onde estava antes que o problema ocorresse. Como mencionamos anteriormente, a camada de transporte pode se recuperar de falhas na camada de rede, desde que cada extremidade da conexão tenha uma ideia do ponto em que está.

Esse problema nos leva à questão do que significa de fato a chamada confirmação fim a fim. Em princípio, o protocolo de transporte é fim a fim, pois não é encadeado como as camadas inferiores. Considere agora o caso de um usuário que solicita transações relativas a um banco de dados remoto. Suponha que a entidade de transporte remota esteja programada de modo a passar primeiro os segmentos para a camada imediatamente superior e só então enviar a confirmação. Até mesmo nesse caso, o fato de uma confirmação ter sido recebida na máquina do usuário não quer dizer necessariamente que o host remoto funcionou por tempo suficiente para atualizar o banco de dados. Uma confirmação fim a fim verdadeira, cujo recebimento indica que o trabalho foi realmente realizado e cuja falta indica que ele não foi cumprido, talvez seja algo impossível de alcançar. Esse assunto é discutido com mais detalhes por Saltzer et al. (1984).

6.3 Controle de congestionamento

Se as entidades de transporte em muitas máquinas enviarem muitos pacotes para a rede com muita rapidez, a rede ficará congestionada, com o desempenho degradado enquanto os pacotes são atrasados e perdidos. Controlar o congestionamento para evitar esse problema é responsabilidade conjunta das camadas de rede e transporte. O congestionamento ocorre nos roteadores, de modo que é detectado na camada de rede. Porém, o congestionamento por fim é causado pelo tráfego enviado para a rede pela camada de transporte. O único modo eficaz de controlar o congestionamento é fazer com que os protocolos de transporte enviem pacotes mais lentamente para a rede.

No Capítulo 5, estudamos os mecanismos de controle de congestionamento na camada de rede. Nesta seção, estudaremos a outra metade do problema, os mecanismos de controle de congestionamento na camada de transporte. Depois de descrever os objetivos do controle de congestionamento, descreveremos como os hosts podem regular a taxa com que enviam pacotes para a rede. A Internet conta bastante com a camada de transporte para o controle de congestionamento, e algoritmos específicos são elaborados para TCP e outros protocolos.

6.3.1 Alocação desejável de largura de banda

Antes de descrevermos como regular o tráfego, temos de entender o que estamos tentando alcançar executando um algoritmo de controle de congestionamento. Ou seja, temos de especificar o estado em que um bom algoritmo de controle de congestionamento operará na rede. O objetivo é mais do que simplesmente evitar o congestionamento. É encontrar uma boa alocação de largura de banda para as entidades de transporte que estão usando a rede. Uma boa alocação oferecerá bom desempenho, pois usa toda a largura de banda disponível mas evita congestionamento, será justa entre entidades de transporte concorrentes e rastreará rapidamente as mudanças nas demandas de tráfego. Vamos esclarecer cada um desses critérios por vez.

Eficiência e potência

Uma alocação eficiente de largura de banda por entidades de transporte usará toda a capacidade da rede que se encontra disponível. Porém, não é muito certo pensar que, se existe um enlace de 100 Mbps, cinco entidades de transporte deverão receber 20 Mbps cada uma. Elas normalmente deverão receber menos de 20 Mbps para que tenham um bom desempenho. O motivo é que o tráfego normalmente é feito por rajada. Lembre-se de que, na Seção 5.3, descrevemos o **goodput** (ou vazão normalizada, a taxa de pacotes úteis que chegam ao receptor) como uma função da carga oferecida. Essa curva e uma curva correspondente para o atraso como uma função da carga oferecida são apresentadas na Figura 6.16.

À medida que a carga aumenta na Figura 6.16(a), o goodput inicialmente aumenta na mesma velocidade, mas quando a carga se aproxima da capacidade, o goodput aumenta mais gradualmente. Isso ocorre porque as rajadas de tráfego ocasionalmente podem se acumular e causar mais perdas nos buffers dentro da rede. Se o protocolo de transporte for mal projetado e retransmitir pacotes que foram atrasados mas não perdidos, a rede pode entrar em colapso de congestionamento. Nesse estado, os transmissores estão furiosamente enviando pacotes, mas cada vez menos trabalho útil está sendo realizado.

O atraso correspondente é dado na Figura 6.16(b). Inicialmente, o atraso é fixo, representando o atraso de propagação pela rede. À medida que a carga se aproxima da capacidade, o atraso aumenta, lentamente a princípio e depois muito mais rapidamente. Isso novamente é por causa do tráfego que tende a se acumular em carga alta. O atraso não pode realmente ir até o infinito, exceto em um modelo em que os roteadores possuem buffers infinitos. Em vez disso, os pacotes serão perdidos após experimentarem um atraso máximo de buffering.

Para o goodput e o atraso, o desempenho começa a degradar no início do congestionamento. Intuitivamente, obteremos o melhor desempenho a partir da rede se alocarmos largura de banda até que o atraso comece a cair rapidamente. Esse ponto está abaixo da capacidade. Para identificá-lo, Kleinrock (1979) propôs a métrica da **potência**, onde

$$potência = \frac{carga}{atraso}$$

A potência inicialmente aumentará com a carga oferecida, pois o atraso continua sendo pequeno e relativamente constante, mas alcançará um máximo e cairá à medida

conexões abertas. Cada cliente pode estar em uma das seguintes situações: um segmento pendente, *S1*, ou nenhum segmento pendente, *S0*. Com base apenas nessas informações de estado, o cliente tem de decidir se deve ou não retransmitir o segmento mais recente.

À primeira vista, parece óbvio que o cliente só deve retransmitir se houver um segmento não confirmado pendente (isto é, se ele se encontrar no estado *S1*) durante a queda do servidor. Contudo, uma análise mais detalhada revela as dificuldades dessa abordagem simplista. Por exemplo, considere a situação na qual a entidade de transporte do servidor primeiro envia uma confirmação e depois, quando a confirmação tiver sido enviada, efetua uma gravação no processo de aplicação. Gravar um segmento no fluxo de saída e enviar uma confirmação são dois eventos distintos e indivisíveis que não podem ser realizados simultaneamente. Se ocorrer uma falha após o envio da confirmação, mas antes de a gravação ser feita, o cliente receberá a confirmação e assim ficará no estado *S0* quando chegar o anúncio de que o funcionamento foi recuperado. Consequentemente, o cliente não retransmitirá, porque imagina (incorretamente) que o segmento chegou. Essa decisão do cliente ocasiona a perda de um segmento.

A essa altura, você deve estar pensando: 'É fácil resolver esse problema. Basta reprogramar a entidade de transporte para gravar primeiro o segmento e depois enviar a confirmação'. Tente outra vez. Imagine que a gravação foi feita mas a falha do servidor ocorreu antes de ser possível enviar a confirmação. O cliente estará no estado *S1* e, portanto, retransmitirá, acarretando uma duplicata do segmento não detectada no fluxo de saída para o processo de aplicação do servidor.

Independentemente da forma como o cliente e o servidor são programados, sempre haverá situações em que o protocolo não recuperará o funcionamento de modo apropriado. O servidor poderá ser programado de duas maneiras: para confirmar primeiro ou para gravar primeiro. O cliente poderá ser programado de quatro formas: para sempre retransmitir o último segmento, para nunca retransmitir o último segmento, para retransmitir somente no estado *S0* ou para retransmitir somente no estado *S1*. Isso nos dá oito combinações, mas, como veremos, para cada combinação existe algum conjunto de eventos que faz o protocolo falhar.

Há três eventos possíveis no servidor: enviar uma confirmação (*A*), gravar no processo de saída (*W*) e sofrer uma pane (*C*). Os três eventos podem ocorrer em seis ordens distintas: *AC*(*W*), *AWC*, *C*(*AW*), *C*(*WA*), *WAC* e *WC*(*A*), onde os parênteses são usados para indicar que nem *A* nem *W* podem seguir *C* (ou seja, não há nenhum evento após uma pane). A Figura 6.15 mostra todas as oito combinações de estratégias do cliente e do servidor, e as sequências de eventos válidas para cada uma delas. Observe que, para cada estratégia, existe alguma sequência de eventos que resulta na falha do protocolo. Por exemplo, se o cliente sempre retransmitir, o evento *AWC* gerará uma duplicata não detectada, mesmo que os dois outros eventos funcionem perfeitamente.

Tornar o protocolo mais elaborado também não ajuda muito. Ainda que o cliente e o servidor troquem vários segmentos antes de o servidor tentar gravar, de forma que o cliente saiba exatamente o que está para acontecer, o cliente não terá meios para saber se ocorreu uma pane imediatamente antes ou imediatamente após a gravação. A conclusão é inevitável: sob nossas regras básicas de não haver eventos simultâneos — ou seja, eventos separados acontecem um após o outro, e não ao mesmo tempo —, a queda e a recuperação do host não podem ser realizadas de forma transparente para as camadas mais altas.

Em termos mais genéricos, esse resultado pode ser reafirmado como o fato de que a recuperação de uma queda da camada N só pode ser feita pela camada $N + 1$, e mesmo assim somente se a camada mais elevada mantiver um volume suficiente de informações sobre o *status* para reconstruir

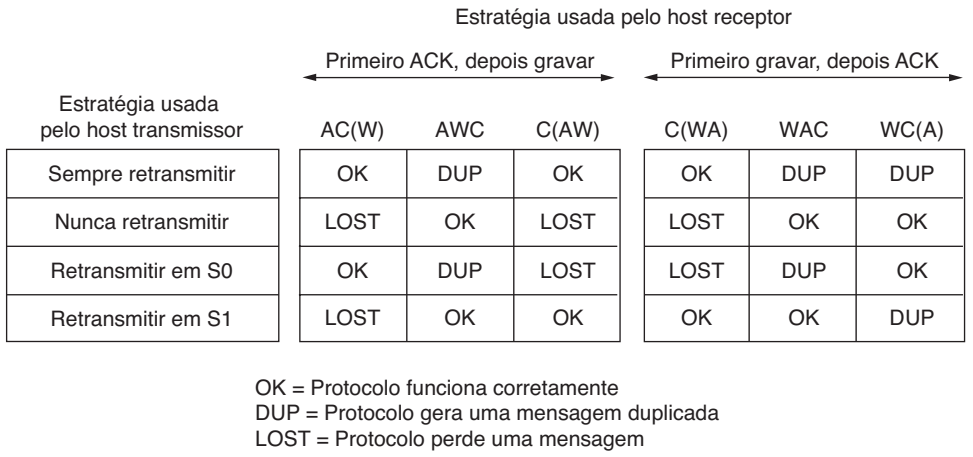

Figura 6.15 | Diferentes combinações de estratégias do cliente e do servidor.

le de congestionamento. Se a rede puder administrar *c* segmentos/s e o tempo de ciclo (incluindo transmissão, propagação, enfileiramento, processamento no receptor e retorno da confirmação) for *r*, a janela do transmissor deverá ser *cr*. Com uma janela desse tamanho, o transmissor normalmente opera com toda a capacidade do pipeline. Qualquer pequena diminuição no desempenho da rede bloqueará o fluxo. Como a capacidade de rede disponível para qualquer fluxo varia com o tempo, o tamanho da janela deverá ser ajustado com frequência, para acompanhar as mudanças na capacidade de transporte. Como veremos mais adiante, o TCP usa um esquema semelhante.

6.2.5 Multiplexação

A multiplexação, ou compartilhamento de várias conversações em conexões, circuitos virtuais e enlaces físicos, tem um papel importante em diversas camadas da arquitetura de rede. Na camada de transporte, a necessidade da multiplexação pode surgir de diversas formas. Por exemplo, se houver apenas um endereço de rede disponível em um host, todas as conexões de transporte nessa máquina terão de utilizá-lo. Ao chegar um segmento, é necessário encontrar algum meio de descobrir a qual processo ele deve ser entregue. Essa situação, denominada **multiplexação**, é ilustrada na Figura 6.14(a). Nessa figura, quatro conexões de transporte distintas utilizam a mesma conexão de rede (por exemplo, um endereço IP) para o host remoto.

A multiplexação também pode ser útil na camada de transporte por outra razão. Por exemplo, suponha que um host tenha vários caminhos de rede que ele possa usar. Se um usuário necessitar de maior largura de banda ou mais confiabilidade do que um dos caminhos de rede pode fornecer, uma saída será uma conexão que distribua o tráfego entre vários caminhos de rede em rodízio, como indica a Figura 6.14(b). Esse modo de operação é chamado **multiplexação inversa**. Com *k* conexões de rede abertas, a largura de banda efetiva é aumentada *k* vezes. Um exemplo comum de multiplexação inversa é o **SCTP** (**Stream Control Transmission Protocol**), que pode trabalhar com uma conexão usando várias interfaces de rede. Ao contrário, o TCP utiliza uma única extremidade de rede. A multiplexação inversa também é encontrada na camada de enlace, quando vários enlaces de baixa velocidade são usados em paralelo como um enlace de alta velocidade.

