, ele infere o congestionamento a partir da perda de pacotes observada. Mais recentemente, foram propostas, implementadas e instaladas extensões do IP e do TCP (RFC 3168) que permitem que a rede sinalize explicitamente o congestionamento para remetentes e destinatários TCP. Além disso, foram propostas diversas variações dos protocolos de controle de congestionamento do TCP que inferem o congestionamento usando medidas de atrasos de pacotes. Nesta seção, analisaremos o controle de congestionamento baseado em atrasos e o assistido pela rede.

#### Notificação explícita de congestionamento

A notificação explícita de congestionamento (ECN, do inglês *Explicit Congestion Notification*) (RFC 3168) é a forma de controle de congestionamento assistida pela rede realizada na Internet. Como mostra a Figura 3.55, ambos o TCP e o IP estão envolvidos. Na camada de rede, dois bits (com quatro valores possíveis no total) no campo Tipo de Serviço do cabeçalho do datagrama IP (que discutiremos na Seção 4.3) são usados para ECN.

Uma configuração dos bits de ECN é usada por um roteador para indicar que este está congestionado. Essa indicação é então levada no datagrama IP marcado até o hospedeiro de destino, que, por sua vez, informa o hospedeiro remetente, como mostra a Figura 3.55. O RFC 3168 não inclui uma definição de quando um roteador está congestionado; a decisão é uma opção de configuração, possibilitada pelo fornecedor do roteador e decidida pelo operador da rede. Contudo, a intuição é que o bit de indicação de congestionamento pode ser configurado para sinalizar o início do congestionamento para o remetente antes da perda ocorrer de fato. Uma segunda configuração dos bits de ECN é usada pelo hospedeiro



**Figura 3.55** Notificação explícita de congestionamento: controle de congestionamento assistido pela rede.

remetente para informar os roteadores de que o destinatário e o remetente têm capacidade para ECN e, logo, conseguir responder ao congestionamento da rede indicado pela ECN.

Como mostra a Figura 3.55, quando recebe a indicação de congestionamento ECN através de um datagrama recebido, o TCP no hospedeiro destinatário informa o TCP no hospedeiro remetente sobre essa indicação marcando o *bit de eco* de notificação explícita de congestionamento (ECE, do inglês *Explicit Congestion Notification Echo*) (ver Figura 3.29) no segmento de ACK do destinatário para o remetente do TCP. O remetente TCP, por sua vez, reage a um ACK com indicação de congestionamento cortando a janela de congestionamento pela metade, assim como reagiria a um segmento perdido usando retransmissão rápida, e define o *bit de janela de congestionamento reduzida* (CWR, do inglês *Congestion Window Reduced*) no cabeçalho do próximo segmento do remetente para o destinatário do TCP enviado.

Além do TCP, outros protocolos da camada de transporte também usam o ECN sinalizado na camada de rede. O Protocolo de Controle de Congestionamento de Datagrama (DCCP, do inglês *Datagram Congestion Control Protocol*) (RFC 4340) oferece um serviço não confiável de baixo consumo, com controle de congestionamento e semelhante ao UDP que utiliza a ECN. O DCTCP (Data Center TCP) (Alizadeh, 2010; RFC 8257) e o DCQCN (Data Center Quantized Congestion Notification) (Zhu, 2015), projetados especificamente para as redes de *datacenters*, também usam a ECN. Medidas recentes da Internet mostram a implementação crescente de recursos ECN em servidores populares e em roteadores nos trajetos até tais servidores (Kühlewind, 2013).

### Controle de congestionamento baseado em atrasos

Na nossa discussão acima sobre ECN, vimos que um roteador congestionado pode determinar que o *bit de indicação de congestionamento* sinalize o início do congestionamento para os remetentes *antes* que os *buffers* cheios façam os pacotes serem descartados no roteador. Isso permite que os remetentes diminuam suas taxas de envio mais cedo, *antes* da perda de pacotes, com sorte, o que evita o processo caro da perda e retransmissão de pacotes. Uma segunda abordagem de prevenção do congestionamento, baseada em atrasos, também detecta proativamente o início do evento *antes* da perda de pacotes ocorrer.

No TCP Vegas (Brakmo, 1995), o remetente mede o RTT do trajeto entre origem e destino para todos os pacotes reconhecidos. Seja  $RTT_{min}$  o valor mínimo dessas medições em um remetente; este ocorre quando o trajeto não está congestionado e os pacotes sofrem atrasos de fila mínimos. Se o tamanho da janela de congestionamento do TCP Vegas é  $cwnd$ , então a taxa de vazão não congestionada seria  $cwnd/RTT_{min}$ . A intuição por trás do TCP Vegas é

que se a vazão real medida pelo remetente fica próxima desse valor, a taxa de envio do TCP pode ser aumentada, pois (por definição e por medição) o trajeto ainda não está congestionado. Contudo, se a vazão real medida pelo remetente for significativamente menor do que a taxa de vazão não congestionada, o trajeto está congestionado e o remetente TCP Vegas reduz a sua taxa de envio. Os detalhes se encontram em Brakmo (1995).

O TCP Vegas opera sob a intuição de que os remetentes TCP devem “*manter o cano cheio, mas não mais do que cheio*” (Kleinrock, 2018). “Manter o cano cheio” significa que os enlaces (em especial, o gargalo que está limitando a vazão de uma conexão) ficam ocupados com a transmissão, realizando trabalho útil; “mas não mais do que cheio” significa que não temos nada a ganhar (exceto um maior atraso!) se permitirmos que filas grandes se acumulem enquanto o tubo permanece cheio.

O protocolo de controle de congestionamento BBR (Cardwell, 2017) se baseia em ideias do TCP Vegas e incorpora mecanismos que permitem que compita em condições iguais (ver Seção 3.7.3) com remetentes TCP não BBR. Segundo Cardwell (2017), em 2016, a Google começou a usar o BBR para todo o tráfego TCP na sua rede privativa B4 (Jain, 2013), que interconecta os *datacenters* da Google, substituindo o CUBIC. Ele também está sendo utilizado nos servidores Web da Google e do YouTube. Outros protocolos de controle de congestionamento no TCP baseados em atrasos incluem o TIMELY para redes de *datacenters* (Mittal, 2015), o Compound TCP (CTPC) (Tan, 2006) e o FAST (Wei, 2006), para redes de alta velocidade e de longa distância.

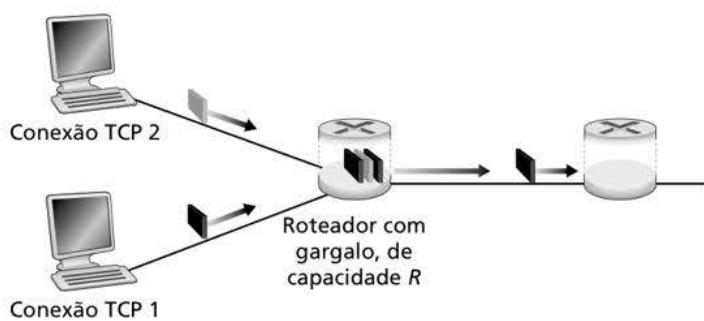
### 3.7.3 Equidade

Considere  $K$  conexões TCP, cada uma com um caminho fim a fim diferente, mas todas passando pelo gargalo em um enlace com taxa de transmissão de  $R$  bits/s (aqui, *gargalo em um enlace* quer dizer que nenhum dos outros enlaces ao longo do caminho de cada conexão está congestionado e que todos dispõem de abundante capacidade de transmissão em comparação à capacidade de transmissão do enlace com gargalo). Suponha que cada conexão está transferindo um arquivo grande e que não há tráfego UDP passando pelo enlace com gargalo. Dizemos que um mecanismo de controle de congestionamento é *justo* se a taxa média de transmissão de cada conexão for mais ou menos  $R/K$ ; isto é, cada uma obtém uma parcela igual da largura de banda do enlace.

O algoritmo AIMD do TCP é justo, considerando, em especial, que diferentes conexões TCP podem começar em momentos diferentes e, assim, ter tamanhos de janela diferentes em um dado instante? Chiu (1989) explica, de um modo elegante e intuitivo, por que o controle de congestionamento converge para fornecer um compartilhamento justo da largura de banda do enlace entre conexões TCP concorrentes.

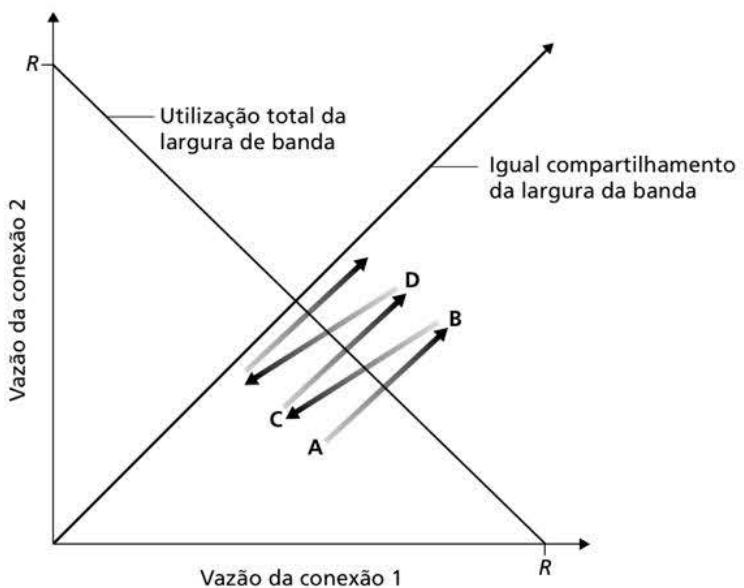
Vamos considerar o caso simples de duas conexões TCP compartilhando um único enlace com taxa de transmissão  $R$ , conforme mostra a Figura 3.56. Admitamos que as duas conexões tenham os mesmos MSS e RTT (de modo que, se o tamanho de suas janelas de congestionamento for o mesmo, eles terão a mesma vazão) e uma grande quantidade de dados para enviar e que nenhuma outra conexão TCP ou datagramas UDP atravesse esse enlace compartilhado. Vamos ignorar também a fase de partida lenta do TCP e admitir que as conexões TCP estão operando em modo prevenção de congestionamento (AIMD) todo o tempo.

A Figura 3.57 mostra a vazão alcançada pelas duas conexões TCP. Se for para o TCP compartilhar equitativamente a largura de banda do enlace entre as duas conexões, a vazão alcançada deverá cair ao longo da linha a 45 graus (igual compartilhamento da largura de banda) que parte da origem. Idealmente, a soma das duas vazões seria igual a  $R$ . (Com certeza, não é uma situação desejável cada conexão receber um compartilhamento igual, mas igual a zero, da capacidade do enlace!) Portanto, o objetivo é que as vazões alcançadas fiquem em algum lugar perto da intersecção da linha de “igual compartilhamento da largura de banda” com a linha de “utilização total da largura de banda” da Figura 3.57.