6.2.6 Recuperação de falhas

Se os hosts e roteadores estiverem sujeitos a interrupções em seu funcionamento ou se as conexões tiverem longa vida (por exemplo, download grande de software ou multimídia), a recuperação dessas panes se tornará uma questão importante. Se a entidade de transporte estiver inteiramente contida nos hosts, a recuperação de falhas na rede ou no roteador será simples. As entidades de transporte esperam segmentos perdidos o tempo todo e sabem como lidar com eles usando retransmissões.

Um problema mais complexo é como recuperar o funcionamento depois de uma pane no host. Em particular, talvez seja preferível que os clientes possam continuar funcionando quando os servidores falharem e forem rapidamente reiniciados em seguida. Para ilustrar a dificuldade, vamos supor que um host, o cliente, esteja enviando um arquivo muito grande a outro host, o servidor de arquivos, utilizando um protocolo stop-and-wait simples. A camada de transporte do servidor simplesmente passa os segmentos que chegam para o usuário de transporte, um a um. No meio da transmissão, o servidor sai do ar. Quando ele volta a funcionar, suas tabelas são reiniciadas, e ele não sabe mais exatamente onde estava.

Na tentativa de recuperar seu estado anterior, o servidor pode transmitir um segmento por broadcast a todos os outros hosts, comunicando seu problema e solicitando que seus clientes o informem sobre o status de todas as

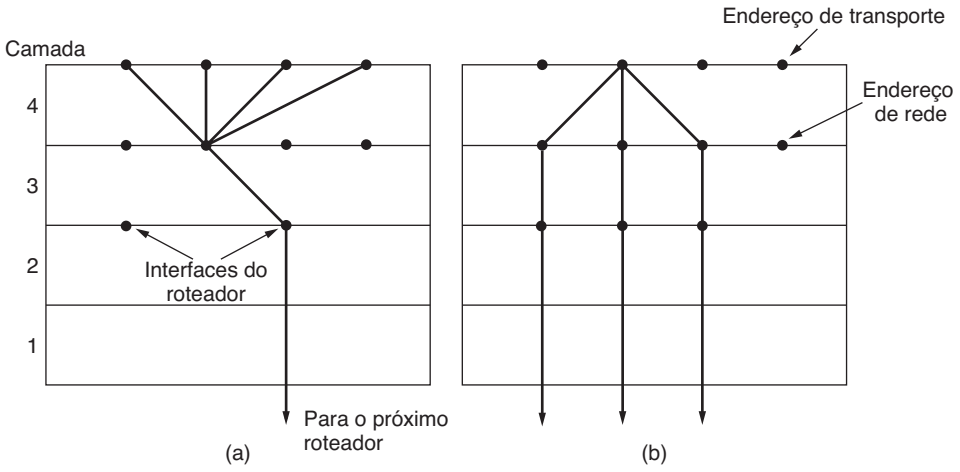

Figura 6.14 | (a) Multiplexação. (b) Multiplexação inversa.

guinte ao de número 1 (isto é, os segmentos 2, 3 e 4). *A* sabe que já enviou o segmento de número 2 e assim imagina que pode enviar os segmentos 3 e 4, o que acaba por fazer. Nesse ponto, ele é bloqueado e deve aguardar a alocação de mais buffers. Entretanto, pode haver retransmissões (linha 9) induzidas por timeouts durante o bloqueio, pois elas utilizam buffers que já haviam sido alocados. Na linha 10, *B* confirma a recepção de todos os segmentos até 4, inclusive, mas se recusa a permitir que *A* continue. Essa situação é impossível com os protocolos de janela fixa do Capítulo 3. O próximo segmento de *B* para *A* aloca outro buffer e permite que A continue. Isso acontecerá quando *B* tem espaço em buffer, provavelmente porque o usuário do transporte aceitou mais segmentos de dados.

Podem surgir problemas potenciais com esquemas de alocação de buffers desse tipo em redes de datagramas, caso ocorra a perda de segmentos de controle — e eles certamente surgem. Observe a linha 16. *B* já alocou mais buffers para *A*, mas o segmento da alocação foi perdido. Como os segmentos de controle não respeitam uma sequência nem sofrem timeout, *A* está em um impasse. Para evitar que isso aconteça, cada host deve enviar periodicamente segmentos de controle informando o status dos buffers e das confirmações em cada conexão. Dessa forma, o impasse será resolvido mais cedo ou mais tarde.

Até agora, partimos da suposição tácita de que o único limite imposto sobre a taxa de dados do transmissor é o espaço em buffer disponível no receptor. Isso normalmente não acontece. A memória já foi muito cara, mas os preços caíram bastante. Os hosts podem ser equipados com memória suficiente, de modo que a falta de buffers raramente ou nunca será um problema, até mesmo para conexões remotas. É claro que isso depende do tamanho do buffer sendo definido como grande o suficiente, o que nem sempre aconteceu no caso do TCP (Zhang et al., 2002).

Quando o espaço em buffer deixar de limitar o fluxo máximo, surgirá outro gargalo: a capacidade de transporte da rede. Se os roteadores adjacentes puderem trocar dados em uma taxa de no máximo *x* pacotes/s, e se houver *k* caminhos disjuntos entre um par de hosts, esses hosts não poderão trocar mais de *kx* segmentos/s, independentemente do espaço em buffer disponível em cada extremidade da conexão. Se o transmissor forçar muito a transmissão (ou seja, enviar mais de *kx* segmentos/segundo), a rede ficará congestionada, pois não será capaz de entregar os segmentos na mesma taxa em que eles chegam.

É necessário um mecanismo que limite as transmissões com base na capacidade de transporte da rede, em vez da capacidade dos buffers do receptor. Belsnes (1975) propôs o uso de um esquema de controle de fluxo com uma janela deslizante, no qual o transmissor ajusta dinamicamente o tamanho da janela à capacidade de transporte da rede. Isso significa que o tamanho da janela deslizante pode implementar controle de fluxo e contro-

	A	Mensagem	B	Comentários
1	→	<request 8 buffers>	→	A deseja 8 buffers
2	←	<ack = 15, buf = 4>	←	B concede mensagens 0-3 apenas
3	→	<seq = 0, data = m0>	→	A tem 3 buffers restantes agora
4	→	<seq = 1, data = m1>	→	A tem 2 buffers restantes agora
5	→	<seq = 2, data = m2>	•••	Mens. perdida, mas A acha que tem 1 restante
6	←	<ack = 1, buf = 3>	←	B confirma 0 e 1, permite 2-4
7	→	<seq = 3, data = m3>	→	A tem 1 buffer restante
8	→	<seq = 4, data = m4>	→	A tem 0 buffers restantes e deve parar
9	→	<seq = 2, data = m2>	→	A atinge o timeout e retransmite
10	←	<ack = 4, buf = 0>	←	Tudo confirmado, mas A ainda bloqueado
11	←	<ack = 4, buf = 1>	←	A pode agora enviar 5
12	←	<ack = 4, buf = 2>	←	B encontrou novo buffer em algum lugar
13	→	<seq = 5, data = m5>	→	A tem 1 buffer restante
14	→	<seq = 6, data = m6>	→	A agora está bloqueado novamente
15	←	<ack = 6, data = m6>	←	A ainda está bloqueado
16	•••	<ack = 6, buf = 4>	←	Impasse em potencial

Figura 6.13 | Alocação dinâmica de buffers. As setas mostram o sentido da transmissão. As reticências (...) indicam a perda de um segmento.

Ainda resta a questão de como organizar o pool de buffers. Se a maioria dos segmentos for quase do mesmo tamanho, é natural organizar os buffers como um pool de buffers de tamanho idêntico, com um segmento por buffer, como na Figura 6.12(a). Porém, se houver uma grande variação no tamanho do segmento, de solicitações curtas para páginas Web e pacotes grandes em transferências de arquivo peer-to-peer, um pool de buffers de tamanho fixo apresentará problemas. Se o tamanho do buffer for escolhido para ser igual ao maior tamanho de segmento possível, o espaço será desperdiçado sempre que um segmento curto chegar. Se o tamanho do buffer for escolhido para ser menor que o tamanho máximo do segmento, vários buffers serão necessários para segmentos longos, com a complexidade criada nesse caso.

Outra abordagem para o problema do tamanho dos buffers é usar buffers de tamanho variável, como mostra a Figura 6.12(b). A vantagem nesse caso é a melhor utilização de memória, à custa de um gerenciamento de buffers mais complicado. Uma terceira possibilidade é dedicar um grande buffer reservado a cada conexão, como ilustra a Figura 6.12(c). Esse sistema é simples e elegante, e não depende do tamanho dos segmentos, mas só faz um bom uso da memória quando todas as conexões têm uma carga muito pesada.

À medida que as conexões são abertas e fechadas e que o padrão de tráfego se altera, o transmissor e o receptor precisam ajustar dinamicamente suas alocações de buffers. Consequentemente, o protocolo de transporte deve permitir que o host transmissor solicite espaço em buffer na outra extremidade da conexão. Os buffers poderiam ser alocados por conexão ou coletivamente, para todas as conexões em execução entre os dois hosts. Em contrapartida, o receptor, conhecendo a situação de seus buffers (mas sem conhecer o tráfego oferecido), poderia dizer ao transmissor: 'Tenho *X* buffers reservados para você'. Se o número de conexões abertas aumentar, talvez seja necessário reduzir uma alocação de buffer; portanto, o protocolo deve oferecer essa possibilidade.

Um modo razoavelmente genérico de gerenciar a alocação dinâmica de buffers é desvincular o gerenciamento dos buffers das confirmações, ao contrário do que ocorre com os protocolos de janela deslizante do Capítulo 3. Na verdade, o gerenciamento dinâmico de buffers significa usar uma janela de tamanho variável. Inicialmente, o transmissor solicita determinado número de buffers, com base em suas necessidades. Em seguida, de acordo com o número solicitado, o receptor oferece todos os buffers de que dispõe. Sempre que enviar um segmento, o transmissor deverá decrementar sua alocação, parando por completo quando a alocação chegar a zero. Em seguida, o receptor transmite (por piggyback) as confirmações e as alocações de buffers no tráfego reverso. O TCP usa esse esquema, transportando alocações de buffer em um campo do cabeçalho chamado *Tamanho de janela*.

A Figura 6.13 mostra um exemplo de como o gerenciamento dinâmico de janelas poderia funcionar em uma rede de datagramas com números de sequência de 4 bits. Nesse exemplo, os dados fluem em segmentos do host *A* para o host *B* e as confirmações e alocações de buffers fluem em segmentos no sentido contrário. Inicialmente, *A* quer oito buffers, dos quais só recebe quatro. Em seguida, *A* envia três segmentos, sendo que o terceiro é perdido. O segmento 6 confirma a recepção de todos os segmentos até o número de sequência 1, inclusive, permitindo que *A* libere esses buffers, e informa a *A* que ele tem permissão para enviar mais três segmentos, começando pelo segmento se-

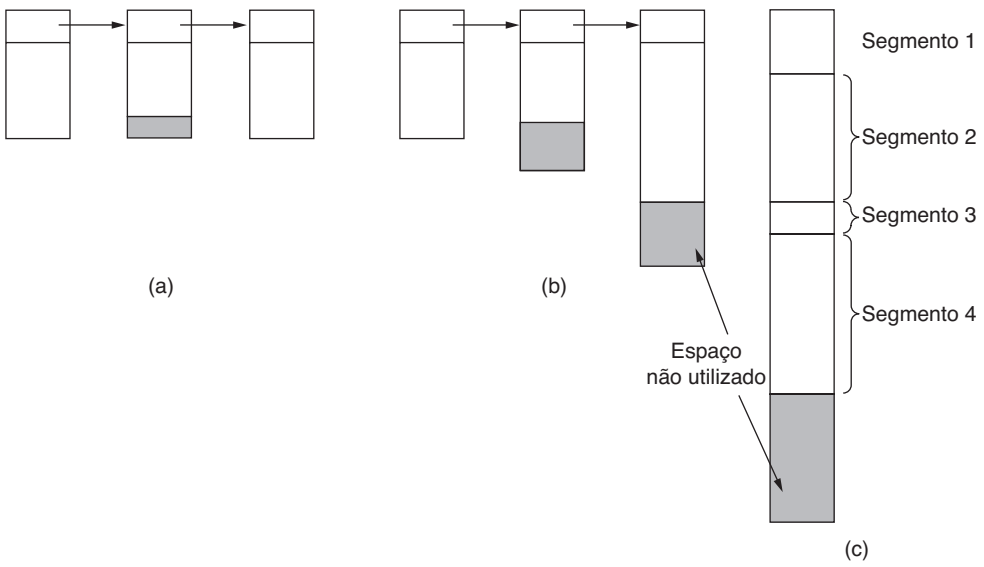

Figura 6.12 | (a) Buffers de tamanho fixo encadeados. (b) Buffers de tamanho variável encadeados. (c) Um grande buffer reservado por conexão.