**Figura 3.56** Duas conexões TCP compartilhando um único enlace congestionado.

Suponha que os tamanhos de janela TCP sejam tais que, em um determinado instante, as conexões 1 e 2 alcancem as vazões indicadas pelo ponto  $A$  na Figura 3.57. Como a quantidade de largura de banda consumida em conjunto pelas duas conexões é menor do que  $R$ , não ocorrerá nenhuma perda, e ambas as conexões aumentarão suas janelas em 1 MSS por RTT como resultado do algoritmo de prevenção de congestionamento do TCP. Assim, a vazão conjunta das duas conexões continua ao longo da linha a 45 graus (aumento igual para as duas), começando no ponto  $A$ . Por fim, a largura de banda do enlace consumida em conjunto pelas duas conexões será maior do que  $R$  e, assim, por fim ocorrerá perda de pacote. Suponha que as conexões 1 e 2 experimentem perda de pacote quando alcançarem as vazões indicadas pelo ponto  $B$ . As conexões 1 e 2 então reduzirão suas janelas por um fator de 2. Assim, as vazões resultantes são as do ponto  $C$ , a meio caminho do vetor que começa em  $B$  e termina na origem. Como a utilização conjunta da largura de banda é menor do que  $R$  no ponto  $C$ , as duas conexões novamente aumentam suas vazões ao longo da linha a 45 graus que começa no ponto  $C$ . Mais cedo ou mais tarde ocorrerá perda, por exemplo, no ponto  $D$ , e as duas conexões reduzirão de novo o tamanho de suas janelas por um fator de 2 – e assim por diante. Você pode ter certeza de que a largura de banda alcançada pelas duas conexões flutuará ao longo da linha de igual compartilhamento da largura de banda. E, também, você pode estar certo de que as duas conexões convergirão para esse comportamento, não importando onde elas começem no espaço bidimensional! Embora haja uma série de suposições idealizadas por trás desse cenário, ainda assim ele dá uma ideia intuitiva de por que o TCP resulta em igual compartilhamento da largura de banda entre conexões.



**Figura 3.57** Vazão alcançada pelas conexões TCP 1 e TCP 2.

Em nosso cenário idealizado, admitimos que apenas conexões TCP atravessem o enlace com gargalo, que elas tenham o mesmo valor de RTT e que uma única conexão TCP esteja associada com um par hospedeiro/destinatário. Na prática, essas condições não são muito encontradas, e, assim, é possível que as aplicações cliente-servidor obtenham porções muito desiguais da largura de banda do enlace. Em especial, foi demonstrado que, quando várias conexões compartilham um único enlace com gargalo, as sessões cujos RTTs são menores conseguem obter a largura de banda disponível naquele enlace mais rapidamente (i.e., aumentam suas janelas de congestionamento mais depressa) à medida que o enlace fica livre. Assim, conseguem vazões mais altas do que conexões com RTTs maiores (Lakshman, 1997).

### Equidade e UDP

Acabamos de ver como o controle de congestionamento no TCP regula a taxa de transmissão de uma aplicação por meio do mecanismo de janela de congestionamento. Diversas aplicações de multimídia, como telefone por Internet e videoconferência, muitas vezes não rodam sobre TCP exatamente por essa razão – elas não querem que sua taxa de transmissão seja limitada, mesmo que a rede esteja muito congestionada. Ao contrário, preferem rodar sobre UDP, que não tem controle de congestionamento. Quando rodam sobre esse protocolo, as aplicações podem passar seus áudios e vídeos para a rede a uma taxa constante e, de modo ocasional, perder pacotes, em vez de reduzir suas taxas a níveis “justos” em horários de congestionamento e não perder nenhum deles. Do ponto de vista do TCP, as aplicações de multimídia que rodam sobre UDP não são justas – elas não cooperam com as outras conexões nem ajustam suas taxas de transmissão de maneira adequada. Como o controle de congestionamento no TCP reduzirá sua taxa de transmissão quando houver aumento de congestionamento (perda), enquanto origens UDP não precisam fazer o mesmo, é possível que essas origens desalojem o tráfego TCP. Foram propostos diversos mecanismos de controle de congestionamento para a Internet que impedem o tráfego de UDP de levar a vazão da Internet a uma parada repentina (Floyd, 1999; Floyd, 2000; Kohler, 2006; RFC 4340).

### Equidade e conexões TCP paralelas

Mesmo que pudéssemos obrigar o tráfego UDP a se comportar com equidade, o problema ainda não estaria resolvido por completo. Isso porque não há nada que impeça uma aplicação de rodar sobre TCP usando múltiplas conexões paralelas. Por exemplo, navegadores Web com frequência usam múltiplas conexões TCP paralelas para transferir os vários objetos de uma página. (O número exato de conexões múltiplas pode ser configurado na maioria dos navegadores.) Quando usa múltiplas conexões paralelas, uma aplicação consegue uma fração maior da largura de banda de um enlace congestionado. Como exemplo, considere um enlace de taxa  $R$  que está suportando nove aplicações cliente-servidor em curso, e cada uma das aplicações está usando uma conexão TCP. Se surgir uma nova aplicação que também utilize uma conexão TCP, então cada aplicação conseguirá aproximadamente a mesma taxa de transmissão igual a  $R/10$ . Porém, se, em vez disso, essa nova aplicação usar 11 conexões TCP paralelas, então ela conseguirá uma alocação injusta de mais do que  $R/2$ . Como a penetração do tráfego Web na Internet é grande, as múltiplas conexões paralelas não são incomuns.

## 3.8 EVOLUÇÃO DA FUNCIONALIDADE DA CAMADA DE TRANSPORTE

Nossa discussão sobre os protocolos específicos de transporte da Internet neste capítulo concentrou-se no UDP e no TCP – os dois “burros de carga” da camada de transporte. Entretanto, três décadas de experiência com os dois identificaram circunstâncias nas quais

nenhum deles é apropriado de maneira ideal, então o projeto e a implementação das funcionalidades da camada de transporte continuam a evoluir.

Vimos como o uso do TCP evoluiu de forma rica durante a última década. Nas Seções 3.7.1 e 3.7.2, aprendemos que além das versões “clássicas” do TCP, como Tahoe e Reno, hoje temos várias novas versões do TCP, que foram desenvolvidas, implementadas e instaladas e que estão em uso significativo na atualidade. Estas incluem TCP CUBIC, DCTCP, CTCP, BBR e mais. Na verdade, segundo as medições de Yang (2014), o CUBIC (e seu predecessor, o BIC [Xu, 2004]) e o CTCP são mais amplamente utilizados em servidores Web do que o TCP Reno clássico; também vimos que o BBR está sendo instalado na rede interna B4 da Google, além de muitos dos servidores voltados para o público da empresa.

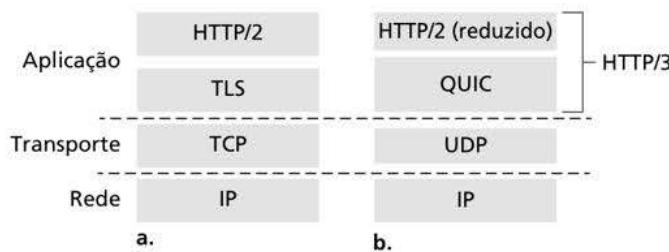
E há muitas (e muitas!) outras versões do TCP! Há versões do TCP projetadas especificamente para uso em enlaces sem fio, sobre trajetos com ampla largura de banda com RTTs grandes, para trajetos com reordenamento de pacotes e para trajetos curtos estritamente internos a *datacenters*. Existem versões do TCP que implementam diferentes prioridades entre as conexões TCP, competindo por banda em um enlace com gargalo, e para conexões TCP cujos segmentos estão sendo enviados por diferentes trajetos origem-destino em paralelo. Também há variações do TCP que lidam com o reconhecimento de pacotes e o estabelecimento/encerramento de sessões TCP de forma diferente daquela estudada na Seção 3.5.6. Na verdade, provavelmente nem deveríamos mais nos referir a “o” protocolo TCP; talvez os únicos recursos em comum entre esses protocolos é que eles usam o formato de segmento TCP que estudamos na Figura 3.29 e que devem competir “justamente” entre si perante o congestionamento da rede! Para uma análise das muitas variantes do TCP, consulte Afanasyev (2010) e Narayan (2018).

## QUIC: Conexões de Internet UDP Rápidas

Se os serviços de transporte de que a aplicação necessita não se encaixam exatamente com os modelos de serviço do UDP e do TCP – talvez uma aplicação precise de mais serviços que o UDP oferece, mas não queira todas as funcionalidades particulares que vêm com o TCP, ou serviços diferentes daqueles oferecidos pelo TCP –, os projetistas de aplicações podem sempre elaborar a sua própria “receita” de protocolo na camada de aplicação. Essa é a abordagem adotada no protocolo QUIC (do inglês *Quick UDP Internet Connections* – Conexões de Internet UDP Rápidas) (Langley, 2017; QUIC 2020). Mais especificamente, o QUIC é um novo protocolo da camada de aplicação, projetado do zero para melhorar o desempenho dos serviços da camada de transporte para o HTTP seguro. O QUIC já é amplamente utilizado, apesar de ainda estar no processo de ser padronizado na forma de RFC da Internet (QUIC 2020). A Google adotou o QUIC para muitos dos servidores voltados para o público, no seu aplicativo de *streaming* de vídeo do YouTube, no seu navegador Chrome e no aplicativo de Busca do Google para Android. Como hoje mais de 7% do tráfego na Internet usa o protocolo QUIC (Langley, 2017), é interessante analisá-lo mais de perto. Nossa estudo sobre o QUIC também serve para concluir o nosso estudo sobre a camada de transporte, pois o QUIC usa muitas das abordagens estudadas neste capítulo para transferência confiável de dados, controle de congestionamento e gerenciamento da conexão.

Como mostrado na Figura 3.58, o QUIC é um protocolo da camada de aplicação que usa UDP como protocolo da camada de transporte subjacente e foi projetado para interagir com a camada superior especificamente com uma versão simplificada, mas evoluída, do HTTP/2. No futuro próximo, o HTTP/3 incorporará nativamente o QUIC (HTTP/3 2020). Alguns dos principais recursos do QUIC incluem:

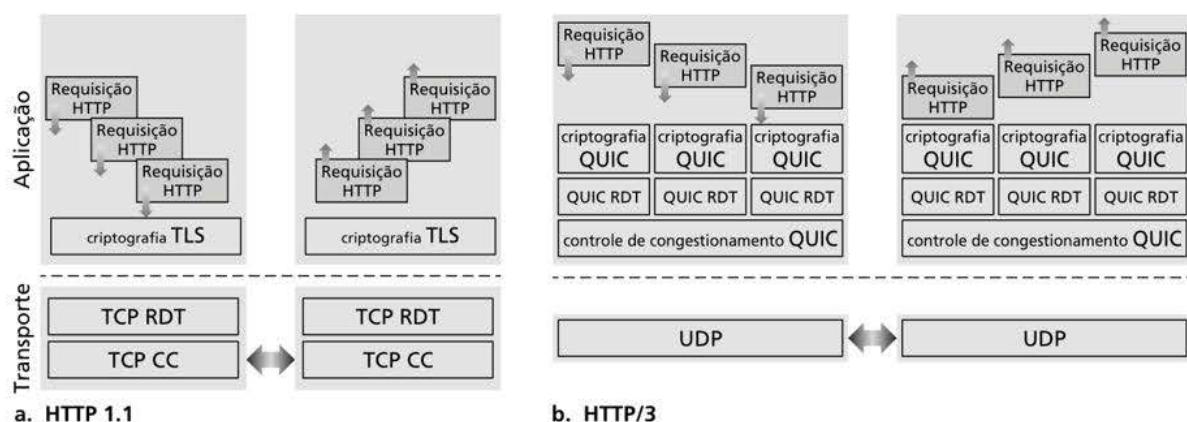
- **Orientado para conexão e seguro.** Assim como o TCP, o QUIC é um protocolo orientado para conexão entre dois pontos finais, o que exige uma apresentação entre os pontos finais para estabelecer o estado de conexão QUIC. Dois elementos do estado de conexão são a identidade da conexão de origem e de destino. Todos os pacotes QUIC são criptografados e, como sugerido na Figura 3.58, o QUIC combina as apresentações



**Figura 3.58** (a) Pilha de protocolos HTTP seguro tradicional e (b) pilha de protocolos HTTP/3 seguro baseado no QUIC.

necessárias para estabelecer o estado de conexão com aquelas necessárias para autenticação e criptografia (tópicos de segurança da camada de transporte que estudaremos no Capítulo 8), o que acelera o estabelecimento da conexão em relação à pilha de protocolos da Figura 3.58(a), na qual múltiplos RTTs são necessários para estabelecer uma conexão TCP, e então para estabelecer uma conexão TLS sobre a conexão TCP.

- **Fluxos.** O QUIC permite que vários “fluxos” diferentes no nível da aplicação sejam multiplexados através de uma única conexão QUIC e, uma vez estabelecida a conexão QUIC, novos fluxos podem ser adicionados. Um fluxo é uma abstração que representa a entrega confiável, ordenada e bidirecional de dados entre dois pontos finais QUIC. No contexto do HTTP/3, haveria um fluxo diferente para cada objeto em uma página Web. Cada conexão possui uma identidade (ID) da conexão e cada fluxo dentro da conexão possui uma ID do fluxo; ambas as IDs estão contidas em um cabeçalho de pacote QUIC (junto com outras informações do cabeçalho). Dados de múltiplos fluxos podem estar contidos em um único segmento QUIC, que é transmitido sobre o UDP. O Protocolo de Controle de Fluxo de Transmissão (SCTP, do inglês *Stream Control Transmission Protocol*) (RFC 4960, RFC 3286) é um protocolo orientado para mensagens confiável anterior, pioneiro na ideia da multiplexação de múltiplos “fluxos” do nível de aplicação através de uma única conexão SCTP. No Capítulo 7, veremos que o SCTP é usado nos protocolos do plano de controle nas redes celulares sem fio 4G/5G.
- **Transferência confiável de dados com controle de congestionamento e amigável ao TCP.** Como ilustrado na Figura 3.59(b), o QUIC oferece uma transferência de dados confiável para cada fluxo QUIC *separadamente*. A Figura 3.59(a) mostra o caso do HTTP/1.1 enviando múltiplas requisições HTTP, todas sobre uma única conexão TCP.



**Figura 3.59** (a) HTTP/1.1: cliente e servidor com uma conexão usando criptografia TLS no nível da aplicação sobre a transferência confiável de dados do TCP (RDT) e controle de congestionamento (CC). (b) HTTP/3: cliente e servidor com múltiplos fluxos (*multi-stream*) usando criptografia, transferência confiável de dados e controle de congestionamento do QUIC sobre o serviço de datagrama não confiável do UDP.

Como o TCP oferece entrega confiável e ordenada de *bytes*, isso significa que múltiplas requisições HTTP devem ser entregues em ordem no servidor HTTP de destino. Assim, se *bytes* de uma requisição HTTP são perdidos, as requisições HTTP restantes não podem ser entregues até os *bytes* perdidos serem retransmitidos e recebidos corretamente pelo TCP no servidor HTTP – o chamado problema de bloqueio HOL que encontramos anteriormente na Seção 2.2.5. Como o QUIC oferece entrega confiável e ordenada *por fluxo*, um segmento UDP perdido impacta apenas os fluxos cujos dados foram transportados naquele segmento; mensagens HTTP em outros fluxos podem continuar a ser recebidas e entregues à aplicação. O QUIC oferece transferência confiável de dados usando mecanismos de reconhecimento semelhantes aos do TCP, como especificado no (RFC 5681).

O controle de congestionamento do QUIC se baseia no TCP NewReno (RFC 6582), uma leve modificação do protocolo TCP Reno que estudamos na Seção 3.7.1. O rascunho das especificações do QUIC (QUIC-recovery 2020) observam que “leitores familiarizados com a detecção de perdas e controle de congestionamento do TCP verão aqui algoritmos semelhantes aos algoritmos TCP conhecidos”. Como estudamos o controle de congestionamento do TCP em detalhes na Seção 3.7.1, estamos familiarizados quando lemos os detalhes do rascunho das especificações do QUIC sobre o seu algoritmo de controle de congestionamento!

Por fim, vale destacar mais uma vez que o QUIC é um protocolo da *camada de aplicação* que oferece transferência de dados confiável e com controle de congestionamento entre dois pontos finais. Os autores do QUIC (Langley, 2017) enfatizam que isso significa que é possível realizar alterações ao QUIC em “escalas temporais de atualização de aplicações”, ou seja, muito mais rapidamente do que as escalas de tempo das atualizações do TCP ou do UDP.

## 3.9 RESUMO

Começamos este capítulo estudando os serviços que um protocolo de camada de transporte pode prover às aplicações de rede. Por um lado, esse protocolo pode ser muito simples e oferecer serviços básicos às aplicações, provendo apenas uma função de multiplexação/demultiplexação para processos em comunicação. O protocolo UDP da Internet é um exemplo desse serviço básico. Por outro lado, o protocolo de transporte pode fornecer uma série de garantias às aplicações, como entrega confiável de dados, garantias contra atrasos e garantias de largura de banda. Não obstante, os serviços que um protocolo de transporte pode prover são muitas vezes limitados pelo modelo de serviço do protocolo subjacente de camada de rede. Se o protocolo de camada de rede não puder proporcionar garantias contra atraso ou garantias de largura de banda para segmentos da camada de transporte, então o protocolo de camada de transporte não poderá fornecer tais garantias para as mensagens enviadas entre processos.

Aprendemos na Seção 3.4 que um protocolo de camada de transporte pode prover transferência confiável de dados mesmo que a camada de rede subjacente seja não confiável. Vimos que há muitos pontos sutis na transferência confiável de dados, mas que a tarefa pode ser realizada pela combinação cuidadosa de reconhecimentos, temporizadores, retransmissões e números de sequência.

Embora tenhamos examinado a transferência confiável de dados neste capítulo, devemos ter em mente que ela pode ser fornecida por protocolos de camada de enlace, de rede, de transporte ou de aplicação. Qualquer uma das camadas superiores da pilha de protocolos pode executar reconhecimentos, temporizadores, retransmissões e números de sequência e prover transferência confiável de dados para a camada situada acima dela. Na verdade, com o passar dos anos, engenheiros e cientistas da computação projetaram e realizaram, independentemente, protocolos de camada de enlace, de rede, de transporte e de

aplicação que fornecem transferência confiável de dados (embora muitos tenham desaparecido silenciosamente).

Na Seção 3.5, examinamos em detalhes o TCP, o protocolo de camada de transporte confiável orientado para conexão da Internet. Aprendemos que o TCP é complexo e envolve gerenciamento da conexão, controle de fluxo, estimativa de tempo de viagem de ida e volta, bem como transferência confiável de dados. Na verdade, o TCP é mais complexo do que nossa descrição – de propósito, não discutimos uma série de ajustes, acertos e melhorias que estão implementados em várias versões do TCP. Toda essa complexidade, no entanto, fica escondida da aplicação de rede. Se um cliente em um hospedeiro quiser enviar dados de maneira confiável para outro hospedeiro, ele apenas abre um *socket* TCP para o servidor e passa dados para dentro desse *socket*. A aplicação cliente-servidor felizmente fica alheia a toda a complexidade do TCP.

Na Seção 3.6, examinamos o controle de congestionamento de um ponto de vista mais amplo, e, na Seção 3.7, demonstramos como o TCP realiza controle de congestionamento. Aprendemos que esse controle é imperativo para o bem-estar da rede. Sem ele, uma rede pode facilmente ficar travada, com poucos ou nenhum dado sendo transportado fim a fim. Na Seção 3.7, aprendemos também que o TCP clássico executa um mecanismo de controle de congestionamento fim a fim que aumenta aditivamente sua taxa de transmissão quando julga que o caminho da conexão TCP está livre de congestionamento e reduz multiplicativamente sua taxa de transmissão quando ocorre perda. Esse mecanismo também luta para dar a cada conexão TCP que passa por um enlace congestionado uma parcela igual da largura de banda da conexão. Também estudamos diversas novas variações no controle de congestionamento do TCP que buscam determinar a taxa de envio do TCP mais rapidamente do que o TCP clássico, usam uma abordagem baseada em atrasos ou notificação explícita de congestionamento da rede (em vez de uma abordagem baseada em perdas) para determinar a taxa de envio do TCP. Examinamos ainda com algum detalhe o impacto do estabelecimento da conexão TCP e da partida lenta sobre a latência. Observamos que, em muitos cenários importantes, o estabelecimento da conexão e a partida lenta contribuem de modo significativo para o atraso fim a fim. Enfatizamos mais uma vez que, embora tenha evoluído com o passar dos anos, o controle de congestionamento no TCP permanece como uma área de pesquisa intensa e, provavelmente, continuará a evoluir nos anos vindouros. Para encerrar este capítulo, na Seção 3.8, estudamos avanços recentes na implementação de diversas funções da camada de transporte – transferência confiável de dados, controle de congestionamento, estabelecimento de conexão e mais – na camada de aplicação usando o protocolo QUIC.