6.5.8 Janela deslizante do TCP

Como mencionamos antes, o gerenciamento de janelas no TCP desvincula as questões de confirmação do recebimento correto dos segmentos da alocação de buffer pelo receptor. Por exemplo, suponha que o receptor tenha um buffer de 4.096 bytes, como ilustra a Figura 6.34. Se o transmissor enviar um segmento de 2.048 bytes e este for recebido de forma correta, o receptor confirmará o segmento. Porém, como agora ele só tem 2.048 bytes de espaço disponível em seu buffer (até que alguma aplicação retire alguns dados do buffer), o receptor anunciará uma janela de 2.048 bytes começando no próximo byte esperado.

Agora, o transmissor envia outros 2.048 bytes, que são confirmados, mas a janela anunciada tem tamanho 0. O transmissor deve parar até que o processo da aplicação no host receptor tenha removido alguns dados do buffer, quando o TCP poderá anunciar uma janela maior e mais dados poderão ser enviados.

Quando a janela é 0, o transmissor não pode enviar segmentos da forma como faria sob condições normais, mas há duas exceções. Primeira, os dados urgentes podem ser enviados para, por exemplo, permitir que o usuário encerre o processo executado na máquina remota. Segunda, o transmissor pode enviar um segmento de 1 byte para fazer com que o receptor anuncie novamente o próximo byte esperado e o tamanho da janela. Esse pacote é chamado **window probe**. O padrão TCP oferece essa opção de forma explícita para evitar um impasse no caso de uma atualização da janela se perder.

Os transmissores não são obrigados a enviar os dados assim que os recebem da aplicação. Nem os receptores têm a obrigação de enviar as confirmações imediatamente. Por exemplo, na Figura 6.34, quando os primeiros 2 KB de dados chegaram, o TCP, sabendo que havia uma janela de 4 KB disponível, estaria completamente correto se apenas armazenasse os dados no buffer até chegarem outros 2 KB, a fim de poder transmitir um segmento com 4 KB de carga útil. Essa liberdade pode ser explorada para melhorar o desempenho das conexões.

Considere uma conexão para um terminal remoto, por exemplo, usando SSH ou Telnet, que reage a cada tecla pressionada. Na pior das hipóteses, quando um caractere chegar à entidade TCP receptora, o TCP cria um segmento TCP de 21 bytes, que será repassado ao IP para ser enviado como um datagrama IP de 41 bytes. No lado receptor, o TCP envia imediatamente uma confirmação de 40 bytes (20 bytes de cabeçalho TCP e 20 bytes de cabeçalho IP). Mais tarde, quando o terminal remoto tiver lido o byte, o

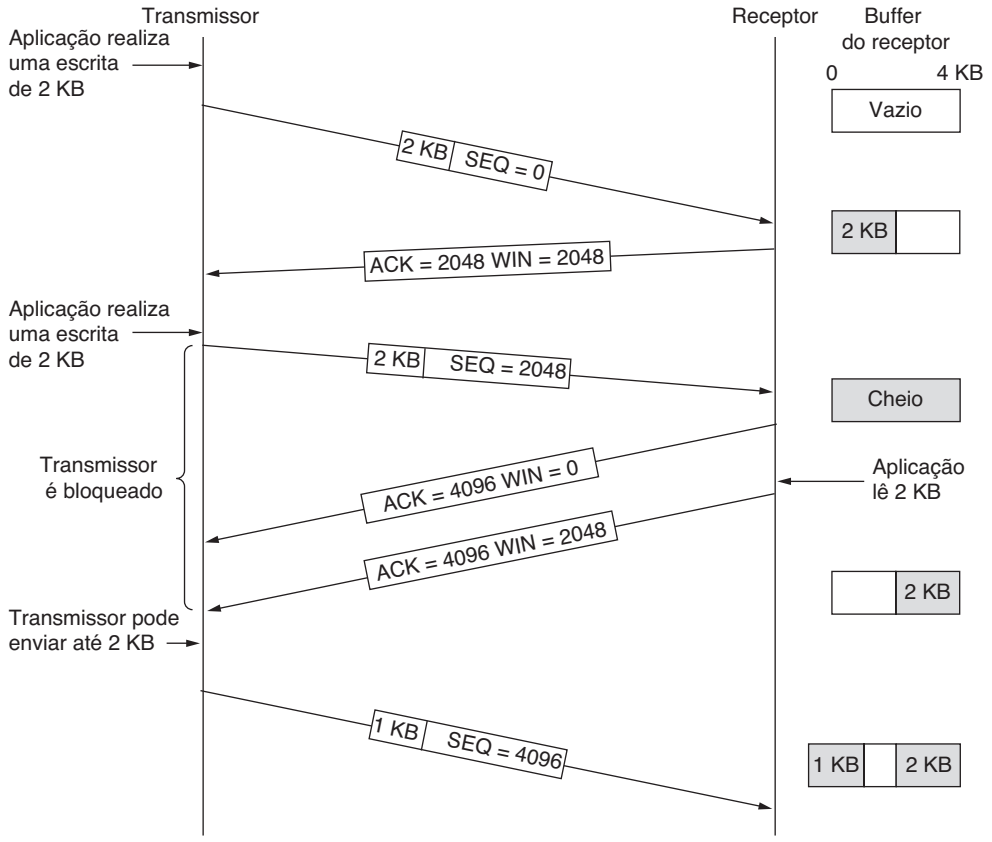

Figura 6.34 | Gerenciamento de janelas no TCP.

TCP enviará uma atualização da janela, movendo a janela um byte para a direita. Esse pacote também tem 40 bytes. Por último, quando o terminal tiver processado o caractere, ele o ecoará para exibição local usando um pacote de 41 bytes. No total, 162 bytes de largura de banda são utilizados e quatro segmentos são enviados para cada caractere digitado. Quando a largura de banda é escassa, esse método não se mostra uma boa opção.

Uma abordagem usada por muitas implementações do TCP para otimizar essa situação é chamada **confirmações adiadas.** A ideia é retardar as confirmações e atualizações de janelas durante 500 ms, na esperança de encontrar algum dado que lhes dê 'uma carona'. Supondo que o terminal ecoe dentro de 500 ms, apenas um pacote de 41 bytes precisará ser retornado ao usuário remoto, reduzindo à metade a contagem de pacotes e o uso da largura de banda.

Embora essa regra reduza a carga imposta à rede pelo receptor, o transmissor ainda estará operando de modo ineficiente, enviando pacotes de 41 bytes que contêm apenas um byte de dados. Uma forma de reduzir esse uso é conhecida como **algoritmo de Nagle** (Nagle, 1984). A sugestão de Nagle é simples: quando os dados chegarem ao transmissor em pequenas partes, basta enviar o primeiro byte e armazenar no buffer todos os outros, até que o byte pendente tenha sido confirmado. Em seguida, envie todos os caracteres armazenados no buffer em um único segmento TCP e comece a armazenar no buffer novamente, até que o próximo segmento seja confirmado. Ou seja, somente um pacote pequeno pode estar pendente a qualquer momento. Se muitas partes dos dados forem enviadas pela aplicação em um tempo de ida e volta, o algoritmo de Nagle colocará as muitas partes em um segmento, reduzindo bastante a largura de banda utilizada. O algoritmo diz ainda que um novo segmento deveria ser enviado se dados bastantes completassem um segmento máximo.

O algoritmo de Nagle é muito utilizado por implementações TCP, mas há ocasiões em que é melhor desativá-lo. Em particular, em jogos interativos executados na Internet, os jogadores normalmente desejam um fluxo rápido de pequenos pacotes de atualização. Reunir as atualizações para enviá-las em rajadas faz com que o jogo responda indevidamente, o que deixa os usuários insatisfeitos. Um problema mais sutil é que o algoritmo de Nagle às vezes pode interagir com confirmações adiadas e causar um impasse temporário: o receptor espera por dados nos quais pode enviar uma confirmação de carona, e o transmissor espera a confirmação para enviar mais dados. Essa interação pode adiar os downloads de páginas Web. Devido a esses problemas, o algoritmo de Nagle pode ser desativado (uma opção chamada *TCP_NODELAY*). Mogul e Minshall (2001) discutem essa e outras soluções.

Outro problema que pode arruinar o desempenho do TCP é a **síndrome do janelamento inútil** (Clark, 1982). Esse problema ocorre quando os dados são passados para a entidade TCP transmissora em grandes blocos, mas uma aplicação interativa no lado receptor lê os dados apenas um byte por vez. Para entender o problema, observe a Figura 6.35. Inicialmente, o buffer TCP no lado receptor está cheio (ou seja, ele tem uma janela de tamanho 0) e o transmissor sabe disso. Em seguida, uma aplicação interativa lê um caractere do fluxo TCP. Essa ação faz com que o TCP receptor fique satisfeito e envie uma atualização da janela ao transmissor, informando que ele pode enviar 1 byte. O transmissor agradece e envia 1 byte. Agora, o buffer se enche outra vez; portanto, o receptor confirma o segmento de 1 byte e atribui o valor 0 ao tamanho da janela. Esse comportamento pode durar para sempre.

A solução apresentada por Clark é evitar que o receptor envie uma atualização da janela de 1 byte. Em vez disso, ele é forçado a aguardar até que haja um espaço considerável na janela para então anunciar o fato. Para ser mais específico, o receptor não deve enviar uma atualização da janela até que possa lidar com o tamanho máximo do segmento que anunciou quando a conexão foi estabelecida, ou até que seu buffer esteja com metade de sua capacidade livre; o que for menor. Além disso, o transmissor também pode ajudar não enviando segmentos muito pequenos. Em vez disso, deve tentar aguardar até que possa enviar um segmento inteiro ou pelo menos um que contenha dados equivalentes à metade da capacidade do buffer no lado receptor.

O algoritmo de Nagle e a solução de Clark para a síndrome do janelamento inútil são complementares. Nagle tentava resolver o problema causado pelo fato de a aplicação transmissora entregar dados ao TCP um byte por vez. Já Clark tentava acabar com o problema criado pelo fato de a aplicação receptora retirar os dados do TCP um byte por vez. Ambas as soluções são válidas e podem funcionar juntas. O objetivo é evitar que o transmissor envie segmentos pequenos e que o receptor tenha de solicitá-los.

O TCP receptor pode fazer mais para melhorar o desempenho da rede do que apenas realizar a atualização das janelas em unidades grandes. Assim como o TCP transmissor, ele também tem a capacidade de armazenar dados em buffer; portanto, pode bloquear uma solicitação READ da aplicação até ter um bom volume de dados a oferecer. Isso reduz o número de chamadas ao TCP (e também o overhead). Isso também aumenta o tempo de resposta; no entanto, para aplicações não interativas, como a transferência de arquivos, a eficiência pode ser mais importante que o tempo de resposta a solicitações individuais.

Outro problema para o receptor é o que fazer com os segmentos fora de ordem. O receptor armazenará os dados em buffer até que possam ser passados para a aplicação em ordem. Na realidade, nada de mau aconteceria se os segmentos fora de ordem fossem descartados, pois eles por fim seriam retransmitidos pelo transmissor, mas isso seria um desperdício.

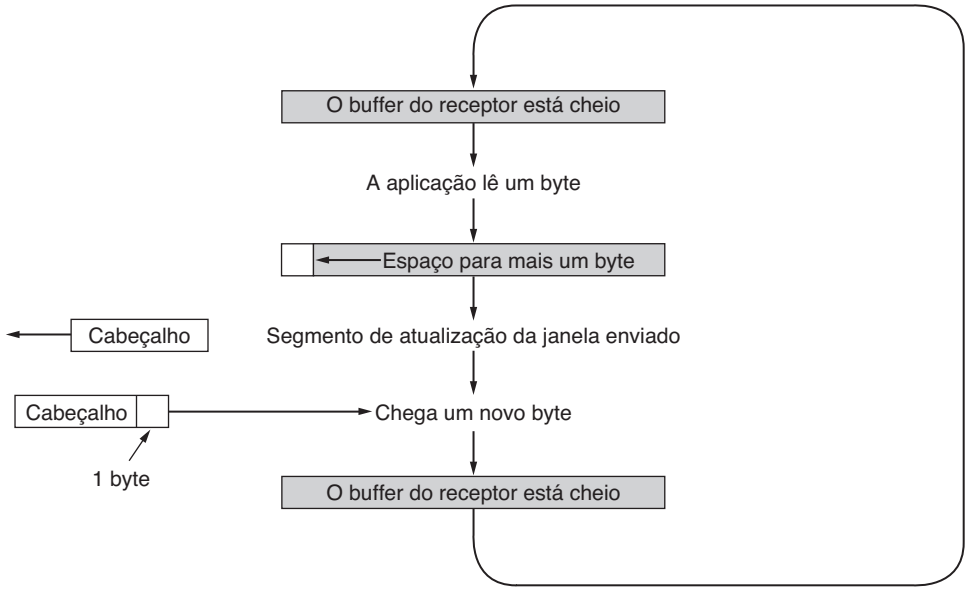

Figura 6.35 | Síndrome do janelamento inútil.

As confirmações só podem ser enviadas quando todos os dados até o byte confirmado tiverem sido recebidos. Isso é chamado **confirmação acumulativa**. Se o receptor receber os segmentos 0, 1, 2, 4, 5, 6 e 7, ele poderá confirmar tudo até o último byte do segmento 2, inclusive. Quando o transmissor sofrer um timeout, ele retransmitirá o segmento 3. Se tiver armazenado os segmentos de 4 a 7, o receptor poderá, ao receber o segmento 3, confirmar todos os bytes até o fim do segmento 7.

6.5.9 Gerenciamento de contadores do TCP

O TCP utiliza vários timers (pelo menos conceitualmente) para realizar seu trabalho. O mais importante deles é o timer de retransmissão, ou **RTO** (**Retransmission Timeout**). Quando um segmento é enviado, um timer de retransmissão é iniciado. Se o segmento for confirmado antes de o timer expirar, ele será interrompido. Por outro lado, se o timer expirar antes de a confirmação chegar, o segmento será retransmitido (e o timer disparado mais uma vez). Com isso, surge a seguinte pergunta: Qual deve ser esse período de tempo?

Esse problema é muito mais difícil na camada de transporte do que nos protocolos de enlace de dados, como o 802.11. Nesse último caso, o atraso esperado é medido em microssegundos e é altamente previsível (ou seja, tem pouca variância), de modo que o timer pode ser definido para expirar pouco depois da confirmação esperada, como mostra a Figura 6.36(a). Como as confirmações raramente são adiadas na camada de enlace de dados (devido à falta de congestionamento), a ausência de uma confirmação no momento esperado geralmente significa que ou o quadro ou a confirmação foram perdidos.

O TCP encontra um ambiente radicalmente distinto. A função densidade de probabilidade para o tempo gasto para uma confirmação TCP voltar é mais semelhante à Figura 6.36(b) do que à Figura 6.36(a). Ela é maior e mais variável. Determinar o tempo de ida e volta ao destino é complicado. Mesmo quando é conhecido, decidir sobre o intervalo de tempo também é difícil. Se o período de tempo for muito curto, digamos T_1 na Figura 6.36(b), haverá retransmissões desnecessárias, enchendo a Internet com pacotes inúteis. Se ele for muito longo (por exemplo, T_2), o desempenho sofrerá devido ao longo atraso de retransmissão sempre que o pacote se perder. Além do mais, a média e a variância da distribuição de chegada da confirmação podem mudar rapidamente dentro de alguns segundos, enquanto o congestionamento se acumula ou é resolvido.

A solução é usar um algoritmo dinâmico que adapte constantemente o intervalo de tempo, com base em medições contínuas do desempenho da rede. O algoritmo geralmente usado pelo TCP é atribuído a Jacobson (1988) e funciona da seguinte forma. Para cada conexão, o TCP mantém uma variável, *SRTT* (Smoothed Round-Trip Time), que é a melhor estimativa atual do tempo de ida e volta até o destino em questão. Quando um segmento é enviado, um timer é iniciado, tanto para ver quanto tempo a confirmação leva como também para disparar uma retransmissão se o tempo for muito longo. Se a confirmação retornar antes que o timer expire, o TCP mede o tempo gasto para

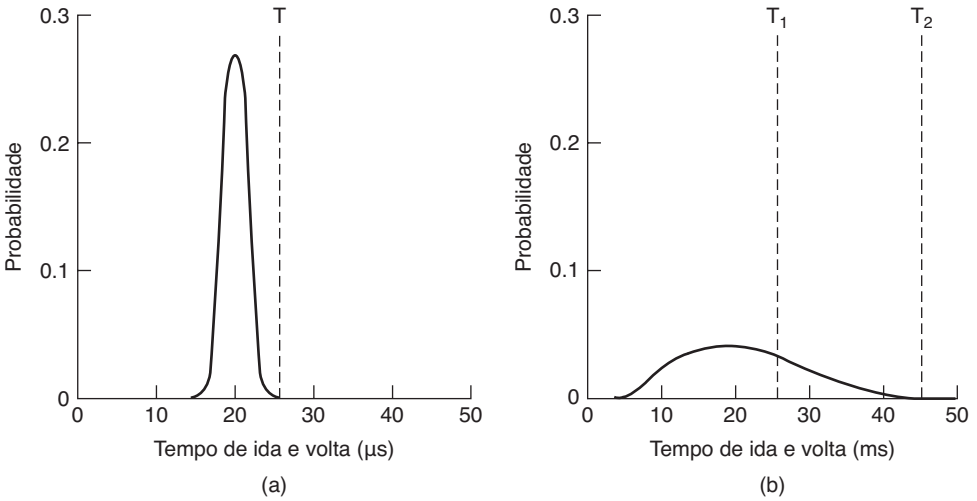

Figura 6.36 | (a) Densidade de probabilidade de tempos de chegada de confirmações na camada de enlace de dados. (b) Densidade de probabilidade de tempos de chegada de confirmações para o TCP.

a confirmação, digamos, *R*. Depois, ele atualiza o *SRTT* de acordo com a fórmula

$$\text{SRTT} = \alpha\,\text{SRTT} + (1 - \alpha)\,\text{R}$$

onde α é um fator de nivelamento que determina a rapidez com que os valores antigos são esquecidos. Normalmente, $\alpha = 7/8$. Esse tipo de fórmula é uma média móvel ponderada exponencialmente, ou **EWMA** (**Exponentially Weighted Moving Average**), ou filtro passa-baixa, que descarta o ruído nas amostras.

Mesmo com um bom valor de *SRTT*, escolher um período de tempo de retransmissão adequado é uma questão não trivial. As implementações iniciais do TCP usavam 2*xRTT*, mas a experiência mostrou que um valor constante era muito inflexível, pois deixava de responder quando a variância subia. Em particular, modelos de enfileiramento de tráfego aleatório (ou seja, de Poisson) preveem que, quando a carga se aproxima de sua capacidade, o atraso se torna grande e altamente variável. Isso pode fazer com que o timer de retransmissão dispare e uma cópia do pacote seja retransmitida, embora o pacote original ainda esteja transitando na rede. Isso é mais provável que aconteça sob condições de alta carga, que é o pior momento para enviar pacotes adicionais para a rede.

Para resolver esse problema, Jacobson propôs tornar o valor do período de tempo sensível à variância nos tempos de ida e volta, bem como o tempo de ida e volta nivelado. Essa mudança exige registrar outra variável nivelada, *RTTVAR* (Round-Trip Time VARiation), que é atualizada usando-se a fórmula:

$$\text{RTTVAR} = \beta\,\text{RTTVAR} + (1 - \beta)\ |\,\text{SRTT} - \text{R}\,|$$

Esta é uma EWMA, como antes, e normalmente $\beta = 3/4$. O período de tempo de retransmissão, *RTO*, é definido como

$$\text{RTO} = \text{SRTT} + 4 \times \text{RTTVAR}$$

A escolha do fator 4 é de certa forma arbitrária, mas a multiplicação por 4 pode ser feita em um único deslocamento, e menos de 1 por cento de todos os pacotes vem atrasado com um desvio-padrão maior que quatro. Observe que a *RTTVAR* não é exatamente o mesmo que o desvio-padrão (na realidade, é o desvio da média), mas na prática é bastante próxima. O artigo de Jacobson está repleto de truques inteligentes para calcular os períodos de tempo usando apenas somas, subtrações e deslocamentos de inteiros. Essa economia agora é necessária para os hosts modernos, mas se tornou parte da cultura que permite que o TCP funcione sobre todos os tipos de dispositivos, desde supercomputadores até pequenos dispositivos. Até agora, ninguém a incluiu em um chip de RFID, mas quem sabe algum dia?

Outros detalhes de como calcular esse período de tempo, incluindo os valores iniciais das variáveis, são dados na RFC 2988. O timer de retransmissão também é mantido em um mínimo de 1 segundo, independentemente das estimativas. Esse é um valor conservador, escolhido para impedir retransmissões falsas, com base nas medições (Allman e Paxson, 1999).

Um problema que ocorre com a coleta das amostras, *R*, do tempo de ida e volta, é o que fazer quando um segmento expira seu período de tempo e é enviado novamente. Quando a confirmação chega, não fica claro se ela se refere à primeira transmissão ou a alguma outra. A decisão errada pode contaminar seriamente o período de tempo de retransmissão. Phil Karn descobriu esse problema pelo modo mais difícil. Karn é um radioamador interessado em

transmitir pacotes TCP/IP via radioamador, um meio notoriamente não confiável. Ele fez uma proposta simples: não atualizar as estimativas sobre quaisquer segmentos que tiverem sido retransmitidos. Além disso, o período de tempo é dobrado a cada retransmissão bem-sucedida, até que os segmentos passem pela primeira vez. Esse reparo é chamado **algoritmo de Karn** (Karn e Partridge, 1987). A maioria das implementações do TCP o utiliza.

O timer de retransmissão não é o único timer que o TCP utiliza. Um segundo timer é o **timer de persistência**. Ele é projetado para impedir o impasse a seguir. O receptor envia uma confirmação com um tamanho de janela 0, dizendo ao transmissor para esperar. Mais tarde, o receptor atualiza a janela, mas o pacote com a atualização se perde. Agora, o transmissor e o receptor estão esperando que o outro faça algo. Quando o timer de persistência expira, o transmissor envia uma consulta ao receptor. A resposta dessa consulta indica o tamanho da janela. Se ainda for 0, o timer de persistência é definido novamente e o ciclo se repete. Se for diferente de zero, os dados agora podem ser enviados.

O terceiro timer que algumas implementações utilizam é o **timer keepalive**. Quando uma conexão estiver ociosa por muito tempo, o timer keepalive pode expirar e fazer com que um lado verifique se o outro lado ainda está lá. Se ele não responder, a conexão é encerrada. Esse recurso é discutível, pois aumenta o overhead e pode terminar uma conexão ativa devido a uma interrupção transitória da rede.

O último timer usado em cada conexão TCP é aquele usado no estado *TIME WAIT* durante o encerramento. Ele usa o dobro do tempo de vida máximo do pacote para garantir que, quando uma conexão for fechada, todos os pacotes criados por ela terão expirado.

6.5.10 Controle de congestionamento do TCP

Deixamos uma das principais funções do TCP para o final: o controle de congestionamento. Quando a carga oferecida a qualquer rede é maior que sua capacidade, acontece um congestionamento. A Internet não é exceção a essa regra. A camada de rede detecta o congestionamento quando as filas se tornam grandes nos roteadores e tenta gerenciá-lo, mesmo que apenas descartando pacotes. Cabe à camada de transporte receber o feedback do congestionamento da camada de rede e diminuir a velocidade do tráfego que está enviando para a rede. Na Internet, o TCP desempenha o principal papel no controle do congestionamento, bem como no transporte confiável. É por isso que esse protocolo é tão especial.

Abordamos a situação geral do controle de congestionamento na Seção 6.3. Um detalhe importante foi que um protocolo de transporte usando uma lei de controle AIMD (Additive Increase Multiplicative Decrease) em resposta aos sinais de congestionamento binários da rede convergiria para uma alocação de largura de banda imparcial e eficiente. O controle de congestionamento do TCP é baseado na implementação dessa técnica usando uma janela e com perda de pacotes como sinal binário. Para fazer isso, o TCP mantém uma **janela de congestionamento** cujo tamanho é o número de bytes que o transmissor pode ter na rede a qualquer momento. A velocidade correspondente é o tamanho da janela dividido pelo tempo de ida e volta da conexão. O TCP ajusta o tamanho da janela de acordo com a regra AIMD.