No Capítulo 1, dissemos que uma rede de computadores pode ser dividida em “periferia da rede” e “núcleo da rede”. A periferia abrange tudo o que acontece nos sistemas finais. Com o exame da camada de aplicação e da camada de transporte, nossa discussão sobre a periferia da rede está completa. É hora de explorar o núcleo! Essa jornada começa nos próximos dois capítulos, em que estudaremos a camada de rede, e continua no Capítulo 6, em que estudaremos a camada de enlace.

## Exercícios de fixação e perguntas

### Questões de revisão do Capítulo 3

#### SEÇÕES 3.1–3.3

- R1. Suponha que uma camada de rede forneça o seguinte serviço. A camada de rede no computador-fonte aceita um segmento de tamanho máximo de 1.200 *bytes* e um endereço de computador-alvo da camada de transporte. A camada de rede, então, garante encaminhar o segmento para a camada de transporte no computador-alvo. Suponha

que muitos processos de aplicação de rede possam estar sendo executados no hospedeiro de destino.

- a. Crie, da forma mais simples, o protocolo da camada de transporte possível que levará os dados da aplicação para o processo desejado no hospedeiro de destino. Suponha que o sistema operacional do hospedeiro de destino determinou um número de porta de 4 bytes para cada processo de aplicação em execução.
- b. Modifique esse protocolo para que ele forneça um “endereço de retorno” ao processo-alvo.
- c. Em seus protocolos, a camada de transporte “tem de fazer algo” no núcleo da rede de computadores?

- R2. Considere um planeta onde todos possuam uma família com seis membros, cada família viva em sua própria casa, cada casa possua um endereço único e cada pessoa em certa casa possua um único nome. Suponha que esse planeta possua um serviço postal que entregue cartas da casa-fonte à casa-alvo. O serviço exige que (1) a carta esteja em um envelope e que (2) o endereço da casa-alvo (e nada mais) esteja escrito claramente no envelope. Imagine que cada família possua um membro representante que recebe e distribui cartas para os demais. As cartas não apresentam necessariamente qualquer indicação dos destinatários das cartas.
  - a. Utilizando a solução do Problema R1 como inspiração, descreva um protocolo que os representantes possam utilizar para entregar cartas de um membro remetente de uma família para um membro destinatário de outra família.
  - b. Em seu protocolo, o serviço postal precisa abrir o envelope e verificar a carta para fornecer o serviço?
- R3. Considere uma conexão TCP entre o hospedeiro A e o hospedeiro B. Suponha que os segmentos TCP que trafegam do hospedeiro A para o hospedeiro B tenham número de porta da origem  $x$  e número de porta do destino  $y$ . Quais são os números de porta da origem e do destino para os segmentos que trafegam do hospedeiro B para o hospedeiro A?
- R4. Descreva por que um desenvolvedor de aplicação pode escolher rodar uma aplicação sobre UDP em vez de sobre TCP.
- R5. Por que o tráfego de voz e de vídeo é frequentemente enviado por meio do UDP e não do TCP na Internet de hoje? (*Dica:* a resposta que procuramos não tem nenhuma relação com o mecanismo de controle de congestionamento no TCP.)
- R6. É possível que uma aplicação desfrute de transferência confiável de dados mesmo quando roda sobre UDP? Caso a resposta seja afirmativa, como isso acontece?
- R7. Suponha que um processo no hospedeiro C possua um *socket* UDP com número de porta 6789 e que o hospedeiro A e o hospedeiro B, individualmente, enviem um segmento UDP ao hospedeiro C com número de porta de destino 6789. Os dois segmentos serão encaminhados para o mesmo *socket* no hospedeiro C? Se sim, como o processo no hospedeiro C saberá que os dois segmentos vieram de dois hospedeiros diferentes?
- R8. Suponha que um servidor da Web seja executado no computador C na porta 80. Esse servidor utiliza conexões persistentes e, no momento, está recebendo solicitações de dois computadores diferentes, A e B. Todas as solicitações estão sendo enviadas por meio do mesmo *socket* no computador C? Se estão passando por diferentes *sockets*, dois deles possuem porta 80? Discuta e explique.

## SEÇÃO 3.4

- R9. Em nossos protocolos rdt, por que precisamos introduzir números de sequência?
- R10. Em nossos protocolos rdt, por que precisamos introduzir temporizadores?

- R11. Suponha que o atraso de viagem de ida e volta entre o emissor e o receptor seja constante e conhecido para o emissor. Ainda seria necessário um temporizador no protocolo rdt 3.0, supondo que os pacotes podem ser perdidos? Explique.
- R12. Visite a animação interativa Go-Back-N no *site* de apoio do livro.
- A origem enviou cinco pacotes e depois interrompeu a animação antes que qualquer um dos cinco pacotes chegasse ao destino. Então, elimine o primeiro pacote e reinicie a animação. Descreva o que acontece.
  - Repita o experimento, mas agora deixe o primeiro pacote chegar ao destino e eliminate o primeiro reconhecimento. Descreva novamente o que acontece.
  - Por fim, tente enviar seis pacotes. O que acontece?
- R13. Repita a Questão R12, mas agora com a animação interativa Selective Repeat. O que difere o Selective Repeat do Go-Back-N?

## SEÇÃO 3.5

- R14. Falso ou verdadeiro?
- O hospedeiro A está enviando ao hospedeiro B um arquivo grande por uma conexão TCP. Suponha que o hospedeiro B não tenha dados para enviar para o hospedeiro A. O hospedeiro B não enviará reconhecimentos para A porque ele não pode dar carona aos reconhecimentos dos dados.
  - O tamanho de *rwnd* do TCP nunca muda enquanto dura a conexão.
  - Suponha que o hospedeiro A esteja enviando ao hospedeiro B um arquivo grande por uma conexão TCP. O número de *bytes* não reconhecidos que o hospedeiro A envia não pode exceder o tamanho do *buffer* de recepção.
  - Imagine que o hospedeiro A esteja enviando ao hospedeiro B um arquivo grande por uma conexão TCP. Se o número de sequência para um segmento dessa conexão for  $m$ , então o número de sequência para o segmento subsequente será necessariamente  $m + 1$ .
  - O segmento TCP tem um campo em seu cabeçalho para *rwnd*.
  - Suponha que o último SampleRTT de uma conexão TCP seja igual a 1 s. Então, o valor corrente de *TimeoutInterval* para a conexão será necessariamente ajustado para um valor  $\geq 1$  s.
  - Imagine que o hospedeiro A envie ao hospedeiro B, por uma conexão TCP, um segmento com o número de sequência 38 e 4 *bytes* de dados. Nesse mesmo segmento, o número de reconhecimento será necessariamente 42.
- R15. Suponha que o hospedeiro A envie dois segmentos TCP um atrás do outro ao hospedeiro B sobre uma conexão TCP. O primeiro segmento tem número de sequência 90 e o segundo, número de sequência 110.
- Quantos dados tem o primeiro segmento?
  - Suponha que o primeiro segmento seja perdido, mas o segundo chegue a B. No reconhecimento que B envia a A, qual será o número de reconhecimento?
- R16. Considere o exemplo do Telnet discutido na Seção 3.5. Alguns segundos após o usuário digitar a letra “C”, ele digitará a letra “R”. Depois disso, quantos segmentos serão enviados e o que será colocado nos campos de número de sequência e de reconhecimento dos segmentos?

## SEÇÃO 3.7

- R17. Suponha que duas conexões TCP estejam presentes em algum enlace congestionado de velocidade  $R$  bits/s. As duas conexões têm um arquivo imenso para enviar (na

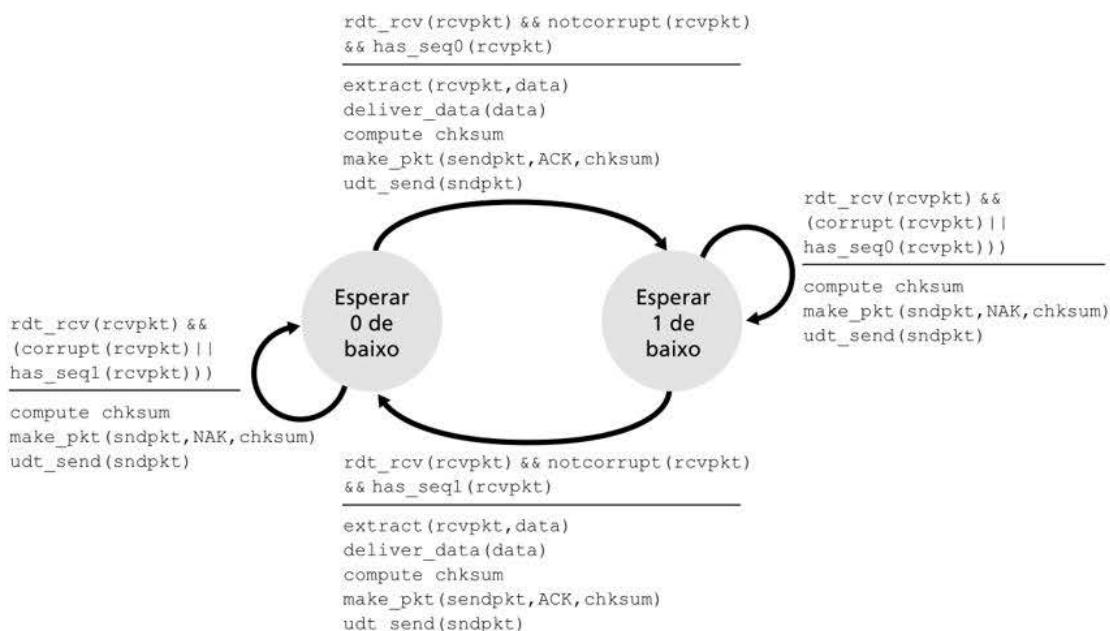
mesma direção, pelo enlace congestionado). As transmissões dos arquivos começam exatamente ao mesmo tempo. Qual é a velocidade de transmissão que o TCP gostaria de dar a cada uma das conexões?

- R18. Verdadeiro ou falso: considere o controle de congestionamento no TCP. Quando um temporizador expira no remetente, o valor de  $ssthresh$  é ajustado para a metade de seu valor anterior.
- R19. Na discussão sobre divisão do TCP, na nota em destaque da Seção 3.7, afirmamos que o tempo de resposta com a divisão do TCP é aproximadamente  $4 \cdot RTT_{FE} + RTT_{BE} +$  tempo de processamento. Justifique essa afirmação.