Lembre-se de que a janela de congestionamento é mantida *além* da janela de controle de fluxo, que especifica o número de bytes que o receptor pode armazenar em buffer. As duas janelas são acompanhadas em paralelo, e o número de bytes que podem ser enviados é a menor das duas janelas. Assim, a janela efetiva é o menor entre o que o transmissor pensa ser o correto e o que o receptor pensa ser o correto. Se um não quer, dois não brigam. O TCP deixará de enviar dados se a janela de congestionamento ou a janela de controle de fluxo estiverem temporariamente cheias. Se o receptor disser 'envie 64 KB', mas o transmissor souber que rajadas de mais de 32 KB se acumulam na rede, ele enviará 32 KB. Por outro lado, se o receptor disser 'envie 64 KB' e o transmissor souber que rajadas de até 128 KB são enviadas sem nenhum esforço, ele enviará os 64 KB solicitados. A janela de controle de fluxo foi descrita anteriormente, e a seguir descreveremos apenas a janela de congestionamento.

O controle de congestionamento moderno foi acrescentado ao TCP em grande parte por meio dos esforços de Van Jacobson (1988). Essa é uma história fascinante. A partir de 1986, a popularidade crescente da Internet de então levou à primeira ocorrência do que passou a ser conhecido como **colapso de congestionamento**, um período prolongado durante o qual o goodput caiu bruscamente (ou seja, por um fator de mais de 100) devido ao congestionamento na rede. Jacobson e muitos outros começaram a entender o que estava acontecendo e remediaram a situação.

O reparo de alto nível que Jacobson implementou foi aproximar uma janela de congestionamento AIMD. A parte interessante, e grande parte da complexidade do controle de congestionamento do TCP, é como ele acrescentou isso a uma implementação existente sem alterar nenhum formato de mensagem, tornando-o instantaneamente implementável. Para começar, ele observou que a perda de pacote é um sinal adequado de congestionamento. Esse sinal chega um pouco tarde (quando a rede já está congestionada), mas é bastante confiável. Afinal, é difícil montar um roteador que não descarte pacotes quando está sobrecarregado. Esse fato provavelmente não mudará. Mesmo quando aparecerem memórias de terabytes para manter grandes quantidades de pacotes em buffer, provavelmente teremos redes de terabits/s para preencher essas memórias.

Porém, usar a perda de pacotes como um sinal de congestionamento depende de os erros de transmis-

são serem relativamente raros. Isso normalmente não acontece para redes sem fios, como 802.11, motivo pelo qual elas incluem seu próprio mecanismo de retransmissão na camada de enlace. Devido às retransmissões sem fios, a perda de pacotes da camada de rede normalmente é mascarada. Ela também é rara em outros enlaces, pois os fios e as fibras ópticas normalmente têm baixas taxas de erro de bit.

Todos os algoritmos TCP da Internet consideram que os pacotes perdidos são causados por congestionamento e monitoram os períodos de tempo, procurando sinais de problemas da forma como os mineiros observam seus canários. É preciso que haja um bom timer de retransmissão para detectar sinais de perda de pacotes com precisão e em tempo. Já discutimos como o timer de retransmissão do TCP inclui estimativas da média e variação nos tempos de ida e volta. Consertar esse timer, incluindo um fator de variação, foi um passo importante no trabalho de Jacobson. Dado um bom período de tempo de retransmissão, o transmissor TCP pode acompanhar o número de bytes pendentes, que estão sobrecarregando a rede. Ele simplesmente observa a diferença entre os números de sequência que são transmitidos e confirmados.

Agora, parece que nossa tarefa é fácil. Tudo o que precisamos fazer é acompanhar a janela de congestionamento, usando números de sequência e confirmação, e ajustar a janela de congestionamento usando uma regra AIMD. Como você poderia esperar, a coisa é mais complicada. Uma consideração inicial é que o modo como os pacotes são enviados para a rede, até mesmo por pequenos períodos de tempo, deve corresponder ao caminho da rede. Caso contrário, o tráfego causará congestionamento. Por exemplo, considere um host com uma janela de congestionamento de 64 KB anexada a uma Ethernet comutada de 1 Gbps. Se o host enviar a janela inteira ao mesmo tempo, essa rajada de tráfego poderá passar por uma linha ADSL lenta de 1 Mbps mais adiante no caminho. A rajada que levava meio milissegundo na linha de 1 Gbps prenderá a linha de 1 Mbps por meio segundo, tumultuando completamente protocolos como o VoIP. Esse comportamento poderia ser uma boa ideia em um protocolo projetado para causar congestionamento, mas não em um protocolo para controlá-lo.

Entretanto, acontece que podemos usar pequenas rajadas de pacotes para o nosso proveito. A Figura 6.37 mostra o que acontece quando um transmissor em uma rede rápida (o enlace de 1 Gpbs) envia uma pequena rajada de quatro pacotes para um receptor em uma rede lenta (o enlace de 1 Mbps), que é o gargalo ou a parte mais lenta do caminho. Inicialmente, os quatro pacotes atravessam o enlace o mais rapidamente quanto o transmissor puder enviá-los. No roteador, eles são enfileirados enquanto são enviados, pois leva mais tempo para enviar um pacote pelo enlace lento do que para receber o próximo pacote pelo enlace rápido. Mas a fila não é grande, pois somente um pequeno número de pacotes foi enviado ao mesmo tempo. Observe o maior tamanho dos pacotes no enlace lento. O mesmo pacote, digamos, de 1 KB, agora é maior, pois leva mais tempo para ser enviado em um enlace lento do que em um enlace rápido.

Por fim, os pacotes chegam ao receptor, onde são confirmados. Os tempos para as confirmações refletem os tempos em que os pacotes chegaram ao receptor depois de cruzar o enlace lento. Eles são espalhados em comparação com os pacotes originais no enlace rápido. À medida que essas confirmações trafegam pela rede e retornam ao transmissor, elas preservam esse tempo.

A observação principal é esta: as confirmações retornam ao transmissor aproximadamente na velocidade em que os pacotes podem ser enviados pelo enlace mais lento no caminho. Essa é exatamente a velocidade que o transmissor deseja usar. Se ele injetar novos pacotes na rede nessa velocidade, eles serão enviados na velocidade que o enlace mais lento permite, mas não ficarão enfileirados nem congestionarão nenhum roteador ao longo do caminho. Esse tempo é conhecido como **clock ACK**. Essa é uma parte essencial do TCP. Usando um clock ACK, o TCP nivela o tráfego e evita filas desnecessárias nos roteadores.

A segunda consideração é que a regra AIMD levará um tempo muito grande para alcançar um bom ponto de operação em redes rápidas se a janela de congestionamento for iniciada a partir de um tamanho pequeno. Considere uma rede modesta que pode dar suporte a 10 Mbps com um RTT de 100 ms. A janela de congestionamento apropriada é o produto largura de banda-atraso, que é 1 Mbit ou 100 pacotes de 1.250 bytes cada um. Se a janela de congestionamento começar em 1 pacote e aumentar em 1 pacote a

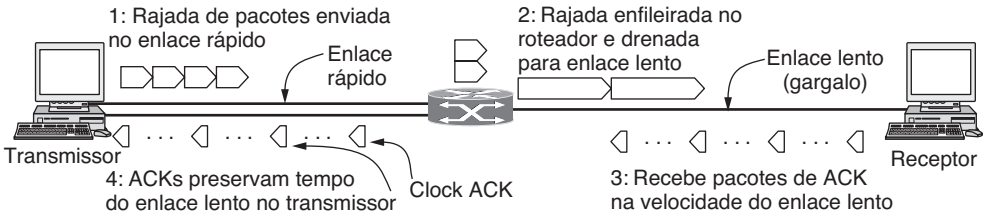

Figura 6.37 | Uma rajada de pacotes de um transmissor e o clock ACK retornando.

cada RTT, ela será de 100 RTTs ou 10 segundos antes que a conexão esteja rodando na velocidade correta. Esse é um longo tempo a esperar só para chegar à velocidade certa para uma transferência. Poderíamos reduzir esse tempo inicial começando com uma janela inicial maior, digamos, de 50 pacotes. Mas essa janela seria muito grande para enlaces lentos ou curtos. Isso causaria congestionamento se fosse usado tudo ao mesmo tempo, conforme acabamos de descrever.

Em vez disso, a solução que Jacobson achou para lidar com essas duas considerações é uma mistura de aumento linear e multiplicativo. Quando a conexão é estabelecida, o transmissor inicia a janela de congestionamento em um valor inicial pequeno de, no máximo, quatro segmentos; os detalhes são descritos na RFC 3390, e o uso de quatro segmentos é um aumento de um segmento a partir do valor inicial, com base na experiência. O transmissor então envia a janela inicial. Os pacotes levarão um tempo de ida e volta para ser confirmados. Para cada segmento que é confirmado antes que o timer de retransmissão expire, o transmissor soma os bytes correspondentes a um segmento na janela de congestionamento. Além disso, quando esse segmento tiver sido confirmado, existe um a menos na rede. O resultado é que cada segmento confirmado permite que mais dois sejam enviados. A janela de congestionamento está dobrando a cada tempo de ida e volta.

Esse algoritmo é chamado **partida lenta**, mas não é nada lento — ele tem crescimento exponencial —, exceto em comparação com o algoritmo anterior, que permitia que uma janela de controle de fluxo inteira fosse enviada ao mesmo tempo. A partida lenta pode ser vista na Figura 6.38. No primeiro tempo de ida e volta, o transmissor injeta um pacote na rede (e o receptor recebe um pacote). Dois pacotes são enviados no próximo tempo de ida e volta, depois quatro pacotes no terceiro tempo de ida e volta.

A partida lenta funciona bem para uma faixa de velocidades de enlace e tempos de ida e volta e utiliza um clock ACK para combinar a velocidade do transmissor com o caminho na rede. Dê uma olhada no modo como as confirmações retornam do transmissor ao receptor na Figura 6.38. Quando o transmissor recebe uma confirmação, ele aumenta a janela de congestionamento em um e imediatamente envia dois pacotes para a rede. (Um pacote é o aumento de um; o outro pacote é um substituto para o pacote que foi confirmado e saiu da rede. Em todo o tempo, o número de pacotes confirmados é dado pela janela de congestionamento.) Porém, esses dois pacotes não necessariamente chegarão ao receptor tão próximos um do outro quanto foram enviados. Por exemplo, suponha que o transmissor esteja em uma Ethernet de 100 Mbps. Cada pacote de 1.250 bytes leva 100 μs para ser enviado. Assim, o atraso entre os pacotes pode ser tão pequeno quanto 100 μs. A situação muda se esses pacotes atravessarem um enlace ADSL de 1 Mbps durante o caminho. Agora, são gastos 10 ms para enviar o mesmo pacote. Isso significa que o espaçamento mínimo entre os dois pacotes aumentou por um fator de 100. A menos que os pacotes tenham que esperar juntos em uma fila de um enlace mais adiante, o espaçamento continuará sendo grande.

Na Figura 6.38, esse esforço é mostrado impondo-se um espaçamento mínimo entre os pacotes de dados que chegam ao receptor. O mesmo espaçamento é mantido quando o receptor envia confirmações e, consequentemente, quando o transmissor recebe as confirmações. Se o caminho da rede for lento, as confirmações chegarão lentamente (após um atraso de um RTT). Se o caminho na rede for rápido, as confirmações chegarão rapidamente (novamente, após o RTT). Tudo o que o transmissor precisa fazer é seguir o tempo do clock ACK enquanto injeta novos pacotes, que é o que a partida lenta faz.

Como a partida lenta causa crescimento exponencial, em algum momento (mas não muito depois) ele enviará muitos pacotes para a rede muito rapidamente. Quando isso acontece, as filas se acumulam na rede. Quando as filas estiverem cheias, um ou mais pacotes serão perdidos. Depois que isso acontecer, o transmissor TCP expirará seu período de tempo quando uma confirmação não chegar em tempo. Na Figura 6.38, há evidência de um crescimento muito rápido da partida lenta. Depois de três RTTs, quatro pacotes estão na rede. Eles usam um RTT inteiro para chegar ao receptor. Ou seja, uma janela de congestionamento de quatro pacotes tem o tamanho certo para essa conexão. Porém, à medida que esses pacotes são confirmados, a partida lenta continua a aumentar a janela de congestionamento, alcançando oito pacotes em outro RTT. Apenas quatro deles podem alcançar o receptor em um RTT, não importa quantos sejam enviados; ou seja, o canal da rede está cheio. Pacotes adicionais colocados na rede pelo transmissor se acumularão nas filas do roteador, pois não podem ser entregues ao receptor com rapidez suficiente. Logo, haverá congestionamento e perda de pacotes.

Para conservar a partida lenta sob controle, o transmissor mantém um limite para a conexão chamado **limite**

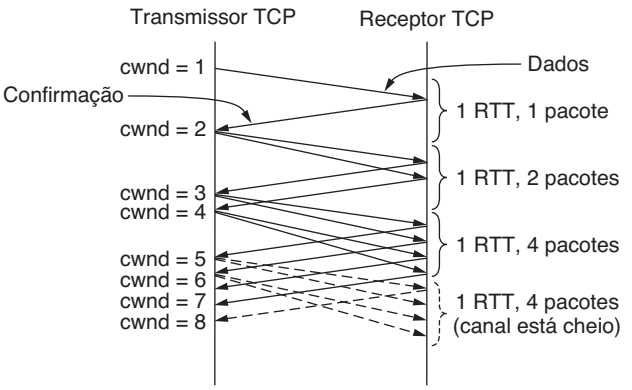

Figura 6.38 | Partida lenta de uma janela de congestionamento inicial de um segmento.

de partida lenta. Inicialmente, isso é definido com um valor arbitrariamente alto, com o tamanho da janela de controle de fluxo, de modo que não limitará a conexão. O TCP continua aumentando a janela de congestionamento na partida lenta até que um período de tempo expire ou a janela de congestionamento ultrapasse o limite (ou a janela do receptor esteja cheia).

Sempre que uma perda de pacote é detectada, por exemplo, por um período de tempo expirado, o limite de partida lenta é definido para a metade da janela de congestionamento e o processo inteiro é reiniciado. A ideia é que a janela atual é muito grande, pois causou um congestionamento anteriormente que só agora foi detectado por um período de tempo expirado. Metade da janela, que foi usada com sucesso em um momento anterior, provavelmente é uma melhor estimativa para uma janela de congestionamento que está próxima da capacidade do caminho, mas não causará perda. Em nosso exemplo da Figura 6.38, aumentar a janela de congestionamento para oito pacotes pode causar perda, enquanto a janela de congestionamento de quatro pacotes no RTT anterior foi o valor correto. A janela de congestionamento, então, retorna ao seu valor mínimo inicial e a partida lenta é retomada.

Sempre que o limite de partida lenta é ultrapassado, o TCP passa de partida lenta para aumento aditivo. Nesse modo, a janela de congestionamento é aumentada em um segmento a cada tempo de ida e volta. Assim como a partida lenta, isso normalmente é implementado com um aumento para cada segmento que é confirmado, ao invés de um aumento único por RTT. Sejam a janela de congestionamento *cwnd* e o tamanho máximo de segmento *MSS*. Uma aproximação comum é aumentar *cwnd* em (*MSS* × *MSS*)/*cwnd* para cada um dos *cwnd*/*MSS* pacotes que podem ser confirmados. Esse aumento não precisa ser rápido. A ideia completa é que uma conexão TCP gasta muito tempo com sua janela de congestionamento próxima do valor ideal — não tão pequeno para que a vazão seja baixa, nem tão grande para que ocorra congestionamento.

O aumento aditivo aparece na Figura 6.39 para a mesma situação da partida lenta. Ao final de cada RTT, a janela de congestionamento do transmissor terá crescido o suficiente para que possa injetar um pacote adicional na rede. Em comparação com a partida lenta, a taxa de crescimento linear é muito mais lenta. Ela faz pouca diferença para janelas de congestionamento pequenas, como acontece aqui, mas uma diferença grande no tempo gasto para aumentar a janela de congestionamento para 100 segmentos, por exemplo.

Há mais uma coisa que podemos fazer para melhorar o desempenho. O defeito no esquema até aqui é esperar por um período de tempo. Os períodos de tempo são relativamente longos, pois precisam ser conservadores. Após a perda de um pacote, o receptor não pode confirmar além dele, de modo que o número de confirmação permanecerá fixo, e o transmissor não poderá enviar novos pacotes para a rede, pois sua janela de congestionamento permanece cheia. Essa condição pode continuar por um período relativamente longo, até que o timer seja disparado e o pacote perdido seja retransmitido. Nesse estágio, o TCP inicia novamente a partida lenta.

Há um modo rápido para o transmissor reconhecer que um de seus pacotes se perdeu. Quando os pacotes além do pacote perdido chegam ao receptor, eles disparam confirmações que retornam ao transmissor. Essas confirmações contêm o mesmo número de confirmação. Elas são chamadas **confirmações duplicadas**. Toda vez que o transmissor recebe uma confirmação duplicada, é provável que outro pacote tenha chegado ao receptor e o pacote perdido ainda não tenha aparecido.

Como os pacotes podem tomar caminhos diferentes pela rede, eles podem chegar fora de ordem. Isso disparará confirmações duplicadas, embora nenhum pacote tenha sido perdido. Contudo, isso é raro na Internet na maior parte do tempo. Quando existe reordenação por vários caminhos, os pacotes recebidos normalmente não são reordenados em demasia. Assim, o TCP, de uma forma um tanto arbitrária, considera que três confirmações duplicadas significam que um pacote foi perdido. A identidade do pacote perdido também pode ser deduzida pelo número de confirmação. Ele é o próximo na sequência. Esse pacote pode então ser retransmitido imediatamente, antes que o período de tempo para retransmissão expire.

Essa heurística é chamada **retransmissão rápida**. Depois que ela é disparada, o limite de partida lenta ainda está definido como metade da janela de congestionamento atual, assim como em um período de tempo. A partida lenta pode ser reiniciada definindo-se a janela de congestionamento para um pacote. Com essa janela de congestionamento, um novo pacote será enviado após um tempo de ida e volta que é gasto para confirmar o pacote retransmitido junto

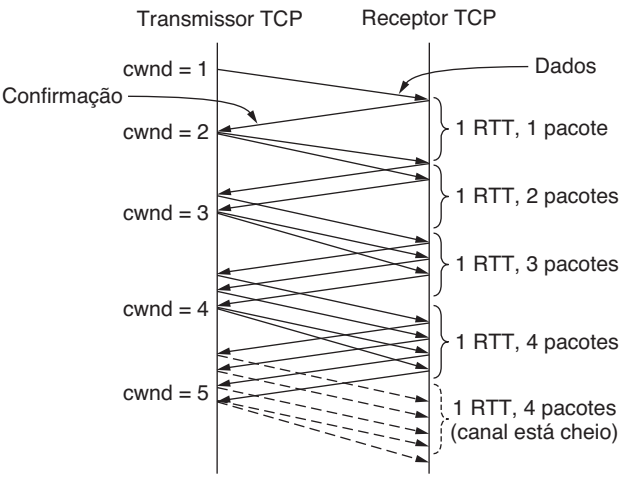

Figura 6.39 | Aumento aditivo a partir de uma janela de congestionamento inicial de um segmento.

com todos os dados que foram enviados antes que a perda fosse detectada.

Uma ilustração do algoritmo de congestionamento que criamos até aqui aparece na Figura 6.40. Essa versão do TCP é chamada TCP Tahoe, devido à versão 4.2BSD Tahoe em que ela foi incluída, em 1988. O tamanho máximo do segmento aqui é 1 KB. Inicialmente, a janela de congestionamento era de 64 KB, mas houve um período de tempo expirado, de modo que o limite é definido como 32 KB e a janela de congestionamento para 1 KB para a **transmissão 0**. A janela de congestionamento cresce exponencialmente até atingir o limite (32 KB). A janela aumenta toda vez que uma nova confirmação chega, e não de forma contínua, o que leva ao padrão descontínuo em escada. Após o limite ser ultrapassado, a janela cresce linearmente. Ela aumenta em um segmento a cada RTT.

As transmissões na rodada 13 não têm sorte (elas deveriam saber disso), e uma delas se perde na rede. Isso é detectado quando chegam três confirmações duplicadas. Nesse momento, o pacote perdido é retransmitido, o limite é definido para metade da janela atual (no momento, 40 KB, de modo que a metade é 20 KB) e a partida lenta é iniciada novamente. Reiniciar com uma janela de congestionamento de um pacote exige um tempo de ida e volta para todos os dados previamente transmitidos saírem da rede e ser confirmados, incluindo o pacote retransmitido. A janela de congestionamento cresce com a partida lenta, como fazia anteriormente, até alcançar o novo limite de 20 KB. Nesse momento, o crescimento se torna linear novamente. Ele continuará dessa forma até que outra perda de pacote seja detectada por confirmações duplicadas ou até que um período de tempo seja expirado (ou a janela do receptor chegue ao extremo).

O TCP Tahoe (que incluía bons timers de retransmissão) ofereceu um algoritmo de controle de congestionamento funcional, que resolveu o problema do colapso do congestionamento. Jacobson observou que é possível fazer ainda melhor. No momento da retransmissão rápida, a conexão está trabalhando com uma janela de congestionamento muito grande, mas ainda está trabalhando com um clock ACK funcional. Toda vez que outra confirmação duplicada chega, é provável que outro pacote tenha saído da rede. Usando confirmações duplicadas para contar os pacotes na rede, é possível permitir que alguns pacotes saiam da rede e continuem a enviar um novo pacote para cada confirmação duplicada adicional.

Recuperação rápida é a heurística que implementa esse comportamento. Esse é um modo temporário que visa a manter o clock ACK funcionando com uma janela de congestionamento que é o novo limite, ou metade do valor da janela de congestionamento no momento da retransmissão rápida. Para fazer isso, confirmações duplicadas são contadas (incluindo as três que dispararam a retransmissão rápida) até que o número de pacotes na rede tenha caído para o novo limite. Isso leva cerca de metade de um tempo de ida e volta. Daí em diante, um novo pacote pode ser enviado para cada confirmação duplicada recebida. Um tempo de ida e volta após a retransmissão rápida e o pacote perdido terá sido confirmado. Nesse momento, o fluxo de confirmações duplicadas terminará e o modo de recuperação rápida será encerrado. A janela de congestionamento será definida para o novo limite de partida lenta e crescerá pelo aumento linear.

O resultado dessa heurística é que o TCP evita a partida lenta, exceto quando a conexão é iniciada e quando um período de tempo expira. O último também pode acontecer quando mais de um pacote é perdido e a retransmissão rápida não se recupera adequadamente. Em vez de partidas lentas repetidas, a janela de congestionamento de uma conexão em execução segue um padrão **dente de serra** de aumento aditivo (por um segmento a cada RTT) e diminuição multiplicativa (pela metade em

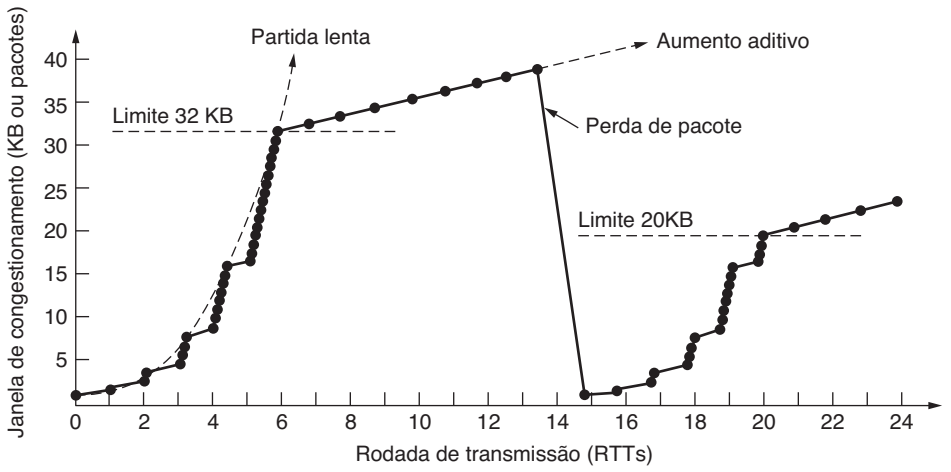

Figura 6.40 | Partida lenta seguida pelo aumento aditivo no TCP Tahoe.

um RTT). Essa é exatamente a regra AIMD que procuramos implementar.

Esse comportamento dente de serra aparece na Figura 6.41. Ele é produzido pelo TCP Reno, que tem o nome da versão 4.3BSD Reno de 1990, na qual foi incluído. O TCP Reno é basicamente o TCP Tahoe com recuperação rápida. Após uma partida lenta inicial, a janela de congestionamento sobe linearmente até que uma perda de pacote seja detectada pelas confirmações duplicadas. O pacote perdido é retransmitido e a recuperação rápida é usada para manter o clock ACK funcionando até que a retransmissão seja confirmada. Nesse momento, a janela de congestionamento é retomada a partir do novo limite de partida lenta, ao invés de 1. Esse comportamento continua indefinidamente, e a conexão gasta a maior parte do tempo com sua janela de congestionamento próxima do valor ideal do produto largura de banda-atraso.