## Problemas

---

- P1. Suponha que o cliente A inicie uma sessão Telnet com o servidor S. Quase ao mesmo tempo, o cliente B também inicia uma sessão Telnet com o servidor S. Forneça possíveis números de porta da fonte e do destino para:
- Os segmentos enviados de A para S.
  - Os segmentos enviados de B para S.
  - Os segmentos enviados de S para A.
  - Os segmentos enviados de S para B.
  - Se A e B são hospedeiros diferentes, é possível que o número de porta da fonte nos segmentos de A para S seja o mesmo que nos de B para S?
  - E se forem o mesmo hospedeiro?
- P2. Considere a Figura 3.5. Quais são os valores da porta de fonte e da porta de destino nos segmentos que fluem do servidor de volta aos processos clientes? Quais são os endereços IP nos datagramas de camada de rede que carregam os segmentos de camada de transporte?
- P3. O UDP e o TCP usam complementos de 1 para suas somas de verificação. Suponha que você tenha as seguintes três palavras de 8 bits: 01010011, 01100110 e 01110100. Qual é o complemento de 1 para as somas dessas palavras? (Note que, embora o UDP e o TCP usem palavras de 16 bits no cálculo da soma de verificação, nesse problema solicitamos que você considere somas de 8 bits.) Mostre todo o trabalho. Por que o UDP toma o complemento de 1 da soma, isto é, por que não toma apenas a soma? Com o esquema de complemento de 1, como o destinatário detecta erros? É possível que um erro de 1 bit passe despercebido? E um erro de 2 bits?
- P4. a. Suponha que você tenha os seguintes bytes: 01011100 e 01100101. Qual é o complemento de 1 da soma desses 2 bytes?  
b. Suponha que você tenha os seguintes bytes: 11011010 e 01100101. Qual é o complemento de 1 da soma desses 2 bytes?  
c. Para os bytes do item (a), dê um exemplo em que um bit é invertido em cada um dos 2 bytes e, mesmo assim, o complemento de um não muda.
- P5. Suponha que o receptor UDP calcule a soma de verificação da Internet para o segmento UDP recebido e identifique que essa soma coincide com o valor transportado no campo da soma de verificação. O receptor pode estar absolutamente certo de que não ocorreu nenhum erro de bit? Explique.
- P6. Considere nosso motivo para corrigir o protocolo rtd2.1. Demonstre que o destinatário apresentado na Figura 3.60, quando em operação com o remetente mostrado na Figura 3.11, pode levar remetente e destinatário a entrar em estado de travamento, em que cada um espera por um evento que nunca vai ocorrer.



**Figura 3.60** Um receptor incorreto para o protocolo rdt 2.1.

- P7. No protocolo rdt3.0, os pacotes ACK que fluem do destinatário ao remetente não têm números de sequência (embora tenham um campo de ACK que contém o número de sequência do pacote que estão reconhecendo). Por que nossos pacotes ACK não requerem números de sequência?
- P8. Elabore a FSM para o lado destinatário do protocolo rdt3.0.
- P9. Elabore um diagrama de mensagens para a operação do protocolo rdt3.0 quando pacotes de dados e de reconhecimento estão truncados. Seu diagrama deve ser semelhante ao usado na Figura 3.16.
- P10. Considere um canal que pode perder pacotes, mas cujo atraso máximo é conhecido. Modifique o protocolo rdt2.1 para incluir esgotamento de temporização do remetente e retransmissão. Informalmente, argumente por que seu protocolo pode se comunicar de modo correto por esse canal.
- P11. Considere o rdt2.2 destinatário da Figura 3.14 e a criação de um novo pacote na autotransição (i.e., a transição do estado de volta para si mesmo) nos estados “Esperar 0 de baixo” e “Esperar 1 de baixo”: `sndpkt=make_pkt(ACK,1,checksum)` e `sndpkt=make_pkt(ACK,0,checksum)`. O protocolo funcionaria corretamente se essa ação fosse removida da autotransição no estado “Esperar 1 de baixo”? Justifique sua resposta. E se esse evento fosse removido da autotransição no estado “Esperar 0 de baixo”? (Dica: neste último caso, considere o que aconteceria se o primeiro pacote do remetente ao destinatário fosse corrompido.)
- P12. O lado remetente do rdt3.0 simplesmente ignora (i.e., não realiza nenhuma ação) todos os pacotes recebidos que estão errados ou com o valor errado no campo acknum de um pacote de reconhecimento. Suponha que em tais circunstâncias o rdt3.0 devesse apenas retransmitir o pacote de dados corrente. Nesse caso, o protocolo ainda funcionaria? (Dica: considere o que aconteceria se houvesse apenas erros de bits; não há perdas de pacotes, mas podem ocorrer esgotamentos de temporização prematuros. Imagine quantas vezes o enésimo pacote é enviado, no limite em que  $n$  se aproximasse do infinito.)
- P13. Considere o protocolo rdt3.0. Elabore um diagrama mostrando que, se a conexão de rede entre o remetente e o destinatário puder alterar a ordem de mensagens (i.e., se for possível reordenar duas mensagens que se propagam no meio entre o remetente e o

destinatário), então o protocolo *bit* alternante não funcionará corretamente (lembre-se de identificar com clareza o sentido no qual o protocolo não funcionará de modo correto). Seu diagrama deve mostrar o remetente à esquerda e o destinatário à direita; o eixo do tempo deverá estar orientado de cima para baixo na página e mostrar a troca de mensagem de dados (D) e de reconhecimento (A). Não se esqueça de indicar o número de sequência associado com qualquer segmento de dados ou de reconhecimento.

- P14. Considere um protocolo de transferência confiável de dados que use somente reconhecimentos negativos. Suponha que o remetente envie dados com pouca frequência. Um protocolo que utiliza somente NAKs seria preferível a um protocolo que utiliza ACKs? Por quê? Agora suponha que o remetente tenha uma grande quantidade de dados para enviar e que a conexão fim a fim sofra poucas perdas. Nesse segundo caso, um protocolo que utilize somente NAKs seria preferível a um protocolo que utilize ACKs? Por quê?
- P15. Considere o exemplo em que se atravessa os Estados Unidos mostrado na Figura 3.17. Que tamanho deveria ter a janela para que a utilização do canal fosse maior do que 98%? Suponha que o tamanho de um pacote seja 1.500 bytes, incluindo os campos do cabeçalho e os dados.
- P16. Suponha que uma aplicação utilize rdt3.0 como seu protocolo da camada de transporte. Como o protocolo pare e espere possui uma utilização do canal muito baixa (mostrada no exemplo de travessia dos Estados Unidos), os criadores dessa aplicação permitem que o receptor continue enviando de volta um número (mais do que dois) de ACKs 0 e ACKs 1 alternados, mesmo que os dados correspondentes não cheguem ao receptor. O projeto dessa aplicação aumentaria a utilização do canal? Por quê? Há possíveis problemas com esse método? Explique.
- P17. Considere duas entidades de rede, A e B, que estão conectadas por um canal bidirecional perfeito (i.e., qualquer mensagem enviada será recebida corretamente; o canal não corromperá, perderá nem reordenará pacotes). A e B devem entregar mensagens de dados entre si de forma alternada: primeiro, A deve entregar uma mensagem a B, depois B deve entregar uma mensagem a A, e assim por diante. Se uma entidade estiver em um estado onde não deve tentar entregar uma mensagem ao outro lado e houver um evento como a chamada `rdt_send(data)` de cima que tente transmitir dados para baixo, para o outro lado, essa chamada de cima pode apenas ser ignorada com uma chamada a `rdt_unable_to_send(data)`, que informa à camada de cima que atualmente não é possível enviar dados. (Nota: essa suposição simplificada é para que você não tenha de se preocupar com o armazenamento de dados em *buffer*.) Elabore uma especificação FSM para este protocolo (uma FSM para A e uma para B!). Observe que você não precisa se preocupar com um mecanismo de confiabilidade aqui; o ponto importante da questão é criar uma especificação FSM que reflita o comportamento sincronizado das duas entidades. Você deverá usar os seguintes eventos e ações que possuem o mesmo significado do protocolo rdt1.0 da Figura 3.9: `rdt_send(data)`, `packet = make_pkt(data)`, `udt_send(packet)`, `rdt_rcv(packet)`, `extract(packet, data)`, `deliver_data(data)`. Cuide para que o protocolo reflita a alternância estrita de envio entre A e B. Além disso, não se esqueça de indicar os estados iniciais de A e B em suas especificações FSM.
- P18. No protocolo genérico SR que estudamos na Seção 3.4.4, o remetente transmite uma mensagem assim que ela está disponível (se ela estiver na janela), sem esperar por um reconhecimento. Suponha, agora, que queiramos um protocolo SR que envie duas mensagens de cada vez. Isto é, o remetente enviará um par de mensagens, e o par de mensagens subsequente somente deverá ser enviado quando o remetente souber que ambas as mensagens do primeiro par foram recebidas corretamente.

Suponha que o canal possa perder mensagens, mas que não as corromperá nem as reordenará. Elabore um protocolo de controle de erro para a transferência confiável

unidirecional de mensagens. Dê uma descrição FSM do remetente e do destinatário. Descreva o formato dos pacotes enviados entre o remetente e o destinatário e vice-versa. Se você usar quaisquer procedimentos de chamada que não sejam os da Seção 3.4 (p. ex., `udt_send()`, `start_timer()`, `rdt_rcv()` etc.), esclareça as ações desses procedimentos. Dê um exemplo (um diagrama de mensagens para o remetente e para o destinatário) mostrando como seu protocolo se recupera de uma perda de pacote.