O TCP Reno, com seus mecanismos para ajustar a janela de congestionamento, formou a base para o controle de congestionamento TCP por mais de duas décadas. A maior parte das mudanças nos anos intervenientes ajustou esses mecanismos de algumas formas, por exemplo, alterando as escolhas da janela inicial e removendo diversas ambiguidades. Algumas melhorias foram feitas para a recuperação de duas ou mais perdas em uma janela de pacotes. Por exemplo, a versão TCP NewReno usa um avanço parcial do número de confirmação após uma transmissão encontrar e reparar outra perda (Hoe, 1996), conforme descrito na RFC 3782. Desde meados da década de 90, surgiram diversas variações que seguem os princípios que descrevemos, mas que usam leis de controle ligeiramente diferentes. Por exemplo, o Linux usa uma variante chamada CUBIC TCP (He et al., 2008) e o Windows inclui uma variante chamada TCP composto (Tan et al., 2006).

Duas mudanças maiores também afetaram as implementações do TCP. Primeiro, grande parte da complexidade do TCP vem da dedução, por um fluxo de confirmações duplicadas, de quais pacotes chegaram e quais pacotes se perderam. O número de confirmação acumulativo não oferece essa informação. Um reparo simples é o uso de **SACK** (**Selective ACKnowledgements**), que lista até três intervalos de bytes que foram recebidos. Com essa informação, o transmissor pode decidir mais diretamente quais pacotes retransmitir e acompanhar os pacotes a caminho para implementar a janela de congestionamento.

Quando transmissor e receptor estabelecem uma conexão, cada um deles envia a opção do TCP *SACK permitido* para sinalizar que eles entendem as confirmações seletivas. Quando o SACK é ativado para uma conexão, ela funciona como mostra a Figura 6.42. Um receptor usa o campo de *Número de confirmação* do TCP normalmente, como uma confirmação acumulativa do byte mais alto na ordem que foi recebido. Ao receber o pacote 3 fora de ordem (porque o pacote 2 se perdeu), ele envia uma *opção SACK* para os dados recebidos junto com a confirmação acumulativa (duplicada) para o pacote 1. A *opção SACK* oferece os intervalos de bytes que foram recebidos acima do número dado pela confirmação acumulativa. O primeiro intervalo é o pacote que disparou a confirmação duplicada. Os próximos intervalos, se houver, são blocos mais antigos. Até três intervalos normalmente são usados. Quando o pacote 6 é recebido, dois intervalos de bytes SACK são usados para indicar que o pacote 6 e os pacotes 3 e 4 foram recebidos, além de todos os pacotes até o pacote 1. Pela informação em cada *opção SACK* que recebe, o transmissor pode decidir quais pacotes retransmitir. Nesse caso, a retransmissão dos pacotes de 2 a 5 seria uma boa ideia.

O SACK é informação estritamente consultiva. A detecção real de perda usando confirmações duplicadas e

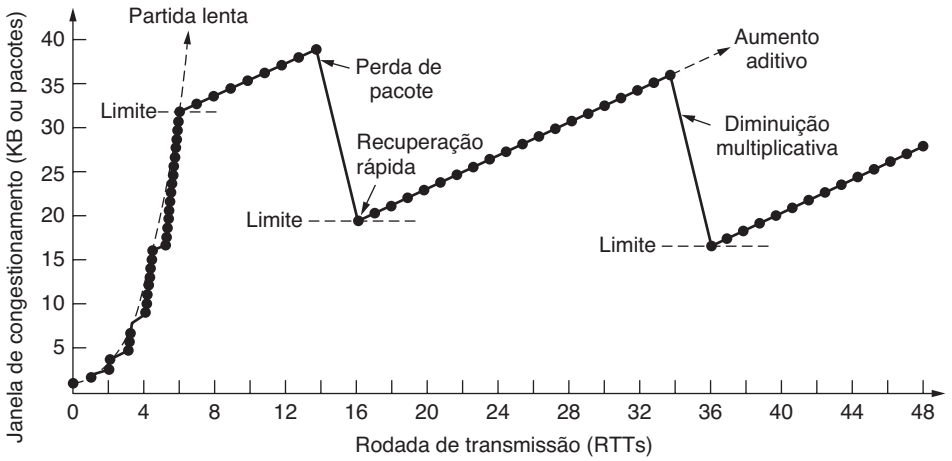

Figura 6.41 | Recuperação rápida e padrão dente de serra do TCP Reno.

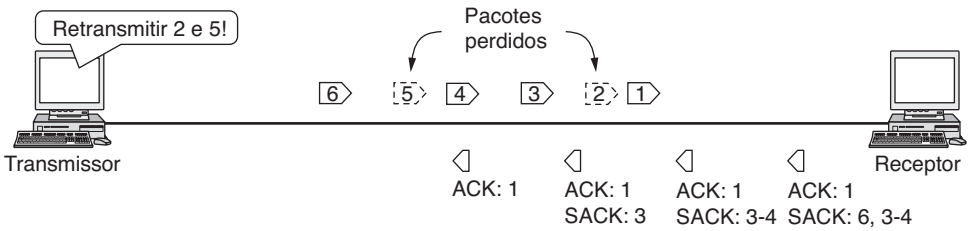

Figura 6.42 | Confirmações seletivas.

ajustes na janela de congestionamento prossegue exatamente como antes. Porém, com SACK, o TCP pode se recuperar mais facilmente de situações em que vários pacotes se perdem aproximadamente ao mesmo tempo, pois o transmissor TCP sabe quais pacotes não foram recebidos. O SACK agora é muito utilizado. Ele é descrito na RFC 2883, e o controle de congestionamento TCP usando SACK é descrito na RFC 3517.

A segunda mudança é o uso de ECN (Explicit Congestion Notification), além da perda de dados como um sinal de congestionamento. ECN é um mecanismo da camada IP para notificar os hosts quanto ao congestionamento que descrevemos na Seção 5.3.4. Com ele, o receptor TCP pode receber sinais de congestionamento do IP.

O uso de ECN é ativado para uma conexão TCP quando transmissor e receptor indicam que são capazes de usar ECN definindo os bits *ECE* e *CWR* durante o estabelecimento da conexão. Se o bit ECN for usado, cada pacote que transporta um segmento TCP é marcado no cabeçalho IP para mostrar que pode transportar um sinal ECN. Os roteadores que dão suporte ao mecanismo ECN definirão um sinal de congestionamento nos pacotes que podem transportar flags ECN quando o congestionamento estiver se aproximando, em vez de descartar esses pacotes após a ocorrência do congestionamento.

O receptor TCP é informado se qualquer pacote que chega transportar um sinal de congestionamento ECN. O receptor, então, usa a flag *ECE* (ECN-Echo) para sinalizar ao transmissor TCP que seus pacotes sofreram congestionamento. O transmissor diz ao receptor que ele ouviu o sinal por meio da flag *CWR* (Congestion Window Reduced).

O transmissor TCP reage a essas notificações de congestionamento exatamente da mesma maneira como faz com a perda de pacotes que é detectada por confirmações duplicadas. Porém, a situação é estritamente melhor. O congestionamento foi detectado e nenhum pacote foi prejudicado de forma alguma. O mecanismo ECN é descrito na RFC 3168. Ele exige suporte do host e do roteador, e ainda não é muito usado na Internet.

Para obter mais informações sobre o conjunto completo de comportamentos de controle de congestionamento implementados no TCO, consulte a RFC 5681.

6.5.11 O futuro do TCP

Como força motriz da Internet, o TCP tem sido usado em muitas aplicações e estendido com o passar do tempo para oferecer bom desempenho em topologias de redes cada vez maiores. Muitas versões são implantadas com implementações ligeiramente diferentes dos algoritmos clássicos que descrevemos, especialmente para controle de congestionamento e robustez contra ataques. É provável que o TCP continue a evoluir com a Internet. Nesta seção, vamos mencionar dois aspectos em particular.

O primeiro é que o TCP não oferece a semântica de transporte que todas as aplicações desejam. Por exemplo, algumas aplicações desejam enviar mensagens ou registros cujos limites precisam ser preservados. Outras aplicações trabalham com um grupo de conversações relacionadas, como um navegador Web que transfere vários objetos a partir do mesmo servidor. Ainda outras aplicações desejam melhorar o controle sobre os caminhos na rede que elas utilizam. O TCP com suas interfaces de soquetes-padrão não atende muito bem a essas necessidades. Basicamente, a aplicação tem o peso de lidar com qualquer problema não resolvido pelo TCP. Isso gerou propostas para novos protocolos que oferecessem uma interface ligeiramente diferente. Dois exemplos são SCTP (Stream Control Transmission Protocol), definido na RFC 4960, e SST (Structured Stream Transport) (Ford, 2007). Porém, toda vez que alguém propõe mudar algo que tenha funcionado tão bem por tanto tempo, há sempre uma imensa batalha entre estes dois lados: 'os usuários estão exigindo mais recursos' e 'se não quebrou, não conserte'.

O segundo é o controle de congestionamento. Você poderia esperar que esse fosse um problema resolvido após nossas deliberações e os mecanismos que foram desenvolvidos com o tempo. Não é bem assim. A forma de controle de congestionamento TCP que descrevemos, que é mais usada, é baseada em perdas de pacote como um sinal de congestionamento. Quando Padhye et al. (1998) modelaram a vazão do TCP com base no padrão dente de serra, eles descobriram que a taxa de perda de pacotes precisa cair rapidamente com o aumento de velocidade. Para alcançar uma vazão de 1 Gbps com um tempo de ida e volta de 100 ms e pacotes de 1.500 bytes,

6.6.6 Protocolos para redes longas de banda larga

Desde a década de 90, tem havido redes de gigabits que transmitem dados a grandes distâncias. Devido à combinação de uma rede rápida de banda larga 'fat networks' e de longo atraso, essas redes são chamadas **redes longas de banda larga**. Quando essas redes surgiram, a primeira reação das pessoas foi usar os protocolos existentes nelas, mas diversos problemas apareceram rapidamente. Nesta seção, discutiremos alguns dos problemas com o aumento da velocidade e o atraso dos protocolos de rede.

O primeiro problema é que muitos protocolos utilizam números de sequência de 32 bits. Quando a Internet começou, as linhas entre os roteadores eram principalmente linhas concedidas de 45 kbps, de modo que um host transmitindo em velocidade plena levava mais de uma semana para percorrer todos os números de sequência. Para os projetistas do TCP, 2^{32} era uma aproximação do infinito, pois havia pouco perigo de pacotes antigos ficarem rodando uma semana depois que fossem transmitidos. Com a Ethernet de 10 Mbps, esse tempo passou para 57 minutos, muito menor, mas ainda assim manejável. Com uma Ethernet de 1 Gbps jogando dados na Internet, o tempo é cerca de 34 segundos, muito abaixo do tempo de vida máximo do pacote de 120 segundos na Internet. De repente, 2^{32} não é uma aproximação tão boa do infinito, pois um transmissor rápido pode percorrer o espaço de sequência enquanto pacotes antigos ainda existem na rede.

O problema é que muitos projetistas de protocolo simplesmente consideraram, sem afirmar isso, que o tempo exigido para ocupar o espaço de sequência inteiro seria muito maior que o tempo de vida máximo do pacote. Consequentemente, não havia necessidade sequer de se preocupar com o problema de duplicatas antigas ainda existindo quando os números de sequência fossem reiniciados. Em velocidades de gigabits, essa suposição não declarada falha. Felizmente, é possível estender o número de sequência efetivo tratando o período de tempo que pode ser transportado como uma opção no cabeçalho TCP de cada pacote como os bits de alta ordem. Esse mecanismo é chamado PAWS (Protection Against Wrapped Sequence numbers) e é descrito na RFC 1323.

O segundo problema é que o tamanho da janela de controle de fluxo precisa ser bastante aumentado. Por exemplo, considere o envio de uma rajada de 64 KB de dados de San Diego para Boston a fim de preencher o buffer de 64 KB do receptor. Suponha que o enlace seja de 1 Gbps e o atraso unidirecional da velocidade da luz na fibra seja de 20 ms. Inicialmente, a $t = 0$, o canal está vazio, conforme ilustra a Figura 6.48(a). Somente 500 μs depois, na Figura 6.48(b), todos os segmentos estão na fibra. O segmento inicial agora estará em algum ponto nas proximidades de Brawley, ainda ao sul da Califórnia. Porém, o transmissor precisa parar até que ele receba uma atualização de janela.

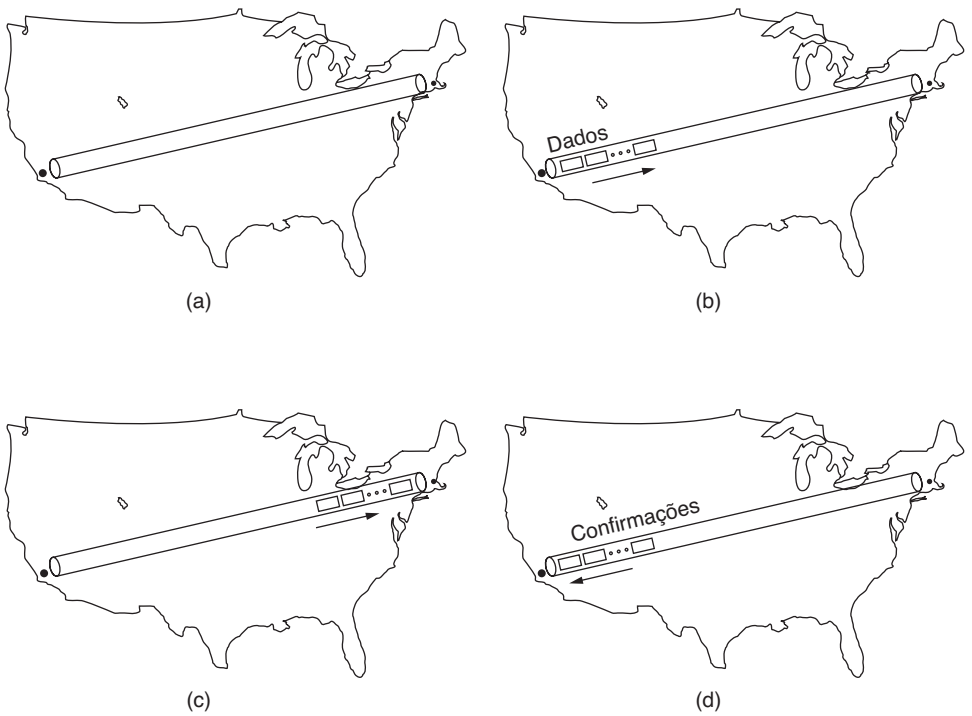

Figura 6.48 | O estado da transmissão de 1 Mbit de San Diego para Boston. (a) Em $t = 0$. (b) Após 500 μs. (c) Após 20 ms. (d) Após 40 ms.

Após 20 ms, o segmento inicial alcança Boston, como mostra a Figura 6.48(c), e é confirmado. Finalmente, 40 ms depois de iniciar, a primeira confirmação retorna ao transmissor e a segunda rajada pode ser transmitida. Como a linha de transmissão foi usada por 1,25 ms dos 100, a eficiência é de cerca de 1,25 por cento. Essa situação é típica de um protocolo mais antigo rodando por linhas de gigabits.

Uma quantidade útil para ter em mente ao analisar o desempenho da rede é o **produto largura de banda-atraso**. Ele é obtido multiplicando-se a largura de banda (em bits/s) pelo tempo de atraso de ida e volta (em segundos). O produto é a capacidade do canal do transmissor ao receptor e de volta (em bits).

Para o exemplo da Figura 6.48, o produto largura de banda-atraso é de 40 milhões de bits. Em outras palavras, o transmissor teria que transmitir uma rajada de 40 milhões de bits para poder continuar em velocidade plena até que a primeira confirmação retornasse. É necessário que haja esse número de bits para encher o canal (nas duas direções). É por isso que uma rajada de meio milhão de bits só atinge uma eficiência de 1,25 por cento: isso significa apenas 1,25 por cento da capacidade do canal.

A conclusão que podemos tirar aqui é que, para obter um bom desempenho, a janela do receptor precisa ser pelo menos tão grande quanto o produto largura de banda-atraso, e de preferência um pouco maior, pois o receptor pode não responder instantaneamente. Para uma linha de gigabit transcontinental, pelo menos 5 MB são necessários.

O terceiro problema, também relacionado, é que esquemas de retransmissão simples, como o protocolo go-back-n, não funcionam bem em linhas com um produto largura de banda-atraso grande. Considere o enlace transcontinental de 1 Gbps com um tempo de transmissão de ida e volta de 40 ms. Um transmissor pode enviar 5 MB em uma viagem de ida e volta. Se um erro for detectado, isso será 40 ms antes que o transmissor seja informado a respeito. Se o protocolo go-back-n for usado, o transmissor terá que retransmitir não apenas o pacote com problema, mas também os 5 MB de pacotes que vieram depois. É claro que isso é um grande desperdício de recursos. Protocolos mais complexos, como a repetição seletiva, são necessários.

O quarto é que as linhas de gigabits são fundamentalmente diferentes das linhas de megabits, pois as longas linhas de gigabits são limitadas por atraso em vez de limitadas por largura de banda. Na Figura 6.49, mostramos o tempo gasto para transferir um arquivo de 1 Mbit por 4.000 km em diversas velocidades de transmissão. Em velocidades de até 1 Mbps, o tempo de transmissão é dominado pela taxa em que os bits podem ser enviados. Por volta de 1 Gbps, o atraso de ida e volta de 40 ms é superior ao tempo de 1 ms gasto para colocar os bits na fibra. Outros aumentos na largura de banda dificilmente terão algum efeito.

A Figura 6.49 tem implicações infelizes para os protocolos de rede. Ela diz que os protocolos stop-and-wait, como RPC, têm um limite superior inerente em seu desempenho. Esse limite é ditado pela velocidade da luz. Nenhuma quantidade de progresso tecnológico na óptica conseguirá melhorar as coisas (porém, novas leis da física ajudariam). A menos que possa ser encontrado algum outro uso para uma linha de gigabits enquanto um host está esperando uma resposta, a linha de gigabits não é melhor do que uma linha de megabits, apenas mais cara.

Um quinto problema é que as velocidades de comunicação têm melhorado com mais rapidez que as velocidades de computação. (Nota para os engenheiros de computação: saiam e vençam os engenheiros da comunicação! Estamos contando com vocês.) Na década de 70, a ARPANET funcionava a 56 kbps e tinha computadores que funcionavam

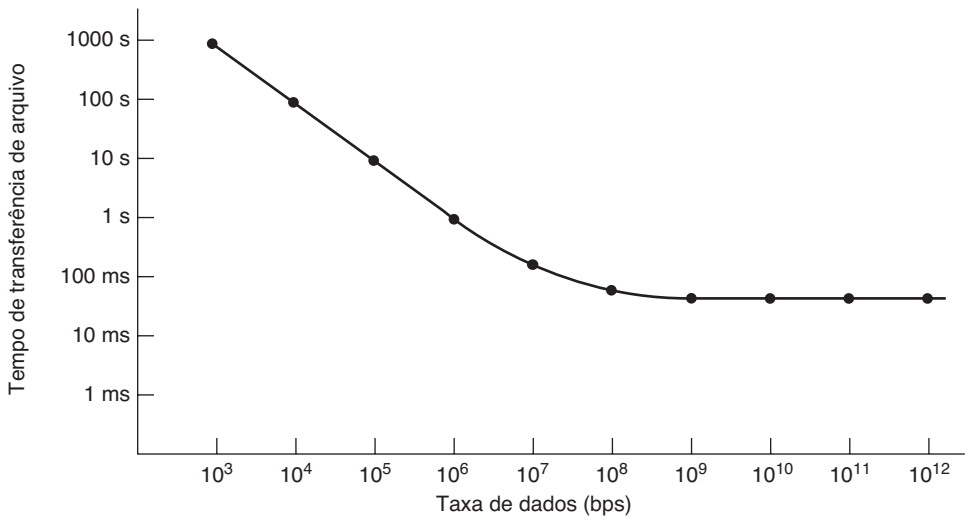

Figura 6.49 | Tempo para transferir e confirmar um arquivo de 1 Mbit por uma linha de 4.000 km.

em aproximadamente 1 MIPS. Compare esses números com computadores de 1.000 MIPS trocando pacotes por uma linha de 1 Gbps. O número de instruções por byte diminuiu por um fator de mais de 10. Os números exatos são discutíveis, dependendo das datas e cenários, mas a conclusão é esta: há menos tempo disponível para processamento de protocolo do que havia antes, de modo que os protocolos se tornaram mais simples.

Agora, vamos passar dos problemas para as maneiras de lidar com eles. O princípio básico que todos os projetistas de redes de alta velocidade precisam aprender de cor é:

Projete visando à velocidade, e não à otimização da largura de banda.

Os protocolos antigos normalmente eram projetados para minimizar o número de bits nos enlaces, geralmente usando campos pequenos e compactando-os em bytes e palavras. Essa preocupação ainda é válida para redes sem fios, mas não para redes de gigabits. O processamento de protocolo é o problema, de modo que os protocolos precisam ser projetados para minimizá-lo. Os projetistas do IPv6 certamente entenderam esse princípio.

Um modo atraente de aumentar a velocidade é criar interfaces de rede rápidas em hardware. A dificuldade com essa estratégia é que, a menos que o protocolo seja incrivelmente simples, o hardware simplesmente significa uma placa com uma segunda CPU e seu próprio programa. Para garantir que o coprocessador de rede seja mais barato que a CPU principal, ele normalmente é um chip mais lento. A consequência desse projeto é que grande parte do tempo na CPU principal (rápida) fica ocioso aguardando que a segunda CPU (lenta) realize o trabalho crítico. É um mito pensar que a CPU principal tem outro trabalho a fazer enquanto espera. Além do mais, quando duas CPUs de uso geral se comunicam, pode haver condições de *race*, de modo que protocolos complicados são necessários entre os dois processadores para sincronizá-los corretamente e evitar *races*. Normalmente, a melhor técnica é tornar os protocolos simples e deixar que a CPU principal faça o trabalho.

O layout de pacotes é uma consideração importante nas redes de gigabits. O cabeçalho deve conter o mínimo possível de campos, a fim de reduzir o tempo de processamento. Esses campos devem ser grandes o suficiente para realizar o trabalho e ter alinhamento de palavras com a finalidade de facilitar o processamento. Nesse contexto, 'grande o suficiente' significa que não ocorrerão problemas como repetição de números de sequência enquanto ainda existem pacotes antigos, receptores incapazes de anunciar espaço de janela suficiente porque o campo da janela é muito pequeno e assim por diante.

O tamanho máximo dos dados deve ser grande, para reduzir o overhead de software e permitir uma operação eficiente. Para redes de alta velocidade, 1.500 bytes é muito pouco, motivo pelo qual a Ethernet de gigabit admite quadros jumbo de até 9 KB e o IPv6 admite pacotes jumbograma com mais de 64 KB.

Agora, vamos examinar a questão do feedback nos protocolos de alta velocidade. Devido ao loop de atraso (relativamente) longo, o feedback deverá ser evitado: o receptor gasta muito tempo para sinalizar o transmissor. Um exemplo de feedback é controlar a taxa de transmissão usando um protocolo de janela de deslizante. Os protocolos do futuro poderão passar para protocolos baseados em taxa, para evitar os (longos) atrasos inerentes ao envio de atualizações de janela do receptor ao transmissor. Nesse protocolo, o transmissor pode enviar tudo o que desejar, desde que não envie mais rápido do que alguma taxa que o transmissor e o receptor combinaram antecipadamente.

Um segundo exemplo de feedback é o algoritmo de partida lenta de Jacobson. Esse algoritmo promove várias sondagens para verificar quanto a rede pode manipular. Em uma rede de alta velocidade, fazer meia dúzia de pequenas sondagens para ver como a rede se comporta desperdiça um grande volume de largura de banda. Um esquema mais eficiente é fazer com que o transmissor, o receptor e a rede reservem os recursos necessários no momento de estabelecer a conexão. Reservar recursos antecipadamente também oferece a vantagem de tornar mais fácil reduzir o jitter. Em resumo, buscar altas velocidades leva o projeto inexoravelmente em direção a uma operação orientada a conexões, ou algo bem parecido.

Outro recurso valioso é a possibilidade de enviar um volume normal de dados junto com a solicitação de conexão. Desse modo, é possível economizar o tempo de uma viagem de ida e volta.

6.7 Redes tolerantes a atrasos

Vamos terminar este capítulo descrevendo um novo tipo de transporte que um dia poderá ser um componente importante da Internet. O TCP e a maioria dos outros protocolos de transporte são baseados na hipótese de que o transmissor e o receptor estão continuamente conectados por algum caminho funcional, ou então o protocolo falha e os dados não podem ser entregues. Em algumas redes, normalmente não existe um caminho fim a fim. Um exemplo é uma rede espacial, como satélites LEO (Low-Earth Orbit) entrando e saindo do alcance das estações terrestres. Determinado satélite pode ser capaz de se comunicar com uma estação terrestre somente em certos momentos, e dois satélites podem nunca ser capazes de se comunicar um com o outro em momento algum, mesmo por meio da estação terrestre, pois um dos satélites pode sempre estar fora de alcance. Mais exemplos de redes envolvem submarinos, ônibus, telefones móveis e outros dispositivos com computadores para os quais existe conectividade intermitente, devido à mobilidade ou a condições extremas.