- P19. Considere um cenário em que o hospedeiro A queira enviar pacotes para os hospedeiros B e C simultaneamente. O hospedeiro A está conectado a B e a C por um canal *broadcast* – um pacote enviado por A é levado pelo canal a B e a C. Suponha que o canal *broadcast* que conecta A, B e C possa, de modo independente, perder e corromper mensagens (e assim, p. ex., uma mensagem enviada de A poderia ser recebida corretamente por B, mas não por C). Projete um protocolo de controle de erro do tipo pare e espere para a transferência confiável de um pacote de A para B e para C, tal que A não receba novos dados da camada superior até que saiba que B e C receberam corretamente o pacote em questão. Dê descrições FSM de A e C. (*Dica:* a FSM para B deve ser a mesma que para C.) Também dê uma descrição do(s) formato(s) de pacote usado(s).
- P20. Considere um cenário em que os hospedeiros A e B queiram enviar mensagens ao hospedeiro C. Os hospedeiros A e C estão conectados por um canal que pode perder e corromper (e não reordenar) mensagens. Os hospedeiros B e C estão conectados por outro canal (independente do canal que conecta A e C) com as mesmas propriedades. A camada de transporte no hospedeiro C deve alternar o envio de mensagens de A e B para a camada acima (i.e., ela deve primeiro transmitir os dados de um pacote de A e depois os dados de um pacote de B, e assim por diante). Elabore um protocolo de controle de erro pare e espere para transferência confiável de pacotes de A e B para C, com envio alternado em C, como descrito acima. Dê descrições FSM de A e C. (*Dica:* a FSM para B deve ser basicamente a mesma de A.) Dê, também, uma descrição do(s) formato(s) de pacote utilizado(s).
- P21. Suponha que haja duas entidades de rede A e B, e que B tenha um suprimento de mensagens de dados que será enviado a A de acordo com as seguintes convenções: quando A recebe uma solicitação da camada superior para obter a mensagem de dados (D) seguinte de B, A deve enviar uma mensagem de requisição (R) para B no canal A para B; somente quando B receber uma mensagem R, ele poderá enviar uma mensagem de dados (D) de volta a A pelo canal B para A; A deve entregar uma cópia de cada mensagem D à camada superior; mensagens R podem ser perdidas (mas não corrompidas) no canal A para B; mensagens (D), uma vez enviadas, são sempre entregues corretamente; o atraso entre ambos os canais é desconhecido e variável.
- Elabore um protocolo (dê uma descrição FSM) que incorpore os mecanismos apropriados para compensar a propensão à perda do canal A a B e implemente passagem de mensagem para a camada superior na entidade A, como discutido antes. Utilize apenas os mecanismos absolutamente necessários.
- P22. Considere o protocolo GBN com um tamanho de janela remetente de 4 e uma faixa de números de sequência de 1.024. Suponha que, no tempo  $t$ , o pacote seguinte na ordem, pelo qual o destinatário está esperando, tenha um número de sequência  $k$ . Admita que o meio não reordene as mensagens. Responda às seguintes perguntas:
- Quais são os possíveis conjuntos de números de sequência dentro da janela do remetente no tempo  $t$ ? Justifique sua resposta.
  - Quais são todos os possíveis valores do campo ACK em todas as mensagens que estão atualmente se propagando de volta ao remetente no tempo  $t$ ? Justifique sua resposta.

- P23. Considere os protocolos GBN e SR. Suponha que o espaço de números de sequência seja de tamanho  $k$ . Qual será o maior tamanho de janela permissível que evitará que ocorram problemas como os da Figura 3.27 para cada um desses protocolos?
- P24. Responda verdadeiro ou falso às seguintes perguntas e justifique resumidamente sua resposta:
- Com o protocolo SR, é possível o remetente receber um ACK para um pacote que caia fora de sua janela corrente.
  - Com o GBN, é possível o remetente receber um ACK para um pacote que caia fora de sua janela corrente.
  - O protocolo *bit* alternante é o mesmo que o protocolo SR com janela do remetente e do destinatário de tamanho 1.
  - O protocolo *bit* alternante é o mesmo que o protocolo GBN com janela do remetente e do destinatário de tamanho 1.
- P25. Dissemos que uma aplicação pode escolher o UDP para um protocolo de transporte, pois oferece um melhor controle às aplicações (do que o TCP) de quais dados são enviados em um segmento e quando isso ocorre.  
Por que uma aplicação possui mais controle de quais dados são enviados em um segmento?  
Por que uma aplicação possui mais controle de quando o segmento é enviado?
- P26. Considere a transferência de um arquivo enorme de  $L$  bytes do hospedeiro A para o hospedeiro B. Suponha um MSS de 536 bytes.
- Qual é o máximo valor de  $L$  tal que não sejam esgotados os números de sequência TCP? Lembre-se de que o campo de número de sequência TCP tem 4 bytes.
  - Para o  $L$  que obtiver em (a), descubra quanto tempo demora para transmitir o arquivo. Admita que um total de 66 bytes de cabeçalho de transporte, de rede e de enlace de dados seja adicionado a cada segmento antes que o pacote resultante seja enviado por um enlace de 155 Mbits/s. Ignore controle de fluxo e controle de congestionamento de modo que A possa enviar os segmentos um atrás do outro e continuamente.
- P27. Os hospedeiros A e B estão se comunicando por meio de uma conexão TCP, e o hospedeiro B já recebeu de A todos os bytes até o byte 126. Suponha que A envie, então, dois segmentos para B sucessivamente. O primeiro e o segundo segmentos contêm 80 e 40 bytes de dados, respectivamente. No primeiro segmento, o número de sequência é 127, o número de porta de origem é 302, e o número de porta de destino é 80. O hospedeiro B envia um reconhecimento ao receber um segmento do hospedeiro A.
- No segundo segmento enviado do hospedeiro A para B, quais são o número de sequência, da porta de origem e da porta de destino?
  - Se o primeiro segmento chegar antes do segundo, no reconhecimento do primeiro segmento que chegar, qual é o número do reconhecimento, da porta de origem e da porta de destino?
  - Se o segundo segmento chegar antes do primeiro, no reconhecimento do primeiro segmento que chegar, qual é o número do reconhecimento?
  - Suponha que dois segmentos enviados por A cheguem em ordem a B. O primeiro reconhecimento é perdido, e o segundo chega após o primeiro intervalo do esgotamento de temporização. Elabore um diagrama de temporização, mostrando esses segmentos, e todos os outros, e os reconhecimentos enviados. (Suponha que não haja qualquer perda de pacote adicional.) Para cada segmento de seu desenho, apresente o número de sequência e o número de bytes de dados; para cada reconhecimento adicionado por você, informe o número do reconhecimento.
- P28. Os hospedeiros A e B estão diretamente conectados com um enlace de 100 Mbits/s. Existe uma conexão TCP entre os dois hospedeiros, e A está enviando a B um arquivo

enorme por meio dessa conexão. O hospedeiro A pode enviar seus dados da aplicação para o *socket* TCP a uma taxa que chega a 120 Mbits/s, mas o hospedeiro B pode ler o *buffer* de recebimento TCP a uma taxa de 50 Mbits/s. Descreva o efeito do controle de fluxo do TCP.

- P29. Os *cookies* SYN foram discutidos na Seção 3.5.6.
- Por que é necessário que o servidor use um número de sequência especial no SYNACK?
  - Suponha que um atacante saiba que um hospedeiro-alvo utilize SYN *cookies*. O atacante consegue criar conexões semiabertas ou completamente abertas apenas enviando um pacote ACK para o alvo? Por quê?
  - Suponha que um atacante receba uma grande quantidade de números de sequência enviados pelo servidor. O atacante consegue fazer o servidor criar muitas conexões totalmente abertas enviando ACKs com esses números de sequência? Por quê?
- P30. Considere a rede mostrada no Cenário 2 na Seção 3.6.1. Suponha que os hospedeiros emissores A e B possuam valores de esgotamento de temporização fixos.
- Analise o fato de que aumentar o tamanho do *buffer* finito do roteador pode possivelmente reduzir a vazão ( $\lambda_{out}$ ).
  - Agora suponha que os hospedeiros ajustem dinamicamente seus valores de esgotamento de temporização (como o que o TCP faz) com base no atraso no *buffer* no roteador. Aumentar o tamanho do *buffer* ajudaria a aumentar a vazão? Por quê?
- P31. Suponha que os cinco valores de SampleRTT medidos (ver Seção 3.5.3) sejam 106 ms, 120 ms, 140 ms, 90 ms e 115 ms. Calcule o EstimatedRTT depois que forem obtidos cada um desses valores de SampleRTT, usando um valor de  $\alpha = 0,125$  e supondo que o valor de EstimatedRTT seja 100 ms imediatamente antes que a primeira dessas cinco amostras seja obtida. Calcule também o DevRTT após a obtenção de cada amostra, considerando um valor de  $\beta = 0,25$  e que o valor de DevRTT seja 5 ms imediatamente antes que a primeira dessas cinco amostras seja obtida. Por fim, calcule o TimeoutInterval do TCP após a obtenção de cada uma dessas amostras.
- P32. Considere o procedimento do TCP para estimar o RTT. Suponha que  $\alpha = 0,1$ . Considere que  $\text{SampleRTT}_1$  seja a amostra de RTT mais recente,  $\text{SampleRTT}_2$  seja a próxima amostra mais recente, e assim por diante.
- Para uma dada conexão TCP, suponha que quatro reconhecimentos foram devolvidos com as amostras RTT correspondentes  $\text{SampleRTT}_4$ ,  $\text{SampleRTT}_3$ ,  $\text{SampleRTT}_2$  e  $\text{SampleRTT}_1$ . Expresse EstimatedRTT em termos das quatro amostras RTT.
  - Generalize sua fórmula para  $n$  amostras de RTTs.
  - Para a fórmula em (b), considere  $n$  tendendo ao infinito. Comente por que esse procedimento de média é denominado média móvel exponencial.
- P33. Na Seção 3.5.3 discutimos estimativa de RTT para o TCP. Em sua opinião, por que o TCP evita medir o SampleRTT para segmentos retransmitidos?
- P34. Qual é a relação entre a variável SendBase na Seção 3.5.4 e a variável LastByteRcvd na Seção 3.5.5?
- P35. Qual é a relação entre a variável LastByteRcvd na Seção 3.5.5 e a variável y na Seção 3.5.4?
- P36. Na Seção 3.5.4, vimos que o TCP espera até receber três ACKs duplicados antes de realizar uma retransmissão rápida. Em sua opinião, por que os projetistas do TCP preferiram não realizar uma retransmissão rápida após ser recebido o primeiro ACK duplicado para um segmento?
- P37. Compare o GBN, SR e o TCP (sem ACK retardado). Admita que os valores do esgotamento de temporização para os três protocolos sejam longos o suficiente de tal modo que cinco segmentos de dados consecutivos e seus ACKs correspondentes possam ser

recebidos (se não perdidos no canal) por um hospedeiro receptor (hospedeiro B) e por um hospedeiro emissor (hospedeiro A), respectivamente. Suponha que A envie cinco segmentos de dados para B, e que o segundo segmento (enviado de A) esteja perdido. No fim, todos os cinco segmentos de dados foram corretamente recebidos pelo hospedeiro B.

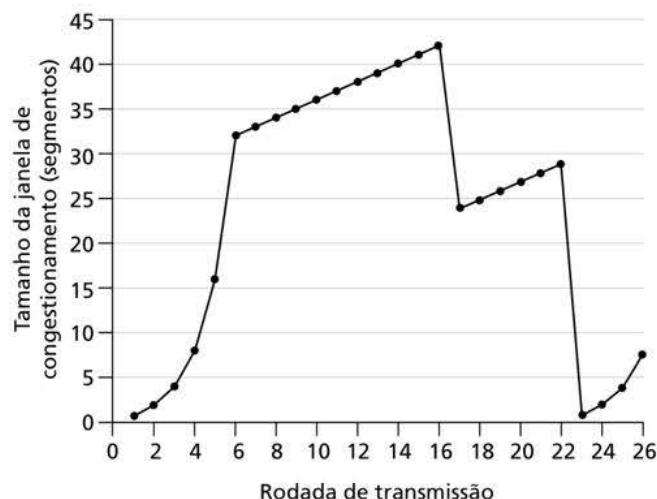
- Quantos segmentos A enviou no total e quantos ACKs o hospedeiro B enviou no total? Quais são seus números de sequência? Responda essa questão para todos os três protocolos.
- Se os valores do esgotamento de temporização para os três protocolos forem muito maiores do que 5 RTT, então qual protocolo envia com sucesso todos os cinco segmentos de dados em um menor intervalo de tempo?

P38. Em nossa descrição sobre o TCP na Figura 3.53, o valor do limiar,  $ssthresh$ , é definido como  $ssthresh = cwnd/2$  em diversos lugares, e o valor  $ssthresh$  é referido como sendo definido para metade do tamanho da janela quando ocorreu um evento de perda. A taxa pela qual o emissor está enviando quando ocorreu o evento de perda deve ser mais ou menos igual a segmentos  $cwnd$  por RTT? Explique sua resposta. Se for negativa, você pode sugerir uma maneira diferente pela qual  $ssthresh$  deva ser definido?

P39. Considere a Figura 3.46(b). Se  $\lambda'_{in}$  aumentar para mais do que  $R/2$ ,  $\lambda'_{out}$  poderá aumentar para mais do que  $R/3$ ? Explique. Agora considere a Figura 3.46(c). Se  $\lambda'_{in}$  aumentar para mais do que  $R/2$ ,  $\lambda'_{out}$  poderá aumentar para mais de  $R/4$  admitindo-se que um pacote será transmitido duas vezes, em média, do roteador para o destinatário? Explique.

P40. Considere a Figura 3.61. Admitindo-se que TCP Reno é o protocolo que experimenta o comportamento mostrado no gráfico, responda às seguintes perguntas. Em todos os casos, você deverá apresentar uma justificativa resumida para sua resposta.

- Quais os intervalos de tempo em que a partida lenta do TCP está em execução?
- Quais os intervalos de tempo em que a prevenção de congestionamento do TCP está em execução?
- Após a 16<sup>a</sup> rodada de transmissão, a perda de segmento será detectada por três ACKs duplicados ou por um esgotamento de temporização?
- Após a 22<sup>a</sup> rodada de transmissão, a perda de segmento será detectada por três ACKs duplicados ou por um esgotamento de temporização?



**Figura 3.61** Tamanho da janela TCP em relação ao tempo.

- e. Qual é o valor inicial de  $ssthresh$  na primeira rodada de transmissão?
- f. Qual é o valor inicial de  $ssthresh$  na 18<sup>a</sup> rodada de transmissão?
- g. Qual é o valor de  $ssthresh$  na 24<sup>a</sup> rodada de transmissão?
- h. Durante qual rodada de transmissão é enviado o 70º segmento?
- i. Admitindo-se que uma perda de pacote será detectada após a 26<sup>a</sup> rodada pelo recebimento de três ACKs duplicados, quais serão os valores do tamanho da janela de congestionamento e de  $ssthresh$ ?
- j. Suponha que o TCP Tahoe seja usado (em vez do TCP Reno), e que ACKs duplicados triplos sejam recebidos na 16<sup>a</sup> rodada. Quais são o  $ssthresh$  e o tamanho da janela de congestionamento na 19<sup>a</sup> rodada?
- k. Suponha novamente que o TCP Tahoe seja usado, e que exista um evento de esgotamento de temporização na 22<sup>a</sup> sessão. Quantos pacotes foram enviados da 17<sup>a</sup> sessão até a 22<sup>a</sup>, inclusive?

- P41. Consulte a Figura 3.55, que ilustra a convergência do algoritmo AIMD do TCP. Suponha que, em vez de uma diminuição multiplicativa, o TCP reduza o tamanho da janela de uma quantidade constante. O AIMD resultante convergiria como um algoritmo de igual compartilhamento? Justifique sua resposta usando um diagrama semelhante ao da Figura 3.55.
- P42. Na Seção 3.5.4, discutimos a duplicação do intervalo de temporização após um evento de esgotamento de temporização. Esse mecanismo é uma forma de controle de congestionamento. Por que o TCP precisa de um mecanismo de controle de congestionamento que utiliza janelas (como estudado na Seção 3.7) além desse mecanismo de duplicação do intervalo de esgotamento de temporização?
- P43. O hospedeiro A está enviando um arquivo enorme ao hospedeiro B por uma conexão TCP. Nessa conexão, nunca há perda de pacotes e os temporizadores nunca se esgotam. Seja  $R$  bits/s a taxa de transmissão do enlace que liga o hospedeiro A à Internet. Suponha que o processo em A consiga enviar dados para seu *socket* TCP a uma taxa de  $S$  bits/s, em que  $S = 10 \cdot R$ . Suponha ainda que o *buffer* de recepção do TCP seja grande o suficiente para conter o arquivo inteiro, e que o *buffer* de envio possa conter apenas 1% do arquivo. O que impediria o hospedeiro A de passar dados continuamente para seu *socket* TCP à taxa de  $S$  bits/s: o controle de fluxo do TCP; o controle de congestionamento do TCP; ou alguma outra coisa? Elabore sua resposta.
- P44. Considere enviar um arquivo grande de um computador a outro por meio de uma conexão TCP em que não haja perda.
- a. Suponha que o TCP utilize AIMD para seu controle de congestionamento sem partida lenta. Admitindo que  $cwnd$  aumenta 1 MSS sempre que um lote de ACK é recebido e os tempos da viagem de ida e volta são constantes, quanto tempo leva para  $cwnd$  aumentar de 6 MSS para 12 MSS (admitindo nenhum evento de perda)?
  - b. Qual é a vazão média (em termos de MSS e RTT) para essa conexão, sendo o tempo = 6 RTT?
- P45. Considere a Figura 3.54. Suponha que em  $t_3$ , a taxa de envio pela qual a perda por congestionamento ocorre a seguir cai para  $0,75*W_{\max}$  (sem o conhecimento dos remetentes TCP, é claro). Mostre a evolução do TCP Reno e do TCP CUBIC para duas rodadas adicionais de cada (*Dica: observe que os instantes em que o TCP Reno e o TCP CUBIC reagem à perda por congestionamento podem não ser mais os mesmos*).
- P46. Considere a Figura 3.54 novamente. Suponha que em  $t_3$ , a taxa de envio pela qual a perda por congestionamento ocorre a seguir aumenta para  $1,5*W_{\max}$ . Mostre a evolução do TCP Reno e do TCP CUBIC para duas rodadas adicionais de cada (*Dica: veja a dica da P45*).

- P47. Relembre a descrição macroscópica da vazão do TCP. No período de tempo transcorrido para a taxa da conexão variar de  $W/(2 \cdot RTT)$  a  $W/RTT$ , apenas um pacote é perdido (bem ao final do período).
- Mostre que a taxa de perda (fração de pacotes perdidos) é igual a

$$L = \text{taxa de perda} = \frac{1}{\frac{3}{8}W^2 + \frac{3}{4}W}$$

- Use o resultado anterior para mostrar que, se uma conexão tiver taxa de perda  $L$ , sua largura de banda média é dada aproximadamente por:

$$\approx \frac{1,22 \cdot MSS}{RTT \sqrt{L}}$$

- P48. Considere que somente uma única conexão TCP (Reno) utiliza um enlace de 10 Mbits/s que não armazena nenhum dado. Suponha que esse enlace seja o único congestionado entre os hospedeiros emissor e receptor. Admita que o emissor TCP tenha um arquivo enorme para enviar ao receptor e o buffer de recebimento do receptor é muito maior do que a janela de congestionamento. Também fazemos as seguintes suposições: o tamanho de cada segmento TCP é 1.500 bytes; o atraso de propagação bidirecional dessa conexão é 150 ms; e essa conexão TCP está sempre na fase de prevenção de congestionamento, ou seja, ignore a partida lenta.

- Qual é o tamanho máximo da janela (em segmentos) que a conexão TCP pode atingir?
- Qual é o tamanho médio da janela (em segmentos) e a vazão média (em bits/s) dessa conexão TCP?
- Quanto tempo essa conexão TCP leva para alcançar sua janela máxima novamente após se recuperar da perda de um pacote?

- P49. Considere o cenário descrito na questão anterior. Suponha que o enlace de 10 Mbits/s possa armazenar um número finito de segmentos. Demonstre que para o enlace sempre enviar dados, teríamos que escolher um tamanho de buffer que é, pelo menos, o produto da velocidade do enlace  $C$  e o atraso de propagação bidirecional entre o emissor e o receptor.

- P50. Repita a Questão P48, mas substituindo o enlace de 10 Mbits/s por um de 10 Gbits/s. Observe que em sua resposta ao item (c), verá que o tamanho da janela de congestionamento leva muito tempo para atingir seu tamanho máximo após se recuperar de uma perda de pacote. Elabore uma solução para resolver o problema.

- P51. Suponha que  $T$  (medido por RTT) seja o intervalo de tempo que uma conexão TCP leva para aumentar seu tamanho de janela de congestionamento de  $W/2$  para  $W$ , sendo  $W$  o tamanho máximo da janela de congestionamento. Demonstre que  $T$  é uma função da vazão média do TCP.

- P52. Considere um algoritmo AIMD do TCP simplificado, sendo o tamanho da janela de congestionamento medido em número de segmentos e não em bytes. No aumento aditivo, o tamanho da janela de congestionamento aumenta por um segmento em cada RTT. Na diminuição multiplicativa, o tamanho da janela de congestionamento diminui para metade (se o resultado não for um número inteiro, arredondar para o número inteiro mais próximo). Suponha que duas conexões TCP,  $C_1$  e  $C_2$  compartilhem um único enlace congestionado com 30 segmentos por segundo de velocidade. Admita que  $C_1$  e  $C_2$  estejam na fase de prevenção de congestionamento. O RTT da conexão  $C_1$  é de 50 ms e o da conexão  $C_2$  é de 100 ms. Suponha que quando a taxa de dados no enlace excede a velocidade do enlace, todas as conexões TCP sofrem perda de segmento de dados.

- a. Se  $C_1$  e  $C_2$  no tempo  $t_0$  possuem uma janela de congestionamento de 10 segmentos, quais são seus tamanhos de janela de congestionamento após 1.000 ms?
- b. No final das contas, essas duas conexões obterão a mesma porção da largura de banda do enlace congestionado? Explique.
- P53. Considere a rede descrita na questão anterior. Agora suponha que as duas conexões TCP,  $C_1$  e  $C_2$ , possuam o mesmo RTT de 100 ms e que, no tempo  $t_0$ , o tamanho da janela de congestionamento de  $C_1$  seja 15 segmentos, e que o de  $C_2$  seja 10 segmentos.
- Quais são os tamanhos de suas janelas de congestionamento após 2.200 ms?
  - No final das contas, essas duas conexões obterão a mesma porção da largura de banda do enlace congestionado?
  - Dizemos que duas conexões são sincronizadas se ambas atingirem o tamanho máximo e mínimo de janela ao mesmo tempo. No final das contas, essas duas conexões serão sincronizadas? Se sim, quais os tamanhos máximos de janela?
  - Essa sincronização ajudará a melhorar a utilização do enlace compartilhado? Por quê? Elabore alguma ideia para romper essa sincronização.
- P54. Considere uma modificação ao algoritmo de controle de congestionamento do TCP. Em vez do aumento aditivo, podemos utilizar o aumento multiplicativo. Um emissor TCP aumenta seu tamanho de janela por uma constante positiva pequena  $\alpha$  ( $0 < \alpha < 1$ ) ao receber um ACK válido. Encontre a relação funcional entre a taxa de perda  $L$  e a janela máxima de congestionamento  $W$ . Para esse TCP modificado, demonstre, independentemente da vazão média do TCP, que uma conexão TCP sempre gasta a mesma quantidade de tempo para aumentar seu tamanho da janela de congestionamento de  $W/2$  para  $W$ .
- P55. Quando discutimos TCPs futuros na Seção 3.7, observamos que, para conseguir uma vazão de 10 Gbits/s, o TCP apenas poderia tolerar uma probabilidade de perda de segmentos de  $2 \cdot 10^{-10}$  (ou, equivalentemente, uma perda para cada 5 milhões de segmentos). Mostre a derivação dos valores para  $2 \cdot 10^{-10}$  (ou 1: 5.000.000) a partir dos valores de RTT e do MSS dados na Seção 3.7. Se o TCP precisasse suportar uma conexão de 100 Gbits/s, qual seria a perda tolerável?
- P56. Quando discutimos controle de congestionamento em TCP na Seção 3.7, admitimos implicitamente que o remetente TCP sempre tinha dados para enviar. Agora considere o caso em que o remetente TCP envia uma grande quantidade de dados e então fica ocioso em  $t_1$  (já que não há mais dados a enviar). O TCP permanecerá ocioso por um período relativamente longo e então irá querer enviar mais dados em  $t_2$ . Quais são as vantagens e desvantagens de o TCP utilizar os valores `cwnd` e `ssthresh` de  $t_1$  quando começar a enviar dados em  $t_2$ ? Qual alternativa você recomendaria? Por quê?
- P57. Neste problema, verificamos se o UDP ou o TCP apresentam um grau de autenticação do ponto de chegada.
- Considere um servidor que receba uma solicitação dentro de um pacote UDP e responda a essa solicitação dentro de um pacote UDP (p. ex., como feito por um servidor DNS). Se um cliente com endereço IP X envia para o servidor um endereço de origem falso, Y, para onde o servidor enviará sua resposta?
  - Suponha que um servidor receba um SYN de endereço IP de origem Y, e depois de responder com um SYNACK, recebe um ACK com o endereço IP de origem Y com o número de reconhecimento correto. Admitindo que o servidor escolha um número de sequência aleatório e que não haja um *man-in-the-middle*, o servidor pode ter certeza de que o cliente realmente está em Y (e não em outro endereço X que está se passando por Y)?

- P58. Neste problema, consideramos o atraso apresentado pela fase de partida lenta do TCP. Considere um cliente e um servidor da Web diretamente conectados por um enlace de taxa  $R$ . Suponha que o cliente queira recuperar um objeto cujo tamanho seja exatamente igual a  $15 S$ , sendo  $S$  o tamanho máximo do segmento (MSS). Considere RTT o tempo de viagem de ida e volta entre o cliente e o servidor (admitindo que seja constante). Ignorando os cabeçalhos do protocolo, determine o tempo para recuperar o objeto (incluindo o estabelecimento da conexão TCP) quando
- $4 S/R > S/R + RTT > 2S/R$
  - $S/R + RTT > 4 S/R$
  - $S/R > RTT$ .

## Tarefa de programação

---

### Implementando um protocolo de transporte confiável

Nesta tarefa de programação de laboratório, você escreverá o código para a camada de transporte do remetente e do destinatário no caso da implementação de um protocolo simples de transferência confiável de dados. Há duas versões deste laboratório: a do protocolo de *bit alternante* e a do GBN. Essa tarefa será muito divertida, já que a sua realização não será muito diferente da que seria exigida em uma situação real.

Como você provavelmente não tem máquinas autônomas (com um sistema operacional que possa modificar), seu código terá de rodar em um ambiente de *hardware/software* simulado. Contudo, a interface de programação fornecida a suas rotinas – isto é, o código que chamará suas entidades de cima e de baixo – é muito próxima ao que é feito em um ambiente UNIX real. (Na verdade, as interfaces do *software* descritas nesta tarefa de programação são muito mais realistas do que os remetentes e destinatários de laço infinito descritos em muitos livros.) A parada e o acionamento dos temporizadores também são simulados, e as interrupções do temporizador provocarão a ativação da sua rotina de tratamento de temporização.

## Wireshark Lab: explorando o TCP

---

Neste laboratório, você usará seu navegador para acessar um arquivo de um servidor Web. Como nos anteriores, você usará Wireshark para capturar os pacotes que estão chegando ao seu computador. Mas, diferentemente daqueles laboratórios, também poderá baixar, do mesmo servidor Web do qual baixou o arquivo, um relatório (*trace*) de pacotes que pode ser lido pelo Wireshark. Nesse relatório do servidor, você encontrará os pacotes que foram gerados pelo seu próprio acesso ao servidor Web. Você analisará os diagramas dos lados do cliente e do servidor de modo a explorar aspectos do TCP. Em especial, fará uma avaliação do desempenho da conexão TCP entre seu computador e o servidor Web. Você analisará o comportamento da janela TCP e deduzirá comportamentos de perda de pacotes, de retransmissão, de controle de fluxo e de controle de congestionamento e do tempo de ida e volta estimado.

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

## Wireshark Lab: explorando o UDP

---

Neste pequeno laboratório, você realizará uma captura de pacote e uma análise de sua aplicação favorita que utiliza o UDP (p. ex., o DNS ou uma aplicação multimídia, como o Skype). Como aprendemos na Seção 3.3, o UDP é um protocolo de transporte simples. Neste laboratório, você examinará os campos do cabeçalho no segmento UDP, assim como o cálculo da soma de verificação.

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

## ENTREVISTA

### Van Jacobson

Van Jacobson trabalha na Google e foi Research Fellow no PARC. Antes disso, foi cofundador e cientista-chefe da Packet Design. Antes ainda, foi cientista-chefe na Cisco. Antes de entrar para a Cisco, chefiou o Network Research Group, no Lawrence Berkeley National Laboratory, e lecionou na Universidade da Califórnia, em Berkeley e Stanford. Van recebeu o prêmio ACM SIGCOMM em 2001 pela destacada contribuição de toda uma vida para o campo de redes de comunicação e o prêmio IEEE Kobayashi em 2002 por “contribuir para o conhecimento do congestionamento de redes e por desenvolver mecanismos de controle de congestionamento de rede que permitiram a escalada bem-sucedida da Internet”. Em 2004, foi eleito para a Academia Nacional de Engenharia dos Estados Unidos.



Imagem cortesia de Van Jacobson

**Por favor, descreva um ou dois dos projetos mais interessantes em que você já trabalhou durante sua carreira. Quais foram os maiores desafios?**

A escola nos ensina muitas maneiras de achar respostas. Em cada problema interessante em que trabalhei, o desafio tem sido achar a pergunta certa. Quando Mike Karels e eu começamos a examinar o congestionamento do TCP, gastamos meses encarando o protocolo e os *traces* de pacotes, perguntando “por que ele está falhando”? Um dia, no escritório de Mike, um de nós disse: “O motivo pelo qual não consigo descobrir por que ele falha é porque não entendo como ele chegou a funcionar, para começar”. Aquela foi a pergunta certa e nos forçou a compreender a “temporização dos reconhecimentos (ACK)” que faz o TCP funcionar. Depois disso, o resto foi fácil.

**De modo geral, qual é o futuro que você imagina para as redes e a Internet?**

Para a maioria das pessoas, a Web é a Internet. Nós, que trabalhamos com redes, sorrimos educadamente, pois sabemos que a Web é uma aplicação rodando sobre a Internet, mas, e se eles estiverem certos? A Internet trata de permitir conversações entre pares de hospedeiros. A Web trata da produção e do consumo de informações distribuídas. “Propagação de informações” é uma visão muito geral da comunicação, da qual a “conversa em pares” é um minúsculo subconjunto. Precisamos pensar além do que vemos. As redes de hoje lidam com a mídia de *broadcast* (rádios, PONs, etc.) fingindo que ela é um fio de ponto a ponto. Isso é tremendamente ineficaz. *Terabits* de dados por segundo estão sendo trocados pelo mundo inteiro por meio de *pendrives* ou *smartphones*, mas não sabemos como tratar isso como “rede”. Os ISPs estão ocupados montando *caches* e CDNs para distribuir vídeo e áudio em escala. O *caching* é uma parte necessária da solução, mas não há uma parte das redes de hoje – desde Informações, Filas ou Teoria de Tráfego até as especificações de protocolos da Internet – que nos diga como projetá-lo e distribuí-lo. Acho e espero que, nos próximos anos, as redes evoluam para abranger a visão muito maior de comunicação, que é a base da Web.

**Quais pessoas o inspiraram profissionalmente?**

Quando eu cursava a faculdade, Richard Feynman nos visitou e deu um seminário acadêmico. Ele falou sobre uma parte da teoria quântica que eu estava lutando para entender durante todo o semestre, e sua explicação foi tão simples e lúcida que aquilo que parecia um lixo sem sentido para mim tornou-se óbvio e inevitável. Essa capacidade de ver e transmitir a simplicidade que está por trás do nosso mundo complexo me parece uma dádiva rara e maravilhosa.

Quais são suas recomendações para estudantes que desejam seguir carreira em computação e tecnologia da informação?

Este é um campo maravilhoso – computadores e redes provavelmente tiveram mais impacto sobre a sociedade do que qualquer invenção desde a imprensa. Redes conectam coisas, e seu estudo o ajuda a fazer conexões intelectuais: a busca de alimento das formigas e as danças das abelhas demonstram o assunto de projeto de protocolo melhor do que RFCs; engarrafamentos de trânsito ou pessoas saindo de um estádio lotado são a essência do congestionamento; e motoristas procurando o melhor caminho de volta para casa após uma tempestade que alagou a cidade constituem o núcleo do roteamento dinâmico. Se você estiver interessado em obter muito material e quiser causar impacto, é difícil imaginar um campo melhor do que este.

Esta página foi deixada em branco intencionalmente.